Martin Deixelberger

Benchmarking for Adaptive Shielding

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur(in)

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisors

Roderick Bloem

Bettina Könighofer

Institute of Applied Information Processing and Communications

Graz, August 2021

AFFIDAVIT

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 indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Acknowledgements

This master thesis was realized at the Institute of Applied Information Processing and Communications at the Graz University of Technology. I would like to thank everyone at the Institute who supported me with advice, knowledge, and coffee breaks. Thanks to my advisers and supporters Roderick Bloem, Bettina Könighofer, Stefan Pranger, Martin Tappler, and Nils Jansen.

I want to thank my friends and family for their support in the time of my studies. Especially, I want to thank my cluster for their attention, caring, and encouragement in a hard time of the COVID pandemic and isolation.

Finally, to my loving and supportive partner, Eva: my deepest gratitude. Your encouragement when the times got rough and forgiving for my failures. My heartfelt thanks.

Abstract

The application of artificial intelligence (AI) technologies heavily increased over the last year. These technologies are continuously adapted and find application in many fields. Intelligent controller, which uses technologies like reinforcement learning, learns behaviour from observations of the environment. In an open environment application, the learning enabled controller operates under uncertainty, which can end in unintended behaviour. In terms of safety, uncertainty is a problem, and to aim to ensure the controller, we prescribe safety requirements. Previous works create so-called safety shields that correct an existing controller if it is about to take unbearable safety risks. In this work, we focus on optimal shields, and unlike safety shields, they aim to prevent the controller from taking performance harming actions. However, so far, shields do not consider that an environment may not be fully known in advance and may evolve for complex control and learning tasks. We propose a new method for shield adaption to a changing environment. In particular, we base our method on problems that are sufficiently captured by potentially infinite Markov decision processes (MDP) and quantitative specifications such as mean payoff objectives. The shield is independent of the controller, which may take the form of a reinforcement learning agent. At runtime, our method builds an internal abstract representation of the MDP and continuously adapts this abstraction and the shield based on observations from the environment.

This thesis shows the adaptive shielding method via an urban traffic control problem. It targets the design and implementation of the shielding framework, evaluating the shielding method with different scenarios. The results contributed to the paper "Adaptive Shielding under Uncertainty" [Pra+20].

Keywords: Formal Verification, Markov Decision Process, Model Checking, Optimal Shielding, Reinforcement Learning, Software Design

Kurzfassung

In den letzten Jahren hat die Anwendung von Technologien, die künstliche Intelligenz (KI) nutzen, stark zugenommen. Diese Technologien werden kontinuierlich adaptiert und kommen in immer mehr Bereichen zum Einsatz. Zum Beispiel nutzen intelligente Regler Technologien wie Reinforcement Learning, um ihr Verhalten selbständig zu erlernen - basierend auf Beobachtungen der Umgebung. Bei der Verwendung von KI Reglern in offener Umgebung kann es aber auch zu Unsicherheiten kommen, die zu unvorhersehbarem Verhalten führen können. Problematisch wird dies, wenn besagtes Fehlverhalten zu Verletzungen von Menschen oder zu Sachschäden führt. Deshalb werden dem Regler Sicherheitsanforderungen vorgegeben, die verhindern sollen, dass es zu solchen unsicheren Zuständen kommt.

Wissenschaftliche Papers beschreiben in diesem Zusammenhang sogenannte Shields. Safety Shields sollen den Regler korrigieren, wenn er dabei ist, ein Sicherheitsrisiko einzugehen. In dieser Arbeit konzentrieren wir uns auf optimale Shields, dessen Ziel es ist, den Regler zu optimieren und Performance kritische Zustände zu korrigieren. Shields werden auf Basis der erfassten Umgebung generiert. Verändert sich diese Umgebung kann es passieren, dass die Shields nicht mehr optimal schützen. Wir präsentieren eine neue Methode zur Anpassung der Shields, die auf verändernde, dynamische Umgebung reagiert: Adaptive Shielding. Insbesondere stützen wir unsere Methode auf Problemerfassung durch Markov Decision Processes (MDP) und quantitative Spezifikationen wie Mean Payoff Objectives. Das Shield ist unabhängig vom Regler, der die Form eines Reinforcement Learning Agents annehmen kann. In der Laufzeit wird eine interne abstrakte Repräsentation des MDPs aufgebaut. Diese Abstraktion und das Shield werden mithilfe von Beobachtungen der Umgebung ständig angepasst. Diese Arbeit zeigt die Adaptive Shielding Methode anhand eines städtischen Verkehrssteuerungsproblems. Sie zielt auf den Entwurf und die Implementierung des Shielding Frameworks ab und evaluiert die Shielding Methode in verschiedenen Verkehrsszenarien. Die Ergebnisse sind in das Paper "Adaptive Shielding under Uncertainty" [Pra+20] eingeflossen.

Schlagwörter: Formale Verifikation, Markov Decision Processes, Model Checking, Optimal Shielding, Reinforcement Learning, Software-Entwurf

Contents

Chapter 1

Introduction

The artificial intelligence (AI) software market is a fast-growing business. The market growth forecast was over 50% last few years [Tra19], which leads to revenue of 10.1 billion USD in 2018 [Omd20].

The future of mobility becomes autonomous, and AI technologies will be the key for autonomous vehicles. Most major automobile manufacturers have advanced driver-assist systems already installed in vehicles. Furthermore, they become standard equipment over the last years. The automotive industry experiments with fully autonomous vehicles, while a significant challenge is to provide reliability and safety.

This trend leads to controllers which 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 [FP18a]. However, due to such controllers' complexity, it is practically infeasible to cover the entire input space of a system with test cases to guarantee safety or any level of performance. Learned controllers introduce additional challenges. In particular, it is difficult to achieve acceptable performance when encountering untrained scenarios, add new features to the without retraining or induce negative side effects.

One suitable technique that delivers theoretical guarantees regarding qualitative or quantitative objectives at runtime is **shielding** [Avn+19; Blo+15; Als+18]. Safety-shields use a qualitative objective that can already enforce safety by preventing an agent's unsafe actions. In this work, we consider shields that enforce *quantitative* measures [Avn+19]. At any time point, the shield reads the controller's command and may choose to alter it before passing it on to the environment.

This thesis's results contributed to the paper "Adaptive Shielding under Uncertainty" [Pra+20], which deals with the same adaptive shielding method in an urban traffic control problem. The source code and several videos of shielding in action can be found on `https://shieldai.github.io/AdaptiveShielding/`.

1.1 Problem Statement

We assume that the controller generally performs well, and the shield should only intervene when required. Therefore, shields should always be lightweight, and in combination with the controller, the controller should be active most of the time. A controller with learned behaviour can be shielded very effectively when it comes to unintended behaviour, which violates the wanted objectives.

Note, that if the controller and the shield have objectives that strongly diverse, any interference of the shield may cause a performance drop in the controller's objective. However, if the shield and the controller have similar performance measures, the shield may even improve the controller's performance by interfering (especially in unusual situations).

We base the decision whether the shield should interfere at any point in time on *quantitative measures*. In particular, we formalize a *performance objective* to be minimized by the shield, and an *interference cost* for changing the output of the controller. Intuitively, the shield needs to balance the cost of interfering with the decrease in performance of not interfering.

Our synthesis procedure assumes a stochastic environment, which includes the controller. It captures the task of shielding in an MDP, where actions denote changing the controller output. A policy for choosing actions is a concrete shield in this interpretation. We compute shields that maximize a single mean payoff objective obtained from combining the performance and the interference measure, thus guaranteeing maximal performance with minimal interference.

1.1.1 Need for Adaptivity

The computation of shields requires a faithful abstraction of the physical environment dynamics. In the case of an inaccurate model, the interference of the shield may be unjustified, which can result in drops for both objectives. Even if the model was accurate initially, in real-world scenarios, the environment in which the shield operates might change over time. Therefore, the shield has to adapt to the new situation and needs to adjust its environment model.

1.1.2 Need for Shielding Framework

For evaluating adaptive shields, we need a framework that allows us to synthesise shields and enforces the quantitative objectives at runtime. Furthermore, the framework has to support shield refinement. We have to improve the model continuously.

1.2 Proposed Solution

In this section, we want to discuss the idea of adaptive shielding and give an introduction to the model and the shielding framework. We use quantitive shields on MDPs, which models decision-making under uncertainty [Whi85].

1.2.1 Idea of Adaptive Shielding

In this work, we aim to address the problem of inaccurate modelling by proposing an approach based on model refinement and online estimation of transition probabilities. Initially, a first shield is synthesized from an initial abstract model of the environment (including the controller). During runtime, the shield is applied. Additionally, a monitor observes the behaviour of the environment and the controller. Every t time units, the synthesis procedure updates the abstract model and computes a new shield.

The adaptive synthesis process checks the discrepancies between the current model and the observed environment and controller. Due to continuous monitoring, the observation is continuously changing. Suppose discrepancies between model and observation that exceed some threshold were detected, the next synthesis procedure creates the new shield. The new shield replaces the previous one until further discrepancies are detected.

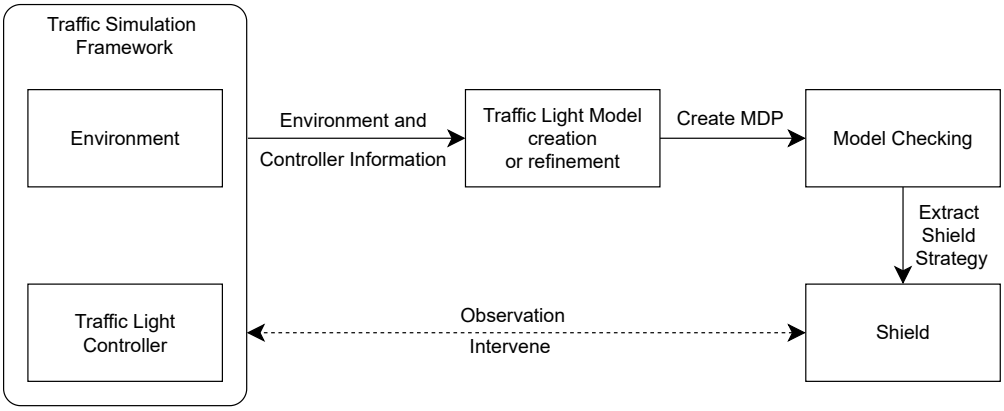

Figure 1.1: Workflow of first shield construction and shield refinement.

Figure 1.1 shows the workflow of the shield construction and the shield refinement after the first creation. We create the traffic light model from the environment and controller information with a model description and further refine it in a defined time interval. With model checking, we first build up the MDP and extract a shield strategy. The dashed line shows the interaction between the Shield and the Traffic Simulation Framework.

1.2.2 Updating the Models

In this work, we implement effective and efficient shielding based on two models capturing our environment's observations.

1. **Infinite-State MDP $\mathcal{M}$:**

   During runtime, we continuously update an infinite-state MDP $\mathcal{M}$ using our knowledge of the whole environment. We base an online estimation of the transition probabilities of $\mathcal{M}$ on observations of the environment.

2. **Abstracted Finite-State MDP $\mathcal{M}_{\square}$:**

   An abstraction $\mathcal{M}_{\square}$ of $\mathcal{M}$ allows us to compute shields efficiently. Whenever we deduce that our current abstraction $\mathcal{M}_{\square}$ is too coarse, we refine it by taking additional parts of $\mathcal{M}$'s state space into account. We also regularly update $\mathcal{M}_{\square}$ to ensure that changes of $\mathcal{M}$'s transition function are reflected in $\mathcal{M}_{\square}$.

Continuous updates of the probabilistic transition function of $\mathcal{M}$ and $\mathcal{M}_{\square}$ ensure that derived shields perform well in a constantly changing environment.

1.2.3 Generic Shields due to Adaptivity

An additional advantage of our adaptive shielding framework is the possibility to synthesize generic shields and apply them in different concrete settings. Here, initially we assume a generic environment where all environment changes, that is, transitions of $\mathcal{M}$, are equally likely. This model synthesizes a generic shield that can be applied in several concrete settings, and the shields adapt over time.

1.2.4 Case Study: Urban Traffic Control

We demonstrate the applicability of our framework by shielding traffic-light controllers in various road networks. In such a setting, an accident on the highway causes drastic changes in the traffic density on smaller roads, resulting in a traffic jam when using an unshielded traffic light controller. Due to the shielding approach's adaptive nature, the shielded controller can maintain the traffic flow through the city.

1.2.5 Shielding Framework Architecture

The implementation of the shielding framework was performed in the context of this master thesis. We created a shielding framework for shielding a traffic light controller in an urban traffic setup. We want to give an overview of the shielding framework's architecture, the used components, and how they are connected. Figure 1.2 shows a high-level structure of the shielding framework components for the basic setup to shield a static traffic light controller in a simulation. We can identify the following components:

- **Traffic Simulation Framework**

  The "Simulation of Urban Mobility" (`SUMO`) [Lop+18] traffic simulator is the platform for the traffic simulations. It allows us to observe them via the socket-based interface "Traffic Control Interface" (`TraCI`) [Lop+18].

- **Model Checker**

  The shield is constructed with model checking, a model-based verification technique known [CGP01]. The `Storm` model checker [Deh+17] gets a shield model and creates the shield MDP.

- **Shielding Application**

  The shielding framework's central part combines the simulation framework and the model checker. The Shielding application interacts with the simulation, creates a shield model, and handles the model checker results.

The traffic simulation framework and the model checker are standalone programs, and we have to deal with the given interfaces for interaction. The shielding agent has to communicate with both programs. Hence, we can identify two connection between components here:

- **Connect `SUMO` and Shielding Application**

  This connection uses `TraCI`, where `SUMO` provides a `TraCI` server and allows clients to connect.

- **Connect Shielding Application and `Storm`**

  The Shielding application deals with the `Storm` CLI interface and creates a file-based interface.

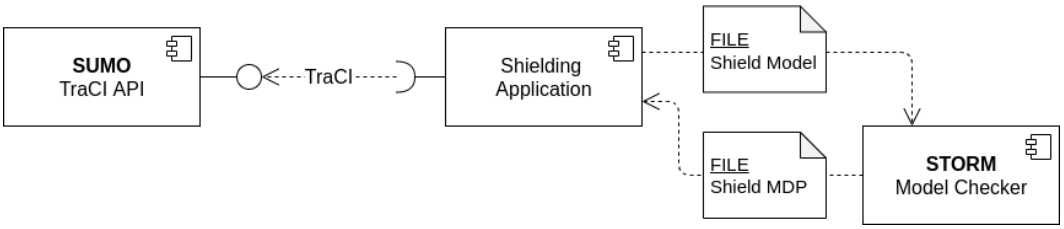

Figure 1.2: The high-level structure of the shielding framework, for shielding of a traffic light controller.

1.2.6 Add a Reinforcement Learning Agent to the Shielding Framework

To enable more advanced shielding scenarios, we can integrate an Intelligent Controller in the System as Traffic Light Controller. This controller is implemented as an RL agent and can optionally add to the system. The RL agent uses the `TraCI` Interface of `SUMO`, similar to the Shielding Application. RL agent and the Shielding Agent cannot communicate directly, and they are only manipulating the environment by setting the traffic light phase. Both agents can change the traffic light phase in one simulation step in a specified order.

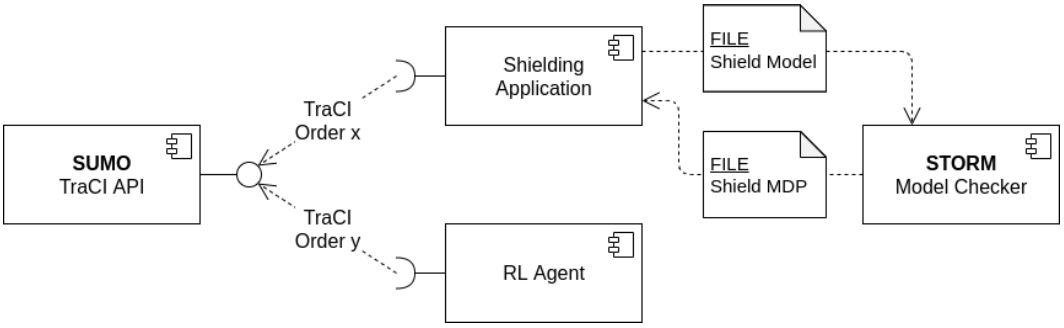

Figure 1.3: The high-level structure of the shielding framework, for shielding of an RL agent traffic light controller.

1.3 Outline

We presented our new method, discussed the adaptive shield idea with the problem and the proposed solution, and gave an overview of the shielding framework.

