Safe Reinforcement Learning Using Formally Verified Abstract Policies

George Rupert Mason

Doctor of Philosophy

University of York

Computer Science

February 2018

Abstract

Reinforcement learning (RL) is an artificial intelligence technique for finding optimal solutions for sequential decision-making problems modelled as Markov decision processes (MDPs). Objectives are represented as numerical rewards in the model where positive values represent achievements and negative values represent failures. An autonomous agent explores the model to locate rewards with the goal to learn behaviour which will cumulate the largest reward possible.

Despite RL successes in applications ranging from robotics and planning systems to sensing, it has so far had little appeal in mission- and safety-critical systems where unpredictable agent actions could lead to mission failure, risks to humans, itself or other systems, or violations of legal requirements. This is due to the difficulty of encoding non-trivial requirements of agent behaviour through rewards alone.

This thesis introduces *assured reinforcement learning* (ARL), a safe RL approach that restricts agent actions, during and after learning. This restriction is based on formally verified policies synthesised for a high-level, abstract MDP that models the safety-relevant aspects of the RL problem. The resulting actions form overall solutions whose properties satisfy strict safety and optimality requirements. Next, ARL with knowledge revision is introduced, allowing ARL to still be used if the initial knowledge for generating action constraints proves to be incorrect. Additionally, two case studies are introduced to test the efficacy of ARL: the first is an adaptation of the benchmark flag collection navigation task and the second is an assisted-living planning system. Finally, an architecture for runtime ARL is proposed to allow ARL to be utilised in real-time systems.

ARL is empirically evaluated and is shown to successfully satisfy strict safety and optimality requirements and, furthermore, with knowledge revision and action reuse, it can be successfully applied in environments where initial information may prove incomplete or incorrect.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my deepest gratitude to my supervisors Dr Radu Calinescu and Dr Daniel Kudenko for their invaluable support, help and advice throughout the entirety of this project. They have always made themselves available whenever I have needed help and have made great efforts to provide me with essential feedback at all stages of this project. Their expertise and patience has helped me in ways that includes and goes beyond becoming a good researcher.

I am very grateful to Dstl for funding this project, without which I could not possibly have pursued a PhD. In particular, I am very grateful to Dr Alec Banks who has continuously given me useful technical feedback and words of encouragement. I am especially appreciative of the significant effort he expended to allow me the opportunity to apply my research in a real-world setting.

I am also very thankful to my examiners Dr Suresh Manandhar and Prof Daniel Neagu whose insights and feedback helped strengthen this thesis. Additionally, I am thankful to Dr Marco Wiering for his feedback on my research and for his advice which helped me to complete this project.

Thank you to my colleagues in both the Artificial Intelligence and Enterprise Systems research groups with whom I had many enjoyable and interesting discussions which helped me determine a trajectory for my research.

Finally, I would like to thank my family and friends for their loyalty and support throughout this project, as well as their understanding during my absences when I have been too preoccupied with work to spend time with them. Specifically, I am especially grateful to my brother, Edward, whose diligence and desire to learn has been the inspiration that led me to start and finish this project.

Declaration

This thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree other than Doctor of Philosophy of the University of York. This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by explicit references.

Some of the material contained in this thesis has appeared in the following published papers:

- G. Mason, R. Calinescu, D. Kudenko, and A. Banks, 'Assurance in reinforcement learning using quantitative verification', in *Advances in Hybridization of Intelligent Methods: Models, Systems and Applications*, ser. Smart Innovation, Systems and Technologies, I. Hatzilygeroudis and V. Palade, Eds. Springer International Publishing AG, 2018, vol. 85, pp. 71–96.
- G. R. Mason, 'An autonomous system safety-monitor architecture using reinforcement learning and runtime quantitative verification', Defence Science and Technology Laboratory (Dstl), Platform Systems, Porton Down, Salisbury, Wiltshire, SP4 0JQ, Tech. Rep. DSTL/TR106266, December 2017, version 1.0.
- G. Mason, R. Calinescu, D. Kudenko, and A. Banks, 'Assured reinforcement learning for safety-critical applications', in *Doctoral Consortium on Agents and Artificial Intelligence (DCAART 2017)*. Porto, Portugal: SciTePress, February 2017, pp. 9–16, best PhD project award.
- G. Mason, R. Calinescu, D. Kudenko, and A. Banks, 'Assured reinforcement learning with formally verified abstract policies', in *Proceedings of the 9th International Conference on Agents and Artificial Intelligence (ICAART*

2017), vol. 2. Porto, Portugal: SciTePress, February 2017, pp. 105–117, best student paper award.

- G. Mason, R. Calinescu, D. Kudenko, and A. Banks, 'Combining reinforcement learning and quantitative verification for agent policy assurance', in *Proceedings of the 6th International Workshop on Combinations of Intelligent Methods and Applications (CIMA 2016)*, I. Hatzilygeroudis and V. Palade, Eds., The Hague, Holland, August 2016, pp. 45–52.

