Probabilistic Least-violating Control Strategy Synthesis with Safety Rules

LUDVIG JANIUK, JOHAN SJÖLÉN
1 Abstract

We consider the problem of automatic control strategy synthesis for discrete models of robotic systems, where the goal is to travel from some region to another while obeying a given set of safety rules in an environment with uncertain properties. This is a probabilistic extension of the work by Jana Tumová et al. [Tům+13] that is able to handle uncertainty by modifying the least-violating strategy synthesis algorithm. The first novel contribution is a way of modelling uncertain events in a map as a Markov decision process with a specific structure, using what we call “Ghost States”. We then introduce a way of constructing a Product Automaton analogous to the original work, on which a modified probabilistic version of Dijkstra’s algorithm can be run to synthesize the least-violating plan. The result is a synthesis algorithm that works similarly to the original, but can handle probabilistic uncertainty. It could be used in cases where e.g. uncertain weather conditions or the behaviour of external actors can be modelled as stochastic variables.

2 Sammanfattning

Vi undersöker automatisk kontrollstrategisyntes (automatic control strategy synthesis) av diskreta robotsystem där målet för roboten är att färdas från en region till en annan medan den följer en mängd säkerhetsregler i en miljö med probabilistiskt osäkra egenskaper. Detta är en uppföljning av arbete gjort av Jana Tumová et al. i [Tům+13]. Vi utvidgar deras arbete genom att modifiera strategisyntesalgoritmen så att den kan hantera probabilistiska situationer. Vårt första bidrag är ett sätt att modellera probabilistiska situationer i en karta genom en så kallad "markov decision process" med en specifik struktur som vi kallar för "Ghost States" (spöktillstånd). Vi bidrar även med ett sätt att konstruera en produktautomat som är analog till originalarbetets produktautomat. På vår produktautomat kan en probabilistisk variant av Dijkstras algoritm köras för att framställa en plan som är "least-violating" (bryter mot säkerhetsreglerna minst). Resultatet är en syntesalgoritm som fungerar som originalet men som även kan hantera stokastiska osäkerheter. Syntesalgoritmen skulle till exempel kunna användas i de fall där ovisa väderlekar eller beteendet av externa aktörer kan modelleras som stokastiska variabler.
## Contents

1 Abstract 2

2 Sammanfattning 2

3 Introduction 4

4 Preliminaries 4
   4.1 Environment models 5
   4.2 Logical models 6
   4.3 Automata 7
   4.4 Probabilistic Dijkstra 8

5 Problem statement 10
   5.1 Motivation 10
   5.2 Problem formulation 11

6 Methods 13
   6.1 Construction of the MDP utilizing Ghost States 13
   6.2 Extend NFA into weighted NFA 15
   6.3 Build all NFA into one 16
   6.4 Probabilistic Product automaton 16
   6.5 Introduction of intermediate states 16
   6.6 Reasoning behind the product automaton and illustrations 18
   6.7 Formal definition of Product Automaton 19
   6.8 Finding the best plan 20
   6.9 Using the plan 20
   6.10 Motivation of correctness 21
   6.11 Memory complexity of product automaton 22
   6.12 Algorithm 22

7 Analysis and discussion 22
   7.1 Possible future work 23
      7.1.1 More frugal PA definition 23
      7.1.2 Implementation and case studies 23

8 Conclusions 23
3 Introduction

The world's usage of robots is at an all-time high. Areas of industry and public life which previously did not use robots are starting to have robotic solutions developed for them, such as autonomous driving solutions for vehicles. With this increase in the relevance of robotics in public life the importance of robots behaving in a way that minimizes the danger to humans has increased. The need for robotic solutions that are able to minimize the risk of its actions and not only its efficiency in doing the task given to it has thus also increased. Robots should therefore try to follow certain constraints as to minimize the harm they can do.

Temporal logic presents itself as a way to express the constraints that we want robots to adhere to, and facilitate generating optimal runs under these constraints. When modeling the robot’s environment, some properties never change. However, certain properties might not always be true, but only have a probability of occurring. For example, there might be a probability that a pedestrian is crossing the road while the robot is moving there, resulting in an accident. We can express the fact that hitting a pedestrian is unwanted behaviour through temporal logic. If we then make a model of the above mentioned environment, we can generate robot movement in it that satisfies the constraints, or minimizes the risk of breaking them. This is an example of model checking-based control strategy synthesis, which is what our paper is based around. [Din+14]

Our thesis will explore the theoretical situation where an autonomous agent has the goal of moving from a region A to another region B in an environment where some events have a probability of occurring at certain locations. This can be viewed as an extension to [Tům+13]. In that paper the authors formalize the safety requirements as a prioritized system of rules, on an environment that never changes. This paper will instead be focused on expressing and handling events that are uncertain, but with a simpler system of rules. Syntactically co-safe linear temporal logic (scLTL) will be used to express statements regarding the events in the environment and these statements will be seen as statements that should not hold true (that is, they are unwanted).

4 Preliminaries

To solve the problem of modelling a robot in an uncertain environment this paper will define certain mathematical structures, as well as a specification language (which will be a formal logic) to make statements regarding the environment. This section defines these mathematical concepts, and gives short motivations for their use. These definitions will serve as a theoretical context for the rest of the thesis.
4.1 Environment models

**Definition 4.1.1** (Weighted Labelled Transition System). A weighted labelled transition system is defined as a tuple

\[ T = (S, s_0, R, W, AP, L) \]

where

- \( S \) is a finite set of states,
- \( s_0 \in S \) is the initial state,
- \( R \subseteq S \times S \) are the possible transitions between states,
- \( W : R \to \mathbb{R}_{\geq 0} \) is a weight function assigning each transition a non-negative weight,
- \( AP \) is a set of atomic propositions, and
- \( L : S \to 2^{AP} \) is a labelling function assigning a subset of the propositions in \( AP \) to each state.

