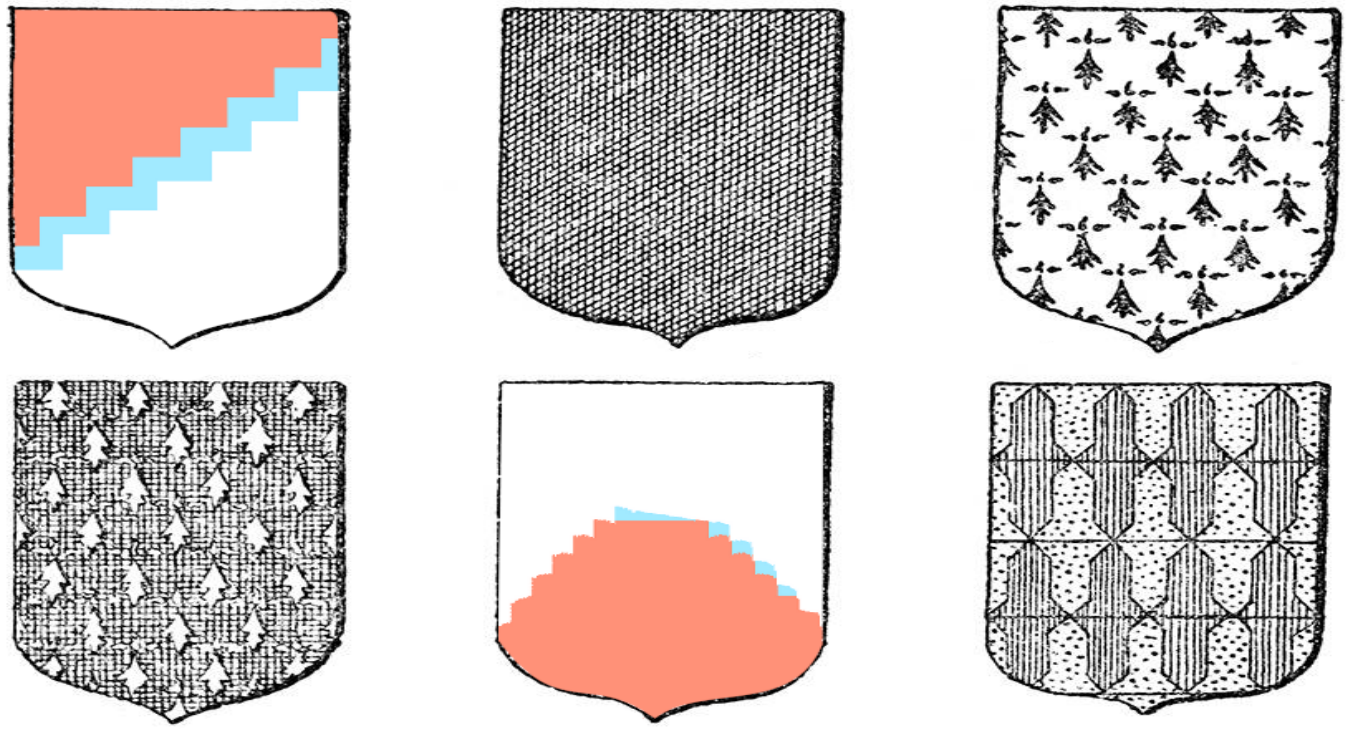

Shielded AI for Hybrid Systems

Asger Horn Brorholt

Computer Science International

Thesis

Department of Computer Science
Aalborg University
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Shielded AI for Hybrid Systems

Theme:
Semantics and Verification

Project Period:
Spring Semester 2022

Project Group:
Individual

Participant(s):
Asger Horn Brorholt

Supervisor(s):
Kim G. Larsen
Florian Lorber
Peter G. Jensen

Copies: 0

Page Numbers: 50

Date of Completion:
June 10, 2022

Abstract:

When safe behaviour of a reinforcement learning agent or other complex model cannot be guaranteed through verification methods, a shield can be put in place to enforce a given safety property. Safe actions suggested by the model are taken without modification, while potentially unsafe actions are corrected according to a synthesized safety strategy.
A technique is developed for hybrid systems, to distinguish between safe and unsafe actions, for a given safety property.
The technique was applied to two problems of a finite and infinite time horizon respectively, and the resulting safe strategy was successfully imported into Uppaal Stratego to make use of the tool's capabilities for machine learning and statistical model checking while running experiments on the effects of shielding.
When learning occurs with the developed shield in place, learning outcomes are in many instances seen to be as good, or better than, their unshielded counterparts.

agreement with the author.

Summary

When safe behaviour of a reinforcement learning agent or other complex model cannot be guaranteed through verification methods, a shield can be put in place to enforce a given safety property. Safe actions suggested by the model are taken without modification, while potentially unsafe actions are corrected according to a synthesized safety strategy. When a learning component is present, safety can either be enforced through pre-shielding, where a shield is in place during both the training and operation phase, or post-shielding, where a shield is applied after training has taken place.

A technique is developed for hybrid systems, to distinguish between safe and unsafe actions, for a given safety property. The state space of the system is partitioned into squares (or their higher-dimensional equivalent) of equal size, and a function is developed to compute, for a given initial square and action, the set of squares reachable by that action from one or more points within the initial square. This reachability function can be used to over-approximate the set of unsafe actions for any point inside a given square.

Any square containing a state that is in violation of the safety property is considered unsafe, and from any given square, an action that has the potential to reach an unsafe square, is considered unsafe. Additionally, there may be points where any action potentially leads to a violation of the safety property, and so any square with no safe actions is considered unsafe in turn. By computing the maximal fixed point of safe actions for all squares, the most permissive strategy under the abstraction given by the partition can be reached.

The over-approximation was seen to become more pessimistic at coarser granularities, and this could reach a point where there were no safe actions from the initial position of the model. Thus, memory restrictions make this technique most useful for low-dimensionality state spaces with fixed bounds, which allow for a more fine-grained representation of the state space.

The technique was applied to two problems of a finite and infinite time horizon respectively, and the resulting safe strategy was successfully imported into Uppaal Stratego to make use of the tool's capabilities for machine learning and statistical model checking.

Experiments were conducted in Uppaal Stratego using models of the two problems. The results showed no safety violations when the shields were in place. Additionally, pre-shielding was seen to improve learning outcomes over some configurations that did not have a shield in place. Post-shielding saw penalties to the model's performance that were potentially quite large, but it was still effective at enforcing safety properties for models that previously did not respect them.

Contents

Introduction

Software systems are increasingly used to automate or optimize tasks, but for some problems, such as nuclear power, medical systems, robotic equipment, and smart energy grids, high safety standards must be met before if a system is to be deployed in good conscience.

Model checking is a broad category of scientific research, which strives to verify that certain properties hold for a given (abstraction of a) system. Synthesis takes this one step further, by providing a strategy for guaranteeing that a given property will hold in the system.

One popular solution for problem-solving using software is reinforcement learning, and other types of machine learning. Though verification techniques for these do exist, the complexity of some machine learning models can make it prohibitively difficult. Still, the nature of the optimization problem may be such, that hard-to-verify models are able to achieve a greater overall performance, compared to ones that can be more easily verified using formal methods.

In this case, shielding can be employed to provide safety guarantees by synthesizing a safe strategy from the given safety property. By employing the most permissive strategy for guaranteeing a safety property, the shield can intervene where the actions of a machine learning agent would lead to a safety violation, while keeping as many options as possible open to the agent, for optimizing the outcome.

In the following, a technique for generating a safe strategy in a hybrid context will be presented, that is shown to work in both a finite and an infinite time horizon. The technique is applied to two problems, and the effects of applying the shield to an agent are explored using machine learning algorithms supplied by Uppaal Stratego.

1 Preliminaries

This section introduces the concept of shielding, as well as the Euclidian Markov Decision process, which forms the basis for stating the two example problems presented in later sections. It ends by presenting Uppaal Stratego's approach to shielding, and how it is supplemented by this thesis.

1.1 Shielding

The software tool Uppaal Stratego was introduced by David et. al. [5], and counts among its features the option to compute strategies both by synthesis, to guarantee a safety property, and by machine learning, to optimize outcomes. Notably,

strategies can be layered to find safe and optimized behaviour. A year later, the term *shielding* was coined in a paper by Alshiekh et. al. [1] which showed a way to combine the benefits of machine learning and formal methods.

Focusing on reinforcement learning, shielding brings guarantees of safety into systems that cannot be formally verified. This might be due to the complexity of the learning algorithm employed, which can make formal verification intractable. In the extreme case, state of the art neural network solutions achieve their results by relying on billions of parameters [16]. Even in networks of more modest size, their full behaviour can be difficult to reason about, though some techniques exist [15].

While this may be acceptable for systems whose goal is simply to optimize some performance metric, other systems may have safety properties which must hold true at all times, to which any optimization metric is secondary. One example might be of a self-driving car, which should ensure the comfort of the driver by avoiding sudden changes in velocity, but which should take all necessary measures to avoid obstacles. Another might be a nuclear power plant that could be optimized to match its energy output to demand, but must avoid a reactor meltdown at any cost.

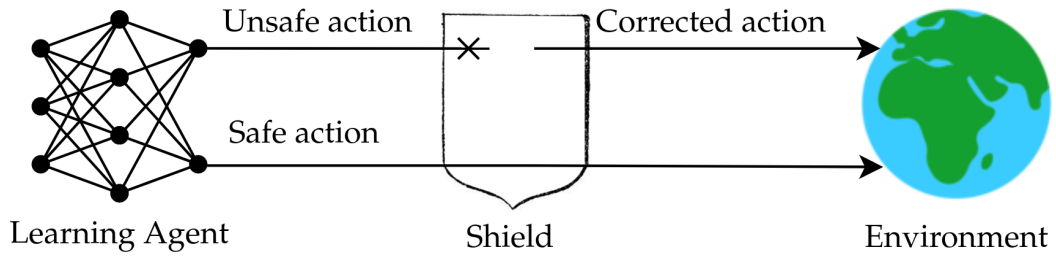

Figure 1: The first action taken by the learning agent is potentially unsafe, and gets corrected as a result. The second action chosen by the agent is safe, and therefore not altered by the shield.

This is where shielding can be employed, to preserve some of the benefits machine learning might offer. A permissive strategy – dubbed a shield – is synthesized to preserve a given safety property, such that actions which lead to a potential violation of the property are disallowed. Preferably, the shield should be the *most* permissive strategy for the given abstraction of the system, since this leaves the greatest amount of choices to the machine learning agent, with which to optimize outcomes. How a shield could be applied to a learning agent is shown in Figure 1. Actions permitted by the shield are taken without modification, while actions that are not permitted has to be altered. In cases where multiple other actions are permitted, selection can happen either randomly or according to some backup policy. If only one other action remains, the choice becomes trivial.

Depending on whether the machine learning agent has been trained with or without the restrictions of the shield, the term pre-shielding or post-shielding ap-

plies. The former describes the case where a shield is in place during training. In this case, the learning agent never gives rise to a safety violation, since the shield prevents this. Instead, the agent learns to take into account the interference of the shield, when optimizing for outcomes.

In the latter case – post-shielding – the shield is applied to an agent which has already been trained without a shield in place. This might be relevant if a trained agent is provided by a third party, or if safety concerns arise after expensive learning has already taken place. The degree of interference by the shield in this case will be determined partially by how permissive the shield is, and partially by how well the agent has been trained to avoid the safety violations. If the learned agent's policy involves entering states that are not allowed by the shield, it might perform poorly with regard to the optimization criterion.

1.2 Euclidian Markov Decision Processes

For some systems, such as population growths, physics, or the spread of a pandemic, continuous values will best capture their dynamics and stochasticity. One model, which captures well the dynamics of many such systems, is an Euclidian Markov Decision Process [10] (EMDP). It can be defined as a tuple $(\mathcal{S}, s_0, Act, T, C)$ where

$$
\begin{aligned}
\mathcal{S} &\text{ is a subset of } \mathbb{R}^K \text{ for some fixed dimensionality } K \\
s_0 &\text{ is the initial state, } s_0 \in \mathcal{S} \\
Act &\text{ is a finite set of actions } \{a, b, \dots\} \\
T &\text{ is a transition function } \mathcal{S} \times Act \times \mapsto \mathbf{\Delta} \\
&\quad \text{where } \mathbf{\Delta} \text{ is the set of probability density functions } \mathcal{S} \mapsto [0;1]. \\
C &\text{ is the immediate cost of taking an action } Act \times \mathcal{S} \mapsto \mathbb{R}_{\geq 0}
\end{aligned}
$$

Here, $\mathbb{R}$ refers to the set of real numbers, while $\mathbb{R}_{\geq 0}$ refers to the set of positive real numbers. A state can be expressed as a vector of dimensionality K, while the probability of transitioning from one state s_1 to a set of states $A \subseteq \mathcal{S}$ is given by the transition function T. For the state and an action $a \in Act$, the function $T(s_1, a)$ returns a probability density function $\Delta \in \mathbf{\Delta}$, such that $\int_{s \in \mathcal{S}} \Delta(s) ds = 1$. Then, the probability of transitioning into A is given as $\int_{s \in A} \Delta(s) ds$.

A trace $s_0, a_1, s_1, a_2, s_2, \dots$ is a series of states, s_i and the actions that lead to them, a_i. The total cost of a trace is the sum of all the immediate costs, given by C. That is, $\sum_i C(s_i, a_{i+1})$.

1.3 UPPAAL STRATEGO

Uppaal Stratego [5] is a software application which among other things can synthesize safety strategies for Stochastic Priced Timed Games (SPTGs). Using Stratego, the most permissive strategy for guaranteeing a property can be found, regardless of the random outcomes of the model. These strategies are non-deterministic, since they can allow multiple actions for a given state, excluding only those which have a chance of leading to a (future) violation of the given property. In addition to this, strategies can also be found for achieving the highest (or lowest) expected time or price in a run of the model. Such strategies will be deterministic, prescribing for each state a single action which has the highest likelihood of producing a favourable outcome.

Stratego achieves this by combining features from and building upon other applications in the Uppaal family. The synthesis capabilities of Uppaal Tiga is used to find winning strategies on Timed Game Automata (a 2-player timed game) to guarantee given properties. In addition to this, Uppaal Stratego provides machine learning algorithms to compute strategies which optimize for some given parameter. Lastly, Uppaal SMC is used for computing expected outcomes, when applying a strategy to the non-deterministic choices of a model.

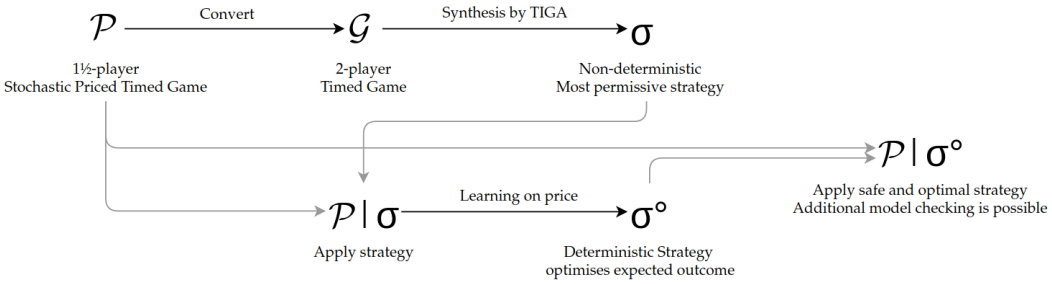

Figure 2: The process of creating a safe and optimized strategy using Uppaal Stratego. Black lines represent actions that produce the elements, while gray lines show how the elements are further used.

The process of Stratego generating a safe and optimal strategy is shown in Figure 2. Given an SPTG $\mathcal{P}$, it is possible to enforce an invariant property by converting it to a Timed Game Automaton $\mathcal{G}$. The conversion happens by simply ignoring the price of edges, and turning the stochastic $1\frac{1}{2}$player game into a 2-player game where the opponent controls the edges which were previously random, and had a non-zero probability of occurring. This conversion allows the opponent to win only if there is a non-zero chance that the same outcome could occur in the SPTG.