Chapter 1

Introduction

Computers are ubiquitous in the modern world and are present in most areas of society, ranging from home appliances and transport to healthcare and business. In recent years there has been particular interest in *artificial intelligence* (AI), which is generally defined as a computer-based system that can learn how to behave rationally, i.e. given an input the system will produce the 'best' possible output from the knowledge it has acquired [9].

The defining characteristic of AI techniques is their capability of *learning* sensible outputs, eliminating the need for a system designer to define outputs and to devise functions for every possible input which may be impractical to do for large and complex systems [10]. Furthermore, learning identifies an *optimal* output for each input, which may not even be known to the designer. This ability of AI has profound and far-reaching benefits that has seen its utilisation in an ever-increasing number of applications, including in areas such as robotics and vehicles [11, 12, 13], business and finance [14, 15], medicine and healthcare [16, 17], speech recognition [18, 19] and gaming [20, 21].

A significant challenge for AI is to learn behaviour that can be trusted as *safe* in situations where certain behaviour could risk catastrophic consequences [22]. This problem arises when it is difficult to express safety requirements as optimisation objectives which are neither too vague to ensure safety nor too restrictive to allow any useful behaviour at all [23]. Therefore, this thesis explores how strict safety requirements can be incorporated into an AI solution whilst still being useful.

1.1 Safe Reinforcement Learning

The field of AI research encompasses a wide variety of techniques. This thesis focusses on the subfield of *reinforcement learning* (RL), a popular approach which

simulates the learning mechanisms observed in living beings [2, 24, 25].

RL identifies an optimal set of outputs (henceforth referred to as *actions* or *behaviours*) for a system by using an artificial decision-maker, termed *agent*, to interact with the system environment [2]. In its interaction with the environment, the agent continuously receives feedback for its action choices: beneficial actions yield a reward and detrimental actions return a punishment. The agent retains knowledge of these rewards and punishments and by doing so learns about the quality of its action choices. Over time, the agent improves its behaviour to the extent that it will produce the largest expected cumulative reward possible from the system or, alternatively, incur the least expected amount of punishment.

There is a large body of research spanning many years with the goal to improving the efficiency and scalability of RL for its use with increasingly large and complex problems [26, 27, 28, 29, 30, 31, 32, 33]. These advancements and many more have shown RL to be a practical technique for solving a multitude of real-world problems, such as for control systems [34, 35, 36], gaming [37, 38] and robotics [39, 40, 41, 42], amongst others [43, 44, 45, 46, 47].

However, despite its continuing successes RL still has the major limitation that it lacks any guarantees that the behaviour learned by the agent can confidently be considered 'safe' where it is essential that it will not risk injury to humans, damage itself or other systems, cause financial loss or violate other regulatory or legal requirements. Furthermore, even though the agent may be acting within safety requirements, unpredictable or 'quirky' behaviour can make the system difficult to trust. This limitation prevents RL from being a viable choice in the class of safety-critical applications which mandate software certification [48, 49, 50, 51].

There has been growing interest to overcome this limitation and a variety of techniques have been proposed towards resolving it [52]. However, existing approaches are still largely theoretical, suffer from scalability issues and have difficulty when expressing non-trivial safety requirements. Furthermore, these approaches are still unable to provide firm guarantees that the RL solutions will satisfy strict safety requirements without unnecessarily and severely reducing the optimality of the solution. They may mitigate the problem, but do not resolve it.

The hypothesis of this thesis is that this limitation can be resolved through the use of *formal verification* techniques. Specifically, the use of *quantitative verification* (QV) [53] can provide assurances that an RL agent's behaviour will not violate strict safety requirements whilst maintaining levels of solution optimality. To this end, QV can be used to identify when a high-level action would cause the agent to violate its safety requirements and therefore disallow it, forcing the

agent to optimise over the remaining, safe actions.

1.2 Contributions

This thesis introduces *assured reinforcement learning* (ARL), a two-stage method for producing safe RL solutions which satisfy strict safety requirements and optimisation objectives. A high-level description of the ARL method and its individual elements is shown in Figure 1.1.