A WLTS is well suited to modelling a typical 2D-environment such as a grid, and the robot’s state in it. Each state can be defined to be a position in the grid and the weight function can represent the cost of moving between positions in the grid. The set \( AP \) can describe which properties hold true in certain parts of the grid, such as one part being muddy for example.

Importantly, a WLTS can also be used to model more than just positions. Each state could for example represent a position and an orientation. This way, the WLTS can model the whole state space of the robot, that is the whole range of all situations the robot could find itself in when moving in the grid.

**Definition 4.1.2** (Markov Decision Process). A Markov Decision Process (MDP) is defined as a tuple

\[ M = (S, s_0, U, U^S, AP, L, W, P) \]

where \( S \) is a finite set of states, \( s_0 \in S \) is the initial state, \( U \) is a finite set of actions, \( U^S : S \to 2^U \) maps each state to the set of possible actions in that state, \( AP \) is a set of atomic propositions, \( L : S \to 2^{AP} \) labels each state with a set of atomic propositions, \( W \) is a weight function analogous to the one in 4.1.1, and finally \( P : S \times U \times S \to [0, 1] \) is a transition probability function satisfying that for all \( s \in S \) and \( u \in U \), if \( u \in U^S(s) \) then we have \( \Sigma_{s' \in S} P(s, u, s') = 1 \), otherwise \( P(s, u, s') = 0 \) for every \( s' \in S \).

The main purpose of MDP’s in this paper is to introduce the concept of probability to the environment model. Contrary to the WLTS, each state has a certain probability to transition into one of their neighboring states, given an action that is performed. This opens up the possibility to develop interesting
ways of representing the risk of something unwanted happening, when we decide to move a certain way. To emphasize: The difference between a WLTS and a MDP is that a state in a MDP may have multiple edges with the same action, each with a different probability of it being chosen.

Definition 4.1.3 (Plan over state and action space). Assume a state space $S$, a set of actions $U$, and an action space $U^S : S \rightarrow 2^U$. A memoryless plan $\pi : S \rightarrow U$ assigns a planned action for each state, so that $\pi(s) \in U^S(s)$. A memory-dependent plan $\pi_{\text{mem}} : S^* \rightarrow U$ assigns a planned action for each scenario of movement through the state space so that $\pi_{\text{mem}}(s_0, s_1, ..., s_k) \in U^S(s)$. This means a memoryless plan may only assign one action to each state, while a memory-dependent plan could assign any number of actions for the same state, depending on how one arrived there.

In this thesis, plans will be executed over MDP’s. If nothing else is specified, a plan is taken to be memoryless by default.

4.2 Logical models

Definition 4.2.1 (scLTL Syntax). \cite{BYA17} A syntactically co-safe LTL formula $\phi$ over a set of atomic propositions $AP$ is defined recursively as follows:

$$\phi = \top \mid o \mid \neg \phi \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid X\phi \mid \phi_1 U \phi_2$$

where $o \in AP$, the operators $\neg, \land, \lor$ are boolean operators and $X$ and $U$ are temporal operators $\text{next}$ and $\text{Until}$. Further, the following operator is defined in terms of the existing ones:

$$F\phi := \top U \phi$$
Definition 4.2.2. (scLTL Semantics) [BYA17]
The satisfaction of a formula $\phi$ over a set of atomic propositions $AP$ at position $k \in \mathbb{N}_+$ in word $w_{AP} = w_{AP}(1)w_{AP}(2)w_{AP}(3)... \in AP^\omega$ denoted as $w_{AP}(k) \models \phi$, is defined recursively as follows:

- $w_{AP}(k) \models \top$
- $w_{AP}(k) \models o$ for an $o \in AP$ if $w_{AP}(k) = o$
- $w_{AP}(k) \models \phi_1 \land \phi_2$ if $w_{AP}(k) \models \phi_1$ and $w_{AP}(k) \models \phi_2$
- $w_{AP}(k) \models \phi_1 \lor \phi_2$ if $w_{AP}(k) \models \phi_1$ or $w_{AP}(k) \models \phi_2$
- $w_{AP}(k) \models X \phi$ if $w_{AP}(k + 1) \models \phi$
- $w_{AP}(k) \models \phi_1 U \phi_2$ if there exists $j \geq k$ such that $w_{AP}(j) \models \phi_2$ and, for all $i | k \leq i < j$ we have $w_{AP}(i) \models \phi_1$

A word $w_{AP}$ satisfies formula $\phi$, written as $w_{AP} \models \phi$, if $w_{AP}(1) \models \phi$.

There exist several logic systems that are defined over time, for example finite linear temporal logic (FLTL), and linear temporal logic (LTL). scLTL is used in this thesis because it presents several guarantees, namely:

- The formula can be validated in finite time if it is true. This is because every word that satisfies a scLTL formula contains a “good” prefix such that no matter what comes after that prefix, the formula would be satisfied. This is important because a robot path for example would always be a finite number of steps, and so we want to be able to say that the robot can satisfy the safety rules.