A winning strategy for the controller σ can then be applied back to $\mathcal{P}$ since the edges of $\mathcal{G}$ correspond to those of $\mathcal{P}$. This allows σ to enforce the desired

property with absolute certainty, while also considering the expected price under this strategy. A near-optimal strategy σ^o can then be learned, to minimise (or maximise) the expected price given the random outcomes of the model.

This safe and optimal strategy is a form of pre-shielding, with the winning strategy of the deterministic 2-player game acting as a shield that restricts the learning agent, as it searches for the optimal strategy in a stochastic system for some desired property.

This method of shielding holds for SPTGs, but does not extend to hybrid automata. The modelling language of UPPAAL supports hybrid automata by stating clock rates as ordinary differential equations, but doing so makes UPPAAL TIGA's approach to synthesizing a safe strategy undecidable. Optimization though machine learning is still supported however, and instead, a non-deterministic safe strategy can be synthesized and applied through other means.

In the following, it will be described how a continuous state space can be partitioned into squares, and how a strategy *SafeAct* can be synthesized based on this partitioning. As shown in Figure 3, the strategy *SafeAct* will be used in place of σ. The strategy will be synthesized outside of UPPAAL, and applied by importing it as an external function.

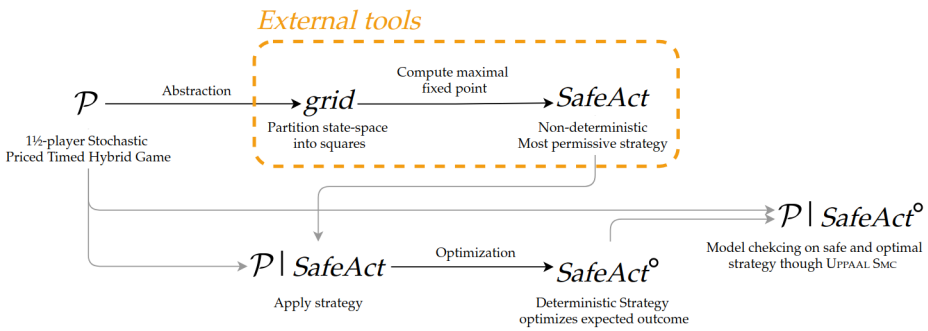

Figure 3: The process of creating a safe and optimal strategy in a hybrid context using UPPAAL STRATEGO and a safe strategy synthesized using external code.

If re-training a learned strategy is not an option, it is possible that post-shielding can introduce safety guarantees while preserving some of the performance achieved by the learned behaviour. The process of post-shielding a learned strategy is shown in Figure 4. Learning is applied to the model alone, without any restrictions to its behaviour. Instead, a penalty for safety violations should be added to the cost function, to guide the learning agent towards safe behaviour. *SafeAct* is then applied to the model, altering the behaviour of the system after learning has occurred, to only allow safe actions.

The technique for partitioning a state-space and synthesizing a safe strategy will be described using the example of a problem dubbed Random Walk. In a later

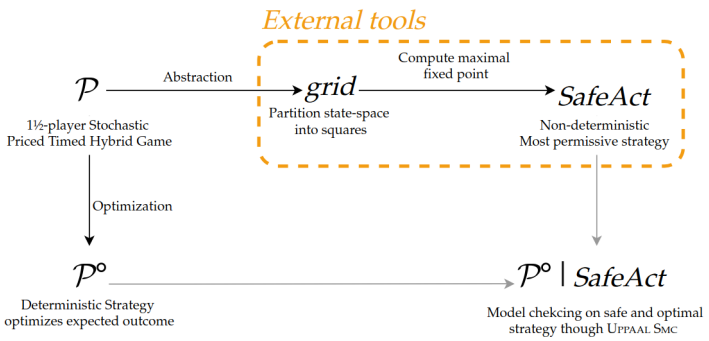

Figure 4: Post-shielding of an optimal strategy learned in Uppaal Stratego, with a safe strategy synthesized using external code.

section, the technique will be applied to an infinite time horizon as illustrated by a second problem, called Bouncing Ball.

2 Related Works

This thesis builds upon and expands work done in a previous semester project [4]. Parts of Section 1.3 are copied verbatim from the project, and other sections and ideas presented, are revised or expanded upon.

The concept of shielding – in the way it is used in this thesis – is also known as *runtime enforcement* in the literature. A comprehensive survey can be found in [11], which provides a taxonomy of runtime monitors with regards to the security properties they can enforce, and the ways in which they can interfere with the system. The paper's focus is on security properties (safety and liveness) that can be expressed as either finite, or countably infinite automata, and does not touch on hybrid systems. It does, however, describe how more recent research into runtime enforcement has concerned itself with memory and computational constraints, as well as higher standards for how the monitor is allowed to intervene. By the terminology of the survey, the shielding method presented in this thesis provides precise enforcement, in that it does not alter actions if they are part of a valid execution. It can enforce safety properties, and it does so in a uniform context – that is, without knowledge of the possible behaviour of the shielded agent.

Runtime enforcement in a timed context is described in [13]. Safety and co-safety properties are defined using timed automata, and a *delay automaton* is generated which is able to either delay an action or halt the system, to ensure the property is maintained. Definitions are given of soundness, transparency and op-

timality, and the delay automaton is proven to have these properties. Soundness requires that all traces that pass through the delay automaton are safe, while transparency means that traces which already satisfy the safety property pass through it unaltered. Optimality requires that if the monitors corrects behaviour by delaying, it does this using the smallest possible delay.

The effects on learning of restricting actions is explored in [17]. The paper provides no method for discarding invalid or unsafe actions, but assumes that many real-world systems already have controllers in place that will do this. Assuming that such a controller is available, it proposes a modification of Deep Q-learning to better learn from actions that are disallowed by the system. By altering the Q-values for actions that are disallowed, faster convergence is achieved compared to ordinary Deep Q-learning. In contrast, this thesis provides a method for shielding actions for certain systems and safety properties, and examines the effects of shielding on a different type of Q-learning, but without any special-case handling of rejected actions. Instead, the outcome of a rejected action will simply appear to the agent as as whichever action was picked in its place.

The Random Walk problem, which is used as a case study in this thesis, was originally introduced by [9]. The paper shows a method for approximating Euclidian Markov Decision Processes using finite-state imprecise MDPs, by partitioning the state space into a grid of square partitions of equal size. This partitioning is used as a basis to approximate the cost function of the Random Walk game, and other finite-time horizon problems. The approximation presented by [9] is shown to become arbitrarily precise, as increasingly fine partitioning is applied. However, it is not examined how such a partitioning can be used to model safety properties, and no safety property is provided for the Random Walk problem in the paper. This thesis uses a similar concept of partitioning to provide a permissive strategy to guarantee a safety property for the Random Walk problem. It was not proven whether the partitioning used in this thesis has the same property of becoming arbitrarily precise when the granularity is increased, but the shields do seem to become more permissive for finer granularities.

3 Random Walk

In [9], the idea of approximating continuous, finite time horizon systems using discrete partitions was applied on a *random walk* example. Safety was not considered in the original paper, but a penalty was applied in the cost function, if a game was „lost."

Using a similar partition-based approach, the following will employ the simple but non-trivial Random Walk problem as a running example. Losing the game will instead be considered a safety violation, and the cost function is modified to create

a more complex learning problem.

3.1 Description

The goal of the Random Walk problem is to cover a certain distance within a given time limit. Call the distance travelled x, and the time elapsed t. Time and distance both start at 0.0, and a distance of 1.0 has to be covered before the timer exceeds 1.0. Even if the actions of the player reaches $x > 1.0$, the game is still considered lost if at the same time $t > 1.0$. Only the states $x > 1.0 \wedge t \leq 1.0$ are part of the winning region.

The player can either move slow, spending a large amount of time to move a relatively short distance, or fast, to cover a greater distance in a shorter time. The cost of each action will vary depending on the current distance, but as a rule moving fast will be more expensive. The player should choose the cheapest possible actions, while making sure to reach the goal in time. An action $a \in Act = \{fast, slow\}$ covers a distance given by $\delta(a)$ and time $\tau(a)$. These constant values are augmented by an uncertainty ϵ, leading to a uniform probability distribution over the values shown in Equation 1.

$$[x + \delta(a) - \epsilon; x + \delta(a) + \epsilon] \times [t + \tau(a) - \epsilon; t + \tau(a) + \epsilon] \tag{1}$$

Figure 5 illustrates the choice between a fast and a slow action. Choosing to go slow will land the player somewhere in the yellow area, while going fast will lead to a random place and time in the blue area.

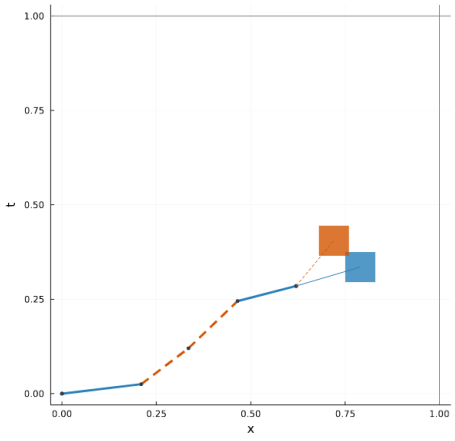

Figure 5: Illustration of the possible outcomes of going slow (yellow area) and fast (blue area) from the last point in the line plot. The other dots in the plot show results of previous actions, and the color of the line segments indicate the type of action taken.

The game ends once t or x reaches 1.0, and this puts a natural upper bound on both values. For t, this upper bound is $1.0 + \tau(slow) + \epsilon$, and as for x it is $1.0 + \delta(fast) + \epsilon$.

Equation 2 shows the values that will be used for τ, δ and ϵ. They have been chosen to create an overlap between the two possible areas, and to ensure the game will always be possible to win from the initial position.

$$\delta(a) = \begin{cases} 0.17, & \text{if } a = fast \\ 0.10, & \text{if } a = slow \end{cases}, \quad \tau(a) = \begin{cases} 0.05, & \text{if } a = fast \\ 0.12, & \text{if } a = slow \end{cases}, \quad \epsilon = 0.04 \tag{2}$$

One winning strategy would simply be to always take the *fast* action, which will be more than enough to always reach the goal without the time passing. This is where the cost function comes in, nudging the player to regulate their choice of fast and slow actions in order to achieve an optimal score. The cost depends on the progress the player has made so far, and could symbolise different terrain which favours moving either fast or slow. The cost C is based on a sinus function, and it has been chosen to create a large variance in costs, where going fast is usually – but not always – the more expensive option.

$$C(x,t) = \begin{cases} 4.5 + \sin(4\pi x), & \text{if } a = fast \\ 2.5 + \sin(\pi + 4\pi x), & \text{if } a = slow \end{cases} \tag{3}$$

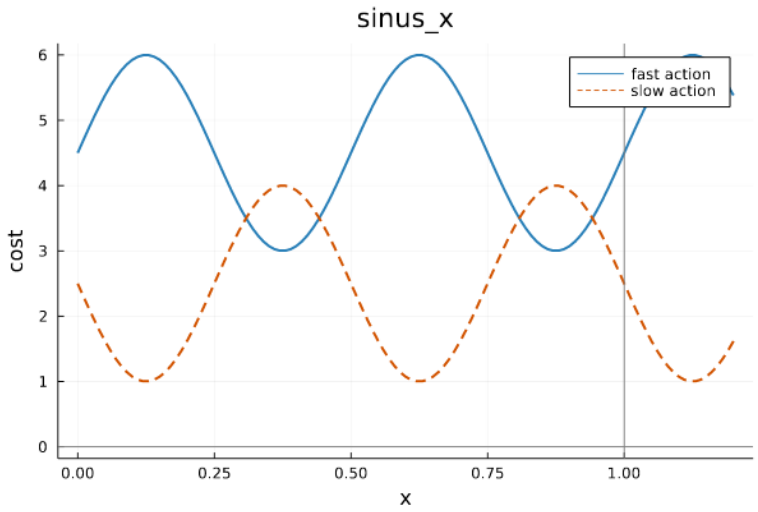

Figure 6: The cost of taking a specific action depends on the current distance x that has been covered.

Now, the problem has both a safety invariant, namely that the time should stay below 1.0, and a cost function to optimize on. The following section will lay the

theoretical groundwork for developing a shield that enforces the safety invariant, which will be implemented in later sections, and transferred to UPPAAL STRATEGO for evaluation.

3.2 Theory

Recall, that an Euclidian Markov Decision process is a tuple $(\mathcal{S}, s_0, Act, T, C)$. Let the vectors $\bar{v}, \bar{w}$ be elements of $\mathcal{S}$ and ϕ be a statement over some or all of the values in $\bar{v}$ or $\bar{w}$, then $\bar{v} \models \phi$ shall mean that ϕ is satisfied given the values in $\bar{v}$. For the random walk problem, ϕ should be the LTL formula over (x, t) that $A\Box\ t < 1.0$.

The probability of reaching a state $\bar{w}$ from an initial state $\bar{v}$ by taking some action a is given by the probability density function, $P(a, \bar{v}, \bar{w})$. If this probability is greater than zero, it is said that $\bar{w}$ is reachable by $\bar{v}$. However, the uncountably infinite state space $\mathcal{S}$ poses a challenge when, for example, a safety property has to be enforced.

The number of reachable states can be uncountably infinite, thus checking if an action is safe by computing $\forall \bar{w}, P(a, \bar{v}, \bar{w}) > 0\ .\ w \models \phi$ is not feasible in every case. Additionally, it may not be enough to simply check whether or not the next action will lead to a state that satisfies ϕ. Even if the current state satisfies ϕ, it is possible one or more of the successor states could lead to the violation of ϕ, no matter which action is taken afterwards. In the Random Walk it is possible to reach a state where taking *fast* actions exclusively still carries a risk of running out of time before reaching the goal.

One possible solution is to split the state space $\mathcal{S}$ into discrete partitions, and compute a set of actions that are safe for all states in a given partition. This would lead to an over-approximation of the set of unsafe actions, but has the benefit of not having to compute any uncountably infinite sets.

3.2.1 Partitioning the State Space

This section will define the concept of a *grid*, which represents some partitioning of the state space, and a set of allowed actions for each partition.

For the Random Walk, this would mean splitting the state space (x, t) into a set of squares that have a given lower and upper bound. For each square it is then determined, if there for either action *slow* or $fast$ exists some set of consecutive actions that are guaranteed to reach $x > 1$ without reaching $t > 1$.

They are written with the common notation of two bounds within brackets, such as $(l; u]$. A square bracket indicates that the bound is included in the set, while a parenthesis indicates a strict bound. A special case of intervals will be considered, which have a fixed size defined by some granularity constant $G \in \mathbb{R}$. Intervals of fixed size G, are those which belong to the corresponding set $\mathcal{I}_G$, described in Equation 4.

$$\mathcal{I}_G = \{[i \cdot G; i \cdot G + G) \mid i \in \mathbb{N}\} \tag{4}$$

These intervals are always open in the upper bound, such that for any granularity G and real number x, there exists a unique interval $I \in \mathcal{I}_G$ such that $x \in I$.

A set $A \in (\mathcal{I}_G)^K$ consisting of such fixed-size intervals, forms a subset of $\mathbb{R}^K$. Such tuples are dubbed squares, referring to the case $K = 2$, where the geometric area describes a square. The meaning of the symbol $\models$ is extended to apply to squares, such that a square is said to respect a property ϕ if every state in the square respects the property. That is, $A \models \phi \iff \forall \bar{v} \in A \,.\, \bar{v} \models \phi$

Recall that the state space $\mathcal{S}$ is a subset of $\mathbb{R}^K$. A partitioning of $\mathcal{S}$, $partition_{\mathcal{S}}^{G} \subseteq (\mathcal{I}_G)^K$ is defined to describe the set of squares which cover the volume of $\mathcal{S}$.

$$partition_{\mathcal{S}}^{G} = \left\{ A \in (\mathcal{I}_G)^K \mid A \cap \mathcal{S} \neq \emptyset \right\} \tag{5}$$

If $\mathcal{S}$ has both an upper and a lower bound along all axes – as is the case for the Random Walk – the set $partition_{\mathcal{S}}^{G}$ will be finite, since Equation 4 relies on natural numbers. In the following, it is assumed that such bounds on $\mathcal{S}$ exist.