The remainder of this thesis is structured as follows. We review the related work and the journey to adaptive shielding in Chapter 2. Then, we establish notation and background in Chapter 3. We explain the adaptive shielding method and quantitative shielding in chapter 4. The implementation of the shielding framework will be discussed in Chapter 5. We present our experiments and results in Chapter 6 and, discuss the future work in Chapter 7. Finally, we give our conclusions in Chapter 8.

Chapter 2

Related Work

The general set of techniques that ensure the correctness of a controller at runtime is referred to as runtime enforcement (RE) [FP19; Ren+19]. The concept of a correct-by-construction *safety-shield* to enforce such correctness concerning a temporal logic specification, with the additional goal of minimal interference with the controller's output, was proposed first in [Blo+15; Als+18]. Several extensions exist [Bha+19; Wu+19]. Closest to our approach is [Avn+19] which proposed *optimal-shields* that enforce quantitative objectives at runtime.

Finally, shields for MDPs are relevant for the area of safe reinforcement learning [GF15; FP18b], and safe reinforcement learning via shielding has been considered in [Als+18; Jan+20; Bou+19]. Some approaches guide reinforcement learning via temporal logic constraints towards the verification of MDPs [HAK20; Hah+19].

2.1 Runtime Enforcement

Runtime Enforcement is a verification technique that uses an enforcement mechanism that modifies a running system's executions. It enforces the correct behaviour of running systems concerning the desired property. Schneider [Sch00] pioneered RE with security automata, enforcing safety properties with a runtime mechanism. Yli'es Falcone et al. [FP19] have focused on timed properties and present a framework for the RE of timed properties where timed automata describe specifications. Matthieu Renard et al. [Ren+19] deals with RE of untimed and timed properties with uncontrollable events.

2.2 Safety-Shield

Bloem et al. [Blo+15] presented the synthesis of safety-shields as a correct-by-construction concept that enforces correctness concerning a temporal logic specification. The work includes a framework to synthesise safety-shields, which enforcement monitors for reactive systems, from a set of safety properties. Where model checking and reactive synthesis fail, Shield synthesis can succeed. Instead of the complex design, it considers a small set of critical properties. The shield continuously monitors the input and output and corrects a wrong output if necessary.

2.3 Safe Reinforcement Learning

Reinforcement learning allows finding optimal policies while exploring an environment and evaluating the result due to the action's reward. There is no guarantee that the learned strategy is correct or does not violate a given specification, defined safe RL as follows:

> "Safe Reinforcement Learning can be defined as the process of learning policies that maximize the expectation of the return in problems in which it is important to ensure reasonable system performance and/or respect safety constraints during the learning and/or deployment processes." [GF15]

Since shielding aims to provide formal correctness and safety guarantees to a system, a shield for MDPs is relevant for safe reinforcement learning [GF15; FP18b].

Alshiekh et al. [Als+18] introduce a new approach to safe reinforcement learning via shielding. A shield will be added to the system and acts each time the learning agent makes a decision. The shield observes the reinforcement learning agent and can alter its actions when it violates the safety properties. The shield enforcing properties expressed in temporal logic. Jansen et al. [Jan+20] work focuses on the efficient construction of a safety shield for reinforcement learning. The shield construction is based on MDPs as an underlying model and aims to incorporate uncertainty. The probabilistic shields prevent the agent from taking unsafe actions while optimizing the performance objective.

The discussed safe reinforcement learning via shielding is not taking changing environments into account. In this work, we consider a changing environment and act by adopting the shield according to the observations.

Through online estimation of transition probabilities, we learn an MDP model on-the-fly during runtime. Active automata learning techniques for MDPs [CN12; Mao+16; Tap+19] also estimate transition probabilities on-the-fly and learn a structure in parallel. While we assume a known structure, we consider a larger state-space, and we learn from a single prefix of a path. In contrast, automata learning techniques usually sample many finite paths.

2.4 Quantitative-Shield

Safety shields follow a qualitative objective and aim to prevent the agent from taking an unsafe action. Guy Avni et al. [Avn+19] propose optimal shields, which guaranteeing certain performance and interfere as little as possible. The model is based on weighted automata, and the shields are generated by finding an optimal strategy in a stochastic 2-player game (controller versus shield). The optimal shields are demonstrated and automatically constructing for learned traffic-light controllers in various road networks.

We extend this work by dealing with the consequences of an incorrect or incomplete model used for the shield's computation. In particular, we use abstraction refinement and online probability estimation to adjust to new situations.

Chapter 3

Preliminaries

In this chapter, we introduce the required models and properties considered in this work. Furthermore, we give an introduction to model checking, Design Patterns and Reinforcement Learning.

A **tuple** is a finite ordered list $t = (t_0, t_1, .., t_n)$ of elements t_i. We use 1-based indexed access on tuples which we denote by $t[i]$, where i is an index of the element t_i of the tuple. The notation $t[I \leftarrow k]$ denotes overwriting of the values in a tuple t at indexes given by the set I with values given by a function k, that is, $(t[I \leftarrow k])[i] = k(i)$ for $i \in I$ and $(t[I \leftarrow k])[i] = t[i]$ for $i \notin I$.

A **word** is a finite or infinite sequence of elements from some alphabet Σ. The set of finite words over Σ is denoted 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^∞.

A **probability distribution** over a (finite) 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 $Distr(X)$.

A **Markov decision process** (MDP) [Hen+12; BK08] is a tuple $\mathcal{M} = (S, s_0, \mathcal{A}, \mathcal{P})$ where

- $\mathcal{S}$ is a finite set of states,
- $s_0 \in \mathcal{S}$ is a unique initial state,
- $\mathcal{A} = \{a_1 \ldots, a_n\}$ is a set of actions,
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to Distr(\mathcal{S})$ is a *probabilistic transition function.*

MDPs can model systems that have both probabilistic and non-deterministic state transition. Therefore, MDPs can be used to model non-deterministic choices of actions at each state. For all $s \in \mathcal{S}$ we have available actions $\mathcal{A}(s)$ for the possible action states.

A **path fragment** in an MDP $\mathcal{M}$ is a finite (or infinite) sequence $\rho = s_0 a_0 s_1 a_1 \ldots$ such that $\mathcal{P}(s_i, a_i, s_{i+1}) > 0$ for all $i \geq 0$.

A **policy** can resolve non-deterministic choices in an MDP. For the properties that we consider in this work, memoryless deterministic policies are sufficient [BK08]. These are functions $\pi : \mathcal{S} \to \mathcal{A}$ with $\pi(s) \in \mathcal{A}(s)$. We denote the set of all memoryless deterministic policies π of an MDP by Π.

A **Markov chain** [BK08] (MC) is a tuple $\mathcal{D} = (\mathcal{S}, s_0, \mathcal{P})$ where

- $\mathcal{S}$ is a finite set of states,
- $s_0 \in \mathcal{S}$ is a unique initial state,
- $\mathcal{P} : \mathcal{S} \to \mathit{Distr}(\mathcal{S})$ is a *probabilistic transition function.*

Applying a policy π on an MDP resolves all non-determinism is resolved. An MDP with resolved non-determinism becomes an MC, a fully probabilistic system.

A **cost function** $c : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_{\geq 0}$ for an MDP $\mathcal{M}$ adds a cost to every state s and action a enabled in s. For an infinite path $\rho = s_0 a_0 s_1 a_1 \ldots$, the cost function $c(s_i, a_i)$ returns the cost for every transition at step $i \geq 0$.

An **objective** [Brá+15] makes it possible to evaluate the MDP paths according to the costs or reward added by the cost function to the MDP. It can be seen as a function that takes all cost of the path and outputs a single value.

A **mean-payoff objective [CKK17; Brá+15]** assigns to every infinite path the long-run average of the costs of the path. With a given a strategy π, the *n-step average cost* then is

$$v_n^\pi(s) = \mathbb{E}_s^\pi\left[\frac{1}{n}\sum_{i=0}^{n-1} c(s_i, a_i)\right],$$

We denote by $\mathbb{E}_s^\pi[\cdot]$ the strategy's expectation measure [Brá+15] given a starting state s. And the *long-run average cost* of the strategy π is

$$v^\pi(s) = \limsup_{n\to\infty} v_n^\pi(s).$$

For finite MDPs, the optimal limit-superior (also called the *value*) is obtained by some memoryless deterministic strategy $\pi^\star \in \Pi$.

$$v^\star(s) = \min_{\pi\in\Pi} v^\pi(s) = \lim_{n\to\infty} v^{\pi^\star}.$$

In this work, we compute an approximate solution for the mean-payoff optimization problem [Brá+15] with two objectives. We compute the solution for a single mean-payoff objective, where the objective is obtained as a weighted sum of the objectives for which the Pareto curve is generated [Brá+15]. The weights are selected in a way similar to [Avn+19], allowing us to obtain the approximation of the curve. Given two cost functions c_1 and c_2 and a factor $\gamma \in [0, 1]$ with which we weigh the two costs, we compute the *n-step average weighted cost* with

$$v_n^\pi(s) = \mathbb{E}_s^\pi\left[\frac{1}{n}\sum_{i=0}^{n-1}(\gamma \cdot c_1(s_i, a_i) + (1-\gamma)\cdot c_2(s_i, a_i))\right].$$

An *optimal* policy π^* minimizes the long-run weighted average cost in the initial state, for instance, with an initial state s_0 where

$$\pi^* = \operatorname*{argmin}_{\pi\in\Pi} \limsup_{n\to\infty} v_n^\pi(s_0).$$

An optimal shield implements an optimal policy π^*.

3.1 Model Checking

Errors and flaws in a system can lead to fatal indices and considerable costs to correct the issues. Therefore, we want to verify that software and hardware are correct in designing complex systems. The time and effort to verify the system's correctness is usually high and often more extensive than the construction time. The early integration of formal methods [BK08] in the design process can reduce verification time and increasing coverage.

Model checking [BK08; Bai16] aims to find errors in the system with a mathematical model that describes the system behavior. We can assume that a system is correct when the model checker does not find errors. Nevertheless, this does not guarantee that there are no errors in the system. Model checking has limits, and discrepancies between the model and the system can lead to flaws. Therefore, error reports should be checked against the actual implementation.

The system model is usually automatically generated from a model description and provided to the model checker with a property. The model checker explores all states of the model and checks if the specific property is satisfied. The properties are typical of a qualitative nature, like can the system reach a certain state. If the model can not satisfy the property, the model checker returns debugging information [BK08]. The property violations cause the model checker to provide a counterexample, which describes a path that reaches the violation state that violates the property from the initial state. The debug information and counterexample of the model checker can be used to resolve the violation scenario. In combination with a simulator, it is possible to replay the violation scenario and continuously adapt the model or the property.

In this work, we focus on quantitative model checking, where we check the model of a system against a quantitative property. We want to find a path in the model that leads to the best result.

3.2 Reinforcement Learning

Reinforcement Learning (RL) [SB98; KLM96] demonstrated solving complex tasks like computer games [Sil+16] and applications in robotics [Wan+19]. In an RL learning setting, we have a learning agent that learns its behaviour with trial-and-error by exploring an unknown environment. In each timestep, the agent observes its environment state, and according to the state, the agent chooses an action. The action changes the environment's state, and the agent evaluates the action over a reward signal. The RL problems learn how to map action to a specific situation. The agent tries to archive a specific goal and only explores the environment by choosing actions and observes the influences on the environment. Therefore, we have no prior knowledge of the problem and consider that one action influences the reward in the following steps. The agent should exercise actions that increase the expected sum of reward signal values in the long run. In the exploration phase, the agent should find an optimal policy. After the training, we can not guarantee that the agent does not execute unsafe actions.

Chapter 4

Adaptive Shielding Method

This chapter discusses the Adaptive Shielding Method. First, in section 4.1, we review the quantitative shielding setting with the environment and controller. In section 4.2, we want to highlight the adaptive shielding approach and the requirements for the shielding framework.

4.1 Adaptive Shielding Setting

In this section, we define the setting in which we apply adaptive shields, discuss the quantitative shield synthesis approach and the used models.

4.1.1 Quantitative Shielding

We consider a stochastic environment that includes a global controller that issues local commands or various local controllers collaborating to achieve some global goal optimally. The shielding framework proposes to attach local shields to alter local commands. Hence, we synthesize each local shield individually concerning the environment and a local controller. A shield can be seen as a proxy between the controller and the environment. At each time step, the controller reads the state of the environment and issues a command. The controller's command will not directly feed to the environment, and the shield first reads it along with an abstract state of the environment. Then, the shield can decide to keep the controller's command or change it before issuing the command to the environment. A shield minimizes two quantitative runtime measures: the shield performance objective and the interference costs. When combining the shield and the controller, the controller should be active most of the time, and the shield intervenes only when required. Therefore, the shield needs to balance the cost of interfering with the decrease in the shield's performance of not interfering.

4.1.2 Quantitative Shield Synthesis

We are given an abstraction MDP $\mathcal{M}_{\sqcup}$ with two cost functions: c_1 denotes the performance objective of the shield, and c_2 denotes the costs for interference. A factor $\gamma \in [0, 1]$ weighs the two scores. A quantitative shield is computed by solving the MDP $\mathcal{M}_{\sqcup}$ with respect to the with γ weighted mean payoff objective of c_1 and c_2 and implements an optimal policy $\pi^\star \in \Pi$ that achieves an optimal value $v^\star$.

4.1.3 Quantitative Shielding Example

As a running example, we consider a global traffic light controller that minimizes the total waiting time of all cars in the city. A local shield overwrites the command of the controller of a single junction. The shield performance objective could supplement the controller's objective, e.g., by balancing the number of waiting for cars per incoming road. In unforeseen scenarios, e.g., with unusually dense traffic flow in one direction, the shield may improve the total waiting time (the controller's objective) by interfering. A simple example of an interference score charges the shield 1 for every change of action and charges 0 when no change is made.

4.1.4 Models

In this section, we want to discuss the modelling formalism of the traffic example, including the environment and the controller. We describe the transformation of an infinite-state model to a finite-state model by using abstraction.

Infinite-State Model $\mathcal{M}$

We consider an environment including a controller, for which we assume stochastic behaviour. The stochastic behaviour determines all actions of the controller via probabilities. Thus the only non-determinism occurs in the shield's actions. Combining these models yields an MDP $\mathcal{M} = (S, s_0, \mathcal{A}, \mathcal{P})$. We assume that each state $s \in \mathcal{S}$ of $\mathcal{M}$ is a tuple of n discrete variables $s = (v_1, \ldots, v_n)$. Each variable v_i is associated with a domain of possible values $D_{vi} \subseteq \mathbb{N}$, i.e., $\mathcal{S} = D_{v1} \times \cdots \times D_{vn}$. The state space $\mathcal{S}$ for modelling the environment may be infinite, but it needs to be countable. Furthermore, we assume that for each state-action (s, a) the support of $\mathcal{P}(s, a)$ is finite, that is, there are finitely many possible successor states following (s, a).

Infinite-State Model Example

In our example of urban traffic control, the dimensions of the state space of $\mathcal{M}$ represent the number of cars waiting per road and the current command of the controller, i.e., states have the form (n, s, e, w, ctr), see Figure 4.1. The value ctr denotes the current command of the controller. For instance, NS_c denotes that the north-south direction should get the green light next. Hence, ctr can take finitely many values, whereas the other values are theoretically unbounded. The shield decides the following setting of the traffic light. It can either give the same command as the controller (here $NS_{\square}$) or decide to deviate (here $EW_{\square}$). Upon issuing the action, the probabilities capture the likeliness of the next queue sizes (influenced by the assumed distribution of incoming cars and the current traffic light setting commanded by the shield) combined with the chance of the controller's next command. The assumption of finite successor states is fulfilled, as the number of cars approaching a traffic light in a single time step is limited.

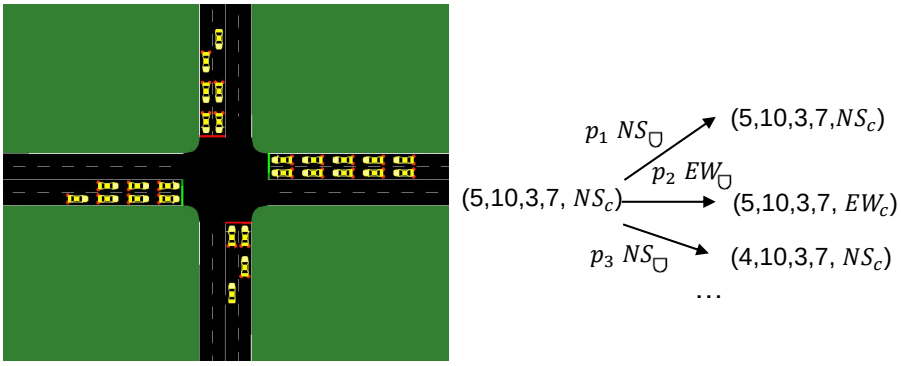

Figure 4.1: On the left, a concrete state depicted in the traffic simulator SUMO. On the right, we depict the corresponding state of $\mathcal{M}$ and some outgoing transitions.