- Every scLTL formula can be converted into an NFA (defined in 4.3.1). LTL, for example, does not give this guarantee.

4.3 Automata

Definition 4.3.1. (Nondeterministic Finite-state Automaton) [BYA17]
A Nondeterministic Finite-state Automaton (NFA) is defined as a tuple

$$A = (Q, q_0, AP, \delta, F)$$

where

- $Q$ is the set of states,
- $q_0 \in Q$ is the initial state,
- $AP$ is the input alphabet,
- $\delta \subseteq Q \times AP \times Q$ is a transition relation, and
- $F \subseteq Q$ is the set of accepting states.
Definition 4.3.2. (Run of $A$ over word) [BYA17]
A run of $A$ over a word $w_o = w_o(1)w_o(2)...w_o(n) \in AP^*$ is a sequence
$$w_s = w_s(1)w_s(2)...w_s(n+1) \in Q^*$$
where $w_s(1) = s_0$ and $(w_s(k),w_o(k),w_s(k+1)) \in \delta$ for all $k = 1,2,...n$. If $w_s(n+1) \in F$, we say that the run is an accepting run.

An NFA can be used to validate whether a word is part of a language or not. Start from the initial state and the first letter of the word. Then, move through the NFA using only the transitions corresponding to the current letter. The goal is to end up in an accepting state. If there is a way to do this, then the word must be part of the language that the NFA accepts.

scLTL formulas are always translatable into an NFA. That is, for every scLTL formula, one can build an NFA that only accepts the words that satisfy the formula.

4.4 Probabilistic Dijkstra

The probabilistic-dijkstra algorithm is used to find a plan for moving in a probabilistic state space, such that one is expected to arrive at the goal as cheaply as possible if following the plan. It is a generalization of the classic Dijkstra’s algorithm, but it works where transitions are probabilistic, that is having chosen an action in a state, one might end up in several other states. An MDP is a good example of such a state space.

The algorithm is an extension of the nondeterministic-dijkstra algorithm. Both are described in [LaV]. It is based on backprojections, this means that the algorithm starts “from the end”, that is from the set of goal states (in the case of this thesis this is a set of size 1). It then grows a plan $\pi$ by moving outwards towards the available starting states.

The algorithm maintains a set of active states $A$, a set of states $C$ having already been processed, asssociations actions with states in the plan, and the currently computed “cost-to-go” $G$ for each state. It is defined very generically in [LaV], but in this thesis we give a specialized definition for running over an MDP.

Definition 4.4.1. Probabilistic Dijkstra algorithm

Given an MDP $(S,s_0,U,U^S,AP,L,W,P)$ and a set of goal states $F$:

1. Initialize $C = \emptyset$ and $A = F$. Associate the null action with every $s \in A$. Assign $G(s) = 0$ for all $s \in A$ and $G(s) = \infty$ for all other states.

2. Unless $A$ is empty, remove the $s_s \in A$ and its corresponding action $u$ for which $G$ is the smallest. If $A$ was empty, then exit (no further progress is possible).

3. Designate $\pi^*(s_s) = u$ as part of the optimal plan and insert $s_s$ into $C$. Declare $G^*(s_s) = G(s_s)$.
4. Compute $G(s)$ for any $s$ in the frontier set, $Front(s, C)$, and insert $Front(s, C)$ into $A$ and with associated actions for each inserted state. For states already in $A$, retain whichever $G$ value is lower, either its original value or the new computed value. Go to Step 2

Let the one-stage forward projection $Proj(s, u) = \{s' | P(s, u, s') > 0\}$. $G(s)$ is computed according to the following formula:

$$G(s) = \min_{u \in U^S(s)} \left\{ E_{s'} \left[ W(s, u, s') + G(s') \right] \right\}$$

The frontier set $Front(s, C)$ is defined as the states whose costs would be immediately affected by the inclusion of $s$ into $C$. These are states $s$ for which their one-stage forward projection fulfills the following criteria: 1). $s \in Proj(s, u)$, and 2). $Proj(s, u) \subseteq C$. 


5 Problem statement

Figure 2: Example of an applicable scenario. The vehicle must travel to the goal and has two roads to choose from. There is a higher probability of pedestrians crossing the road to the right, but maybe the left road is significantly longer?

5.1 Motivation

The motivation for this paper, which builds upon [Tům+13], comes from the following scenario: Given a robotic vehicle in an uncertain environment, and a goal to reach a target position B from a starting point A, automatically generate the optimal path for the robot to take. “Optimal” as defined in [Tům+13], is taken to mean the shortest path that violates certain safety rules as little as possible. In [Tům+13], the map is assumed to be static: certain properties hold true in some positions on the map, and never change. We extend that paper with support for random events. Having done this, we must replace the fixed “shortest path” from the original authors with a plan: what do we do in each state, given how the uncertainties turn out?

A motivating example, (see Fig. 2):

Assume a car needs to drive as quickly between two regions as possible through a well-known area. Given that some roads lead to a shorter path but have more zebra crossings it may be that taking the longer road would give a shorter time of travel. Given statistical analysis of these zebra crossings a probability of people forcing the vehicle to stop can be derived. Such knowledge along with our method could prove to give the car the information necessary to arrive at a better informed choice regarding how to travel.
Figure 3: Example WLTS of a 3x3 map. Each position is represented by four states, since the states encode both position and orientation. For brevity, actions and full state labels have been omitted.