To store a set of permissible actions for each square, a grid is defined to be a partitioning $partition_{\mathcal{S}}^{G}$, along with a set of actions for each square in this partition. Formally, as defined in Equation 6, a grid is a mapping from a set of squares to their allowed actions.

$$grid_{\mathcal{S}}^{G} : partition_{\mathcal{S}}^{G} \mapsto \mathcal{P}(Act) \tag{6}$$

This means that $grid_{\mathcal{S}}^{G}(A) = acts$ defines a partitioning of the state space, as well as a set of actions $acts \subseteq Act$ for every square A. If a grid maps to an action from a specific square, the grid is said to *allow* this action for that square, or alternatively that the square *contains* this action.

3.2.2 The Set of Safe Actions

Now it is possible to formally define a safe action with regards to some safety invariant ϕ, and to define a grid *SafeAct* that describes safe actions for all squares.

The statement that another square B can be reached by taking some action a, is written $A \xrightarrow{a} B$ and defined in Equation 7.

$$A \xrightarrow{a} B \iff \exists \bar{v} \in A \,.\, T(a, \bar{v}) = \Delta \wedge 0 < \int_{\bar{w} \in B} \Delta(\bar{w}) \, d\bar{w} \tag{7}$$

This leads to the definition of a reachability-function which gives all squares reachable from an initial square, by some action:

$$R_a(A) = \left\{ B \mid A \xrightarrow{a} B \right\} \tag{8}$$

An action $a \in Act$ is safe for a square A if for all squares $B \in R_a(A)$ it is the case that $B \models \phi$ and there exists at least one action that is safe for B. If the reachability function is known, it is possible to compute a grid $SafeAct^G_S$, which maps squares to a set of safe actions.

Intuitively, when computing this grid for the Random Walk, the set of squares which must be avoided are the ones where $t > 1$. Any other square in the grid where the action $slow$ could reach such a square, should not have $slow$ in its set of (safe) actions. If the same is the case for $fast$, that means the square has no safe actions, and hence the square must be avoided. Thus, actions should be removed iteratively until the grid stabilises.

Formally, the grid should be the maximal set of actions for each square, such that $SafeAct^G_S$ satisfies the property given in Equation 9.

$$SafeAct^G_S(A) = \left\{ a \mid \forall B \in R_a(A) \,.\, B \models \phi \wedge SafeAct^G_S(B) \neq \emptyset \right\} \tag{9}$$

The values of $SafeAct^G_S$ can be computed as a fixed point. The initial value should allow no actions for those squares which violate the safety property. This initialization step is given in Equation 10 where, for legibility, the values G and $\mathcal{S}$ are left implied.

$$SafeAct_0(A) = \begin{cases} Act, & \text{if } A \models \phi \\ \emptyset, & \text{otherwise} \end{cases} \tag{10}$$

The following steps in the fixed point computation consist of removing actions, which can reach squares that have no safe actions left. These steps are given in Equation 11.

$$SafeAct_n(A) = SafeAct_{n-1} \cap \left\{ a \mid \forall B \in R_a(A) \,.\, SafeAct_{n-1}(B) \neq \emptyset \right\} \tag{11}$$

Since the set of safe actions for a square can only shrink at each step, by Tarski's fixed point theorem [7], a fixed point exists and can be reached by the above computation. This will happen, at the latest, when $SafeAct(A) = \emptyset$ for all $A \in grid$.

3.2.3 Computing Reachability for the Random Walk

Given an initial square A, the upper and lower bounds reachable by random walk from somewhere inside A, can be deduced from Equation 1. This allows one to easily compute the reachability set $R_a(A)$ described in the previous section. If $A = ([x_\ell; x_u) \times [t_\ell; t_u))$, the upper bound for the value x which can be reached by an action a from somewhere within A is $x_u + \delta(a) + \epsilon$. Conversely, the lowest possible value must be $x_\ell + \delta(a) - \epsilon$. The same applies to t, resulting in a total reachable area described in Equation 12.

$$[x_\ell + \delta(a) - \epsilon; x_u + \delta(a) + \epsilon) \times [t_\ell + \tau(a) - \epsilon; t_u + \tau(a) + \epsilon) \tag{12}$$

However, these intervals will almost certainly not match up with the grid. To obtain the squares that fully cover this, one must find the least upper bound divisible by G, as well as the greatest lower bound.

$$\begin{aligned} R_a([x_\ell; x_u) \times [t_\ell; t_u)\ ) = \Bigg\{ & [i \cdot G; i \cdot G + G) \times [j \cdot G; j \cdot G + G) \ \mid \\ & i \in \mathbb{N} \cap \left[\left\lfloor \frac{x_\ell + \delta(a) - \epsilon}{G} \right\rfloor ; \left\lceil \frac{x_u + \delta(a) + \epsilon}{G} \right\rceil \right] \\ & j \in \mathbb{N} \cap \left[\left\lfloor \frac{t_\ell + \tau(a) - \epsilon}{G} \right\rfloor ; \left\lceil \frac{t_u + \tau(a) + \epsilon}{G} \right\rceil \right] \Bigg\} \end{aligned} \tag{13}$$

3.3 Implementation

This section describes the experimental setup used to examine the effects of shielding on the Random Walk problem.

A grid describing safe actions of the system is generated using code written using Pluto Notebooks [14]. Then, the grid is made available through a C library and loaded into Uppaal Stratego (see Section 1.3) which is used to simulate the Random Walk, and run machine learning experiments.

Pluto Notebooks are written in the programming language Julia [2], a dynamically typed, just-in-time compiled language. The notebooks are reactive and without hidden state, which provides a powerful and reproducible development environment for writing and testing code, where changes take effect immediately and the output is determined only by the current state of the code.

3.3.1 Synthesising a Shield

An implementation of the Random Walk problem is developed in a Pluto Notebook, and this is used as the basis for developing a grid that covers the state space,

and computing – for each square in the grid – the set of safe actions, to obtain a grid $SafeAct_{\mathcal{S}}^{G}$.

Grids are implemented as a matrix, wherein each element represents a square. Indexing starts at one, and the matrix element at (m, n) represents the square $([(m-1)\cdot G; m\cdot G) \times [(n-1)\cdot G; n\cdot G))$. The actions in each square of the grid are implemented as a single integer value stored in the matrix. The value one is used for $\{slow, fast\}$ two for $\{fast\}$ and three for $\emptyset$. There is no value assigned to $\{slow\}$, since by the semantics of the game, the action *slow* always carries an equal or greater risk of security violation than *fast*. Thus, there cannot be a case where going fast could lead to a safety violation, but going slow could not.

A function to draw the grid using Plots.jl [3] was created, using the colors white, blue and red for each square, to represent the corresponding sets of safe actions. A granularity of $G = 0.02$ was chosen, and the grid was initialized so that all actions are permitted in the area where $t < 1$, that is, squares which are not in direct violation of the safety property. This grid is drawn in Figure 7.

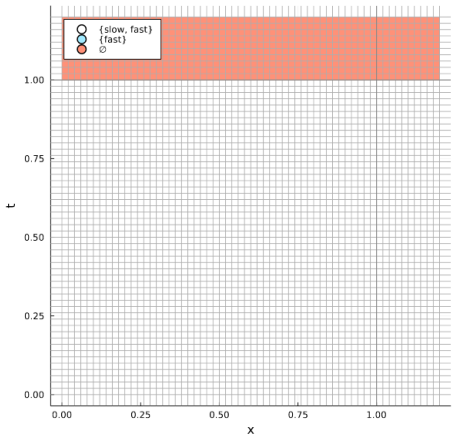

Figure 7: The initial configuration of the grid. The color of each square corresponds to a set of safe actions.

The borders of the grid don't stop at one, even though the game ends after this value is passed. Instead, it goes out to 1.2, so that no matter how much the player overshoots, it is still within the area covered by the grid. This means, that the result will be the grid $SafeAct^{0.02}_{[0;1.2]\times[0;1.2]}$.

As described in Section 3.2, the set of safe actions for each square is computed as a fixed point, described in Equation 11. Figure 8 shows pseudocode describing the function used to determine the updated set of safe actions for a single square.

Starting at line two of the figure, if the set of safe actions for a given square is already empty, it cannot shrink any further, and a number representing the empty set is returned.

```
1  function get_new_value(square, grid)
2        if get_value(square, grid) = ∅
3              return 3 /* ∅ */
4        if lower_bound_x(square) > 1 and lower_bound_t(square) < 1
5              return 1 /* {slow, fast} */
6        allow_slow := true
7        for square' in R_slow(square)
8              if get_value(square', grid) = ∅
9                    allow_slow := false
10       if allow_slow
11             return 1 /* {slow, fast} */
12       allow_fast := true
13       for square' in R_fast(square)
14             if get_value(square', grid) = ∅
15                   allow_fast := false
16       if allow_fast
17             return 2 /* {fast} */
18       else
19             return 3 /* ∅ */
```

Figure 8: The function used to decide the set of safe actions one step into the future. Since the value will be saved directly to the matrix representing a grid, a number is returned corresponding to a set in $\mathcal{P}(Act)$.

Next, in line four, the lower bounds of the square are used to determine if it describes part of the goal area. If this is the case, a full set of permissible actions is always returned, since the game at this point has ended.

From line six, a check is made for whether the *slow* action is safe. The function `R_slow` gives the set of reachable squares from the given action, and is implemented as described in Section 3.2.3. If all squares reachable by going *slow* have at least one safe action, the value corresponding to $\{slow, fast\}$ is returned. This is because by the mechanics of the game, if *slow* is a safe action, *fast* will be as well. Line twelve repeats this check for *fast*, which determines the final result.

The function `get_new_value` is used to update every square in turn, until the result stabilises. The routine for carrying out these updates is given as pseudocode in Figure 9.

Each iteration, an empty grid is used to store the new values, so that the updated value of a square does not interfere with the value of the next. In line six, a new value is calculated based on the grid from the previous iteration, and in line seven this value is written to the same square in the new grid.

When an iteration is done, a check is made to see if the result was identical to the previous iteration, which is the termination condition. And lastly, the old grid is discarded in favour of the updated values, and a new iteration can begin if

```
function make_shield(grid)
    done := false
    while not done
        grid' := new_empty_grid()
        for square in grid
            new_actions := get_new_value(square, grid)
            set_value(square, grid', new_actions)
        done := grid' = grid
        grid := grid'
    return grid
```

Figure 9: Code for computing the fixed point of safe actions, making use of the function defined in Figure 8

necessary.

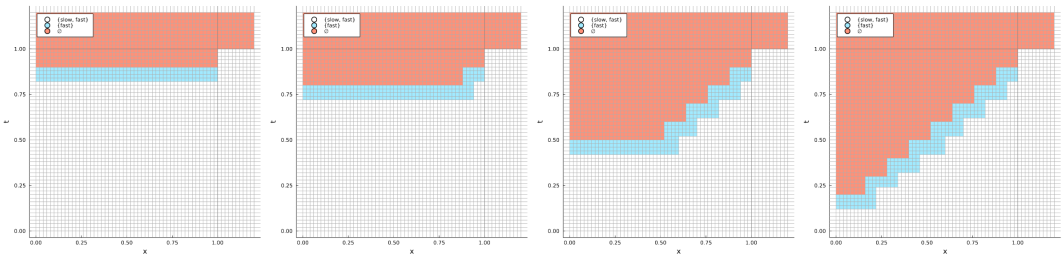

Figure 10: The grid at different points in the iteration

The function `make_shield` is run with the initial grid drawn in Figure 7 as its argument. The value after select iterations n are shown in Figure 10. A sort of staircase is formed, adding one level each iteration. This pattern continues until it stabilises after 10 iterations, which is drawn in Figure 11.

It is worth highlighting that in the starting position, $x = 0, t = 0$, any action is considered safe. If there had been no safe action from the starting position, as has been observed at higher granularity, the usefulness of the shield becomes limited.

The effect of the granularity is examined in Figure 12. It shows granularities of 0.1, 0.05, 0.02 and 0.005, along with the corresponding number of squares that make up the grid. For example, a granularity of $G = 0.1$ yields $1.2 \cdot 1.2 \cdot G = 144$ squares in total.

It can be seen that finer granularities create a significantly more permissive shield. Notice that it is first at the $G = 0.02$ that the initial state has any permissible actions. Further increases of the area where all actions are permitted can be achieved, at the cost of higher computation times and memory usage.

To make use of the result generated in Figure 11, the values are made available

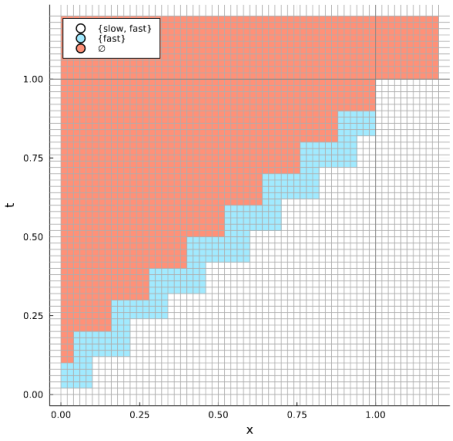

Figure 11: The set of safe actions for the Random Walk problem computed with a granularity of $G = 0.02$.

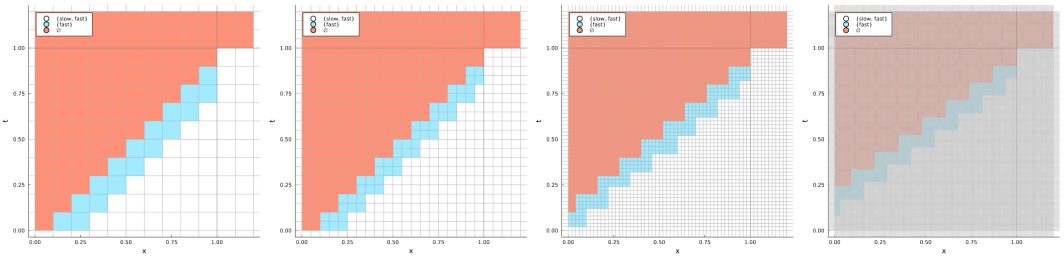

Figure 12: Comparison of different granularities.

through a library written in C. This library can be read by Uppaal Stratego, as described in the next section. The grid is simply exported as a very long string of characters, with corresponding functions to read from the grid at the appropriate places. Each character symbolises a color of the grid, which in turn corresponds to a subset of *Act*. The most important function made available through the library is `get_value`, which performs a look-up in the exported grid at the square containing the function's input values x and t.

3.3.2 Simulation in Uppaal Stratego

A model of Random Walk was created in Uppaal Stratego (see Section 1.3) to make use of its suite of built-in machine learning algorithms, as well as the built-in query language. These elements makes Stratego an excellent choice for evaluating the performance of the shield when applied with and without a machine learning agent.

The variables `double x` and `double t` describe the state of the model. Two `update`-functions are used to update the values of `x` and `t`, depending on the action taken. Which action is taken is decided partly by an automaton, and partly by a function `shield`, which may alter the action, depending on whether shielding is enabled, and if so, whether the action is permitted according to the grid from Section 3.3.1.

```
1  void updateSlow() {
2    if (unlucky) {
3      x = x + delta_slow - epsilon;
4      t = t + tau_slow   + epsilon;
5    } else {
6      x = x + delta_slow - epsilon + random(2*epsilon);
7      t = t + tau_slow   - epsilon + random(2*epsilon);
8    }
9    total_cost = total_cost + cost_function(x, t, SLOW);
10 }
```

Figure 13: The `updateSlow` function, which performs an update to the variables `x` and `t`. The update is either random, or the worst-case outcome.