Finite-State Abstraction $\mathcal{M}_{\cup}$ of $\mathcal{M}$

We use a finite-state abstraction $\mathcal{M}_{\cup}$ as an underlying model to synthesize the shield. The idea is that $\mathcal{M}_{\cup}$ and $\mathcal{M}$ are equivalent on regularly visited states of the state space, and rarely visited states are basically abstracted away. We define the abstraction MDP $\mathcal{M}_{\cup}$ by limiting the values that variables $v_i \in s$ can take. A cut-off function k defines the maximal value for each variable $v_i \in s$, i.e., $v_1 \leq k(1), \ldots, v_n \leq k(n)$.

Definition 1 (Domain-constrained Abstraction)**.** *Given an MDP $\mathcal{M} = (S, s_0, \mathcal{A}, \mathcal{P})$ with $\mathcal{S} = D_{v1} \times \cdots \times D_{vn}$ and $D_{vi} \subseteq \mathbb{N}$ and cut-off function k with $k(i) \in D_{vi}$ for $i \in [1\,..\,n]$. The abstraction MDP $\mathcal{M}_{\cup} = (\mathcal{S}_{\cup}, s_{0_{\cup}}, \mathcal{A}_{\cup}, \mathcal{P}_{\cup})$ has the following components:*

- $\mathcal{S}_{\cup} = D_{v1_{\cup}} \times \cdots \times D_{vn_{\cup}}$ *with* $D_{vi_{\cup}} = \{x \mid x \in D_{vi} \text{ and } x \leq k(i)\}$ *is the state space,*
- $s_{0_{\cup}} = (s_0[I \leftarrow k])$ *with* $I = [1\,..\,n]$ *is the initial state,*
- $\mathcal{A}_{\cup}(s) = \mathcal{A}(s)$ *are actions available in* $s \in \mathcal{S}_{\cup}$*, and*
- $\mathcal{P}_{\cup}$ *is the probabilistic transition function. For all* $I \subseteq [1..n]$ *we define the state sets*

$$\mathcal{S}^I_{\geq} = \{s \in \mathcal{S} \mid \forall j \in I.\ s[j] \geq k(j) \wedge \forall j \notin I.\ s[j] < k(j)\},$$
$$\mathcal{S}^I_{\text{abs}} = \{s[I \leftarrow k] \mid s \in \mathcal{S}^I_{\geq}\}.$$

For all $s \in \mathcal{S}_{\cup}$*,* $a \in \mathcal{A}_{\cup}(s)$*,* $I \subseteq [1..n]$*, and* $s' \in \mathcal{S}^I_{\text{abs}}$*,* $\mathcal{P}_{\cup}$ *is defined by*

$$\mathcal{P}_{\cup}(s,a)(s') = \sum_{s'' \in \mathcal{S}^I_{\geq} : s''[I \leftarrow k] = s'} \mathcal{P}(s,a)(s'').$$

With an empty index set I *we have a special case, where*

$$\mathcal{P}_{\cup}(s,a)(s') = \mathcal{P}(s,a)(s')$$

for all states s*,* s' *where for all* $v_i \in s$*,* $v_j \in s'$ *it holds that* $v_i < k(i)$ *and* $v_j < k(j)$*.*

After the cut-off, the states' transitions are combined transitions of the original MDP $\mathcal{M}$. We sum their probabilities and apply them to the state at the cut-off point. For any index set I, $\mathcal{S}^I_{\geq}$ contains all states whose variables indexed in I have values greater than or equal to their cut-off values. The set $\mathcal{S}^I_{abs}$ contains all states where the indexed variables have exactly the cut-off value. Transitions to states in $\mathcal{S}^I_{abs}$ in $\mathcal{M}_{\square}$ are combined transitions to states in $\mathcal{S}^I_{\geq}$ in $\mathcal{M}$, therefore we sum their probabilities to create $\mathcal{P}_{\square}$.

Finite-State Abstraction Example

The values of the variables n, s, e, and w that form the states of $\mathcal{M}$ are unbounded. To enable synthesis, we lump states in the abstraction $\mathcal{M}_{\square}$ together, where n, s, e, or w exceed specific thresholds. The domain of ctr is the same for $\mathcal{M}$ and the abstraction $\mathcal{M}_{\square}$.

4.2 Adaptive Shielding Framework

In this section, we discuss our adaptive shielding approach for deriving optimal shields concerning our knowledge of the environment.

The approach works as follows: We start from an initial infinite-state MDP $\mathcal{M}$ that models the interaction of the environment, the controller, and the shield. From $\mathcal{M}$, we derive a finite-state abstraction $\mathcal{M}_{\square}$ and synthesize an initial shield that we use during runtime. The following steps ensure adaptivity at runtime.

1. **Continuous Updates of $\mathcal{M}$:**

   During runtime, we monitor the environment and use the collected observations to update the estimation of the transition function of $\mathcal{M}$.

2. **Periodical Updates of $\mathcal{M}_{\square}$:**

   During runtime, we monitor whether states of $\mathcal{M}$ were visited that are abstracted away in the current $\mathcal{M}_{\square}$. Every t time units, we create a new abstraction $\mathcal{M}_{\square}$ from the current model $\mathcal{M}$, with additional refinement if necessary. From the new $\mathcal{M}_{\square}$, we synthesize a new shield and use it to replace the old one.

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

4.2.1 Continuous Updates of $\mathcal{M}$

As introduced above, we use $\mathcal{M} = (S, s_0, \mathcal{A}, \mathcal{P})$ to denote the infinite-state MDP underlying the synthesis environment. Since the structure of $\mathcal{M}$ is known, learning about the environment amounts to estimating $\mathcal{M}$'s probabilistic transition function $\mathcal{P}$.

In the following, we follow Chen and Nielsen's formalization of online learning of transition probabilities [CN12]. For each state s and action a, there is a multinomial distribution $\mathcal{P}(\mathcal{S}^s|s, a, \theta^{(s,a)})$ parameterized by $\theta^{(s,a)}$ over the successor states $\mathcal{S}^s$ of s that we need to estimate. As is common [CN12; BH08], we express the uncertainty about the transition probability distributions through prior Dirichlet densities $\mathcal{P}(\theta^{(s,a)})$. We use a

Dirichlet distribution for this purpose, since it is the conjugate prior of the multinomial distribution. In general, we use symmetric Dirichlet distributions as prior distributions, as we initially assume all environment changes to be equally likely.

A Dirichlet distribution is parameterized by $(\alpha_1^{(s,a)}, \ldots, \alpha_{|\mathcal{S}^s|}^{(s,a)})$, where $\alpha_j^{(s,a)} \in \mathbb{R}^+$. Each $\alpha_j^{(s,a)}$ roughly corresponds to how often a particular successor state is observed after executing action a in state s. Given $\mathcal{P}(\mathcal{S}^s|s, a, \theta^{(s,a)})$ and a Dirichlet distribution $\mathcal{P}(\theta^{(s,a)})$, we can estimate the probability of reaching s_j after executing a in s by $\frac{\alpha_j^{(s,a)}}{\sum_k \alpha_k^{(s,a)}}$.

After observing the successor state s_j, we update $\mathcal{M}$ by computing the posterior distribution $\mathcal{P}(\theta^{(s,a)})$ with updated hyperparameters, which is also a Dirichlet distribution. In contrast to Chen and Nielsen [CN12], we consider a non-stationary environment, that is, the transition probability distributions may change over time. Therefore, we introduce a discounting factor $\lambda < 1$ to discount past observations similarly to Bertuccelli and How [BH08]. We refer to λ also as learning rate. Given such a λ, The update rule of the hyperparameters is defined as follows [BH08]:

$$\alpha_j^{(s,a)} \leftarrow \lambda \alpha_j^{(s,a)} + \delta_{s,a,j},$$

where $\delta_{s,a,j} = 1$ if the observed transition ended in state s_j and $\delta_{s,a,j} = 0$ otherwise. The learning rate λ intuitively lets us control how much we value past observations with respect to new observations. Its effect is to keep the estimator's variance non-zero such that new observations have an impact. However, this hinders convergence unless we let λ approach 1 over time [BH08].

4.2.2 Example to Continuous Updates

In our setting, the number of arriving and leaving cars in a single time step is limited, and this gives a clear structure on the possible successor states. For the sake of simplicity, we assume that the queue sizes of each road can change by at most ± 1 and the controller can choose the command NS_c or EW_c. Suppose, the initial state $s = (1, 1, 2, 2, NS_c)$ and the shield agrees with the command of the controller and gives the action $a = NS_{\cup}$. Let λ be 0.9 and let all prior distributions be initialized with symmetric Dirichlet distributions, where $\alpha^{(s,a)} = (1, \ldots, 1)$. There are potentially 162 successor states for (s, a), thus $\alpha^{(s,a)}$ contains 162 entries. After observing the successor state $s' = (0, 0, 2, 2, NS_c)$ (the cars in the north-south direction left), we update $\alpha^{(s,a)}$ to $(0.9, \ldots, 0.9, 1.9, 0.9, \ldots, 0.9)$. Each entry is multiplied by $\lambda = 0.9$, except for the entry $\alpha_{s'}^{(s,a)} = 1.9$ corresponding to s', to which we also add 1. As a result, we estimate the probability of this transition to be $\frac{\alpha_{s'}^{(s,a)}}{\sum_k \alpha_k^{(s,a)}} = \frac{1.9}{161 \cdot 0.9 + 1.9} \approx 0.013$, whereas the probability of each of the other transitions is approximately 0.006.

4.2.3 Periodical Updates of $\mathcal{M}_{\cup}$

After t time units, we construct a new abstraction $\mathcal{M}_{\cup}$ from the current model $\mathcal{M}$ using the current cut-off function k via definition 1. Given the new abstraction $\mathcal{M}_{\cup}$, we also compute a new shield and immediately deploy it.

Before creating a new abstraction from the current model $\mathcal{M}$, we need to check if *abstraction refinement* is needed. Abstraction refinement is needed, if in the last t time units, states were monitored whose variable values exceeded its cut-off. In this case, these states are lumped together in the current abstraction and the cut-off function k needs to be updated.

Definition 2 (Abstraction Refinement)**.** *Given an MDP $\mathcal{M} = (S, s_0, \mathcal{A}, \mathcal{P})$ and a cut-off function k defining the maximal value of each variable $v_i \in s$ i.e., $v_1 \leq k(1), \ldots, v_n \leq k(n)$. Given the states $\mathcal{S}_t \in \mathcal{S}$ observed in the last t time steps, we formalize abstraction refinement as follows. For all $1 \leq j \leq n$, determine:*

$$
\begin{aligned}
{k'}_j &= \max_{s \in \mathcal{S}_t} \{s[j]\} \text{ and} \\
{k''}_j &= \max({k'}_j, k(j)) \text{ to update } k \text{ by} \\
k(j) &\leftarrow {k''}_j.
\end{aligned}
$$

By updating the cut-off function k, the state space of the new abstraction $\mathcal{M}_{\cup}$ will be refined based on the last observations of the environment.

4.2.4 Detecting Changes

If the current abstraction is consistent with our current knowledge of the environment dynamics, we may avoid computing a new abstraction and a new shield, thus saving computational resources. Discrepancy checks between abstraction $\mathcal{M}_{\cup}$ and $\mathcal{M}$ can be introduced to avoid unnecessary computations.

In general, we may apply statistical tests for each state-action pair to check if recently observed transitions are consistent with the current abstraction of the environment, i.e., there are no significant changes of the transition probabilities. If data is likely to be sparse for most pairs, we propose applying application-specific checks for changes.

Example to Detecting Changes

In traffic control, we consider a large state space. Thus the data for individual state-action pairs are sparse. Since our goal is to detect changes in the traffic flow, we propose to monitor the average number of cars waiting in each lane at a junction. These numbers correspond to the variables v_i making up the state space. Change detection could be initialized by computing the mean number of cars $\overline{v}_i$ for each i from the first t observations. Whenever a change of $\overline{v}_i$ is detected for some i, we can assume a discrepancy between the current abstraction and the current state of the environment. After detecting a change, $\overline{v}_i$ needs to reinitialized for each i using the next t observations.

Chapter 5

Implementation

In this section, we want to discuss the components of our framework and provide an overview. We use two existing standalone programs that suit our purpose: a Simulation framework and a Model Checker. To bring these two programs together and build the new shielding framework, we created the Shielding Application. Due to including existing programs, we have to deal with different interfaces and consider the proper usage. Furthermore, the shielding framework can include a Reinforcement Learning (RL) Agent. This RL Agent can be optionally included in the setup of a shielding scenario. We include remarks to the RL Agent, but it is not active in the experiments and not included in the evaluation. We further want to discuss the Shielding Application's implementation, used design patterns, and evaluate features, workarounds and performance.

The implementation and design of the shielding framework were performed in the context of this master thesis, and the development of the shielding application is a significant part.

5.1 Design Patterns

A Design Pattern [Gam+95] is a proven solution for a recurring problem. A software developer has to deal with problems daily, and some of them are well known and already solved in the past. When the software developer starts to reuse this knowledge, he uses Design Pattern and is not aware of that.

Nevertheless, with the knowledge of design pattern, software development can be better structured and improve the collaboration between developers with a conses on how to talk about the problems by giving them names. We want to give an insight into how we used design pattern and discuss and high light them. We can see these patterns as smaller architectural elements that help abstract more giant architectures and break down the problem.

5.2 Shielding Framework Components

As stated in the introduction to the shielding framework architecture 1.2.5, the shielding framework consists of three main components and can include an RL Agent optionally. We want to discuss the three main components in detail and briefly overview the RL Agent and how it can integrate into the shielding framework.

5.2.1 Traffic Simulation Framework

The traffic simulation framework is an essential component of the shielding framework, and it provides a platform that allows us to create traffic scenarios and observe them. The tool "Simulation of Urban Mobility", short `SUMO` [Lop+18], is a microscopic traffic simulation for traffic research and our choice for the traffic simulation framework.

`SUMO` allows loading networks and additional traffic infrastructure data, like roads, footpaths, traffic lights and traffic demand. Furthermore, we can configure this as a traffic scenario via configuration files. Data can be retrieved via output files and with a socket-based interface, like vehicle positions and speeds, traffic data, and network elements to evaluate the simulation.

Therefore, `SUMO` provides a standardised interface to allow interaction with the simulation. The "Traffic Control Interface", short `TraCI` [Weg+08; Lop+18], is based on a socket connection. In Figure 1.3, the framework structure, `SUMO` is shown with the `TraCI` API, where `SUMO` provides a `TraCI` server and accepts `TraCI` clients. By default, `TraCI` accepts only one client to connect, but it allows multiple clients with the proper configuration. It is necessary to define the client's order in the `TraCI` interface with a multi-client setup. Due to the socket interface, `TraCI` is independent of the platform and programming language. With the `TraCI` interface, we can start and stop the simulation. Furthermore, we can retrieve data and modify the simulation. In this work, we essentially need features to define traffic scenarios and retrieve data.

5.2.2 Model Checker

The model checker is used to synthesize shields for the traffic light controller. We use the `Storm` model checker [Deh+17]. `Storm` is a probabilistic model checker with rich features and supports multiple modelling languages. It features the analysis of discrete- and continuous-time variants for Markov chains and MDPs.

The model checker supports the synthesis of counterexamples and permissive scheduler and implements efficient algorithms [Bai+14] for conditional probabilities and rewards, increasing the performance [Deh+17]. Furthermore, `Storm` supports long-run averages on MDPs [Alf98], which is also needed for this work. In terms of performance, `Storm` was the choice compared with the PRISM model checker [KNP11], which provides similar functionality. `Storm` support also multiple modelling languages, where we only use the Prism input language. To synthesize shields, we have to provide information about the environment and the local controller in the format of a model to the model checker. We have to specify what we want to optimize. To run `Storm`, we have to provide two files, one with the model and one with the property. From the model checker, we get the solving strategy, which will be our shield strategy.

To interact with `Storm` over the command-line, we create the two files, start the process and read the resulting back. This introduces an overhead due to the file operations and the creating of a `Storm` process. Solving the model needs much more time than the command-line handling. We discuss a possible solution in the Future Work section 7.2.2.

5.2.3 Shielding Application

The Shielding Application combines traffic simulation and model checker. As the central part of the shielding framework, this application has to interact with the `SUMO` simulation framework and the `Storm` model checker. A socket connection enables the connection between `SUMO` and the Shielding Application. Therefore we have a TCP based client/server interface, and we access `SUMO` with an IP and port. On top of this connection, `SUMO` provides the `TraCI` API, and the protocol provides multiple commands to interact with the simulation. However, this setup allows the Shielding Application to get simulation information and create shields using a model checker. The `Storm` model checker takes model files created of the traffic light information and outputs a solved file. The Shielding Application uses the `Storm` model checker is over the command-line interface (CLI) and wraps around the CLI with files and arguments and file operations. The shield follows the resulting strategy generated from `Storm`, which is loaded into the Shielding Application. In each time step of the simulation, we check the shielding strategy and review the traffic light controller's action.