5.2 Problem formulation

In the case studies of [Tům+13] the authors model the environment as a Weighted Labelled Transition System (WLTS), where each state represents a position on the map, and the orientation of the robot. This means that a single position (or “cell”) in the map corresponds to four states in the WLTS: one for each direction the robot could be facing there. Figure 3 gives an idea of how this might look. The transitions represent how the robot may move between the states. For example, from the state “Facing east at (0, 0)” there would be a transition to the state “Facing east at (1, 0)” corresponding to the action “move forward”. If that location were a junction then there might be an additional transition to the state “Facing south at (0, 1)” corresponding to the action “Turn right”. We use the same initial model of the environment, but later extend it with the non-permanent properties.

Apart from the environment, the safety rules must be modelled. These are modelled in this paper as scLTL formulas coupled with a nonnegative integer cost for breaking them. The scLTL formulas are transformed into finite state automata. These automata are then adapted into new, weighted automata that accept any word, but associate additional cost with the new transitions added that do not fulfill the original rule. This concept is described in detail later in
Finally, given the WLTS to represent the environment and the automata representing the safety rules, the original authors produce a “product automaton” which is a graph representing both the map and the rules at once. It is mostly defined as a product set of the states in the WLTS, and the states in the automata. Paths in this product automaton map to paths in the original map, but also express rule satisfaction. Therefore, using a comparison function defined in a certain way, it is proven that the shortest path in this product automaton corresponds to the path in the map that fulfills the original problem statement (i.e. is at least as safe as any other route, and is the shortest such path).

We want to extend this method to allow for random events, changing the properties on certain positions. To take a simple example: a neighborhood street is normally empty, so driving on it is safe. However, if a pedestrian walks onto the street, then suddenly the street is not empty and it’s no longer safe to drive there. On the other hand, a highway may be properly walled off, so we could assume that it’s impossible for pedestrians to appear there. This “possibility of random events” is what we want to express in our model of the environment. We want to be able to say that “in this location, two or more different sets of properties might hold, with certain probabilities.”

In order to model this, a Weighted Transition System is not sufficient, as it only allows for one set of labels on each state. Additionally, probabilistic transitions are not expressible. To solve this problem, we move to a Weighted Markov Decision Process which also features states, labels and choice, but additionally incorporates probabilistic transitions. Probabilistic transitions simply mean that given a choice of action, the process might go to any one of the connected states with some probability. The problem of several sets of labels in one state is solved by us by essentially copying the original state into several Ghost States in the MDP, where each may represent a different possible reality of the original state. The transitions on these are set up so that they still all behave like the original state.

With this in mind we will solve two problems which are described as follows:

**Definition 5.2.1.** Problem 1

Given a map expressed as a weighted labelled transition system and a set of uncertain events regarding that map, find a way to model the uncertain events in the map.

**Definition 5.2.2.** Problem 2

Given a map with uncertain events modelled in it, a starting region A, a goal region B, and a set of safety rules expressed in scLTL, find a way to do the following in the map:

1. Move from region A and eventually reach region B with probability 1.
2. Minimize the expected penalty of broken safety rules.
3. Minimize the expected travel distance in the map, maintaining the minimal safety penalty.
To reiterate: We want to solve these problems because we want to extend the method in [Tům+13] to allow for random events through changing the properties of certain positions in the map.

6 Methods

With the preliminaries and problem statement introduced and defined, what now follows is the method. More specifically, the following text is a description of the methods that we have used or developed to solve the problems defined in the problem statement.

Our novel contributions can be found mainly in section 6.1 and 6.4 where the concept of Ghost States and our product automaton are defined respectively. We begin by explaining the idea of Ghost States.

6.1 Construction of the MDP utilizing Ghost States

In [Tům+13], the map is taken to be a weighted labelled transition system. That paper only deals with the case where the map is static, and in every state a set of observations holds which might never change. We want to consider how a more dynamic map might be entered into this process. More specifically, we investigate a map which can still be represented by a finite set of states, but we want to express the possibility of random events occurring at certain locations. For example, “at state $s_5$, observation $p_3$ has a 10% chance of being true”.

To express this uncertainty we need to move from a weighted labelled transition system to a MDP, where the same action in the same state can have a chance of leading to several different new states. We add weights to our MDP which directly correspond to the weights of the WLTS.

Our solution to this is a specific way of structuring MDPs that we call "Ghost States" and is our solution to the first problem described in the problem statement.

The first step in this process is formalizing the idea of “random events occurring at certain locations”. We formalize it as a function $\hat{L}$ which gives the possible alternatives for each state, and their probabilities. For example, consider in a map where there is a state $s_k$ in which the proposition $\text{redlight}$ can either be true or not with a 50/50 probability. That would mean that $\hat{L}(s_k) = \{ (\{\text{redlight}\}, 0.5), (\{\}, 0.5) \}$.

**Definition 6.1.1.** Given a WLTS $\mathcal{T} = (S, s_0, R, W, AP, L)$ we define the possible random property function $\hat{L}$. This function takes a state from the WLTS and returns a set of tuples consisting of a set of atomic propositions and the probability that these are true in that state. The signature of this function can be seen as follows:

$$\hat{L}(s) = \{ (\hat{A}, p) | \hat{A} \subseteq AP, p \in [0, 1] \}$$
Since it expresses probabilities, we require that the probabilities of each state sum up to one. That is, \( \hat{L} \) must satisfy

\[
\forall s \in S : \sum_{(a,p) \in \hat{L}(s)} p = 1
\]

Additionally, since \( \hat{L} \) is meant as an extension of \( L \), we also require that the original labelling of each state as expressed in \( L \) exists represented in \( \hat{L} \). That is, the following must hold:

\[
\forall s \in S : \exists (a, p) \in \hat{L}(s) : a = L(s)
\]

The following describes the process of creating an MDP expressing our random possibilities. Start with a WLTS describing your environment, wherein the states that are static are expressed. Now, for each state \( s \) in the map:

1. Consider all different versions it has been determined that the state can be in. These are the different labellings of the state that could hold true with some probability when the state is visited. They are expressed by the function \( \hat{L} \).