As previously stated, the functions `updateSlow` and `updateFast` govern the behaviour of the Random Walk. Their implementation is nearly identical, latter of which can be seen in Figure 13. In the normal case – in lines six and seven – a choice is made using the the built-in `random` function, which returns a random double-precision number from the range between zero and its argument.

The boolean constant `unlucky` can be used to simulate the worst-case outcome of every action, to see if the strategy is able to handle this. When `unlucky == true`, rather than making a random choice, lines three and four govern the outcome of an action.

```
1  int shield(int action) {
2      if (!shield_enabled || action == FAST) {
3        return action;
4    }
5      if (get_value(x, t) == BLUE) {
6        ++interventions;
7        return FAST;
8    }
9    return action;
10 }
```

Figure 14: The `shield` function, which corrects the chosen actions of the player, if it is enabled.

The model is constructed to include the option of shielding the player actions, using the grid constructed in Section 3.3.1. The grid is exported as an external C library, which makes the set of actions for each square available through the function `get_value`.

The code which governs the shield function is listed in Figure 14. It takes a proposed action as an argument, and returns the action to be carried out based on this. Using a boolean constant `shield_enabled`, it can be made to always return the proposed action. Otherwise, the action is shielded according to the current values of x and t.

If the player proposes to go fast, the action is always unaltered, since by the mechanics of the Random Walk problem, this is always the safest action. Only in the case where the player chooses to go slow, a lookup is made using the function `get_value`. If *slow* is allowed for the given square, the action is kept. Otherwise, the action is changed to $fast$, and the intervention is tallied in the global variable `interventions`.

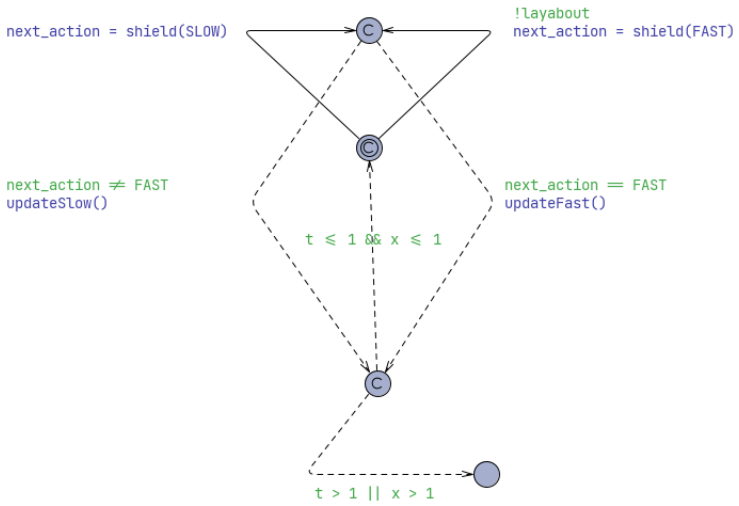

Figure 15: Automaton that simulates the Random Walk problem. The two `update` functions change the values of `t` and `x` by one step.

The update and shielding functions are called from the automaton shown in Figure 15. It was constructed to afford the player the decision of either of the two actions. The decision passes through the `shield` function, whose output decides the final outcome of the action. This can either be `updateSlow` or `updateFast`, depending on said output.

Guards on the uncontrollable (dashed) edges ensure that the selected action is carried out, and the loop continues until either t or x exceeds one. The boolean constant `layabout` acts as a guard on the *fast* player action. When this is enabled, the whole system becomes deterministic, such that only *slow* actions are passed to

the shield, forcing it to interfere as often as possible. This means that decisions to go fast are only made inside the shield itself, which provides a baseline for the performance of other shielded policies.

The final result is a UPPAAL STRATEGO model which can accommodate a diverse range of experimental setups, which can be evaluated using the built-in query language and its machine learning capabilities.

3.4 Description of Experiment

The effects of applying the shield *SafeAct* on the Random Walk problem was explored by running several experimental setups with different parameters. The different ways of applying shielding described in Section 1.1 were examined, as well as an unshielded version for comparison, and a shielded layabout strategy. Additionally, variations were applied in terms of training times, and the penalty applied to the unshielded agent for violating the safety property.

3.4.1 Experimental Setups

Each setup alternates the number of runs used for training, the penalty for a safety violation d, and the configuration, which will be described in the following.

The time it takes for an agent to converge on a near-optimal strategy will vary from model to model, and by adjusting the number of runs, it can be seen if shielding has an impact on the speed of learning. Setups were made using 1500, 3000, 6000 and 12000 training runs.

In the cases where a shield is not in place, the agent should still learn to avoid safety violations, though avoiding them completely may not be a guarantee. To achieve this, a penalty d was applied to the overall cost when a violation of the safety property occurs. That is, if the run finishes with $t > 1$. It is expected that a higher penalty would lead to the agent prioritizing the avoidance of safety violations, but it is possible that a value which is too high could have a negative impact on the learning. Setups were varied to have either $d = 10$, $d = 100$ or $d = 1000$.

Lastly, a *configuration* is composed of either a learning agent, a shield, or some combination of the two. Each setup uses one of the following four configurations:

Shielded Layabout The shield is applied to a basic „agent", which simply takes the most unsafe action no matter the input. For the Random Walk, this simply entails always going slow. This creates a strategy fully dictated by the shield.

No Shield An agent is trained with no shield applied, receiving a penalty d on runs that violate the safety property.

Post-shielded A shield is applied to the same learning agents, that were trained and evaluated in the No Shield model. This means that the agents have been trained without a shield, but are subsequently being shielded during the evaluation phase.

Pre-shielded The Q-learning agents were trained with the shield in place. If the shield intervenes, it apppears to the agent that it's suggested action has the outcome of the shielded action.

The boolean constants of the model described in Section 3.3.2 – as well as functionality for saving and learned strategies – can be used to create these experimental setups. To model the Shielded Layabout, the values `shield_enabled` and `layabout` were set to `true`. In all cases, `unlucky` will be false, but it is useful as a method for discovering safety violations in a strategy after it has been learned.

For No Shield, the parameters will be `shield_enabled, layabout = false`, and the learned strategy will not only be evaluated, but it will also be saved for the next experiment.

The Post-shielded configuration loads in the strategy that was previously trained, and runs it again but with the parameter `shield_enabled = true`. Lastly, the Pre-shielded strategy is trained with this parameter already set.

Varying the number of runs or the penalty d might not be applicable for all configurations. Since the Shielded Layabout is deterministic, no learning occurs, so the number of runs is not applicable. Likewise, the Shielded Layabout and the Pre-shielded configurations will not be affected by the value of d, since the shield prevents this penalty from ever being applied.

3.4.2 Queries Used

For the learning algorithm, the built-in Q-learning of Uppaal Stratego is used, and to force a certain number of training runs, the following flags were set while calling the verifier from the command-line:

```
1   --max-iterations 1 --good-runs 3000 --total-runs 3000 --runs-pr-state 3000
```

Here shown for a 3000-run setup. All other learning parameters were kept at their default value for `Uppaal 4.1.20-stratego-10 Beta-2`.

The Q-learning agent is invoked using Uppaal's powerful query language – with or without `shield_enabled = true` – by running the following query:

```
strategy d1000 = minE (total_cost+(t>1)*1000) [#<=30]{}->{x, t}: <> x>1 or t>=1
```

Here shown for $d = 1000$. Note that boolean expressions are treated as having the value one, if they are true, and zero otherwise. The expression `[#<=30]` indicates that the strategy should train over a time horizon of 30 steps, more than enough to finish a run.

A trained strategy can subsequently be applied to other queries. The following query to computes the average cost of a run for the previous strategy. The expression `[#<=30;1000]` indicates that the query should be evaluated on 1000 runs of at most 30 steps.

```
E[#<=30;1000](total_cost) under d1000                29.2742 +/- 0.15377 (95% CI)
```

This is the number of evaluation runs that were used for all setups. The query does not include the penalty d, since this is only introduced to prevent safety violations. Instead, the number of times a safety violation occurs is computed and reported separately. The following query computes the average number of safety violations, which for each run will be between zero and one for the random walk. Since safety violations are a potentially rare occurrence, the number of runs the query is evaluated over has been increased.

```
E[#<=30;100000] (max:t>1) under d1000                                    ~=0
```

Since at most one safety violation can occur per run, this number becomes a measure for the fraction of runs where a safety violation occurred. In this case it is reported as ≈ 0, indicating that no safety violations were observed. As Figure 14 counts the number of interventions by the shield, this metric was computed for the Post-shielded setups. This gives a metric for how much of the original policy is preserved when the shield is applied, for the average of 8.6 actions that are taken to reach the end of a run. Because the policy was not trained with the shield in place, a high amount of interference could explain a potential drop in performance.

```
E[#<=30;1000](interventions) under d100          0.824 +/-  0.0302265 (95% CI)
```

The query displays the result for a Post-shielded setup, that was trained for 1500 runs with $d = 100$. An average of 0.824 interventions occur per run, out of about 8.6 actions taken in total.

3.4.3 Validating the Model

Since a model of Random Walk was also made in a Pluto Notebook – the one used to synthesize the grid *SafeAct* – it was used to validate the Uppaal Stratego model. Since the shielding function was available in both models, the average score was computed for `shield_enabled, layabout = true`.

The Uppaal Stratego query is given below. Since the player is forced to always pick the *slow* action, no strategy is applied.

```
E[#<=30;100](total_cost)                             27.3077 +/- 0.792499 (95% CI)
```

Below is the result from the Pluto Notebook. A function `evaluate` is supplied the cost function, the cost of losing, the limits to x and t, a tuple `mechanics`, and lastly the policy function. The tuple contains the values for τ, δ and ϵ, while the policy is defined as `shielded_layabout(x, t) = shield_action(:slow)`.

```
evaluate(cost_function, 1000, 1.0, 1.0, mechanics..., shielded_layabout)
# result: 28.919
```

Although the two results are not completely identical, they are similar enough to indicate, that the behaviour of the two models is the same, and that the shield has been exported and applied correctly to the UPPAAL model, from the Pluto notebook.

3.5 Experimental Results

Due to concerns of variance in training outcomes, the experiment was run 10 times, and the results are reported as the median values of each of these 10 experiments. Standard deviations for the results are given in Appendix A Table 3. The median values are given in Table 1 for all setups, which are the 29 possible combinations of configurations, values of d, and the number of runs. Figures 16, 17 and 18 show plots made from said table.

Notably, the fraction of runs lost is zero for all shielded configurations, as expected. Only in the setups that have the No Shield configuration were runs observed where $t > 1$. Figure 16 shows a bar chart of these values. When the penalty is only at $d = 10$, the safety violations are as high as nine percent, for an agent that has only trained for 1500 runs. Safety violations are observed for unshielded agents with $d = 10$ and $d = 100$. They were not observed to occur for $d = 1000$, but that is not to say a safety violation could never occur for these setups.

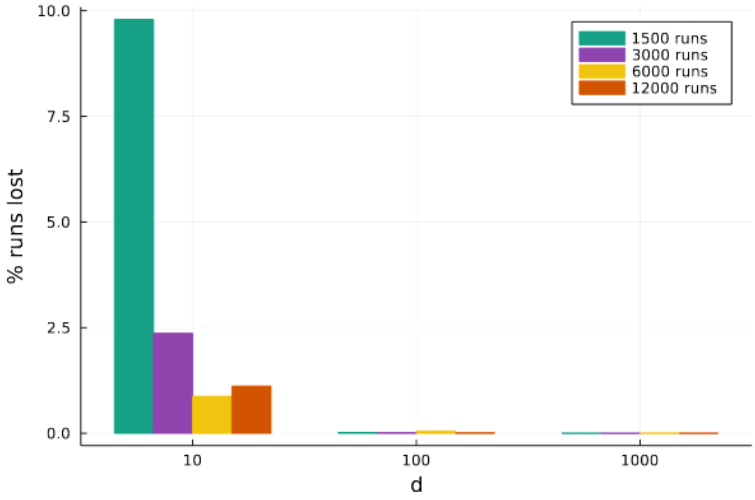

Figure 16: Bar plot that shows fraction of runs lost for the No Shield configuration of the Random Walk problem. Bars are grouped by the penalty d applied for losing, and color-coded by number of training runs.

The average cost of a run is shown in Figure 17 for all setups. The average

#	Configuration	d	Runs	Avg. swings	Avg. deaths	% interventions
1	Shielded Layabout	-	-	69.2190	0.0000	5.7698
2	Pre-shielded	-	1500	38.2755	0.0000	-
3	Pre-shielded	-	3000	37.5740	0.0000	-
4	Pre-shielded	-	6000	37.1590	0.0000	-
5	Pre-shielded	-	12000	36.9270	0.0000	-
6	No Shield	10	1500	0.0295	5.0575	-
7	No Shield	10	3000	0.0270	5.0530	-
8	No Shield	10	6000	0.0000	5.0560	-
9	No Shield	10	12000	0.0000	5.0565	-
10	No Shield	100	1500	42.1865	0.0700	-
11	No Shield	100	3000	41.7590	0.0770	-
12	No Shield	100	6000	40.4740	0.0590	-
13	No Shield	100	12000	40.4510	0.0540	-
14	No Shield	1000	1500	41.0775	0.0110	-
15	No Shield	1000	3000	41.6090	0.0120	-
16	No Shield	1000	6000	40.0670	0.0100	-
17	No Shield	1000	12000	39.5025	0.0060	-
18	Post-shielded	10	1500	69.2260	0.0000	5.7697
19	Post-shielded	10	3000	69.2090	0.0000	5.7683
20	Post-shielded	10	6000	69.2440	0.0000	5.7684
21	Post-shielded	10	12000	69.2460	0.0000	5.7669
22	Post-shielded	100	1500	66.0740	0.0000	3.5966
23	Post-shielded	100	3000	65.6865	0.0000	3.9916
24	Post-shielded	100	6000	66.1435	0.0000	3.9186
25	Post-shielded	100	12000	66.8410	0.0000	4.1085
26	Post-shielded	1000	1500	61.5450	0.0000	2.8524
27	Post-shielded	1000	3000	64.3535	0.0000	3.3076
28	Post-shielded	1000	6000	63.9235	0.0000	3.4481
29	Post-shielded	1000	12000	63.7120	0.0000	3.4162

Table 1: Average cost, fraction of runs lost, and average interventions for different setups of the Random Walk problem. All values are the median result of running the experiment 10 times.

costs are shown as a function of the number of training runs used in the setup, for different combinations of configurations and values of d.