Features

The Shielding Application implements a range of features:

- Shield Generation from `SUMO` Information.
- Dynamic rerouting of vehicles and lane blocking.
- Accurate Lane State for Real-World Maps.
- Accurate Phase Restore for Static Traffic Light Controller.
- Improved `TraCI` Performance.

We will discuss these features in detail in the following sections, but give first an introduction to the general software design and implementation.

5.2.4 Reinforcement Learning Agent

The RL agent is a Python application forked from Github Repository. Lucas N. Alegre's existing work [Ale19] provides us with a good starting point to work with `SUMO` and RL, and it contained an implementation of Q-Learning with NumPy, an example with stable-baselines3 and DQN and an example with Ray and A3C.

We want to discuss the connection of the RL agent with our shielding framework. Since the application supports a variety of algorithms, we focus on the basic implementation.

The RL agent is connected via the `TraCI` API, as shown in figure 1.3, and works with and without the shielding application. In each simulation step of `SUMO`, both `TraCI` clients, shielding application and RL agent have the chance to change the environment. The order of the application is defined via the `TraCI` order. Since we want to support the training with shields, it is better to start the RL agent scripts' shielding application.

We created a wrapper class and held the object together with the environment object, which wraps the `SUMO` simulation. The application can quickly be used for RL evaluation and extended with a new algorithm since it already hooks two large frameworks.

5.3 Shielding Framework on Docker

Over the last years, containers [Mou15] changed how we develop, distribute and execute software. Developers are independent of local environments and can provide the development environment as a container. It reduces the time to spend on correctly setup environments and struggles with dependencies. Principally, a container is a bundle of dependencies and the application, running on top of the services of the Operating System (OS). Therefore, it is different from a virtual machine (VM), which often appears similar. The VM runs a complete OS in a virtual environment, and in contrast, the container uses the services provided by the host OS.

In this work, we use Docker [Mou15] as a container engine. Docker is a successful platform and comes with a whole ecosystem of tools and a cloud platform, and it led the containerization over the last years. We want to discuss three points, the Dockerfile, the Docker Image and the Docker Container. A Docker image consists of multiple layers, which the Dockerfile defines. For each instruction in the Dockerfile, we create a new layer in the image. The Docker image will be turned into a container by running the Docker Image. Depending on the configuration, the Docker engine adds additional settings.

5.3.1 Dockerfile

The Dockerfile [Mou15] is a simple text file that describes the Docker image in layers. With such a file, we define the Shielding Application container image. On top of a base image, we install the Shielding Framework's dependencies like `SUMO`, `Storm` and building dependencies like `CMake`, `BOOST`. The Dockerfile is available in the GIT source tree.

5.3.2 Docker Image

To create a Docker image [Mou15] with the Dockerfile, we have to build the image with the Docker command build. We can specify a tag for the image with the build command, making it easier to address this image.

Docker Hub [Mou15] provides a platform for various public container images and provides users with a repository to distribute their images. We use this platform to get a base image, build ours on top, and further deploy it to avoid building the image on every new host.

The base image for the Shielding Application container image is the `ubuntu:bionic` image from Docker Hub. By starting the image build, Docker will pull the image from Docker Hub. On top of this image, it applies the new layers defined by the Dockerfile. After the successful, build we can push the image to Docker Hub and avoid building the image again on a new host. Furthermore, with the Docker image, we provide our Shielding Framework environment with fixed dependencies.

5.3.3 Docker Container

To create a Docker container [Mou15] from the Docker image, we run Docker with the run command. At the start of a container, we specify a volume, which allows us to share a directory with the container that contains the GIT repository and is our working directory. Moreover, we specify a container name and configure the container to support the `SUMO` GUI via an X-Server [SU95], which manages the graphic displays.

With the Docker image and containers, the only dependencies for the host is a Docker Engine and, if the GUI is needed, an X-Server. Therefore, we support Linux and Windows OS, which can meet the requirements. On Linux, we usually have already an X-Server, which only need to allow connections without authentication. On Windows, we have to install Docker for Windows and an X-Server, for instance, the `Xming`, which also needs to configured without authentication.

5.4 Software Design of the Shielding Application

We want to discuss the Shielding Application's software design, used Design Patterns, and overviews of how the software works. The Shielding Application is written in C++, standard 2011, and will be built with CMake. The project has dependencies against boost library 1.65 and higher and the C++ `TraCI` API client library [DLR+20], which is included in the source tree. We are using GIT hosted on Github as a version control system and Doxygen for the documentation.

The `Storm` model checker is needed to run the Shielding Application's runtime and depend on the runtime environment. `Storm` also provides APIs for Python and C++ [Deh+17], but we have no benefit from using the C++ APIs, considering the model's solving needs much more time than the interaction between the components.

However, we follow the idea of breaking down a problem into smaller pieces in our software design. Therefore, we divide the Shielding Application into three parts:

- **Simulation Scope**
  In this part, we handle everything related to `SUMO` and the simulation. We handle the data from `SUMO` and manipulate the simulation, and one simulation can have multiple traffic light which can be shielded. Therefore, in the simulation, we manage multiple shielding scenarios and manipulate the environment, like creating traffic indecent.

- **Traffic Light Scope**
  In this part, we build a connection between the `SUMO` traffic light and shield. We have here multiple mappers which manage the capsulation between the simulation and shield scope. Loggers fetch and cache data from the simulation. Furthermore, the software's central part will be done here, the shielding of the traffic light. We get the traffic light controller and environment state and fetch the right shield action to change the simulation.

- **Shield Scope**
  In this part, we handle everything shield related, the environment, controller, and strategy and handle shield configuration files. Furthermore, we handle the `Storm` model checker interaction, shield model generation and strategy parsing in this scope. The shield model will be generated from the environment and controller information, tracked by the traffic light scope.

Figure 5.1 shows a high-level structure of the scopes and the primary classes in them. The blocks are classes with the actual names from the code. We want to show the classes' associations and spare out inheritance, generalisation and interfaces, and discuss details in the corresponding scopes. In this structure, we have a main class with the same name as the scope in each scope. The arrows with the continuous line represent a one-way association form, one class, to the arrow pointing class. We have these connections only from one main class to another. The dashed line is an indirect one-way direction, in which abstracts a wrapped call from the main class in the scope or for the `SUMOConnector` a reference.

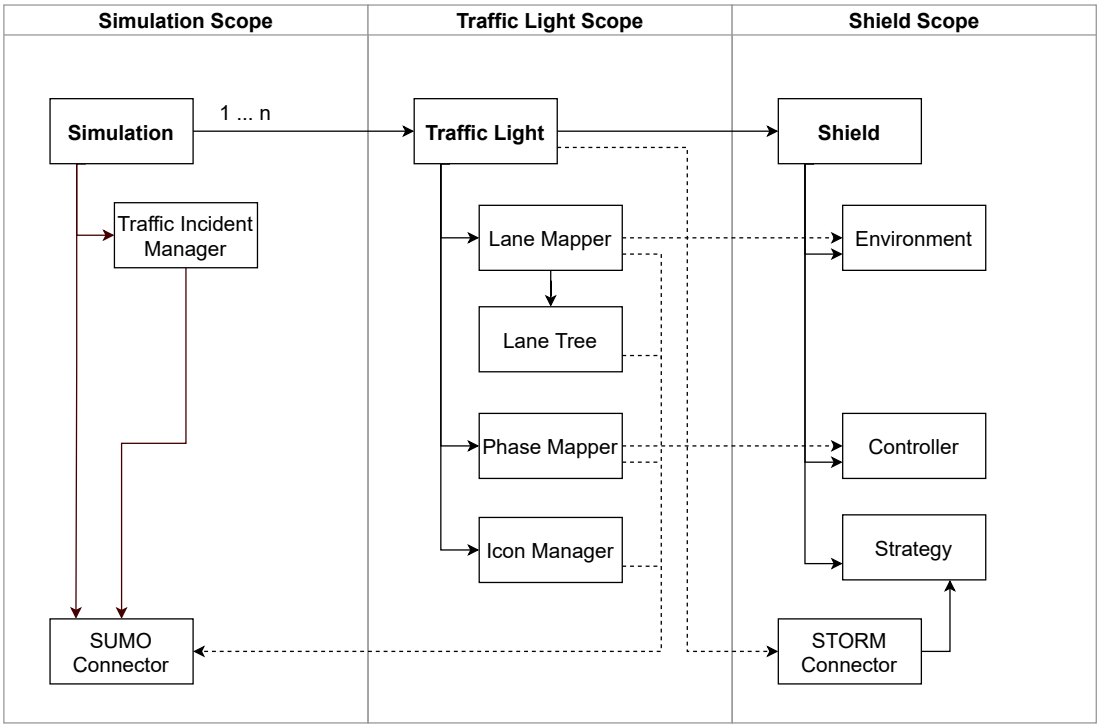

Figure 5.1: The high-level view of all scopes and their classes and associations.

5.4.1 Documentation

In this section, we describe the software implementation, structured in three scopes, in more detail and provide basic class diagrams. Nevertheless, to make the class diagram smaller, we spared unnecessary information, like the most getter and setter. For complete documentation, we suggest taking a look at the Doxygen documentation.

The Shielding Application software project includes Doxygen documentation and can be generated with the GIT source tree. Doxygen [YK17] is a tool that automatically generates Latex and HTML documentation from the source code. It supports a special comment format in the source code, enabling adding additional information right next to the source code. Furthermore, the tool parses header and source files can automatically generate class and UML diagrams. Doxygen is a well-known tool for software documentation and supports a range of popular programming languages, for instance, C, C#, Java, Python.

5.4.2 Simulation Scope

The simulation scope contains all needed classes to connect with the `SUMO` simulation and interact with an active simulation. Figure 5.2 shows a more detailed class diagram of the simulation scope. The `TrafficLight` class also inheres from `ISimulationObject`, but it is not included in this diagram as we will discuss it in the next scope.

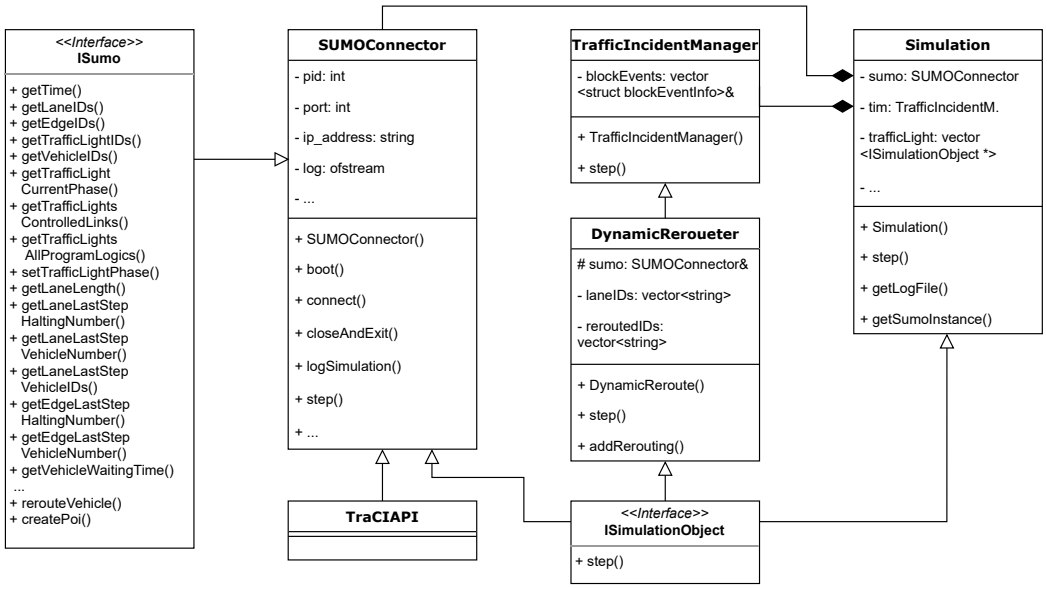

Figure 5.2: A detailed class diagram of the simulation scope.

Simulation and ISimulationObject Interface

The class simulation is the part of starting a shielded `SUMO` simulation. It gets the configuration and exposes only a step method and loop method, which call the step in a loop until the defined time step is reached. The step function is similar to the `TraCI` simulation step method, and it does one timestep in the simulation. Other classes also have a method that should be called in each timestep, and to keep track of this, we introduced the `ISimulationObject` interface to identify such a class. Furthermore, we implemented the composite pattern [Gam+95], which is a Design Pattern that allows treating a group of objects in a similar way collected in one instance. The hierarchies of objects with different granularities can vary and make no difference since we can treat the defined rules in the same way.

> "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." [Gam+95]

Figure 5.3 shows the class diagram of the composite pattern in our software. We have simulation objects in the first and second scope, and we can treat it similarly. The Idea of a composite pattern is to compose functionality in multiple objects to break down the complexity. With different granularities and over different layers, we get a tree structure of components and composites.

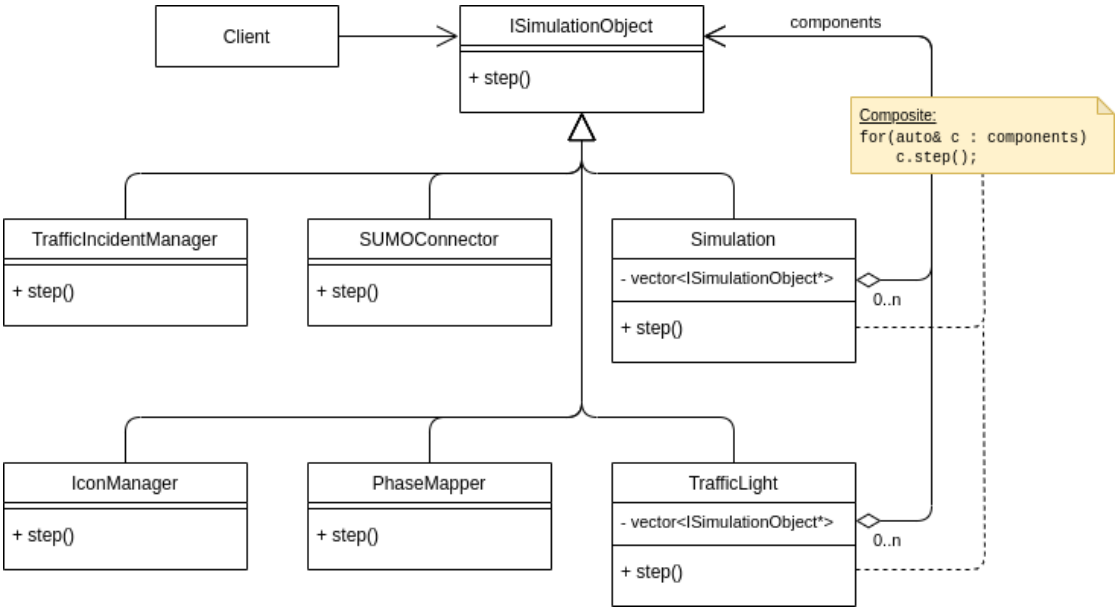

Figure 5.3: The class diagram of the simulation composite pattern.

Figure 5.4, shows the tree structure in our software. In the simulation scope, the simulation class is composite and runs all components in his step method: `SUMOConnector`, `TrafficIncidentManager` and `TrafficLight` objects. We have the traffic light object called from the first simulation composite in the traffic light scope, and it is also an own composite and runs all components in his step method: `PhaseMapper` and `IconManager`.

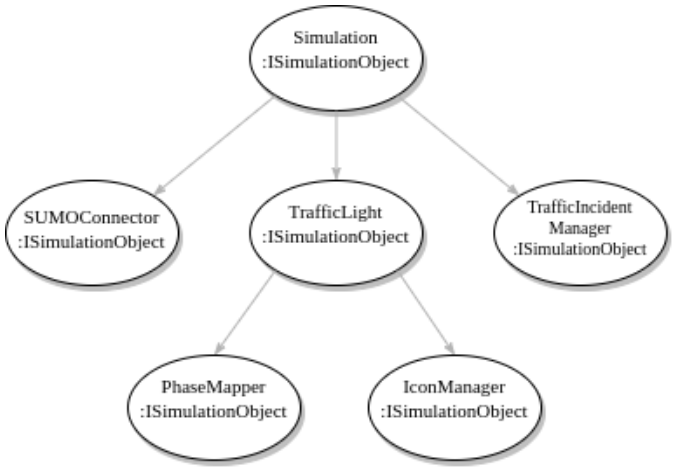

Figure 5.4: The tree structure of the simulation composite pattern.