2. For each such version \( \hat{A} \) and its probability \( p \) of being the case once the state is visited \((\hat{A}, p) \in \hat{L}(s)\):

   (a) Create a new state \( s' \) which will represent this “reality” being true. This new state is what we call a ghost state.

   (b) Label \( s' \) with \( \hat{A} \).

   (c) For each transition going into \( s \), create a transition going into \( s' \) with the probability \( p \).

   (d) For each transition going out from \( s \), create a transition going out from \( s' \).

3. Remove the state \( s \) and all transitions that include it from the MDP.

The end result will be an MDP which represents all the possible versions of every state and their probabilities as “Ghost States”.

**Example 6.1.1.** Consider the following example where we start with the transition system:

\[
\begin{array}{c}
  s_1 \\
  \rightarrow \\
  s_2 \\
  \rightarrow \\
  s_3
\end{array}
\]

To be able to represent a random possibility at \( s_2 \) with the probability 0.2 we run our algorithm to generate Ghost States:
And then we label \( s_2 \) the same as \( s_2 \) before, and \( s_2 \) with the labels of the event. So we now have a MDP that represents our more dynamic environment.

**Lemma 1.** A run over a MDP constructed as above can be projected to a run over the original WLTS.

*Proof.* A state in the MDP is either the same state as in the WLTS or it is a ghost state. If it is a ghost state then the state has a tag which is what state in the WLTS it belongs to and an index for its ghost state, thus the ghost state’s original state is the tag.

A run \( \tau \) over the MDP is a sequence of states. Converting each state as defined above yields a sequence of states in the WLTS. It follows from step 2c that each of the transitions between states in the run will also exist between the states in the WLTS. Thus, the projection can always made. \( \Box \)

### 6.2 Extend NFA into weighted NFA

Given an NFA \( A_\psi \) representing a safety rule \( \psi \) and a cost \( w \in \mathbb{Z} \) for breaking \( \psi \) we construct an NFA \( \overline{A}_\psi \) which accepts all runs but associates a cost with previously un-accepted runs. This is equivalent to the work done in [Tům+13]. \( \overline{A}_\psi \) is defined as a tuple

\[
\overline{A}_\psi = (Q_\psi, \overline{q}_0, \overline{AP}_\psi, \delta_\psi, W_\psi)
\]

where

- \( Q_\psi = Q_\psi \)
- \( \overline{q}_0 = q_0 \psi \)
- \( \overline{AP}_\psi = AP_\psi \)
- \( \delta_\psi = \delta_\psi \cup \{(q, o, q') | q \in Q_\psi, o \in 2^{AP_\psi} \} \)
- \( F_\psi = F_\psi \)
- \( W_\psi(q, o, q') = 0 \) if \( (q, o, q') \in \delta_\psi \)
- \( W_\psi(q, o, q') = w \) if \( (q, o, q') \in \delta_\psi \setminus \delta_\psi \)

**Lemma 2.** Let \( \omega \) be a word over \( 2^{AP_\psi} \). Then \( \omega \) is accepted by \( \overline{A}_\psi \) and the weight of the shortest accepting run of \( \overline{A}_\psi \) over \( \omega \) is equal to the level of unsafety \( \lambda(\omega, \overline{A}_\psi) \). Proof of this and definition of \( \lambda(\omega, \overline{A}_\psi) \) is found in [Tům+13].
6.3 Build all NFA into one

Given a set of \( n \) weighted automata \( \{ A_\psi \mid 1 \leq i \leq n \} \) we combine them into one automaton \( A_\psi \). This allows us to finally combine our safety rules with the MDP to construct our product automaton. Again, the definition is the same as in [Tům+13]:

**Definition 6.3.1** (Combined automaton). The combined automaton \( A_\psi \) is defined as a tuple

\[
A_\psi = (Q_\psi, q_0_\psi, AP_\psi, \delta_\psi, F_\psi, W_\psi)
\]

where

- \( Q_\psi = Q_1 \times Q_2 \times \ldots \times Q_n \)
- \( q_0_\psi = ((q_0)_1, (q_0)_2, \ldots, (q_0)_n) \)
- \( AP_\psi = 2^{AP} \)
- \( (p, \sigma, p') \in \delta_\psi \) and \( W_\psi((p, \sigma, p')) = w \) if \( p = (q_1, \ldots, q_n) \), \( p' = (q'_1, \ldots, q'_n) \), \( (q_i, \sigma, q'_i) \in \delta_i \) for all \( i \in \{1\ldots n\} \) and \( w = \sum_{i=0}^n W_i((q_i, \sigma, q'_i)) \)
- \( F_\psi = \{ (q_{1,i}, q_{2,i}, \ldots, q_{m,i}) \mid q_m \in F_i \text{ for all } i \in \{1, \ldots, n\} \} \)