An exception is Shielded Layabout, which does not have a trained component. Instead, its performance is drawn into the graph as a dotted line, and it can be seen that it is the setup which achieves the next-worst performance. Using the Shielded Layabout as a benchmark, it can be seen that the shield is permissive enough to leave a good amount of room to improve performance. The Pre-shielded

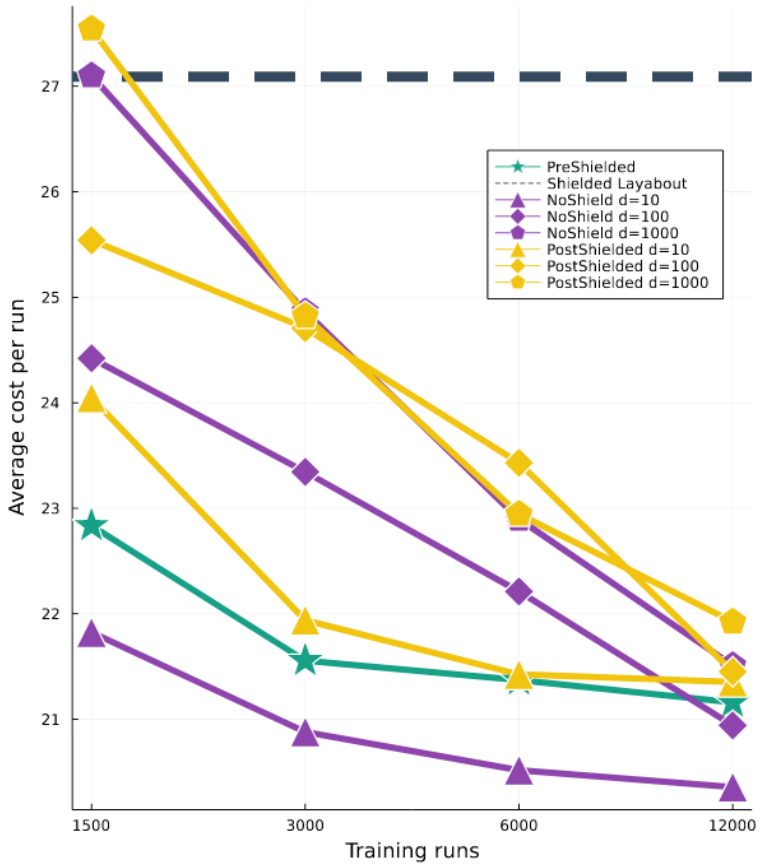

Figure 17: Average cost for the Random walk problem, as a function of training time. Markers indicate configuration, and penalty d when applicable. The average cost of the Shielded Layabout is drawn as a dotted line, since it does not make use of training runs.

configuration initially outperforms the No Shield configuration with $d = 1000$, but catches up when they both use 12000 training runs. With $d = 10$, the unshielded agent performs similarly to the Pre-shielded one, with either the same average performance, or slightly above. The unshielded agent that was penalised only 10 points for losing consistently outperforms the Pre-shielded agent, though this comes with the most significant risk of safety violations, across all setups.

The reason it outperforms the pre-shielded agent might not just be due to the fact it can accept a small risk of safety violation, however. As Figure 12 in Section 3.3.1 shows, a granularity of $G = 0.005$ creates a more permissive shield than the $G = 0.02$ that was used for the experiments. Therefore it is possible that a more efficient Pre-shielded strategy exists, than the one that can be achieved in this experiment.

Looking at the post-shielded configurations, they consistently fare worse than

the unshielded strategies they were based on, but do retain some of their performance. In fact, the Post-shielded configuration with a penalty $d = 10$ consistently outperforms the other No Shield configurations.

The amount of times the shield interferes in the Post-shielded configurations are shown as a bar chart in Figure 18. For comparison, an average of 8.6 are taken before the goal is reached, though this number varies with the outcome of actions. A general trend can be seen where higher training times – and higher penalties for losing – corresponds to fewer unsafe actions taken by the agent. For $d = 100$ and $d = 1000$, the shield interferes a little less than once per run, and for $d = 10$ a little more.

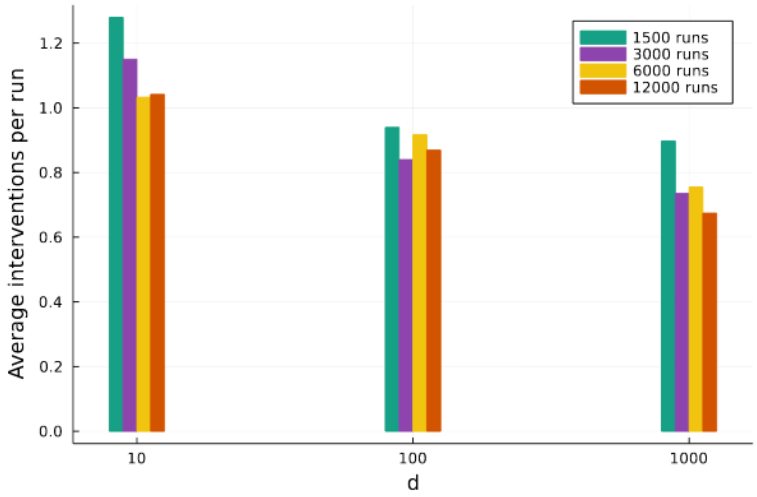

Figure 18: Bar plot that shows average number of interventions for the post-shielded configuration of the Random Walk problem. Bars are grouped by the penalty d that was applied for losing during training, and color-coded by number of training runs. In comparison, the shielded layabout has an average of 2.8835 interventions per run.

It is notable, that harsher penalties for safety violations generally leads to higher training times to reach the same performance. On the other hand, they do correspond to a reduced risk of safety violation. This trade-off can of course be mitigated if a shield is available. Post-shielding an agent had a performance penalty, but post-shielded agents that had been penalised less harshly, were comparable to unshielded ones who had a higher penalty applied. It also provided a guarantee that safety violations would be avoided alltogether, which was not achieved even for training with $d = 1000$.

To achieve even better performance while maintaining safety guarantees, pre-shielding can be employed. Training the agent together with the shield, allows it to learn a safe and efficient strategy. However, the best-performing setups were the ones with the No Shield configuration and $d = 10$, outperforming the Pre-shielded

setups after more than 1500 runs. This performance did not come without an increased risk of safety violations, however. It is possible that a shield generated using a finer granularity, could produce a pre-shielded agent that either matches or outperforms the No Shield configuration. Thus, repeating the experiment after the shield of finer granularity was generated is an obvious avenue of potential further study.

4 Bouncing Ball

The second case is inspired by the titular problem of the paper *Teaching* STRATEGO *to Play Ball* [10]. It is an infinite time horizon game, where a ball is bouncing on a flat surface, and loses a random amount of velocity on each impact. To keep the ball bouncing, a hammer can swing from above, which adds some energy to the ball, if it hits.

4.1 Description

The state of the ball is at any time given by its position p (along the vertical axis) as well as its velocity v. This is illustrated in Figure 19.

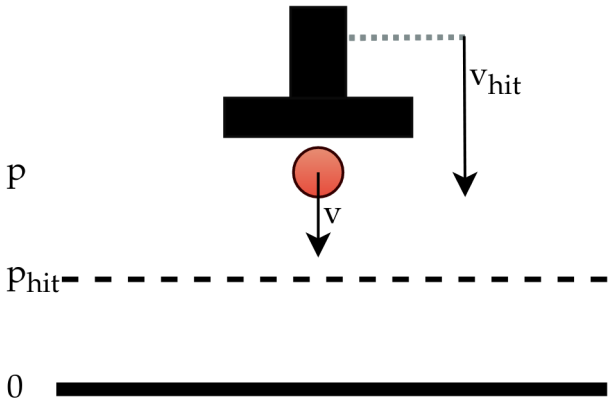

Figure 19: Illustration of the bouncing ball example [10].

Upwards will be considered the positive direction, and downwards negative. The word *velocity* will be used to refer to a signed value, while speed shall refer to an absolute value.

The dynamics of the ball in free fall are described by the set of differential equations in Equation 14, while the solution to them are shown in Equation 15.

$$\begin{aligned} v'(t) &= g \\ p'(t) &= v \end{aligned} \tag{14}$$

$$\begin{aligned} v(t) &= gt + v_0 \\ p(t) &= v(t) \cdot t + p_0 = \frac{1}{2}gt^2 + v_0 t + p_0 \end{aligned} \tag{15}$$

The surface is defined to be at position $p = 0$, and when the ball reaches this position, a bounce occurs as an atomic event. When a bounce happens, the sign of v is flipped, and a fraction of its value is preserved, such that the new value of v is given by the uniform random distribution in Equation 16.

$$[-v \cdot (\beta_1 - \epsilon_1); -v \cdot (\beta_1 + \epsilon_1)] \tag{16}$$

At a regular time interval t_{hit}, the player gets the option to bring down the hammer to try and hit the ball. A hit will only happen if two things are satisfied. Firstly, the ball's position p must be at some minimum height p_{hit}. Second, the ball should not be moving towards the ground at a speed greater than that of the hammer. That is, for some hammer velocity v_{hit} it should be the case that $v \geq v_{hit}$. Note the difference between speed and velocity: The hammer strikes down towards the ground, thus v_{hit} is negative, and the comparison says that v must be greater than this value.

If the ball is not hit, it simply follows its normal freefall trajectory. However, an impact of the hammer will correspond to the atomic action, where v follows the (potentially) random uniform distribution given in Equation 17.

If the ball is already on a downards arc, (that is, $v \leq 0$), then the ball is simply accelerated the same speed as the hammer. If the ball is still on an upwards trajectory, it receives the full speed as the hammer, as well as some velocity that is preserved as it bounces into the hammer.

$$\begin{cases} \min(v, v_{hit}), & \text{if } v < 0 \\ [v_{hit} - v \cdot (\beta_2 - \epsilon_2); v_{hit} - v \cdot (\beta_2 + \epsilon_2)] & \text{otherwise} \end{cases} \tag{17}$$

Informally, the goal will be to keep the ball bouncing. This is specified as the safety invariant, that whenever the ball impacts with the ground, the speed of the ball must be 1m/s or greater. This corresponds to the LTL formula given in Equation 18.

$$A\Box\ p = 0 \rightarrow |v| > 1 \tag{18}$$

Only one version of the bouncing ball problem will be used, and it is dubbed BB for short. The concrete values of the constants described above, are given in Equation 19.

$$t_{hit} = 0.1, \quad p_{hit} = 4, \quad v_{hit} = -4, \quad \beta_1 = 0.91, \quad \epsilon_1 = 0.06, \quad \beta_2 = 0.95, \quad \epsilon_2 = 0.5 \quad (19)$$

An example using these values is shown in Figure 20. To the left is shown the ball's position as a function of time, describing the distinctive arcs of a ball bouncing. Some irregularity is present due to the variance caused by ϵ_1. To the right is shown the position as a function of velocity instead, and here it can be seen how the ball's velocity switches instantly from a negative to a positive value on impact with the ground.

The ball is simply allowed to bounce without being hit, and when the safety property is violated, the ball comes to an immediate stop. In theory, a perfect simulation of the ball should come to a stop in finite time, but doing an infinite amount of bounces in the process [8]. However, since infinitely small arcs might cause issues with floating point precision, it is useful to cut the simulation off at some point.

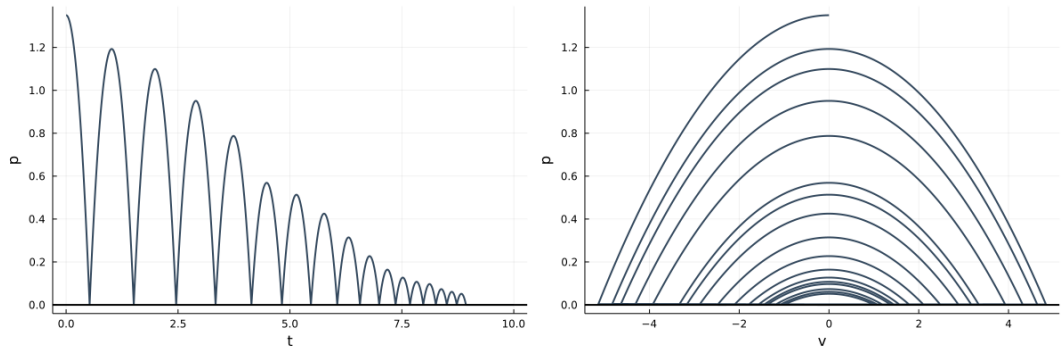

Figure 20: Two graphs showing the same trace of a ball bouncing. The left shows the ball's position as a function of time, while the right shows the position as a function of its velocity. When the ball hits the ground with less than 1m/s, it comes to a stop.

The flat line at the end of the graph shows when the safety property was violated. Note, that the minimum height to hit the ball $p_{hit} = 4$ is significantly greater than the heights that were reached just before the safety violation occurred. This means that long before a safety violation occurs, there is nothing the player can do to transfer additional energy to the ball, and therefore it is inevitable that the ball would come to a stop eventually.

4.2 Computing Reachability for Bouncing Ball

The technique for computing the set of reachable squares $R_a(A)$ from a given initial square A and action a, is different for the Bouncing Ball problem, compared to Random Walk.

It did not prove possible to compute this transition symbolically, despite efforts to do so. Unfortunately, the equations became infeasibly complex when there was a chance of the ball hitting the ground from within a square. The set of squares reachable from an initial square, was expressed using quantifiers, and an attempt was then made to use quantifier elimination to allow computing the set of reachable squares efficiently. This was done successfully in the case where the ball was moving freely through the air, but no solution was found when accounting for the potential of the ball bouncing on the ground. When the ball could potentially hit the ground, the quantifier elimination was not doable by hand. The mathematics software package Maple [12] could not produce a viable answer either, producing intermediary results that were equations which spanned hundreds of lines.

The approach to compute reachability that is used instead, is dubbed the „barbaric" method, in reference to the late Oded Maler [6]. The method is straightforward, but lacks formal guarantees. The technique simulates the behaviour of the ball accross many individual points inside the initial square, to see where these points end up after taking the given action. The set of squares that the points end up in, is then used as an approximation of R_a.

From a single point, a square A is said to be reachable, if after taking some action a, the updated position of the ball after time t_{hit} (when it is time for the player to take another action) is within A. The non-determinism of an action's outcome is removed by only considering the worst case, since this is what is required to determine if an action is safe. Specifically, the random bounciness of the ball – on impact with either the ground or the hammer – is set to $\beta_1 - \epsilon_1$ and $\beta_2 - \epsilon_2$ respectively, such that the minimum amount of energy is assumed to be preserved in each case.

The barbaric approximation of $R_a(A)$ will be called $R_a^k(A)$. A number of points k are placed with regular spacing within A. For each of the k points, the square reachable by the given action a is computed, and the the result is a set of squares reachable by any of these points.

This is illustrated in Figure 21, visualising the approximation for both *hit* and *nohit*. The initial square is coloured blue in the grid, and contains $k = 9$ points (darker blue), that are used to calculate the set of reachable squares. The movement of the ball from the middle point is plotted for each of the two actions, ending up in a dark green point. The neighboring points are shown in green, and the squares that these points intersect with, have been highlighted in a lighter green.

The green points form a parallelogram, explained by the difference in velocity inside the initial square: Depending on where in the square the ball starts, it will

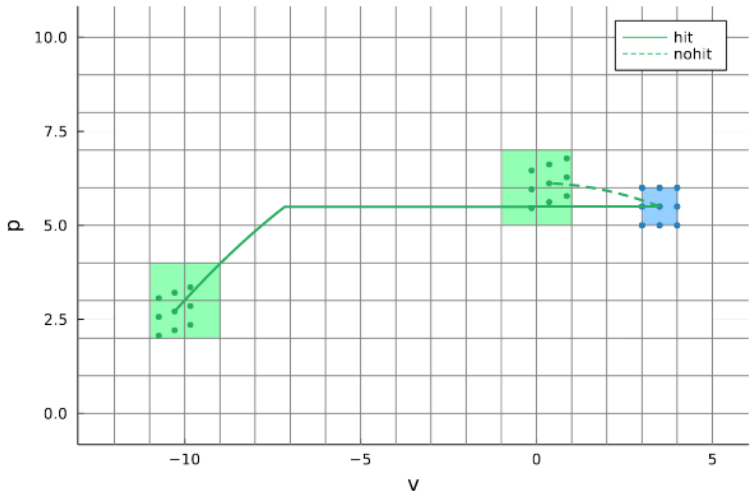

Figure 21: Illustration of the „barbaric" method, showing the computation of R^k_{hit} and R^k_{nohit}. Clouds of points are shown before (blue) and after (green) taking an action, and the trajectory of the ball has been plotted for the middle point in the cloud.

travel further during the same amount of time.