SUMOConnector, ISumo Interface and TraCIAPI Interface

The `SUMOConnector` is responsible for the interaction with the simulation. Therefore, it can create a `SUMO` instance or only connect to an already existing `SUMO` instance. The connection between the Shielding Application and `SUMO` is based on the `TraCI` interface. By connecting with `SUMO`, we open a socket and have a TCP connection, and the `TraCI` API allows us to interact with the simulation. The `SUMO` configuration file will provide all necessary simulation configurations among the net file, which defines the map and the route file, which defines the traffic in the simulation. The `SUMOConnector` also has a step method and implements a simulation component. This step method will call the `TraCI` simulationStep, which triggers the net simulation step.

In this scope, we added and internal standardization and more functionality on top of the `TraCIAPI`:

1. Provide a more straightforward and unified interface to access `TraCIAPI` by introducing `ISumo`.
2. Provide an adapter for the alternative interface `ISumo` interface.
3. Provide additional functionality and better performance when accessing `TraCIAPI`.

The core class which implements this task is the `SUMOConnector`. To interact with `SUMO` in different objects in the Shielding Application, we defined the `ISumo` interface.

The shown class diagram contains not the whole interface but shows the structure. However, by inheriting the `TraCIAPI` private and only exposing the `ISumo` interface, the `SUMOConnector` hides the complexity of the `TraCIAPI` and adapts it to `ISumo` and can also implement additional functionality. For each of the three tasks of `SUMOConnector`, we can observe a design pattern used to provide a solution.

Facade Design Pattern. The facade pattern [Gam+95; w3s17] simplifies an interface and makes it easier to use. With the `TraCI` API, we have to deal with different scopes [DLR+20], for instance, LaneScope, EdgeScope, TrafficLightScope, and SimulationScopee. Each scope is a collection of methods to interact with the simulation. For all of these methods, we need only a few selected methods. Depending on the SUMO simulation, scope available or not. To reduce the complexity of the `TraCI` API, we use a facade implemented by the `SUMOConnector`.

> "Provide an unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." [Gam+95]

The `TraCI` API provides access to the simulation and can be seen as an interface to the simulation objects. When SUMO is the simulation system, the scopes are the simulation subsystem, and with `TraCI` a have access to these subsystems. For other use cases, it is desired to have such a broad range of features to interact with a simulation, but we want to keep it simple for our use case.

Adapter Design Pattern. The adapter pattern [Gam+95; w3s17] converts one interface to another one. We use it to convert the `TraCI` API interface into the `ISumo` interface, used across the Shielding Application. Usually, this pattern is used to be an adapter for existing interfaces. Here, it was helpful to simplify the more extensive interface to a more compact one.

> "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces." [Gam+95]

This pattern usually has two types, the class adapter and the object adapter. We use the class adapter pattern, where we use inheritance to implement the `ISumo` Target interface and the `TraCIAPI` Adaptee class. The `SUMOConnector` Adapter class implements the actual connection between the interfaces.

Figure 5.5, shows the class diagram of the adapter pattern implemented in the Shielding Application. The example spares out all methods in the interfaces and shows the principles whit the SimulationScope method getTime. As discussed with the facade pattern, the `TraCIAPI` has multiple scopes which collect the methods. The `ISumo` interface wants to avoid this and introduces a more straightforward interface. To connect both interfaces, the `SUMOConnector` acts as an adapter. With the new `ISumo` interface, we use the getTime method caring about the scope.

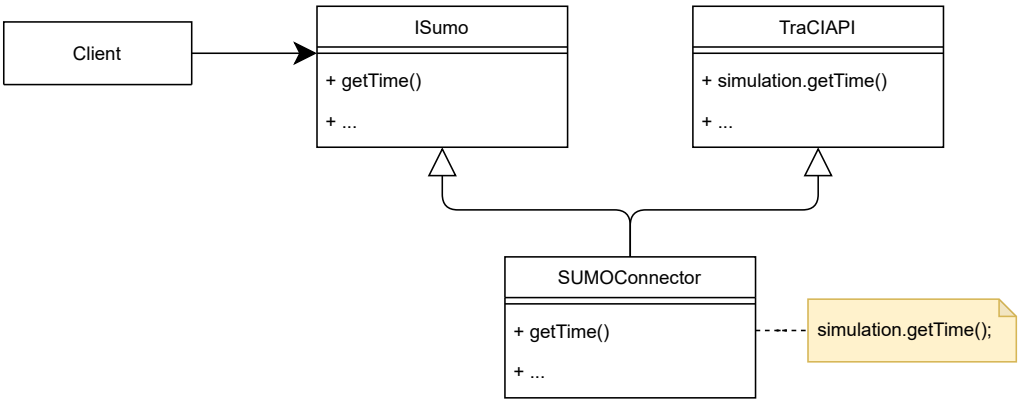

Figure 5.5: The class diagram of the `SUMOConnector` adapter pattern.

Proxy Design Pattern. The proxy pattern [Gam+95; w3s17] controls the access to an object. In this case, we want to access the sumo with the `ISumo` interface. The accessed object is the `SUMOConnector` since the class inheres from the `TraCIAPI` and puts an adaptor top.

> "Provide a surrogate or placeholder for another object to control access to it." [Gam+95]

We deal here with a bit different proxy pattern since we also differed the interface, but still, it is the idea to add complexity to a placeholder. In this case, we want to add functionality to the `TraCIAPI`, but it is accessed with `ISumo`.

However, we added caching functionality to the `SUMOConnector` to reduce the number of `TraCI` request and increase the performance. Furthermore, we replaced the majority of the get request with the subscription interfacer of `TraCI`. The `TraCIAPI` provides functions to subscribe to variables [DLR+20]. Hence, the variable has to be subscribed once and is available after each time step in structure. The variable subscription setup and the retrieving of the variable values need more code than the normal getter function, which generates a `TraCI` request. However, with a high number of request, this is very beneficial for the performance.

This strategy gives the `SUMOConnector` caching proxy behaviour. It reduces the number of requests and is beneficial for the performance of the Shielding Application. The subscription model is used in the background and forward to the typical methods by the `ISumo` interface. The advantage of the subscription model is that it reduces the communication overhead by avoiding polling on variables.

DynamicRerouter

With the `SUMO` GUI, we can place rerouter block on roads, which trigger a rerouting of the vehicles on the lane, and the route of the vehicles will be updated. With the Shielding Application, we have only `TraCI`, which has no functionality to place rerouting

blocks. To solve this problem, we implemented the `DynamicRerouter`, which implements a rerouting block in the Shielding Application. The rereouter has an `ISumo` reference to interact with the simulation and a list of lane IDs where rerouting should be happening. In the step method, we trigger a rerouting of the vehicles on the observed lanes.

TrafficIncidentManager

The `TrafficIncidentManager` implements the dynamic rerouting of vehicles and lane blocking feature. The class manages the lane blocking and rerouting. Where the functionally is distributed over two classes, the rerouting is implemented in the `DynamicRerouter`. The `TrafficIncidentManager` inheres from `DynamicRerouter` and additionally manages the lane blocking. Hence, the class has an event list of blocking events. The step method is triggered in each timestep and checks if a blocking event is active. If the blocking event happens, we block the road, vehicles on the previous road with a static route will be rerouted. Currently, we support only the accident scenario.

5.4.3 Traffic Light Scope

The traffic light scope connects the simulation layer and the shield layer. Figure 5.6 shows a detailed class diagram of the traffic light scope. We can see here that also some classes inherit from `ISimulationObject`, which identifies them as a simulation object. The simulation objects follow the composite pattern, and in this scope is the `TrafficLight` the composite. The composite has the `PhaseMapper` and `IconManager` components. The `ISimulationObject` class is marked grey in the class diagram since it belongs to the simulation scope.

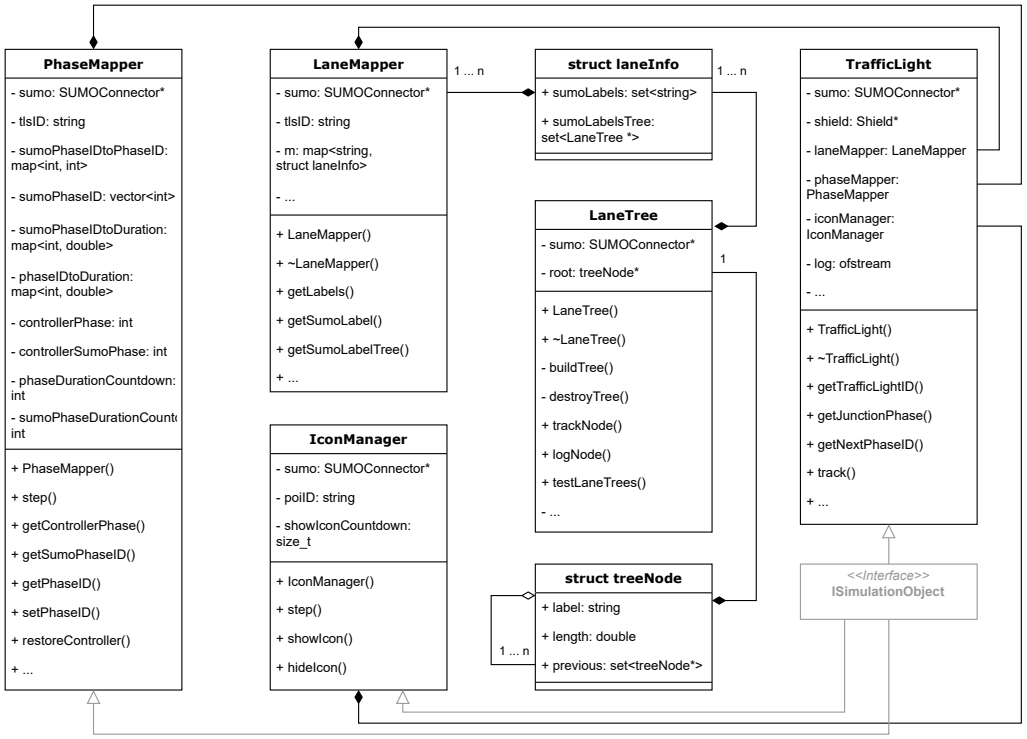

Figure 5.6: A detailed class diagram of the traffic light scope.

TrafficLight

The `TrafficLight` class abstract a real `SUMO` traffic light and acts as a translator between an internal naming convention and the common `SUMO` names. Manly, this is needed for variables that are needed in the model. The model files are PRISM syntax, and we do the mapping of the name convention in this layer. Furthermore, loggers fetch and cache data from the simulation and the central part of the shielding will be done here. The `TrafficLight` get the simulation data and check the shield strategy, and intervenes in the simulation.

LaneMapper and PhaseMapper

The `LaneMapper` and the `PhaseMapper` are adapters for `SUMO`'s simulation environment and the internal conventions for the shield. Both classes implement mapping mechanisms to connect the simulation and the shield.

LaneMapper. The `LaneMapper` implements the mapping of the sumo lane IDs to the internal lane ID. Hence, the internal lane IDs are the lane towards a junction and used in the shield. We also implemented lane merging and can collect multiple parallel lanes under one internal lane ID, which improves the model checking performance by keeping the complexity low. Furthermore, this class implements the Accurate Lane State

for Real-World Maps 5.7. It solves a problem with imported maps from OSM, where more extended lanes are composited of multiple small pieces.

The LaneTree creates and traverses the tress. It has a tree structs of `treeNode`, which corresponds to the lanes in the simulation and has information of sumo lane ID, length and successor (or in this case, the previous lane, with the junction's viewpoint). The `laneInfo` struct helps keep track of the lane trees and supports the mapping mechanism without lane trees.

PhaseMapper. The `PhaseMapper` is similar to the `LaneMapper`. Nevertheless, it implements the mapping of the sumo phases to the internal shield phases, which ignore the yellow phases. Additionally, the class implements the Accurate Phase Restore for Static Traffic Light Controller 5.8, which keeps track of the static traffic light controller internally. Therefore the static internal controller in sync with the static simulation controller allows us to restore the actual phase and duration after a shield interference.

IconManager

The `IconManager` is a graphical feature for the GUI mode and visualises the shield intervening in the simulation. If the Shielding Application is started with the GUI flag, we can see if a shield changes the simulation by a red shield icon. Hence, the `IconManager` creates a point of interest (POI) object [DLR+20] in the `SUMO` simulation and manages its visibility. The POI can load a bitmap image and is placed in a layer. If the shield intervenes in the simulation, the POI will be set visible and set up a count down. With every timestep, the countdown will be decreased, and at zero, the POI will set invisible.

5.4.4 Shield Scope

The shield scope contains all needed classes, is independent of the other scopes, and has no simulation information. The observations of the simulation will be forwarded from the Traffic Light scope. This scope handles the interaction with the STROM model checker, creates shield models and loads it in the shielding program. Figure 5.7 shows a more detailed class diagram of the shield scope. The shield class inheres from two classes which implements some basic functionalities. It collects all classes in scope and handles the interaction with the TrafficLight scope.

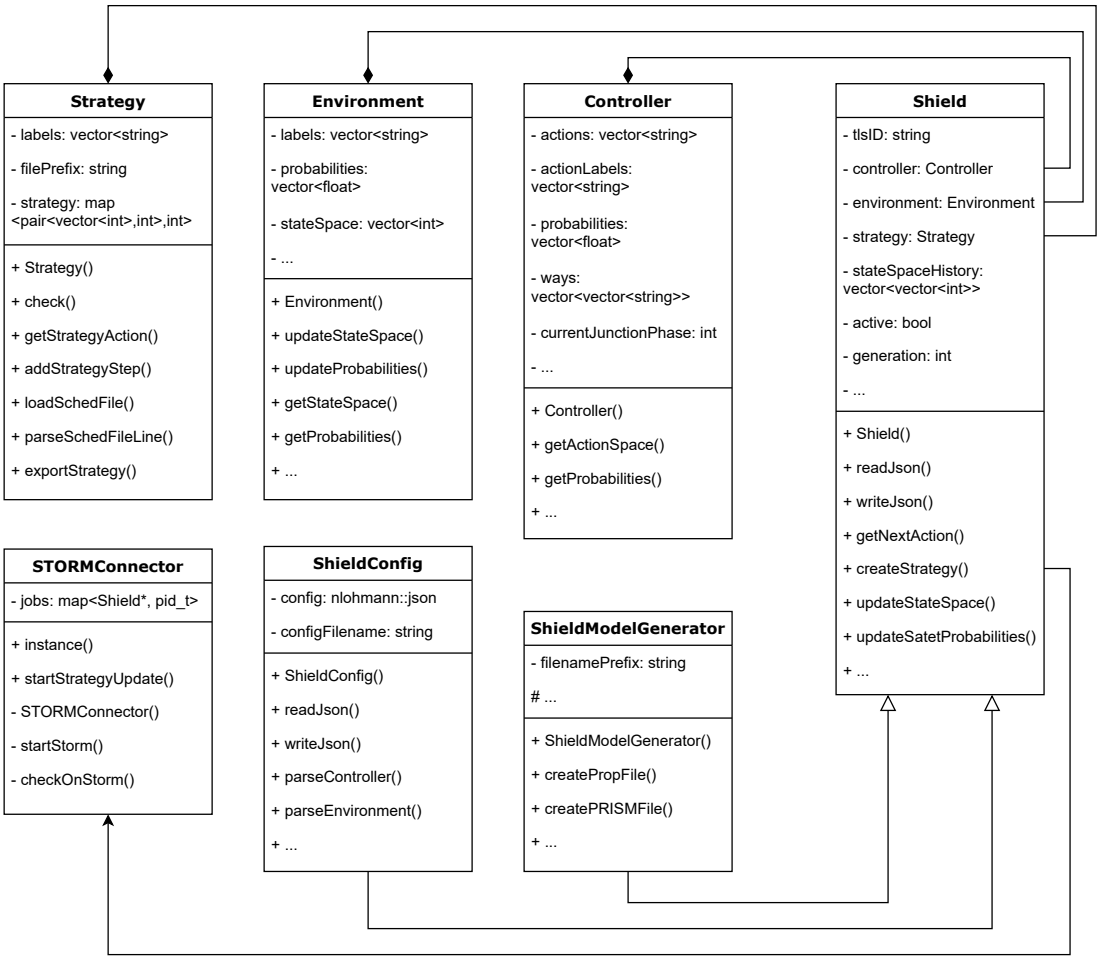

Figure 5.7: A detailed class diagram of the shield scope.

Shield, ShieldConfig and ShieldModelGenerator

The class `Shield` handles everything shield, and it inheres from `ShieldConfig` and `ShieldModelGenerator`. The `ShieldConfig` class contains the functionality to load and save shield configurations and parameter to a JSON file. The `ShieldModelGenerator` class contains the functionality to generate the model checker files. It generated the PRISM syntax and hast the shield model template hard-coded in methods with the `createPRISM` prefix. Furthermore, the shield has an Environment, Controller and Strategy instance. Environment and Controller track the simulation properties and which are forwarded from the Traffic Light Scope. The model checker files will be created with the `ShieldModelGenerator` methods inherited by the shield. The `STORMConnector` runs the model checker with the generated files, and the Strategy loads the result into the application.