**Lemma 3.** Let \( \omega \) be a word over \( 2^{AP} \). Then \( \omega \) is accepted by \( A_\psi \) and the weight of the shortest accepting run \( \tau \) of \( A_\psi \) over \( \omega \) is equal to (with a slight abuse of notation) \( \sum_{A_\psi \in A_\psi} \lambda(\tau, A_\psi) \). Here, \( A_\psi \in A_\psi \) is taken to mean “every rule NFA that was used to construct the combined automaton.” Proof of this can be found in [Tům+13]

6.4 Probabilistic Product automaton

We now construct a product automaton (PA) which is a combination of an MDP and an NFA. As is seen in the formal definition in [6.7.1] the combination is done mainly through using set multiplication, although some specific structure (intermediate states) is added. The result can be considered an MDP that enables us to reason about both the original MDP and NFA simultaneously. Since the formal definition of the PA is quite complex, we will introduce the concept of intermediate states and explain why they are necessary, before delving into the formal definition. Figure 4 will be used as a running example.

6.5 Introduction of intermediate states

The PA’s states are built from the product set of the original MDP and NFA’s states. This product set would contain states such as the ones in Figure 4c: each state is a combination of a state in the MDP and in the NFA. However, for each element in this product, the PA actually creates two versions, tagged with “!” and “?” respectively (see example in Figure 4d). These tags are introduced to
(a) The MDP

(b) The NFA

(c) Part of the naïve PA

(d) The same part of our PA. Edges without actions are assumed to have an individual action with a 100% probability of being chosen

Figure 4: Simplistic comparison between Product Automaton synthesis using a naïve product, and our approach. Only the part of the PA relevant in the first “step” is shown.
separate the non-determinism of the NFA and the probabilistic nature of the MDP. More specifically, when a probabilistic split happens in the MDP the PA combines this with the NFA by introducing an extra state transition (marked with the ?-tag). This allows the PA to keep the probabilities from the MDP in a correct way, which is not possible with the PA as defined in [Tům+13].

In other words, ?-tags represent a choice having been taken in the MDP but yet to be taken in the NFA. For example, if we are in state \( s_i \), then an MPD-action is first chosen and the probability outcome determines which one (out of possibly several) ?-states we arrive in. The next step is to move from the ?-state to a !-state. This transition is always deterministic and represents the choice of how to proceed in the NFA.

### 6.6 Reasoning behind the product automaton and illustrations

Our PA is markedly different from the PA found in [Tům+13]. A major change is the move from a “concurrent” model, to a “turn-based” one. In the original PA, each transition corresponded to movement in the map, but our PA maps each movement in the map to two turns: moving to a ?-state and then to a !-state.

This subsection motivates the choice of PA by first showing how the original is insufficient and then how our PA solves these insufficiencies. These examples are not representative of real-world cases, but illustrate the difference between the PA’s in sufficient detail.

Consider a simple MDP and NFA (Figures 4a and 4b). Figure 4c shows the way one would (partially) construct a PA based on this MDP and NFA naïvely following the definition of [Tům+13] and extending it for MDPs. This would however lead to the paths of action A having a summed probability > 1, which is paradoxical. Why does the problem occur? The original authors were only trying to take the product of a WLTS and an NFA, not an MDP. The MDP has probabilistic splits whereas the NFA is nondeterministic. Mixing these together naïvely does not work because one must actually perform the choice in the NFA only after the probabilistic choice in the MDP takes place. Depending on where one turns out, the available options might be different. This cannot be represented in one single transition.

To remedy this, our PA (Figure 4d) maps each original pair of transitions (the probabilistic ones from the MDP and the deterministic ones from the NFA) into two layers of transitions. We introduce the intermediate ?-states to represent having moved in the MDP, but not yet decided how to move in the NFA. First, by moving from a !-state to a ?-state one makes a choice of action in the MDP which probabilistically leads to one of the intermediate states. Then, the transitions from that state represent the possible courses of action in the NFA, given the apparent labels on the state one arrived at. Chosing one of these and moving from the ?-state to the !-state represents making a choice of action in the NFA, and thus the transition is complete: we have moved in both the MDP and the

---

1See appendix 2 for example
NFA.

We see therefore that this structure is necessary and allows us to represent both the MDP and the NFA in one model, which will allow for running a variation of Dijkstra’s algorithm on it to find the best plan.

6.7 Formal definition of Product Automaton

Note how the actions are added into $\mathcal{P}$.

**Definition 6.7.1.** Let the product automaton $\mathcal{P}$ be defined by

$$\mathcal{P} = M \otimes \mathcal{A}_\psi = (Q_\mathcal{P}, q_{init}, \delta_\mathcal{P}, W_\mathcal{P}, F_\mathcal{P}, P_\mathcal{P})$$