A square B is only included in of $R^k_a(A)$, if a point is discovered inside A, which, after taking action a, ends up inside of B. But since there is an uncountably infinite number of points inside of A, there might exist some additional point that wasn't checked, from which an undiscovered square is reachable. Hence, it is an under-approximation, but one that is made more accurate for higher values of k. Figure 21 shows $k = 9$, but in the following, R^{256}_a will be used. The effects of increasing k on the final result are discussed in Section 4.3.2, but first it will be described how this result is arrived at.

4.3 Implementation

The implementation of the Bouncing Ball problem follows many of the same steps, as that the Ramdom Walk described in Section 3.3. Some additional changes are made, however, to accommodate the specifics of the problem. This includes both additional implementation details in the Pluto Notebook that provides the shield, as well as making better use of Uppaal Stratego features modelling the behaviour of the ball.

4.3.1 Synthesising a shield

Since the grid will have to accommodate negative values for the Bouncing Ball problem, the data structure is generalised to support this. Like in Section 3.3.1, it is implemented as a Julia matrix, but indexing is further offset, such that given some

lower bounds of the grid, v_{min} and p_{min}, the matrix element (m, n) now represents the square $([(m-1) \cdot G + v_{min}; m \cdot G + v_{min}) \times [(n-1) \cdot G + p_{min}; n \cdot G + p_{min}))$. The lower bound of p will still be zero, but the generalisation was added there as well for good measure.

The lower and upper bounds do not have any natural definition, as was the case for the Random Walk. Continually hitting the ball might accelerate it to high speeds and great heights, but the grid is subject to memory constraints.

Fortunately, the goal of the system is to prevent the ball from coming to a stop, and for this, only relatively low speeds and heights are relevant. By defining the allowed actions outside of the grid as always being $\{hit, nohit\}$, a result can still be computed. Past a certain point, all actions should always be permissible, and so lower and upper bounds of the grid can be arrived at experimentally. The bounds simply have to be large enough that there are no squares near the side or upper edges of the grid who do not allow all actions.

Computing R_a^{256}, as described in Section 4.2, requires quite a lot of calculations per square. Therefore, the values are pre-computed, rather than being processed on demand. This has a higher memory footprint, but does not result in any redundant calculations, since the fixed-point computation described in Section 3.3.1 Figure 8 goes over every square in multiple iterations.

And since any approximation R_a^k depends on and makes heavy use of a model of the Bouncing Ball, this model will need to be implemented, preferably in a manner that is fairly efficient. A function is required to determine the outcome of an action, such that the ball's position is calculated after the time t_{hit} has passed. That is, when it is time for the player to decide another action.

A common way to do this is to update the velocity and position of the ball at tiny time increments Δt. The time $i \cdot \Delta t$ at which the ball is seen to reach the ground will be close to the actual time, when using a sufficiently small Δt, and so the impact can be resolved at that increment. Though simple, the method is an approximation, and one which requires many calculations to resolve an action.

Instead, the function to resolve the outcome of an action can be developed analytically. When the ball is simply moving freely through the air, the update to v and p after t seconds is already given in Section 4.1, Equation 15. Thus, if the ball is allowed to move uninterrupted through the air for the entire duration of t_{hit}, these equations can simply be applied.

In the case where the ball is hit with the hammer, this occurs as an atomic event right as the player takes that action. Therefore, v can simply be updated according to Equation 17, before applying Equation 15. More precisely, v should be updated using the worst-case outcome of Equation 17, which is given in Equation 20.

$$\begin{cases} \min(v, v_{hit}), & \text{if } v < 0 \\ v_{hit} - v \cdot (\beta_2 - \epsilon_2) & \text{otherwise} \end{cases} \tag{20}$$

That leaves handling the case where the ball is going to hit the ground at some point during t_{hit}. By solving for $p = 0$ in Equation 15 it is possible to calculate the next time that the ball is going to collide with the ground.

Given the second degree polynomial, $p(t) = \frac{1}{2}gt^2 + v_0t + p_0$, the discriminant becomes $v_0^2 - 4 \cdot (\frac{1}{2}gp_0) = v_0^2 - 2gp_0$. The discriminant will be positive, since g is defined to be a negative value, while the ball always stays above ground level. That is, $p_0 \geq 0$, meaning that the expression $-2gp_0$ will be a positive number. Thus, the equation has two solutions, given in Equations 21a and 21b.

$$t = \frac{-v0 + \sqrt{v_0^2 - 2gp_0}}{g} \tag{21a}$$

$$t = \frac{-v0 - \sqrt{v_0^2 - 2gp_0}}{g} \tag{21b}$$

Since the relevant time of impact should be in the future ($t > 0$) the result can be narrowed down to Equation 21b.

As stated above, the discriminant is positive, and its square root must be greater than v_0, since it consists of v_0^2 plus some positive number $-2gp_0$. Thus, the whole expression $-v0 - \sqrt{v_0^2 - 2gp_0}$ must always represent a negative number, and since it is divided by another negative number g, the solution as a whole will be positive.

Knowing the time of impact t_{impact}, if this value is less than t_{hit}, the velocity of the ball at impact is given as $v = v_0 + gt_{impact}$. The position will of course be $p = 0$, and from here, the position of the ball after the remaining time $t_{hit} - t_{impact}$ can be calculated. This is done as a recursive call to the same function, allowing for multiple bounces during t_{hit}, if applicable.

The soundness of this method can be tested by computing the behaviour of the ball using different values for t_{hit}. Removing the random element from the ball's bounce, the model of the ball should always follow the same arc, regardless of the frequency of t_{hit}.

And indeed this is the case as can be seen in Figure 22. The outcomes of repeatedly taking the action *nohit* are plotted using variants of the Bouncing Ball with different values of t_{hit}. With extremely frequent sampling at $t_{hit} = 0.001$, the graph appears as an unbroken line, forming the distinct shape of a ball bouncing. At greater intervals between actions (and therefore between samples), the ball's trajectory becomes harder to make out, but the points are always perfectly consistent with other values of t_{hit}.

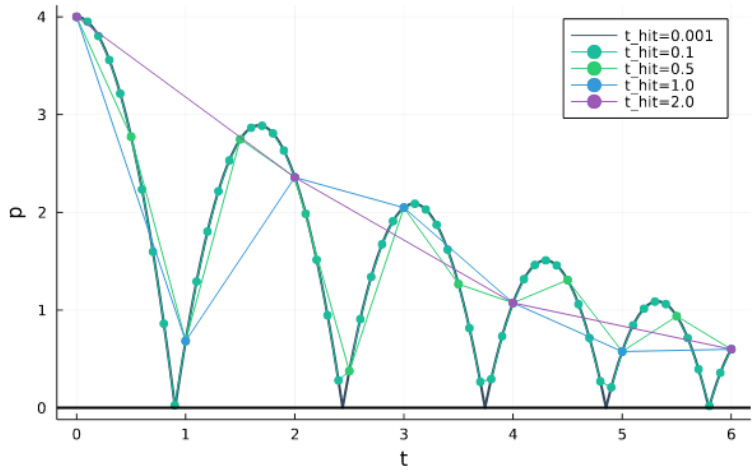

Figure 22: Continually taking the *nohit* action using different values of t_{hit}. As expected, all the different values describe the same movement of the ball, albeit with different sampling frequencies.

Now the approximations of R^{256}_{hit} and R^{256}_{nohit} can be pre-computed with relative efficiency. Next, the code in Figures 8 and 9 in Section 3.3.1 can be used again to arrive at a fixed point for a grid, that describes a set of safe actions for the Bouncing Ball. This requires specifying the initial set of actions for the grid based on the safety invariant, as well as a function to compute – for each iteration – a square's updated set of actions.

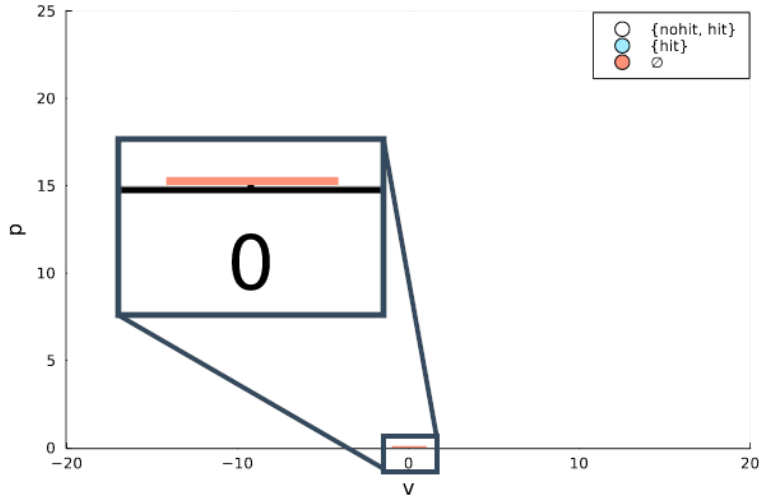

Figure 23: The initial value of the grid.

The initial values of the grid should be given by the safety invariant $A\Box\ p = 0 \rightarrow |v| > 1$ specified in Section 4.1. Either the lower or the upper bound for the velocity could represent the lower bound for the speed, which gives the actions for

the squares described in Equation 22.

$$\text{actions for square } [v_\ell; v_u) \times [p_\ell; p_u) = \begin{cases} \emptyset, & \text{if } \; p_\ell = 0 \wedge (|v_\ell| < 1 \vee |v_u| < 1) \\ \{hit, nohit\} & \text{otherwise} \end{cases} \tag{22}$$

The grid will be defined with a granularity of $G = 0.02$, and the initial values are drawn in Figure 23. At such a fine granularity, magnification is required to make out the squares which must be avoided, but – as it will be shown – the area from which these squares might be reached in one or more steps, is much larger.

The code for updating the set of actions in the grid (to be used in the function `make_shield` that was given in Figure 9) is given in Figure 24.

```
1  function get_new_value(R_hit, R_nohit, square, grid)
2        value = get_value(square, grid)
3        if value = ∅
4          return 2 /* ∅ */
5        nohit_bad = false
6        for square' in R_nohit[square]
7              if get_value(square', grid) = ∅
8                    nohit_bad = true
9        if nohit_bad
10             for square' in R_hit[square]
11                   if get_value(square', grid) = ∅
12                         return 2 /* ∅ */
13             return 1 /* {hit} */
14       else
15             return 0 /* {hit, nohit}
```

Figure 24: The function used to decide the set of safe actions one step into the future. Since the value will be saved directly to the matrix representing a grid, a number is returned corresponding to a set in $\mathcal{P}(Act)$.

Lines five to eight will check if any square without safe actions is reachable by the action *nohit*. If all reachable squares have a safe action in the grid, then the resulting set is simply $\{hit, nohit\}$. Otherwise, the same check is made in lines 10 to 13, for the action *hit* instead. If any square without safe actions are still reachable, the resulting set is $\emptyset$. Otherwise, the set of actions in the square becomes $\{hit\}$.

This is the extent of what is required to synthesize *SafeAct* for the Bouncing Ball. The final fixed point for the grid using a granularity of $G = 0.02$ and a barbaric approximation of reachability R_a^{256} – for the constants given in Section 4.1 Equation 19 – is shown in Figure 25. The grid is drawn with position along one axis, and velocity along the other, the same axes used in the right-hand Figure 20 in Section 4.1.

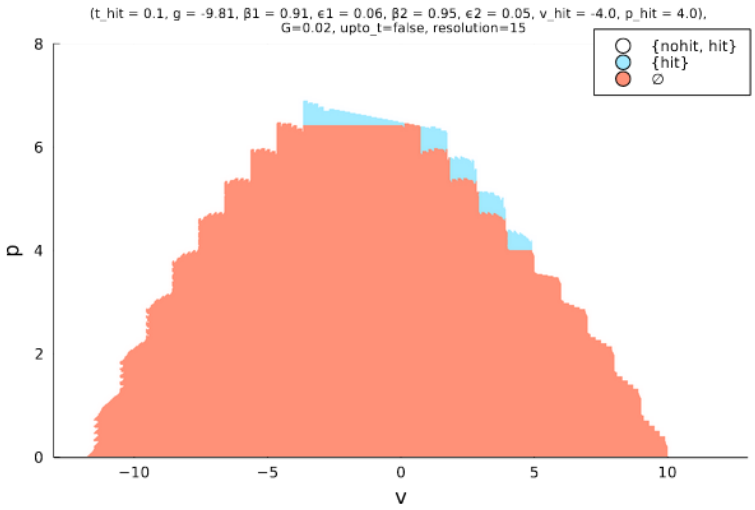

Figure 25: The fixed point of the BB shield.

The figure is not symmetrical around the zero velocity mark, and this can be explained by the fact that hitting the ball on a downwards trajectory has less of an effect than when it is on an upwards trajectory. When it is on an upwards trajectory, the Bouncing Ball preserves a semi-random fraction of its velocity, and gains an additional speed of $v_{hit} = -4$m/s. When on a downards arc however, the ball is simply brought up to the speed of v_{hit}, if it is not moving faster already. This means that hitting the ball makes less of a difference as its velocity approaches -4m/s , but the grid will still mandate it in some cases, up until the point where it makes no difference.

Past this point, where hitting the ball does not help, squares instead have no safe actions, $\emptyset$, if there is a possibility of the ball reaching other unsafe squares from it. This can also be seen with regards to the position of the ball, as the squares which only allow the action $\{hit\}$ are all above the height $p_{hit} = 4$m.

The $\{hit\}$ and $\emptyset$ areas together form a staircase pattern, where the size of the steps match the distance that the ball can travel during the span of t_{hit}. Since the intermediary postions between two actions are not considered in the definition of *SafeAct*, it is possible for the ball to pass through an unsafe square, as long as it does not end up in one. Evidently, it is possible for the ball to pass through squares without any safe actions, while still maintaining the safety property indefinetely, as long as it does not end up in one of them. It is possible the staircase pattern emerges due to the timing of the player actions. At which part of its arc the ball gets hit will have an impact on the energy transferred to it, and the player is only given the option to swing at a ball at a certain interval given by t_{hit}. It may be that for some part of the state space, the player will not get any good opportunities to hit the ball, during the window where this is possible.

4.3.2 Validation of the Barbaric Method

The difference to the final outcomes was examined using 256 points to compute the set of reachable squares, versus 16 points. In total, out of a grid of 520000 squares, there were 88 discrepancies between the two results.

Compared to the granularity used, the differences can be difficult to spot with the naked eye. Figure 26 shows the locations of the 88 differences, as well as a zoom-in showing part of the grids where they differ.

Dotted circles mark places where the actions of one or two squares differ, but where this cannot be seen on the drawn grid. The other circles show small but perceptible discrepancies. Obvious or not, it can be seen that the shield becomes more restrictive as the number of points k, that are used for the approximation, is increased. One might worry that this leaves „gaps" in the approximations of *SafeAct*, where the resulting shield would allow potential safety violations in specific instances.

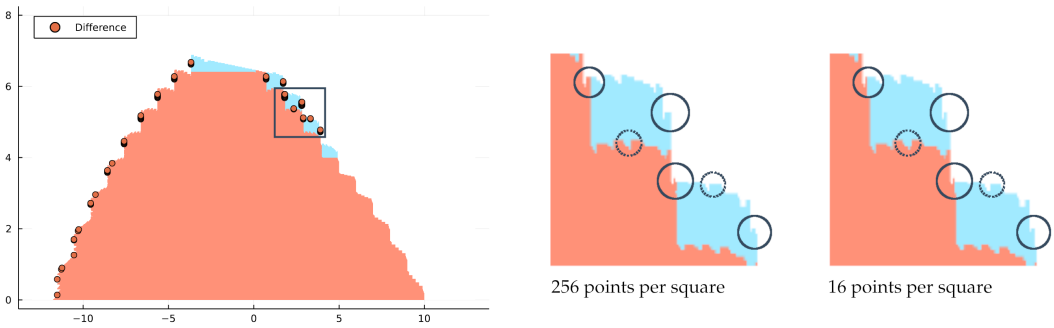

Figure 26: Comparison of the final results, between calculating the set of reachable squares using 256 and 16 points from the initial square. (left) All differences highlighted. (right) Zoom-in of the marked area, with dotted circles showing imperceptible differences, and solid circles showing visible differences.