Environment

The Environment class is a simple container to track the environment state and properties of the shield. It has similar properties as the model with a focus on the shield environment. The environment properties contain the lane labels (state-space labels), the state-space and the state probability. We also track the environment observations and update the state probabilities with the `updateProbabilities` method. With the `updateStateSpace` method, we update the state-space if the current state space limits are reached until a maximum size is reached.

Controller

The Controller class is similar to the Environment class and a container to track the shield's properties. In this case, with a focus on the controller. Since we deal with static controllers, the action-space and action probabilities are fixed and will not be updated like the environment. We track here the controller phase, and the class has some essential properties for the shield generation. To provide a robust shield generation, we identify the actions on their links to avoid that a changed order leads to problems. Therefore, we keep track of the ways of action and check this by changing a property.

Strategy

`Strategy` is a container class for the shield strategy (MDP). It also handles the file operation and parses the strategy. The method `loadSchedFile` starts to read the file line by line and parses it with `parseSchedFileLine`. The retrieved values are added to the shield strategy with `addStrategyStep`. The strategy variable is a complex struct that contains the MDP and where we gat the shield action based on the current environment state and the controller state. The method `getStrategyAction` takes both states as arguments and does the lookup in the strategy. For better debugging of the model checker output, we write the MDP back in a more user friendly in an own file with `exportStrategy`.

STORMConnector

The `STORMConnector` handles the interaction with the `Storm` model checker. The shield triggers and model checking process with the `createStrategy` method and creates the model checker files. The shield triggers the `STORMConnector` with the `startStrategyUpdate` method, which gets a reference to the shield instance. After creating a Storm process, the shield reference and the process id will be stored as a job in the jobs list. The method polls on the `Storm` process and is it the success it calls the shield `updateStrategyCallback` method and the new strategy load into the strategy object. Due to the model's complexity, `Storm` can timeout, and in this case, `Storm` fails, and state-space update will be locked.

The `STORMConnector` is multi-threading ready, but for our use case, we need reproducible results. For more notable scenarios and simulation, it would help run `Storm`

asynchronously and avoid freezing simulations in GUI mode when the Shielding Application waits until `Storm` succeeded. With the `Storm` connector, we want to create a central point to handle the `Storm` interaction. Therefore we want to have only one instance and have it available globally.

Singleton Design Pattern. The singleton pattern [Gam+95; w3s17] meets these requirements and is the perfect solution. It ensures that only one instance exists of the class and solves the problem that this instance of the class should be accessed globally.

> "Ensure a class only has one instance, and provide a global point of access to it." [Gam+95]

Figure 5.7, shows the `STORMConnector` class and the influence of the singleton pattern. The class constructor is private and hidden for the user. Furthermore, we want to hide all operators and copy constructor and also set them private. The creation and the access to the instance is handled by the static method `instance`. It returns the instance of the class if it exists and creates an instance if not. With this strategy, we can be sure that an instance exists only once by shifting the responsibility to the class. With the public static `instance` method, it is easy to access the object globally within an application.

5.5 Shield Generation from `SUMO` Information

For the generation of a shield from `SUMO` information, we need to know the traffic light program, phases and links. Figure 5.8, shows the `SUMO` netedit view of a traffic light with the links from one selected traffic light phase of the traffic light program. A link of the traffic light is the lane into the traffic light's junction to the intended lane. If more than one way is possible, there is an additional link for this target lane. The `SUMO` netedit highlights the state of the selected traffic light phase.

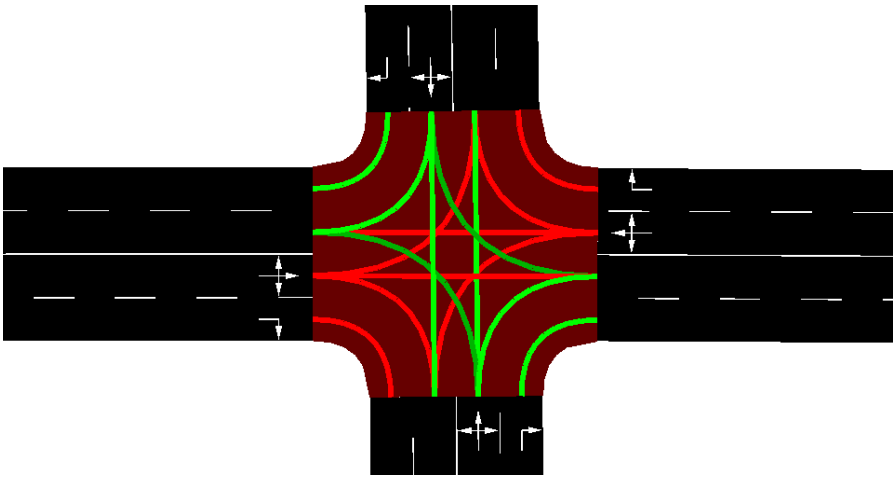

Figure 5.8: Traffic Light Links in `SUMO` netedit.

These phases and their link states define the traffic light program. With the `TraCI` API, we can change the traffic light phase and retrieve the traffic light's needed information to create the shield model. Table 5.1, shows the Program of a traffic light how we can retrieve it with `TraCI`. This traffic light program consists of four phases with a phase ID or phase index, duration and the link state. The link states, from the links $\{l_0 \ldots, l_{15}\}$, can be accessed over the index of the link. For example, for phase one, link l_0 the state G. The phase states are encoded like known from the real world traffic light, for red (=r), yellow (=y), and green (=g). We notice that there are an upper case G and a lower case g used for prioritisation for green. With a lower case g, the link has no priority and may not pass the junction. With the upper case G, the link has priority. This intends to prioritise the link, with a lower case g the link has no priority and may not pass the junction, and with the upper case G, the link has priority.

Traffic Light Program		
Phase	Duration	Link States
0	42	GGGgrrrrGGGgrrrr
1	3	yyyyrrrryyyyrrrr
2	42	rrrrGGGgrrrrGGGg
3	3	rrrryyyyrrrryyyy

Table 5.1: Program of Traffic Light Controller.

In general, we abstract the `SUMO` lane names to internal lane names to conform with given name formats. This allows us to perform lane abstraction and merging of parallel lanes under one internal lane name, which increases the shields' performance and decreases the complexity of the MDP. Currently, this will only be done if two lanes have equal links in the same phase. In this example, we can merge the parallel lanes since the links are active in the same phase. For other cases, we have had to avoid lane merging if the links are equal but active in different phases.

With the information from the traffic light topology and the traffic light controller, we generate the PRISM model of a template. This template is coded in the class `ShieldModelGenerator`, which has methods to generate the PRISM module string and can be adapted if needed. We support a shield configuration file where the PRISM syntax can be adapted more quickly to allow higher flexibility. To load shields from the configuration file, we suggest generating a shield configuration with the Shielding Application and then adust the content. We added some metadata from the traffic light, which have to be correct to success. This should avoid problems with wrong orders, and it is hard to read all the information out of the `SUMO` files.

5.6 Dynamic Rerouting of Vehicles and Lane Blocking

In the SUMO GUI, we can block a road and place rerouting blocks with a right-click. Figure 5.9, shows the GUI with the traffic accident setup, where lanes are blocked and emulating the accident. The red colour indicates that no vehicle can cross the road. The vehicles started with a fixed route and would jam until the lane blocking is resolved. We placed the rerouting block on the lanes where the vehicle would start queuing up to avoid this. The yellow block triggers a rerouting of the vehicle on the lane and enables further flow on the other roads.

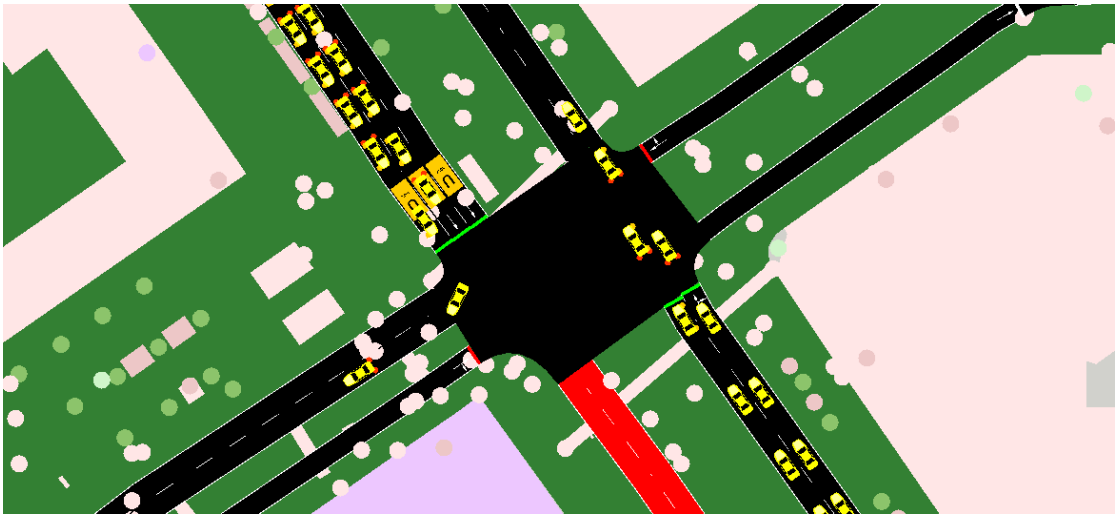

Figure 5.9: SUMO GUI with blocked lanes and rerouting blocks, set trough the menu.

However, with the TraCI interface, we can only block a road and not create rerouting blocks. Still, we have the functionality to trigger a rerouting of vehicles which allowed us to implemented the rerouting block in the Shielding Application. Furthermore, road blocking and rerouting can be configured with a file and provided to the Shielding Application by the indecent argument. The configuration files is a simple text file, where one line is one blocking event. The blocking event has three properties: the lane ID of the lane which will be blocked, the timestep when the event should happen, and the lane ID where the vehicles have to be rerouted.

5.7 Accurate Lane State for Real-World Maps

SUMO provides a tool that allows us to import real-world maps [DLR+20] from Open Street Maps (OSM). A Python script parses the OSM data and creates SUMO net files. With these real-world maps, we run into the issue that the script imports lanes in sections how it sees it on the map. The data we retrieve from SUMO is per lane, which gives us inaccurate information with lanes divided into sections.

Figure 5.10 shows the issue with the divided lanes from OSM imported maps and highlights the lane sections. The lane is split into four-part and on the right sides into two parts on the left side. The here is on the left side, where the lane goes into a traffic light junction. On this lane, we check for waiting for vehicles, and the data we get is only for the first lane. Therefore, vehicles can queue up the whole street, and we notice only up to two vehicles on the lane.

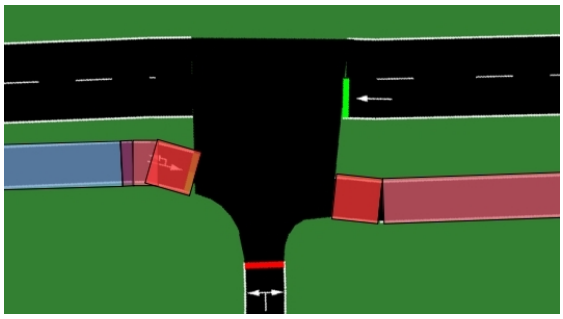

Figure 5.10: The high lighted Issue with imported Maps from Open Street Maps.

We solved that issue by creating lane trees with the lane to the traffic light junction as the root node. These trees contain the needed lanes to reach a lane length minimally needed to reach the maximum state space.

Figure 5.11 shows an example for a traffic light with lane trees. The length values are numerical values in the simulation framework and associated with meters as a physical unit. The maximum lane size is six vehicles which means we need a length of at least 30 when an average vehicle has a length of 5.

`TraCI` has no functionality to get the previous lanes of a lane, and therefore we have to iterate over all lanes and build a lookup table of the next lanes. We can build the trees faster with the lookup table since we do not need a `TraCI` request for each lookup. We get an accurate lane state with the lane trees by traversing the trees and accumulating the lane values concerning parallel lane topologies.

The workaround with the lane trees can lead to erroneous results when trees start to overlap. Therefore, we do some checks to provide here a robust solution. The lanes trees will be checked, and overlaps and cycles will be found. For the past experiment, this was a suitable solution for the OSM maps since we have a limited state space, and the lane tree length was limited.

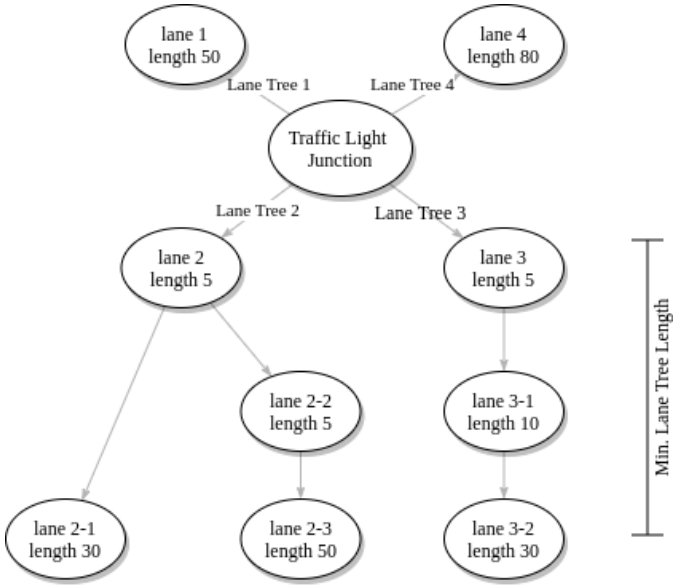

Figure 5.11: The high lighted Issue with imported Maps from Open Street Maps.

5.8 Accurate Phase Restore for Static Traffic Light Controller

For scenarios with Static Traffic Light Controller, we implemented an accurate phase restore after the shield intervened. The controller has a fixed program, shown in table 5.1, which consists of phases with a duration. If the shield intervenes and changes the phase, the static controller will switch to the next phase when the duration is reached. We have no problem with automatically changing phases in this setup since the shield intervene shorter than the phase durations. Nevertheless, if we restore the old controller state, the timing is destroyed. Therefore, we implement a static internal controller, who knows the traffic light program. Since the controller is static, the internal controller is always in sync with the actual controller. This allows us to restore the actual phase and duration after a shield intervene.

5.9 Create new Shield Types

For future development and experiments, it is exciting to create new types of shields. Since the shield generation is done with the Storm model checker, we have several options to create a new shield type. Depending on the new shield model, we can adapt the model in the shield configuration file or apply an update to the current shield model template in the source code. If new shield types change, the model checker output we have to adapt the file operations.

We want to discuss the options and consequences by introducing new model types in the shielding framework. Furthermore, we want to give guidance on how the adaption process can look like.

5.9.1 Modify the Shield Configuration File

The Shielding Application creates JSON files with the shield configuration, and by starting the program without shield additional shield configuration JSON files, this happens by default. The JSON file has multiple properties, and some of them contain the PRISM syntax, which can be modified to derive new shields. With this method, only simple modifications are possible, like changing the reward function. The state-space and action-space definitions are fixed, as well as the probability update. Nevertheless, we can change all PRISM modules in the configuration file.

For this method, we suggest letting the application create the JSON file, applying the modification to the JSON file, and providing this file over the configuration file argument of the Shielding Application.

5.9.2 Modify the Source Code

In the source code, there are two tasks bound to the shield: Create the shield model files, and parse the result from the model checker. The parsing must only be changed if the output changes from the model checker. Figure 5.12, shows the class diagram, which solves the shield adaption, with the essential relations between the `Shield` class, `ShieldModelGenerator` class, and `Strategy` class. The solution to adapt the necessary components is to derive new classes from the existing and use them. For the parsing task, we derive the NewStrategy class, and in the shield, we only need to instance the strategy with the new class. Currently, the `Shield` inherited the `ShieldModelGenerator` functionality.