where

- $Q_\mathcal{P} = S \times Q_A \times \{!, ?\}$
- $q_{init} = (s_{init}, q_0, \exists)$
- $\delta_\mathcal{P} = \delta_{MDP} \cup \delta_{NFA}$
  - $\delta_{MDP} = \{((s, q), u, (s', q, ?)) | P(s, u, s') > 0, u \in U^+(s)\}$
  - $\delta_{NFA} = \{((s, q), (u, i), (s, q', !)) | (q, (u, i), q') \in \delta_{enum}\}$
  - $\delta_{enum} = \{(q, (u, i), q') | (q, u, q') \in \delta\}$ such that
    $\forall (q, (u, i), q') \in \delta_{enum} \nexists j \neq i \land (q, (u, j), q') \in \delta_{enum}$
- $W_\mathcal{P} = W_{MDP} \cup W_{NFA}$
  - $W_{MDP} = \{(((s, q), u, (s', q, ?)) | (s, u, s') \in \delta_{MDP}, (0, W_T(s, u, s')))\}$
  - $W_{NFA} = \{(((s, q), (u, i), (s, q', !)) | (s, q') \in \delta_{NFA}, (W_A(q, L(s), q'), 0))\}$
- $F_\mathcal{P} = S \times F_A \times \{!\}$
- $P_\mathcal{P} = P_{\mathcal{P}_{MDP}} \cup P_{\mathcal{P}_{NFA}}$
  - $P_{\mathcal{P}_{MDP}} = \{(((s, q), u, (s', q, ?)), \rho) | ((s, u, s'), \rho) \in P\}$
  - $P_{\mathcal{P}_{NFA}} = \{(((s, q), u, (s, q', !)), 1)\}$
- State$(s, q, t) = s$ Nfa$(s, q, t) = q$ Tag$(s, q, t) = t$

**Lemma 4.** Given a PA $\mathcal{P}$, a plan $\pi$ and the original MDP $M$ then executing the plan over the PA gives a run over the MDP.

**Proof.** The following proof is a constructive proof and so can also be seen as an algorithm for how to follow along in the map when using a plan.

Given a plan $\pi$ (from $s_0$ to $s_n$), an MDP $M$, a PA $\mathcal{P}$ and some procedure $\mathcal{G}$ (whose purpose is described below) we show an algorithm that can be used to get the path in the MDP that following the plan over the PA gives us. Let $\mathcal{G} : \mathcal{P}_{state} \times O \rightarrow \mathcal{P}_{state}$, that is see $\mathcal{G}$ as a procedure (not function because of probabilistic nature) which gives the next state given some state and an action.
\( \mathcal{G} \) must be provided since state-transitions are probabilistic. \( \mathcal{G} \) can be seen as the PA deciding what the next state should be when an action has been chosen. For example, if the action \( A \) in state \( s_n \) can lead to either \( s_i \) or \( s_j \) then \( \mathcal{G} \) decides which one it is that \( s_n \) leads to for this instance.

Given the above structure as input the algorithm is as follows:

**Result:** Path in MDP

```plaintext
currentState = s_0;
path = [];
path.push(State(currentState));
while currentState \neq s_n do
    nextAction = \pi(currentState);
currentState = \mathcal{G}(currentState, nextAction);
    if tag(currentState) == ? then
        path.push(State(currentState));
    end
end
path.push(State(s_n));
return path;
```

**Algorithm 1:** MDP path  

---

**Lemma 5.** The optimal plan results in the optimal run over the MDP with regards to the expected cost.

**Proof.** Assume that \( \pi \) is an optimal plan and that it does not result in an optimal run over the MDP with regards to expected-cost. Then there must be some sequence of actions that gives a better least expected cost for some sequence of states. Thus a plan can be produced by replacing \( \pi \)'s choice of actions for these states with the better actions, producing a new plan. This plan must be better, but \( \pi \) was optimal, thus we have a contradiction.

---

### 6.8 Finding the best plan

A probabilistic version of Dijkstra's algorithm, outlined in 4.4.1, is used to find the best plan for moving through the PA. As stated earlier in the problem description, the best plan is seen as the plan that minimizes the expected penalty for breaking rules, and additionally has the shortest expected length of all such plans. Lexicographic comparison is used to determine the transition cost from one state to another in the product automaton. Since the PA is in practice an MDP, probabilistic dijkstra can be run on it to get the optimal plan \( \pi \).

### 6.9 Using the plan

So far, we have created an optimal plan on the Product Automaton, but how is one to apply it on the original robot that motivates this whole study?

The authors of [**Tům+13**] prove that, in their version of the Product Automaton, the shortest run corresponds to the solution to their problem. Having
computed the shortest run in the PA, they can then project said run onto a run in the original map. Our case is different. Given the probabilistic nature of the problem, it is not useful to give a static path to follow. Depending on how chance turns out, the best course of action will be different. We must therefore present our solution as a plan. However, this makes the final projection step of Tům+13 impossible, since the best course of action in one particular map location might also be different, depending on how chance turned out several steps ago. For this reason, the solution to our problem is not only a plan, but a plan on the product automaton.

This is not a problem, but it impacts a possible implementation. Instead of having a fixed path, one must also have the product automaton in memory, and keep track of the state in it as the robot moves in the environment.

Nevertheless, any concrete run on the PA can be mapped onto a run through the MDP through a step-by-step process as detailed by lemma 2. Thus, use the process as defined in lemma 2 and define the function $P: P_{state} \times O \rightarrow P_{state}$ to be what happens in the environment as the robot progresses from the starting position to the final position and mimic the robot’s movements according to what happens in the MDP.

### 6.10 Motivation of correctness

According to [LaV], Probabilistic Dijkstra can be successfully applied if there exists some plan $\pi$ such that from any state $s_k \notin F_P$ in the map there is probability 1 that $G_\pi(s_k) > G_\pi(s_{k+1})$. In layman’s terms, every step in the plan should bring us closer to region B no matter what probabilistic outcome occurs.