However, it is likely the case that these so-called gaps are not significant for the $k = 256$ value used in Figure 25. This assumption is especially supported by the fact, that the partitioning into squares is itself an over-approximation of possible traces, and so might hopefully correct for a slight under-approximation of reachability.

4.3.3 Simulation in Uppaal Stratego

A model of the Bouncing Ball was created using UPPAAL STRATEGO to support the same experimental setups that were employed for Random Walk. The model is based directly on the original STRATEGO model of a bouncing ball used in [10].

In the exact same manner as for the Random Walk, the grid from Section 4.3.1 is imported into UPPAAL through an external C library. This is used to create a

function `shield`, that is identical to the one in Figure 14 in Section 3.3.2. It takes a proposed action as its argument, and uses the state of the ball to determine if that action is safe. If it is, the same action will be returned. Otherwise, a safe action is returned. Provided that the ball starts in a safe square, a correct $SafeAct_S^G$ grid should ensure that a situation will never arise, in which there are no safe actions.

Two channels are declared to control the behaviour of the system. The broadcast channel `hit` is used to resolve the *hit* action, while the urgent broadcast channel `bounce` is used to make the ball bounce when it hits the ground.

The constant boolean values `shield_enabled` and `layabout` are used for this model as well, for the same purposes as in Random Walk. The `layabout` parameter will in this instance force the player to deterministically choose the *nohit* action.

Two clocks `v` and `p` represent the state of the system. Despite them being declared as clocks, they can be used to model position and velocity due to UPPAAL's support for stating clock progression as a differential equation.

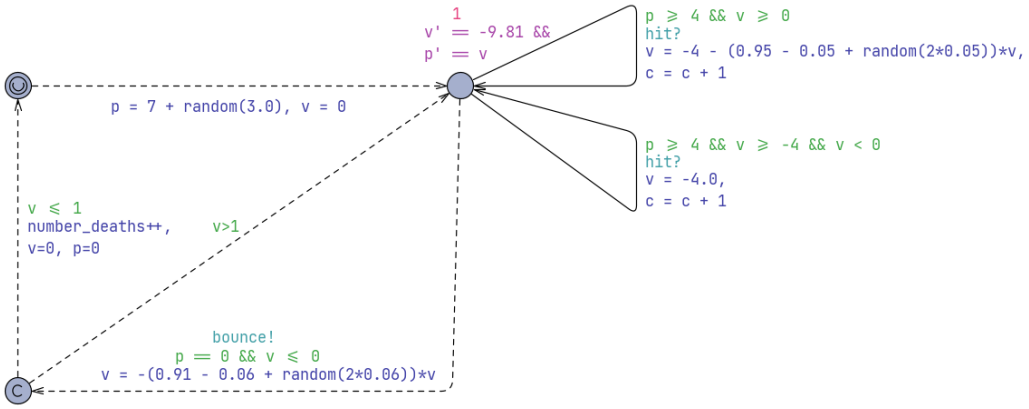

Figure 27: Automaton that models the behaviour of the ball.

Unlike the Random Walk, the mechanics of the ball are modelled as an automaton, shown in Figure 27. The first edge initializes the ball with position sampled randomly from the range $[7; 10]$ and a velocity of zero. After initialization, it reaches the state where the clocks are allowed to advance.

This state has the rate of the clocks `p` and `v` specified through the invariant `v' == -9.81 && p' == v`, which matches Equation 14 in Section 4.1. When p reaches zero, the impact with the ground is handled by the uncontrollable edge that synchronises on the `bounce!` channel. The urgency of the channel means, that `p` and `v` are not allowed to advance while the edge's guard is satisfied – that is, when `p == 0 and v <= 0`. When this is the case, the edge has to be taken, which updates v by multiplying it with $\beta_1 \pm \epsilon_1$. This leads to an intermediary state, where a check is made to ensure that the safety invariant holds. If the ball's speed after bouncing is greater than one, it is allowed to continue. Otherwise, the

`number_deaths` counter is incremented (The word „deaths" is used as a shorthand for safety violations.) and another ball is given up.

The ball getting hit is handled by the two controllable (solid) edges. The two cases for hitting the ball – as described in Equation 17 in Section 4.1 – are handled using two different edges that both have the channel synchronisation `hit?`. The guards both require that the ball must be above $p_{hit} = 4\text{m}$, and it should have a velocity greater than $v_{hit} = -4\text{m/s}$. If neither guard is satisfied, hitting will have no effect.

Since `hit` is a broadcast channel, these edges are taken if and only if their guards are satisfied, and player automaton takes the edge with a `hit!` synchronisation.

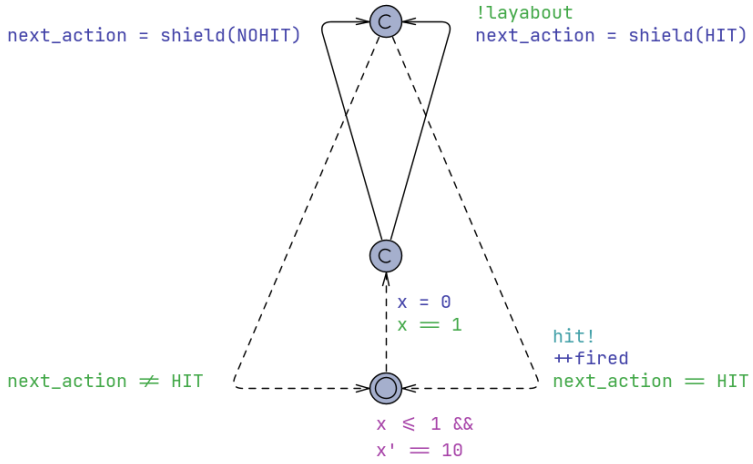

Figure 28: Automaton that models the behaviour of the player.

The behaviour of the player is shown in Figure 28, a similar automaton to Figure 15 in Section 3.3.2. The clock `x` controls how often the player takes an action, with the rate of the clock being set to 10, so that it reaches `x == 1` 10 times a second. This is to emulate $t_{hit} = 0.1$.

When the clock reaches one, the player can choose between the actions *hit* and *nohit*, as long as `layabout == false`. Otherwise, it becomes a deterministic choice, where *nohit* is always taken. The choice that was made is passed through the `shield` function, and the result saved in `next_action` is what determines if the edge representing the `hit!` channel is taken. When this is the case, the `hit!` channel is synchronised on, and the number of times the hammer has been swung at the ball, is tallied in the variable `fired`.

As with the Random Walk, this Uppaal model was validated a similar model of the bouncing ball implemented in Pluto Notebooks. Using the Shielded Layabout in both cases, the same results were obtained, validating that the two models behave the same, and that the shield was transferred successfully to Uppaal Strat-

EGO.

4.4 Description of Experiment

Experiments were conducted in much the same manner as was for the Random Walk, described in Section 3.4.1.

The configurations Shielded Layabout, No Shield, Post-shielded and Pre-shielded are repeated for the Bouncing Ball, with the layabout strategy always picking the *nohit* action. Penalties of $d \in 10$, 100 and 1000 are applied for violating the safety property, that whenever the ball hits the ground, it should bounce back with a speed greater than 1m/s . When this happens, the ball is given up again in order for the game to continue, since it is played over an infinite time horizon. As mentioned previously, the word *death* is applied as a shorthand for this occurrence.

Since UPPAAL SMC queries have to be bounded, an upper bound of 120 seconds is chosen for both training and evaluation queries. Like previously, training will use either 1500, 3000, 6000 or 12000 runs. This corresponds to significantly higher training times for the bouncing ball, since a decision is made 10 times a second. Runs are therefore around 1200 steps each, compared to the Random Walk's upper limit of 30.

Like the previous set of experiments, queries were executed using UPPAAL 4.1.20-stratego-10 Beta-2. The following queries show results for a Pre-shielded configuration with $d = 1000$ that was trained for 12000 runs. The first query states the optimization criterion, while the second query shows it achieves an average performance of 36.9 swings to keep the ball bouncing for 120 seconds. A slight risk of the ball dying is observed in the third query, but much less than once per run on average. Finally, since the configuration is not shielded, no interventions will occur, but this query would be applicable for the Post-shielded configurations.

```
strategy d1000 = minE(swings+number_deaths*1000)[<=120]{}->{p, v}: <> time>=120
E[<=120;1000](max:swings) under d1000              36.921 +/- 0.167364 (95% CI)
E[<=120;1000](max:number_deaths) under d1000        0.005 +/- 0.004379 (95% CI)
E[<=120;1000](max:interventions) under d1000                              ~=0
```

Like for the Random Walk, the penalty d is not applied when evaluating the performance of the model, to allow for better comparison between setups. Instead, the expected number of deaths is computed separately.

4.5 Experimental Results

The experiment was repeated multiple times to account for variance of the results. Due to a bug, some rows were invalid and had to be dropped, causing inconsistencies in the number of times each setup was repeated. As a result, the number varies between eight and 17, and can be seen in Table 4 along with the standard deviations. The resulting mean values are shown in Table 2, and Figures 29, 30

and 31 show plots based on the table. As expected, no safety violations were observed for the shielded configurations. Additionally, the No Shield configuration has some observed safety violations in every setup.

A large number of safety violations are seen in the No Shield configurations with $d = 10$. Looking at setups #6 to #9 in the table, the agent seems to simply learn a layabout policy, accepting the penalties rather than trying to keep the ball from dying. An average between 5.05 and 5.06 safety violations occur per run, which would result in a cost of ≈ 50 from the perspective of the learning agent. Scores below 50 are achieved by other setups, but the learning algorithm does not find these more efficient strategies, possibly due to how short-term rewards are prioritized over long-term ones.

Figure 29 shows a bar plot of the average number of deaths per 120 second run, grouped by d and color-coded to show the number of training runs used. A break in the vertical axis has been introduced to accommodate the high values observed for $d = 10$ relative to the other groups. The ball dies significantly less often for $d = 1000$ compared to $d = 100$, and both groups show a general trend where more training runs correspond to a lower risk of safety violations.

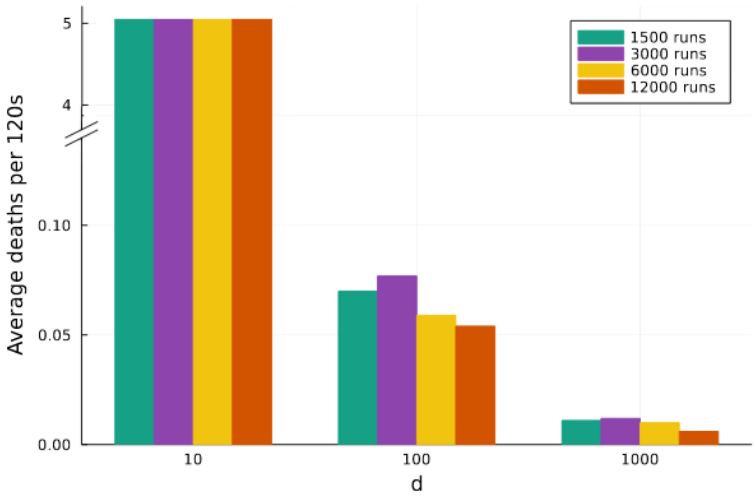

Figure 29: Bar plot that shows number of safety violations for the No Shield configuration of the Bouncing Ball problem. Bars are grouped by the penalty d applied for losing, and color-coded by number of training runs.

The number of swings used to keep the ball bouncing for a 120 second run is shown in Figure 30, as a function of the number of training runs used. The Shielded Layabout configuration does not make use of training runs, and is therefore drawn as a dotted horizontal line.

As was already seen in Table 2, the unshielded agent with $d = 10$ almost never swings the hammer. The metric has certainly been successfully minimized, but not

#	Configuration	d	Runs	Avg. swings	Avg. deaths	% interventions
1	Shielded Layabout	-	-	69.2190	0.0000	5.7698
2	Pre-shielded	-	1500	38.2755	0.0000	-
3	Pre-shielded	-	3000	37.5740	0.0000	-
4	Pre-shielded	-	6000	37.1590	0.0000	-
5	Pre-shielded	-	12000	36.9270	0.0000	-
6	No Shield	10	1500	0.0295	5.0575	-
7	No Shield	10	3000	0.0270	5.0530	-
8	No Shield	10	6000	0.0000	5.0560	-
9	No Shield	10	12000	0.0000	5.0565	-
10	No Shield	100	1500	42.1865	0.0700	-
11	No Shield	100	3000	41.7590	0.0770	-
12	No Shield	100	6000	40.4740	0.0590	-
13	No Shield	100	12000	40.4510	0.0540	-
14	No Shield	1000	1500	41.0775	0.0110	-
15	No Shield	1000	3000	41.6090	0.0120	-
16	No Shield	1000	6000	40.0670	0.0100	-
17	No Shield	1000	12000	39.5025	0.0060	-
18	Post-shielded	10	1500	69.2260	0.0000	5.7697
19	Post-shielded	10	3000	69.2090	0.0000	5.7683
20	Post-shielded	10	6000	69.2440	0.0000	5.7684
21	Post-shielded	10	12000	69.2460	0.0000	5.7669
22	Post-shielded	100	1500	66.0740	0.0000	3.5966
23	Post-shielded	100	3000	65.6865	0.0000	3.9916
24	Post-shielded	100	6000	66.1435	0.0000	3.9186
25	Post-shielded	100	12000	66.8410	0.0000	4.1085
26	Post-shielded	1000	1500	61.5450	0.0000	2.8524
27	Post-shielded	1000	3000	64.3535	0.0000	3.3076
28	Post-shielded	1000	6000	63.9235	0.0000	3.4481
29	Post-shielded	1000	12000	63.7120	0.0000	3.4162

Table 2: Average cost, average number of deaths (safety violations), and average interventions for different setups of the Bouncing Ball problem. All values are the median result of running the experiment between three and nine times.

to the desired outcome. Looking at how the same $d = 10$ setups perform when post-shielded, the average number of swings exactly match that of the Shielded Layabout, since a layabout strategy is what was learned. A marginally better post-shielded performance can be seen when the agents were trained with either $d = 100$ or $d = 1000$, though the average number of swings is still closer to the shielded layabout. Finally, the pre-shielded setups have a similar performance to that of the unshielded setups with $d = 100$ or $d = 1000$, beating them with a small margin.

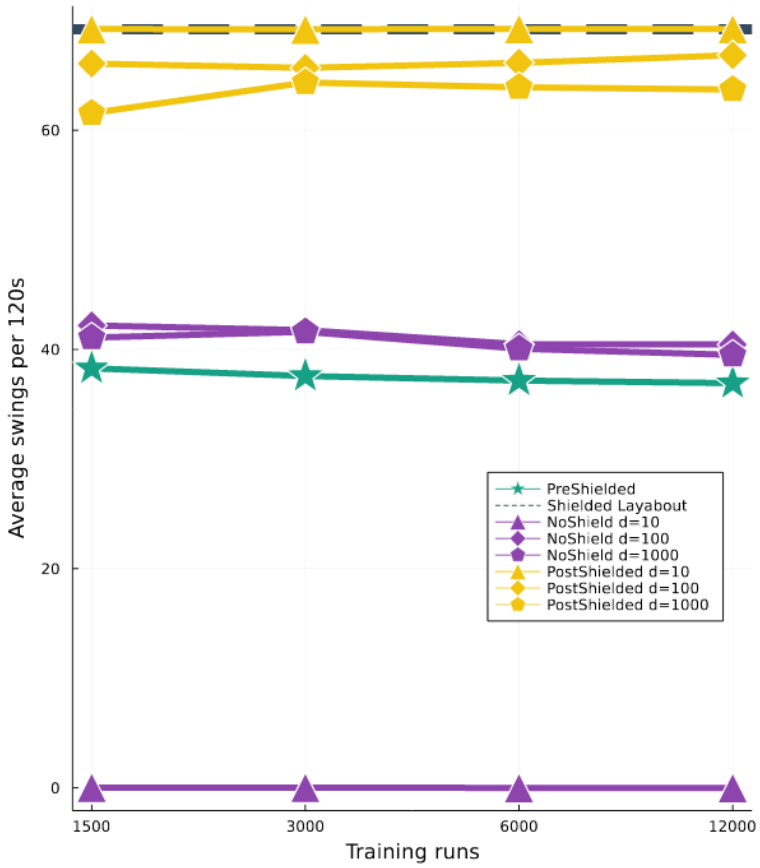

Figure 30: Average number of swings used in 120s, as a function of training time for the Bouncing Ball problem. Markers indicate configuration, and penalty d when applicable. The average cost of the Shielded Layabout is drawn as a dotted line, since it does not make use of training runs.