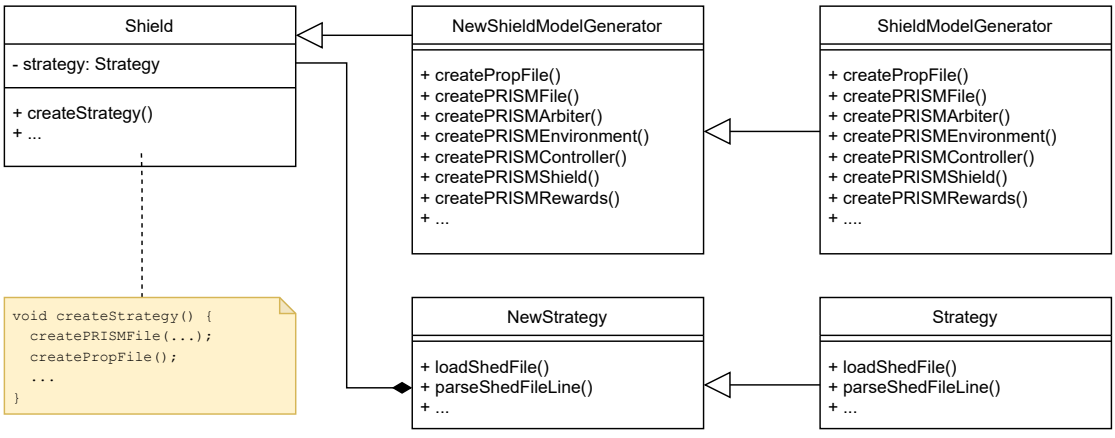

Figure 5.12: Create a new Shield Types in Source Code by deriving new Classes in Shield Scope.

Therefore, we need to derive the NewShieldModelGeneration class and inherit this class in the `Shield` class. With a minimal change on exiting code, we included new classes, NewStrategy and NewShieldModelGeneration, which can be modified for the new shield types. By overwriting the exiting methods, as shown in Figure 5.12, we can adapt the parsing and the shield model generation. The inheritance of the NewShieldGenerator makes it hard to allow configuration of the model generation at run time. Therefore, we want to propose a more flexible solution by introducing an interface for model generation and showing the Strategy Design Pattern's usage.

Strategy Design Pattern. The strategy pattern [Gam+95; w3s17] helps to make the selection of the model generator a bit more flexible. Usually, this pattern considers how to configure an algorithm at run time and avoid direct implementation. It makes the selection and changing of the runtime easier.

> "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." [Gam+95]

In our use case, the model generator creates the model checker's configuration, and the actual algorithm runs there. Still, we have here some similarities and want to have similar flexibilities with this class. Therefore, we should avoid hard-wiring and remove the inheritance of the `ShieldModelGenerator` class.

Figure 5.13 shows the model generator's strategy pattern, with the new interface identifies a model generator. The `Shield` class has a new `IShieldModelGenerator` member instead of inherited the class, which allows us to an instance on that every class inherits from `IShieldModelGenerator`. To get the new code in the class, we follow the same idea to overwrite the `IShieldModelGenerator` methods.

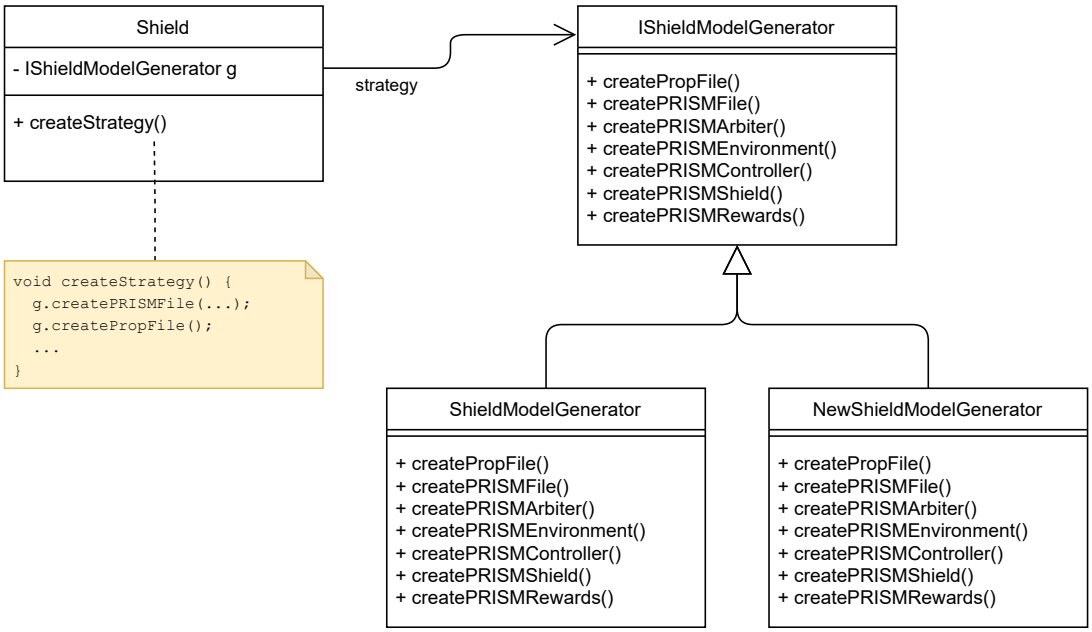

Figure 5.13: Strategy Design Pattern to create new Shield Types.

Chapter 6

Experiments

In this chapter, we evaluate our adaptive shields with experiments in an urban traffic setup. The experiments are performed in the context of this master thesis with the shielding framework. The experiments traffic scenarios will load in the traffic simulator "Simulation of Urban MObility" (`SUMO`) [Lop+18]. We use the shielding application with the `SUMO` C++ API to extract the traffic-light-controlled intersection information and send orders to change the traffic light phases. The shielding application implements the computation of abstraction MDPs. Given abstraction MDPs, we compute shields via value iteration with we the model checker `Storm` [Hen+20]. All simulations were executed on a desktop computer with a 4 x 2.70 GHz Intel Core i7-7500U CPU, 7.7 GB of RAM running Arch Linux. The simulation results are presented to show the effectiveness of our approach. The source code and several videos of shielding in action can be found on `https://shieldai.github.io/AdaptiveShielding/`.

6.1 Scenario 1: Changing traffic density on a single crossing.

In the first experiment, we use the crossing illustrated in Fig. 4.1 for simulations. The intersection is composed of four roads, and every road has two lanes: the right lane allows right turns, and the left lane allows the left turn and through vehicles. During the simulations, we measure the waiting time of cars on the road.

The experiment's simulations last for 6000 time steps, where 1 timestep corresponds to 1 sec. At the beginning of a simulation, the average vehicle arrival rate is uniformly distributed over all lanes with an arrival period of 1.5 cars per second. It captures the normal traffic density at this crossing. After 500 steps, the traffic density on the east-to-west road increases to 65% of all the arriving cars (e.g., due to an accident or a construction site).

6.1.1 The Controller

The controller is a static traffic controller on a two road junction. The traffic light has four phases, where two of them are yellow phases. Effectively, we deal internally with two phases with 45 seconds, including one 1 second yellow phase.

6.1.2 The Shield

We compute adaptive shields for this junction. Its MDP representation is illustrated on the right in Fig. 4.1. Each state of the model $\mathcal{M}$ is a 5-tuple (n, s, e, w, ctr) composed of integers. The variables n, s, e, and w represent the number of waiting cars in each direction and $ctr \in \{NS_C, EW_C\}$ is the issued command of the controller. We use the knowledge about the initial traffic density and the signal's stage duration of the controller for an initial estimate of the transition probabilities of $\mathcal{M}$. To construct the initial abstraction $\mathcal{M}_{\sqcup}$, we use the cut-off function $k(i) = 3$ for $i \in [1\,..\,4]$. As performance measure we charge an abstract state $s = (v_1, v_2, v_3, v_4, ctr)$ with a cost of $c_1 = |\max(v_1, v_2) - \max(v_3, v_4)|$, thus the shield aims to balance the number of waiting cars per direction. Interfering, i.e., altering the controller action, incurs a constant cost value c_2 to the shield. The factor $\gamma = 0.5$ is chosen to weight the costs c_1 and c_2.

6.1.3 Adaptive Updates

During runtime, we continuously monitor the traffic density and update the probability distributions of $\mathcal{M}$ with a learning rate λ. Additionally, we update the cut-off function k if we observe states whose values exceed their respective cut-off values. Every $t = 500$ time steps, after an initial setup time of 1000 steps, we compute a new abstraction $\mathcal{M}_{\sqcup}$ and a new shield based on $\mathcal{M}_{\sqcup}$, which we immediately deploy.

6.1.4 Results

Throughout all our experiments, we use a unified measure of performance: the total waiting time of the cars in the city accumulated over the last 100 time steps. We assume that minimizing this measure is the designer's main objective of the traffic light system for the city. We show the effect of the shield-interference cost w.r.t. the concrete performance. We experimented with shields with a learning rate of $\lambda = 0.3$ and different fixed interference costs $c_2 \in \{5, 7, 9\}$. Note that the same effect could be achieved by varying γ.

Fig. 6.1 shows the effect of the shields on the performance as well as the according to ratios of times where the shields intervene. Table 6.1.4 summarizes the updates of the cut-off functions of the individual shields. In Fig. 6.1 the red line shows the performance of the unshielded controller, which is good until step 500 when the traffic density shifts to one direction. From this time step on, the controller has to face unprecedented situations and therefore drops in performance. The remaining lines show the effect of using shields with $c_2 \in \{5, 7, 9\}$.

We observe that all shields can reduce the total waiting time of the cars by orders of magnitude. For larger values of c_2, interference is too costly in the beginning. Several updates of the cut-off function are needed until the abstraction captures the number of waiting cars with sufficient accuracy. With increasing accuracy, the shield starts to intervene, and the waiting times improve. After several refinement steps, all shields achieve the same level of performance with identical interference rates.

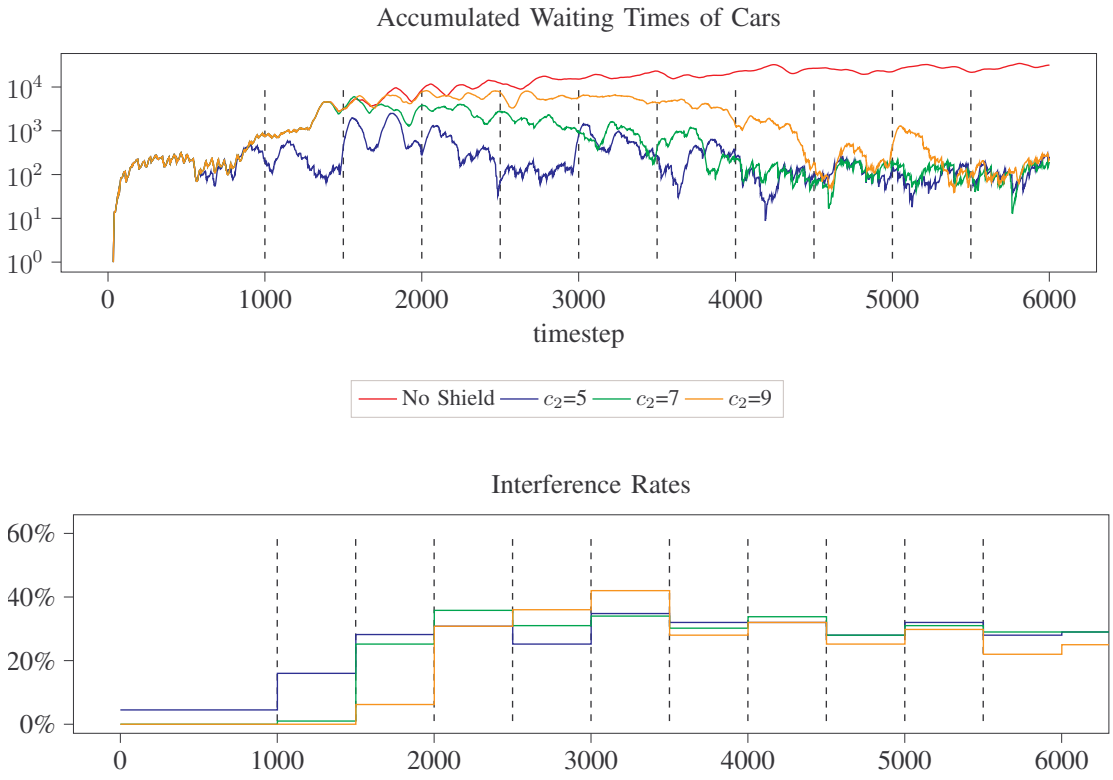

Figure 6.1: Results for Scenario 1: Shielding a single intersection.

In this setting, we used a learning rate of $\lambda = 0.3$. We conducted the same experiment with different values of λ and noticed only minor differences in the accumulated waiting times. For larger λ, the accumulated waiting time dropped a bit sooner. However, we noticed that adjusting the costs for interference has more impact than changing the learning rate. More experiments in different settings are necessary to give a general statement.

time-step	k for shield with $c_2 = 5$	k for shield with $c_2 = 7$	k for shield with $c_2 = 9$
0	(3,3,3,3)	(3,3,3,3)	(3,3,3,3)
1000	(4,4,4,4)	(4,4,4,4)	(4,4,4,4)
1500	(5,4,5,4)	(5,4,5,4)	(5,4,5,4)
2000	(5,4,6,4)	(5,4,6,4)	(5,4,6,4)
2500	(5,5,7,4)	(6,5,7,4)	(5,5,7,4)
3000	(6,6,8,4)	(7,6,8,4)	(6,6,8,4)
3500	(7,6,9,4)	(8,7,9,4)	(7,7,9,4)
4000	(8,6,10,4)	(9,7,10,4)	(8,8,10,4)
4500	(8,7,11,4)	(9,7,11,4)	(9,9,11,4)
5000	(8,7,11,4)	(9,7,11,4)	(10,9,12,4)
5500	(8,7,11,4)	(9,7,11,4)	(10,9,12,4)
6000	(8,7,11,4)	(9,7,11,4)	(10,9,13,4)

Table 6.1: Updates of the cut-off function k for different shields.

6.2 Scenario 2: Changing traffic density on a road network.

We tested our adaptive shields by using a part of the urban traffic network of Helsinki and shielded 6 selected intersections as shown in Figure 6.2.

The main traffic flow across the city is on the highway. We use a traffic light controller that is optimized for that scenario. In the simulation, we block a section of the highway in the north-south direction from time step 1000 to 6000. This causes the cars to take a detour on the smaller roads before getting back on the highway.

The Controller

In this scenario, we deal with multiple controllers on different types of junctions. Every traffic light controller is auto-generated with the OSM imported configuration. Those controllers have fixed durations and, depending on the topology, different phases. The controller generations follow the pattern of 45 seconds duration, including a 3 seconds yellow phases.

6.2.1 The Shield

Due to the detour of the cars that leave the highway, the traffic density on two junctions on the highway and four junctions on side roads changes drastically. We place a local shield at each of these six crossings. Note that the fact that we synthesize local shields makes our approach highly scalable, and we could easily place a shield at any junction in the city. We compute for each crossing a model based on the usual traffic density and

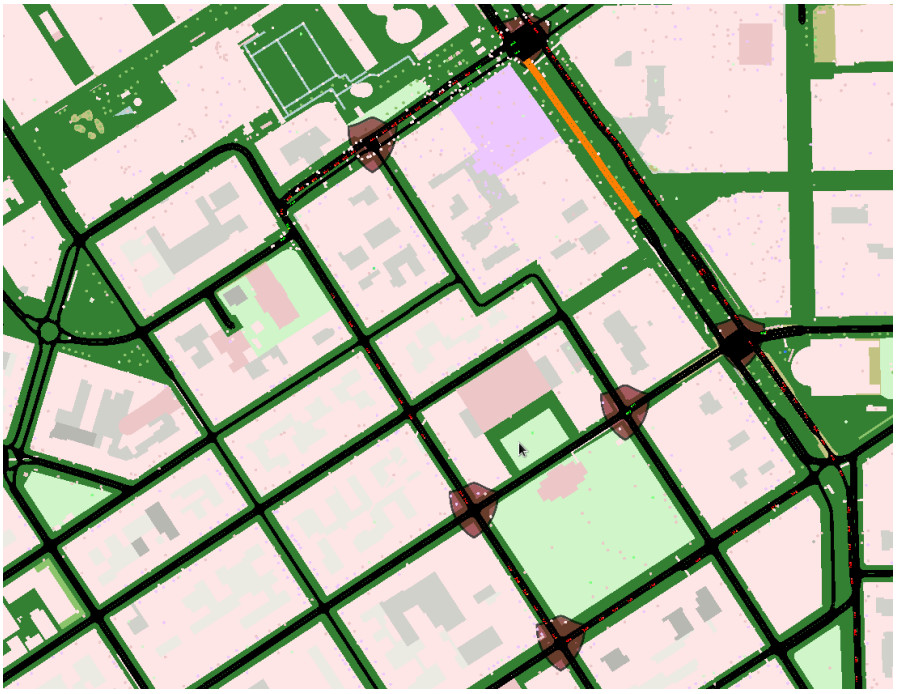

Figure 6.2: SUMO representation of a selected area of the Helsinki traffic network showing 6 shielded junctions (shaded red).

the available controller actions and synthesize initial shields. The cost functions and parameters are the same as in Scenario 1.