Strictly speaking, this condition is not fulfilled in our product automaton, because there can be states where each transition out has weight $(0, 0)$. This means no matter how you go, $G_\pi(s_k) = G_\pi(s_{k+1})$. However, when the authors discuss this requirement, they specify that it ensures that the path one takes following the plan always makes monotonic progress towards the goal. This does hold true in our PA, because even if the weight change is zero, you are still moving closer to the target in the PA. Therefore, we will instead prove that from any state $s_k \notin F_P$ there is probability 1 that either $G_\pi(s_k) > G_\pi(s_{k+1})$, or $G_\pi(s_k) > G_\pi(s_{k+2})$.

**Theorem 6.** There exists a plan $\pi$ such that from any state $s_k \notin F_P$ in $P$ there is probability 1 that $G_\pi(s_k) > G_\pi(s_{k+1})$, or $G_\pi(s_k) > G_\pi(s_{k+2})$.

**Proof:** $G$ is computed over states $s_k \in Q_P, s_k \notin F_P$. There are two options. Either the state is a $?$-state, or a $!$-state.

- If $s_k$ is a $!$-state, then $s_{k+1}$ is a $?$-state. According to the definition of $P$, $l(s_k, u, \theta)$ will be of the form $(0, w)$ where $w$ is the travel length from the MDP. We assume that from every relevant part of the MDP, there is a way to the goal (if there were any isolated parts of the model, these might as well not be modelled). Since they represent geographical locations, we can also assume that a transition to a different state will always have a positive
weight. Therefore, we can let \( \pi(s_k) \) be the action that moves toward the goal. Since \( l(s_k, u, \theta) \) will be positive, we will have \( G_\pi(s_k) > G_\pi(s_{k+1}) \).

- If \( s_k \) is a \( ? \)-state, then \( s_{k+1} \) is a \( ! \)-state. We have already shown that for a \( ! \)-state \( s_{k+1} \), \( G_\pi(s_{k+1}) > G_\pi(s_{k+2}) \) holds. Therefore, \( G_\pi(s_k) > G_\pi(s_{k+2}) \) holds true.

\[ \square \]

6.11 Memory complexity of product automaton

Let \(|M|\) denote the size of the input MDP including Ghost States and \(|A|\) the size of the input automaton. The size of the extended automaton is \(|A| \times |AP|\) and so the size of the combined automaton is \( \Pi_{\psi \in \psi} |A_\psi| \).

The size of the PA \( P \) is \(|P| = |M| \times |A_\psi|\).

6.12 Algorithm

This pseudo-code describes the algorithm used for solving our problem.

**Data:** WLTS \( \mathcal{T} \), uncertainty function \( \hat{L} \), and security rules \( \psi \)
**Result:** Product automaton \( P \), optimized plan \( \pi \)

for \( \psi \) in \( \psi \) do
  Build automaton \( \mathcal{A} \) representing the rule;
  Extend \( \mathcal{A} \) into \( \mathcal{A}_\psi \);
end

Collect all constructed \( \mathcal{A}_\psi \) into \( \mathcal{A}_\psi \);

Construct product automaton \( P \) from \( \mathcal{T} \) and \( \mathcal{A}_\psi \);

Construct plan \( \pi \) using Probabilistic Dijkstra;

Return \( P \) and \( \pi \);

7 Analysis and discussion

A novel contribution in this thesis was the construction of a special Markov decision process (what we call introducing Ghost States) for modelling a changing environment. This allowed us to reason about more complex situations than in \cite{Tum+13} while leaving enough structure in the MDP for it to be usable in a Product Automaton.

Another novelty introduced in our thesis was the product automaton with \( ?/! \)-tags. This allowed for keeping the probabilistic part (the MDP) and the non-deterministic part (the NFA) separate in the same product automaton. The automaton can, as in the original paper, still be used for shortest-path-searching, albeit now with a plan as a result.
Our work requires the gathering of data to be able to construct the probabilities of events at locations. This might be restrictive in that this gathering is difficult to do (must be done by hand) or time-consuming. We think that the largest gains will be for many vehicles operating in the same area (such as delivery trucks) because the same PA can be used for every truck and many attempts are more likely to yield the exact probabilities measured.

7.1 Possible future work

7.1.1 More frugal PA definition

An issue with our solution is the need to store the product automaton (which may be quite large) while traversing the environment. Future work could look into a way of compressing the representation of plan and product automaton to allow for lower memory consumption. We believe there might be large gains to be had from this, as the formulation is currently made to be as simple mathematically as possible. In reality, a large percentage of transitions and states in the PA are unnecessary for the PA to be usable.

For example, one obvious optimization would be including intermediate states only where needed. These are necessary to resolve conflicts between the MDP and the NFA (see Section 6.6 for a detailed explanation), but only when there is a set of Ghost States involved. If large portions of a model are static, those portions could probably be modelled without intermediate states, like on the original PA, since there are no probabilities involved.

We believe there might be several other ways to modify the definitions in this thesis to lower the space and time complexities involved. We have not spent much time and effort on searching for optimisations.

7.1.2 Implementation and case studies

We consider an implementation to be viable future work. Possible case studies would include adapting the ones from [Tům+13]. Another area of research using the implementation would be measuring how long the PA takes to generate and at what scale it becomes useful to employ the PA for real-life situations.

8 Conclusions

We've shown a method for extending a natural way of representing a map (WLTS) into a restricted version of MDP to include the notion of uncertainty. We've shown a method for separating probabilistic (MDP) and non-deterministic (NFA) states and combining them to remain their semantics through a product automaton. We have also described a method for minimizing the expected cost of going from region A to region B in an uncertain map through the usage of off-the-shelf algorithms.
References