To further examine the performance of the Post-shielded configurations, Figure 31 shows the number of times the shield intervened, as a percentage of all actions that were taken by the agent. Since the agent makes a decision ten times a second, 1200 actions are taken per run. Although the percentages seem small, the interventions of the shield happen in the critical moments of decision-making. When an opportunity comes up to hit the ball, the shield might force a hit at less

than optimal points in the ball's arc. Thus, the five percent of interventions in the cases where $d = 10$ account for all swings during those runs. The interventions decrease slightly for agents trained with higher penalties, but likely it is still the shield which decides the majority of *hit* actions.

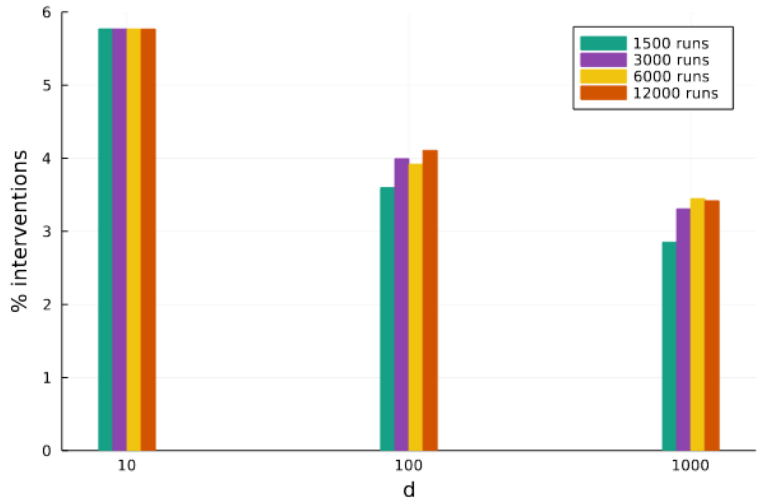

Figure 31: Bar plot that shows average number of interventions for the post-shielded configuration, for the Bouncing Ball problem. Bars are grouped by the penalty d that was applied for losing during training, and color-coded by number of training runs. For comparison, the Shielded Layabout setup has 69.246 interventions on average.

The Pre-shielded configuration was able to achieve the best performance, and did so while keeping the ball from dying at any point. Next-best was the performance of the unshielded agents trained with sufficiently high penalty – and it was seen that a penalty of $d = 10$ was not sufficient to incentivize learning. Lastly, post-shielding did not prove effective for the Bouncing Ball problem, with the shield restricting most of the behaviour of the learned strategies.

A potential area of further study would be to check, if even more efficient strategies exist. One option is to further examine, whether greater values of d lead to better or worse training outcomes, and at which point between $d = 10$ and $d = 100$ the model will begin to keep the ball alive. Genearting a shield using a finer granularity G would be possible on a system with more working memory available. Potentially, this would allow a more efficient pre-shielded strategy to train, and might make post-shielding more successfull than it was with the current setup.

5 Discussion

A safe strategy *SafeAct* was successfully computed for the Random Walk and Bouncing Ball problems, using a fixed-point iteration on a grid of permitted actions. The strategy was employed as a shield, and could successfully prevent safety violations for both the Random Walk and the Bouncing Ball, while allowing further optimization by a Q-learning agent.

In both instances, a pre-shielded configuration was able to learn an optimized strategy while respecting the safety property. Unshielded configurations, which had been nudged towards respecting the given safety property using a penalty d, were evaluated to provide a comparison. Using moderate or large penalties, they were seen to have equal or worse performance compared to the pre-shielded configuration, but showed a better performance when a very low penalty was applied. For the Random Walk, this came with a risk of safety violations of between 1% to 10% of runs. For the Bouncing Ball on the other hand, the learning agent did not optimize for the desired outcome, and chose instead to always pick the cheaper action, accepting the resulting penalties for safety violations.

Post-shielding was seen to successfully enforce safety properties onto learning agents that had already been trained without a shield in place. For the Random Walk problem, the post-shielded setups were able to achieve a performance comparable to other configurations, although it was a post-shielded setup that had the worst performance overall. Compared to the unshielded strategies that the setups were based on, performance was worse in all cases. For the Bouncing Ball, the shield interfered heavily with the learned strategy, causing a large drop in performance as a result.

Thus, if re-training a learned strategy with a shield in place is not an option, post-shielding an existing strategy can provide safety guarantees, but at risk of a serious impact on performance. On the other hand, if a shield is in place during training, the setups were in many cases seen to perform *better* than unshielded agents, making pre-shielding the preferred method of training if a safety property has to be enforced. In [1] it is suggested that the reason for this benefit in learning outcomes can be attributed to the shield acting as a *teacher* for the learning agent, correcting actions to guide it away from unsafe traces. When the shield is in place, it ensures that the agent only has to learn from valid traces, allowing it to optimize outcomes within the safety constraint, without needing to first learn how to follow it.

A finer granularity G of the partitioning used for *SafeAct* was seen to create a more permissive strategy for the Random Walk, while coarser granularities were sometimes not sufficient to generate any safe strategy, over-approximating all actions as unsafe. Due to memory restrictions, this method will therefore have the most success on problems of low dimensionality, where a sufficiently fine granu-

larity G can be applied. The method will likewise be most effective for problems that have tight lower and upper bounds on the state space $\mathcal{S}$. High dimensionality could potentially be mitigated to an extent, by applying different values of G on each axis, for a more efficient partitioning of the state space. Alternatively, dynamic partitioning could be used, to only apply a fine resolution where necessary.

Both the Bouncing Ball and the Random Walk had an action space of size 2. This made selecting an action trivial, in the case where one action was deemed unsafe. Methods for picking among several safe actions to correct an unsafe action were not explored, though this could potentially have a large impact on learning outcomes.

6 Conclusion

A method based on splitting the state space into abstract partitions was developed, which could enforce a safety property onto a hybrid game played over a finite or infinite time horizon. Subsequent experiments showed that learning could occur with the shield in place, matching or outperforming similar unshielded models in most cases.

The grid describing safe actions *SafeAct* can be computed for a model if a reachability function R is given, though the method may work best for problems of low dimensionality where memory restrictions allow for a fine granularity. The result *SafeAct* can be exported to Uppaal Stratego, where an optimized strategy can be computed using machine learning, and the outcomes evaluated using statistical model checking.

The strategy should preferably be applied as pre-shielding, which was seen to provide a better outcome than post-shielding in all cases. The restrictions imposed on a model by pre-shielding it, can prevent it from more optimized outcomes, which are possible without a shield in place. In practice it was seen however, that the pre-shielded model often performed better than its counterparts using the same amount of training runs.

The method provided in this thesis is one possible solution for enforcing a safety property onto a hybrid system. It can add guarantees that pure machine learning cannot, since – even though machine learning can generally achieve a high average case performance – it should not be trusted to always enforce safety properties.

When machine learning is applied for real-world systems, stakeholders should consider the possibility for unsafe or otherwise incorrect behaviour, even if it is a potentially rare occurrence. As methods for verification and shielding of machine learning models are developed, the benefits of optimized behaviour can be taken advantage of safely for an increasingly wide range of systems.

Bibliography

[1] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. "Safe Reinforcement Learning via Shielding". In: *AAAI*. 2018.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1 (2017), pp. 65–98. URL: https://doi.org/10.1137/141000671.

[3] Tom Breloff. *Plots.jl*. Version v1.29.0. May 2022. DOI: 10.5281/zenodo.6523361. URL: https://doi.org/10.5281/zenodo.6523361.

[4] Asger Horn Brorholt. "Safe Learning - Examining the Benets and Drawbacks of Shielded AI". 2022.

[5] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. "Uppaal Stratego". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 206–211. ISBN: 978-3-662-46681-0.

[6] Alexandre Donzé. *HSB'19. From Sensitive to Formal Barbaric Systems Biology - Oded Maler Memorial.* 2019.

[7] H. Gericke. "Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. Pacific journal of mathematics, Bd. 5 (1955), S. 285309." In: *Journal of Symbolic Logic* 22.4 (1957), pp. 370370. DOI: 10.2307/2963937.

[8] Henning Henriksen. *Hvordan hopper tennisbolde og bordtennisbolde?* Fysikforlaget, 1998.

[9] Manfred Jaeger, Giorgio Bacci, Giovanni Bacci, Kim Guldstrand Larsen, and Peter Gjøl Jensen. "Approximating Euclidean by Imprecise Markov decision processes". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2020, pp. 275–289.

[10] Manfred Jaeger, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, Sean Sedwards, and Jakob Haahr Taankvist. "Teaching Stratego to Play Ball: Optimal Synthesis for Continuous Space MDPs". In: *Automated Technology for Verification and Analysis*. Ed. by Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza. Cham: Springer International Publishing, 2019, pp. 81–97. ISBN: 978-3-030-31784-3.

[11] Raphaël Khoury and Nadia Tawbi. “Which security policies are enforceable by runtime monitors? A survey”. In: *Computer Science Review* 6.1 (2012), pp. 27–45. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2012.01.001. URL: https://www.sciencedirect.com/science/article/pii/S1574013712000020.

[12] Maplesoft, a division of Waterloo Maple Inc.. *Maple*. Version 2019. Waterloo, Ontario, 2019. URL: https://hadoop.apache.org.

[13] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Landry Nguena Timo. “Runtime Enforcement of Timed Properties”. In: *Runtime Verification*. Ed. by Shaz Qadeer and Serdar Tasiran. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 229–244. ISBN: 978-3-642-35632-2.

[14] Fons van der Plas et al. *fonsp/Pluto.jl: v0.19.4*. Version v0.19.4. May 2022. DOI: 10.5281/zenodo.6532709. URL: https://doi.org/10.5281/zenodo.6532709.

[15] Christian Schilling, Marcelo Forets, and Sebastian Guadalupe. *Verification of Neural-Network Control Systems by Integrating Taylor Models and Zonotopes*. Dec. 2021.

[16] Julian Schrittwieser et al. “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model”. In: *Nature* 588 (2020). URL: https://doi.org/10.1038/s41586-020-03051-4.

[17] Mathieu Seurin, Philippe Preux, and Olivier Pietquin. “"I’m sorry Dave, I’m afraid I can’t do that" - Deep Q-Learning From Forbidden Actions”. In: *Proc. of IJCNN 2020*. 2020. URL: https://arxiv.org/abs/1910.02078.

A Standard Deviations in Experimental Results

	Configuration	Runs	d	Times repeated	St. d. avg. cost	St. d. % runs lost	St. d. avg. interventions
1	Shielded Layabout	0	-	10	0.12546	0.00000	0.02237
2	Pre-shielded	1500	-	10	1.54574	7.59601	-
3	Pre-shielded	3000	-	10	1.15522	3.50421	-
4	Pre-shielded	6000	-	10	0.27217	1.41883	-
5	Pre-shielded	12000	-	10	0.16899	0.94634	-
6	No Shield	1500	10	10	2.02386	0.36687	-
7	No Shield	3000	10	10	0.79925	0.06434	-
8	No Shield	6000	10	10	0.83882	0.08879	-
9	No Shield	12000	10	10	0.32215	0.00909	-
10	No Shield	1500	100	10	1.88790	0.00063	-
11	No Shield	3000	100	10	2.62906	0.00257	-
12	No Shield	6000	100	10	0.88171	0.01233	-
13	No Shield	12000	100	10	0.52673	0.00544	-
14	No Shield	1500	1000	10	2.04310	0.00000	-
15	No Shield	3000	1000	10	1.67638	0.00000	-
16	No Shield	6000	1000	10	0.30346	0.00000	-
17	No Shield	12000	1000	10	0.26676	0.00000	-
18	Post-shielded	1500	10	10	1.61918	0.00000	-
19	Post-shielded	3000	10	10	0.92251	0.00000	-
20	Post-shielded	6000	10	10	0.88092	0.00000	-
21	Post-shielded	12000	10	10	0.29319	0.00000	-
22	Post-shielded	1500	100	10	1.90272	0.00000	-
23	Post-shielded	3000	100	10	2.48717	0.00000	-
24	Post-shielded	6000	100	10	1.18320	0.00000	-
25	Post-shielded	12000	100	10	0.46515	0.00000	-
26	Post-shielded	1500	1000	10	1.51673	0.00000	0.18950
27	Post-shielded	3000	1000	10	1.14328	0.00000	0.21597
28	Post-shielded	6000	1000	10	0.40135	0.00000	0.15970
29	Post-shielded	12000	1000	10	0.14195	0.00000	0.12829

Table 3: Standard deviation table for the Random Walk experiments. The table shows number of times experiment was repeated, and standard deviations of the average cost, fraction of runs lost and of the average number of interventions.

#	Configuration	d	Runs	Times repeated	St. d. avg. swings	St. d. avg. deaths	St. d. % interventions
1	Shielded Layabout	-	0	9	0.0555	0.0000	0.0063
2	Pre-shielded	-	1500	7	0.1409	0.0122	0.0000
3	Pre-shielded	-	3000	8	0.1729	0.0071	0.0000
4	Pre-shielded	-	6000	7	0.0352	0.0106	0.0000
5	Pre-shielded	-	12000	7	0.0089	0.0092	0.0000
6	No Shield	10	1500	8	2.1872	0.0900	0.0000
7	No Shield	10	3000	7	1.0841	0.0563	0.0000
8	No Shield	10	6000	5	1.6856	0.0524	0.0000
9	No Shield	10	12000	3	0.4135	0.0135	0.0000
10	No Shield	100	1500	8	2.1208	0.0218	0.0000
11	No Shield	100	3000	7	1.3348	0.0126	0.0000
12	No Shield	100	6000	5	1.3000	0.0104	0.0000
13	No Shield	100	12000	3	1.0432	0.0173	0.0000
14	No Shield	1000	1500	8	0.0688	0.0000	0.0060
15	No Shield	1000	3000	7	0.0655	0.0000	0.0047
16	No Shield	1000	6000	5	0.0397	0.0000	0.0043
17	No Shield	1000	12000	3	0.0529	0.0000	0.0034
18	Post-shielded	10	1500	7	3.4036	0.0000	0.4497
19	Post-shielded	10	3000	7	1.4381	0.0000	0.3769
20	Post-shielded	10	6000	5	1.7778	0.0000	0.4186
21	Post-shielded	10	12000	3	0.4353	0.0000	0.1310
22	Post-shielded	100	1500	7	6.4542	0.0000	1.0513
23	Post-shielded	100	3000	7	4.4205	0.0000	0.7079
24	Post-shielded	100	6000	5	3.0799	0.0000	0.5657
25	Post-shielded	100	12000	3	2.5805	0.0000	0.4873
26	Post-shielded	1000	1500	7	0.8199	0.0000	0.0460
27	Post-shielded	1000	3000	7	0.4798	0.0000	0.0264
28	Post-shielded	1000	6000	5	0.3490	0.0000	0.0210
29	Post-shielded	1000	12000	3	0.2144	0.0000	0.0093

Table 4: Standard deviation table for the Bouncing Ball experiments. The table shows number of times experiment was repeated, and standard deviations of the average cost, average deaths, and of the average number of interventions.