6.2.2 Results

Figure 6.3 shows the shields' effect on the accumulated waiting time of all cars in the city. The results of Scenario 2 substantiate the previous results. The controller without a shield (red line) performed well initially until the traffic situation changed and traffic started to congest due to non-adaptive traffic control. The shields adapted to the new traffic density quickly and reduced the waiting times by orders of magnitude.

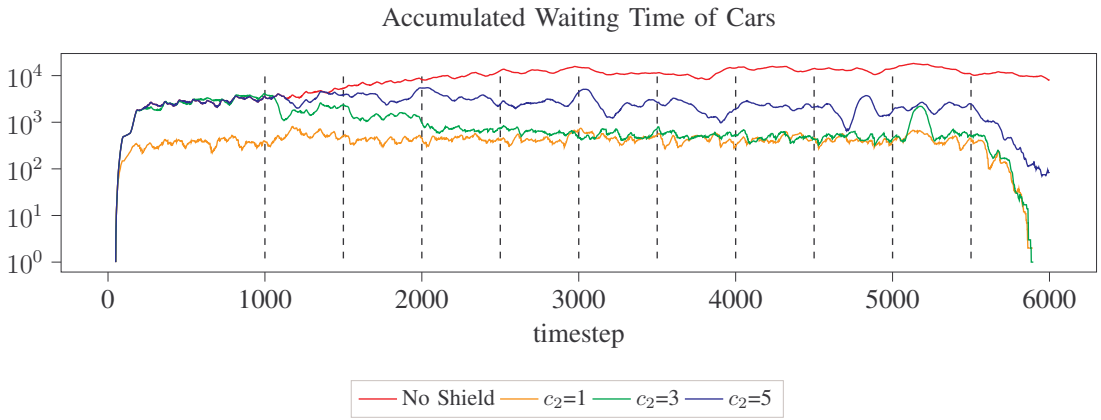

Figure 6.3: Results for Scenario 2: Changing traffic in Helsinki.

6.3 Scenario 3: Prioritizing Public Transport

This scenario demonstrates that shields can be used to add functionality to an existing controller. Mostly when designing learned controllers, shields can be seen as a tool for designers to optimize a secondary objective and keep the learning agent's reward structure simple. The controller types are similar to scenario 2, but we modified the traffic and added busses.

6.3.1 The Shield

The model $\mathcal{M}$ over which the shield is constructed slightly differs from the one used in the other experiments. The state space is likewise composed of 5-tuples $s = (n, s, e, w, ctr)$, but this time the variables n, s, e, and w represent the total waiting time of all buses on the corresponding roads. For example, the state $(0, 21, 0, 0, EW_C)$ represents that buses are waiting in the South queue for a total time of 21 seconds. Outgoing edges take the frequencies of arriving buses into account. As a performance measure, we assign a cost to each abstract state, which is equal to the difference in the bus waiting times per direction. All other parameters kept the same.

6.3.2 Results

Figure 6.4 shows the total waiting time of all vehicles and only busses as a function of the interference cost c_2. We observe the predicted behaviour. The interference of the shield improves the buses' waiting time at the expense of the general waiting time.

Figure 6.4: Results for Scenario 3: Influence on normal traffic.Results for Scenario 3: Prioritizing public transport over normal traffic.

Chapter 7

Future Work

This chapter wants to discuss possible future works on the shielding method and the shielding framework.

7.1 Shielding Method

For future work, we will examine different ways to learn from observations of the environment. For instance, instead of estimating multinomial distributions for the transition probabilities, we may consider estimating binomial distributions for individual transitions probability. Zhao et al. [Zha+20] presented such an approach for interval Markov chains. The authors also presented change-point detection for interval Markov detection. We intend to explore different methods for change-point detection to detect changes in the traffic flow. Among others, we want to explore an adaption of the technique of Zhao et al.

Finally, we will apply our adaptive shields on controllers trained via state-of-the-art reinforcement learning and investigate the performance improvements.

7.2 Shielding Framework

In this section, we want to discuss the future development of the shielding framework. The shielding framework provides a good range of features and performance, but it can still be improved. Especially in combination with the RL agent, there are several improvements possible.

7.2.1 Reinforcement Learning Setup

Currently, the Shielding Application and the RL agent are two separate applications, and they are manipulating the simulation independently in a defined order. Therefore, both applications have to fetch the simulation states, which increase the time per timestep. Furthermore, we have to use `TraCI` and can not use faster methods to interact with the simulation like Libsumo [DLR+20]. Libsumo allows to start `SUMO` as a library and avoids the communication overhead over the socket connection. It increases the performance and disables features like the GUI mode and `TraCI`, which is needed to run Shielding Application and RL agent on the same simulation.

On the RL agent's scope, we have to handle several RL frameworks, which take an abstraction class of the simulation. To better manage the RL agent learning with enabled shields, it would be good to handle the shielding cases on this layer.

Therefore, we want to propose the Shielding Proxy, which should be placed between the `SUMO` simulation and the RL agent. The idea is to provide the Shielding Application as the library and provide it to the RL agent via a python wrapper. With this architecture, we have the same functionality and bring Shielding Application and RL agent together. The Shielding Application can still run without the RL agent with python. In combination with the Shielding Application, we have only one `TraCI` client, which allows us to use the Libsumo mode. Furthermore, the Shielding Application, and in this case, the shielding proxy, has an improved `TraCI` performance due to subscription pattern and caching, which improves the RL agent. The shielding proxy would also provide other features like accurate lane state for OSM imported maps.

We calculate with a minimal development effort for the needed adaption and expect more flexibility with the RL agent.

7.2.2 Increase Model Checking Performance

The `Storm` model checker provides additional interfaces [Deh+17], to the command-line interface. One is for Python, which we do not need, and the other is a C++ API. The C++ API provides options to increase performance and access to internal functions of `Storm`. For future work, this can be used to increase the performance by cache build models. Furthermore, it is possible to speed up probability update by update the probabilities directly in the `Storm` structures. This would need to change to the C++ API and remove the `Storm` process handling from the Shielding Application.

Chapter 8

Conclusion

This thesis aimed to implement the shielding framework, showed the adaptive shielding method, and evaluated the shielding method with different scenarios. Shields can be used to ensure that safety and performance requirements are met or add functionality while minimally interfering with the global controller. Furthermore, adaptive shielding is an approach for adaptive control through local changes of the global control scheme at runtime.

The focus on adaptive shielding lies on shields in changing and uncertain environments. As an underlying model for shield computation, we use MDPs with costs, where our method created an internal abstract representation of the MDP. Adaptivity is ensured by regular updates and refinement steps of this model based on the environment's observations during runtime. We demonstrate our approach using experiments on urban traffic control with changing traffic scenarios. Our experiments show that shielding enables adaptation to change traffic flow.

Furthermore, we presented our shielding framework, containing a traffic simulation platform and a model checker (for the shield synthesis) and the Shielding Application that implements the shields and put everything together. The Shielding Application demanded the significant implementation effort of this work and is designed to enable shielding scenarios beyond our adaptive shielding method. Therefore, we gave a detailed overview of the implementation and features. For a more detailed insight, we provided Doxygen Documentation.

Due to the Docker images, the shielding framework can run on several platforms with minimal efforts, like Linux, Windows, and it can be deployed to the cloud.

Bibliography

[Ale19] Lucas N. Alegre. *SUMO-RL*. `https://github.com/LucasAlegre/sumo-rl`. 2019.

[Alf98] Luca De Alfaro. "How to Specify and Verify the Long-Run Average Behavior of Probabilistic Systems". In: *In Proc. LICS'98*. 1998, pp. 454–465.

[Als+18] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. "Safe Reinforcement Learning via Shielding". In: *AAAI*. AAAI Press, 2018, pp. 2669–2678.

[Avn+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: *CAV (1)*. Vol. 11561. LNCS. Springer, 2019, pp. 630–649.

[Bai+14] Christel Baier, Joachim Klein, Sascha Klüppelholz, and Steffen Märcker. "Computing Conditional Probabilities in Markovian Models Efficiently". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 515–530. ISBN: 978-3-642-54862-8.

[Bai16] Christel Baier. "Probabilistic Model Checking". In: *Dependable Software Systems Engineering*. Vol. 45. IOS Press, 2016, pp. 1–23.

[BH08] Luca F. Bertuccelli and Jonathan P. How. "Estimation of non-stationary Markov Chain transition models". In: *CDC*. IEEE, 2008, pp. 55–60.

[Bha+19] Suda Bharadwaj, Roderick Bloem, Rayna Dimitrova, Bettina Könighofer, and Ufuk Topcu. "Synthesis of Minimum-Cost Shields for Multi-agent Systems". In: *ACC*. IEEE, 2019, pp. 1048–1055.

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

[Blo+15] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. "Shield Synthesis: - Runtime Enforcement for Reactive Systems". In: *TACAS*. Vol. 9035. LNCS. Springer, 2015, pp. 533–548.

[Bou+19] Maxime Bouton, Jesper Karlsson, Alireza Nakhaei, Kikuo Fujimura, Mykel J. Kochenderfer, and Jana Tumova. "Reinforcement Learning with Probabilistic Guarantees for Autonomous Driving". In: *CoRR* abs/1904.07189 (2019).

[Brá+15] Tomás Brázdil, Krishnendu Chatterjee, Vojtech Forejt, and Antonín Kucera. "MultiGain: A Controller Synthesis Tool for MDPs with Multiple Mean-Payoff Objectives". In: *TACAS*. Vol. 9035. LNCS. Springer, 2015, pp. 181–187.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[CKK17] Krishnendu Chatterjee, Zuzana Kretínská, and Jan Kretínský. "Unifying Two Views on Multiple Mean-Payoff Objectives in Markov Decision Processes". In: *Log. Methods Comput. Sci.* 13.2 (2017). DOI: 10.23638/LMCS-13(2:15)2017. URL: https://doi.org/10.23638/LMCS-13(2:15)2017.

[CN12] Yingke Chen and Thomas Dyhre Nielsen. "Active Learning of Markov Decision Processes for System Verification". In: *ICMLA (2)*. IEEE, 2012, pp. 289–294.

[Deh+17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. *A storm is Coming: A Modern Probabilistic Model Checker*. 2017. arXiv: 1702.04311 [cs.SE]. URL: https://arxiv.org/abs/1702.04311v1.

[DLR+20] German Aerospace Center (DLR) et al. *SUMO User Documentation*. 2020. URL: https://sumo.dlr.de/docs/index.html (visited on 02/05/2021).

[FP18a] Nathan Fulton and André Platzer. "Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning". In: *AAAI*. AAAI Press, 2018, pp. 6485–6492.

[FP18b] Nathan Fulton and André Platzer. "Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning". In: *AAAI*. AAAI Press, 2018.

[FP19] Yliès Falcone and Srinivas Pinisetty. "On the Runtime Enforcement of Timed Properties". In: *RV*. Vol. 11757. LNCS. Springer, 2019, pp. 48–69.

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995. ISBN: 0-201-63361-2.

[GF15] Javier Garcıa and Fernando Fernández. "A comprehensive survey on safe reinforcement learning". In: *Journal of Machine Learning Research* 16.1 (2015), pp. 1437–1480.

[Hah+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)*. Vol. 11427. LNCS. Springer, 2019, pp. 395–412.

[HAK20] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. "Cautious Reinforcement Learning with Logical Constraints". In: *AAMAS*. International Foundation for Autonomous Agents and Multiagent Systems, 2020, pp. 483–491.

[Hen+12] David Henriques, Joao G Martins, Paolo Zuliani, André Platzer, and Edmund M Clarke. "Statistical model checking for Markov decision processes". In: *2012 Ninth international conference on quantitative evaluation of systems*. IEEE. 2012, pp. 84–93.

[Hen+20] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. "The Probabilistic Model Checker Storm". In: *arXiv preprint arXiv:2002.07080* (2020).

[Jan+20] Nils Jansen, Bettina Könighofer, Sebastian Junges, Alex Serban, and Roderick Bloem. "Safe Reinforcement Learning Using Probabilistic Shields (Invited Paper)". In: *CONCUR*. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 3:1–3:16.

[KLM96] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[KNP11] Marta Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 585–591. ISBN: 978-3-642-22110-1.

[Lop+18] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. "Microscopic traffic simulation using sumo". In: *ITSC*. IEEE. 2018, pp. 2575–2582.

[Mao+16] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. "Learning deterministic probabilistic automata from a model checking perspective". In: *Mach. Learn.* 105.2 (2016), pp. 255–299.

[Mou15] Adrian Mouat. *Using Docker: Developing and Deploying Software with Containers*. " O'Reilly Media, Inc.", 2015.

[Omd20] Omdia. *Revenues from the artificial intelligence (AI) software market worldwide from 2018 to 2025 (in billion U.S. dollars) [Graph]*. `https://www.statista.com/statistics/607716/worldwide-artificial-intelligence-market-revenues/`. 2020. (Visited on 03/03/2021).

[Pra+20] Stefan Pranger, Bettina Könighofer, Martin Tappler, Martin Deixelberger, Nils Jansen, and Roderick Bloem. *Adaptive Shielding under Uncertainty*. 2020. arXiv: `2010.03842 [cs.LO]`.

[Ren+19] Matthieu Renard, Yliès Falcone, Antoine Rollet, Thierry Jéron, and Hervé Marchand. "Optimal enforcement of (timed) properties with uncontrollable events". In: *Math. Struct. Comput. Sci.* 29.1 (2019), pp. 169–214.

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

[Sch00] Fred B. Schneider. "Enforceable Security Policies". In: *ACM Trans. Inf. Syst. Secur.* 3.1 (Feb. 2000), pp. 30–50. ISSN: 1094-9224. DOI: 10.1145/353323.353382. URL: https://doi.org/10.1145/353323.353382.

[Sil+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[SU95] Stefan Strobel and Thomas Uhl. "X11". In: *LINUX — Vom PC zur Workstation: Grundlagen, Installation und praktischer Einsatz*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 163–206. ISBN: 978-3-642-97575-2. DOI: 10.1007/978-3-642-97575-2_8. URL: https://doi.org/10.1007/978-3-642-97575-2_8.

[Tap+19] Martin Tappler, Bernhard K. Aichernig, Giovanni Bacci, Maria Eichlseder, and Kim G. Larsen. "L^*-Based Learning of Markov Decision Processes". In: *FM*. Vol. 11800. LNCS. Springer, 2019, pp. 651–669.

[Tra19] Tractica. *Forecast growth of the artificial intelligence (AI) software market worldwide from 2019 to 2025*. https://www.statista.com/statistics/607960/worldwide-artificial-intelligence-market-growth/. 2019. (Visited on 03/03/2021).

[w3s17] w3sDesign. *The GoF Design Patterns Reference*. 2017. URL: http://w3sdesign.com/GoF_Design_Patterns_Reference0100.pdf (visited on 02/01/2021).

[Wan+19] Angelina Wang, Thanard Kurutach, Kara Liu, Pieter Abbeel, and Aviv Tamar. "Learning Robotic Manipulation through Visual Planning and Acting". In: *CoRR* abs/1905.04411 (2019). arXiv: 1905.04411. URL: http://arxiv.org/abs/1905.04411.

[Weg+08] Axel Wegener, Michał Piórkowski, Maxim Raya, Horst Hellbrück, Stefan Fischer, and Jean-Pierre Hubaux. "TraCI: An Interface for Coupling Road Traffic and Network Simulators". In: *Proceedings of the 11th Communications and Networking Simulation Symposium*. CNS '08. Ottawa, Canada: Association for Computing Machinery, 2008, pp. 155–163. ISBN: 1565553187. DOI: 10.1145/1400713.1400740. URL: https://doi.org/10.1145/1400713.1400740.

[Whi85] Douglas J White. "Real applications of Markov decision processes". In: *Interfaces* 15.6 (1985), pp. 73–83.

[Wu+19] Meng Wu, Jingbo Wang, Jyotirmoy Deshmukh, and Chao Wang. "Shield Synthesis for Real: Enforcing Safety in Cyber-Physical Systems". In: *FMCAD*. IEEE, 2019, pp. 129–137.

[YK17] Murti Yasodharan and Kumari Krishna. "DOXYGEN: A TOOL USED TO AUTOMATICALLY GENERATES FUNCTION LISTS AND CLASS/UML DIAGRAMS FOR SOFTWARE PROGRAMMING PROJECTS". In: *Journal on Recent Innovation in Cloud Computing, Virtualization & Web Applications [ISSN: 2581-544X (online)]* 1.1 (2017).

[Zha+20] Xingyu Zhao, Radu Calinescu, Simos Gerasimou, Valentin Robu, and David Flynn. "Interval Change-Point Detection for Runtime Probabilistic Model Checking". In: *ASE*. ACM, 2020, 12 pages. DOI: 10.1145/3324884.3416565. URL: https://doi.org/10.1145/3324884.3416565.