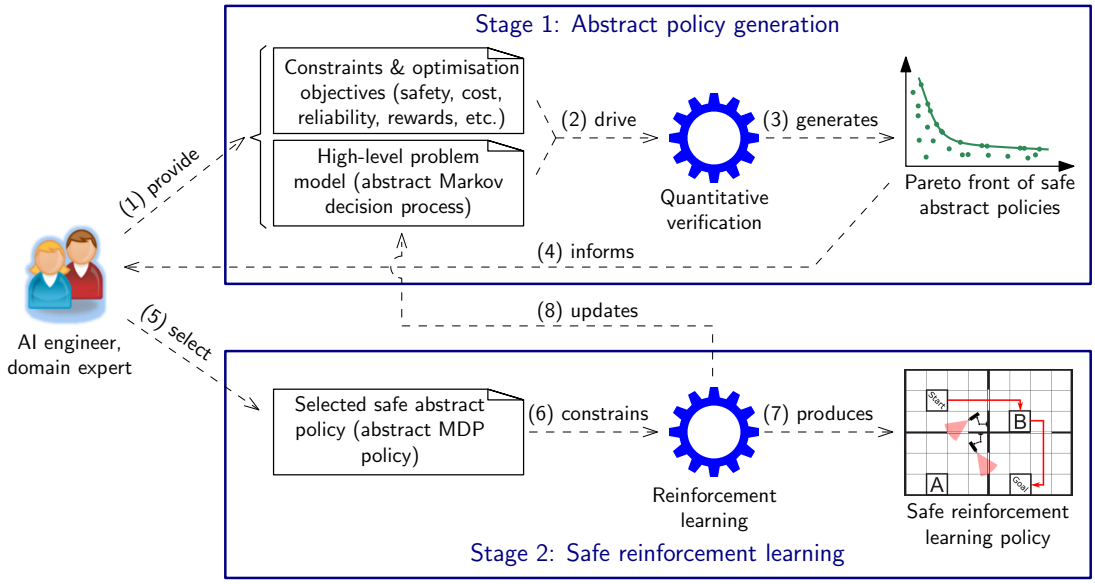


Figure 1.1: The two-stage method for assured reinforcement learning.

In Stage 1, the goal is to generate a set of high-level solutions, termed *abstract policies*, which have been formally verified as satisfying the safety and optimisation requirements. These safe abstract policies specify which high-level actions should, or should not, be done in each high-level state of the problem environment in order to satisfy the requirements. These abstract policies are used in Stage 2 to constrain the RL agent's optimisation of low-level actions in low-level states.

Accordingly, teams comprising both an AI engineer and a domain expert provide (1) the set of safety constraints and optimisation requirements and also an abstracted model of the RL environment. These are used to drive (2) the search for safe abstract policies using QV, a variant of model checking for the analysis and verification of stochastic models [53]. By exploring different areas of the abstract policy space, QV generates (3) a Pareto-optimal set of abstract policies which have been verified as satisfying all the safety constraints and optimisation

objectives. This Pareto front captures the safe abstract policies that are Pareto-optimal with respect to the optimisation criteria and can therefore be used to inform (4) the user's selection (5) of a suitable safe abstract policy.

In Stage 2 of ARL, termed *safe reinforcement learning*, the selected safe abstract policy is translated into a set of safety rules. These rules constrain (6) the RL agent's optimisation to low-level states and actions that map to the high-level states and actions of the abstract model, used in Stage 1, that are known to be safe at a high-level. As a result, the RL agent produces (7) a safe RL policy that when followed will have equal safety levels to those verified for the abstract policy, thus satisfying the safety requirements. If the RL policy does not satisfy the safety requirements then the abstract model used to generate the constraints must contain incomplete or inaccurate knowledge. Therefore, information collected about rewards and transitions of the RL environment is used to update (8) the abstract model, after which the process repeats until the RL policy is safe.

However, it is not guaranteed that a safe RL policy can always be found for every problem and set of safety/optimisation requirements. If it is not possible to satisfy all the requirements then no safe abstract policy can be generated and consequently no safety constraints for the RL process can be enforced.

The main contributions of this thesis are summarised below:

1. The ARL technique with specification of how an abstract model can be constructed for a problem and how abstract safe policies can be generated and verified. Additionally, guidance for how abstract policies can be used as action constraints for the RL agent.

2. An extension of ARL to incorporate *knowledge revision* into the safe RL process so that if the model used to create the initial safe abstract policies is not accurate it can be revised until it correctly reflects the RL environment.

3. A new algorithm for reusing previously optimised actions if they are unaffected by knowledge revision. This is to improve the efficiency of subsequent learning iterations during the knowledge revision process.

4. Two new case studies from two of the main classes of problems tackled by RL to be used for evaluating safe RL techniques. The first case study is a navigation problem based on the benchmark RL 'flag collection' problem [3], modified to incorporate a risk of the agent being captured. The second case study is a planning problem, adapted from an assisted-living system [54],

where the agent must learn the preferences of its user when guiding them to perform an everyday task.

5. An extensive evaluation of ARL, demonstrating its ability to successfully satisfy strict safety and optimisation requirements in each of the case studies.

6. A runtime architecture is proposed, showing how ARL can be used in real-time systems by continuously monitoring the problem environment and safety requirements. If new information or requirements are acquired the system will default to a known safe behaviour, albeit potentially suboptimal, until the RL agent has learned an updated safe solution.

1.3 Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 2 introduces the concepts, techniques and tools that are used to formulate the ARL approach. Specifically, Markov decision processes (MDPs) which form the framework of RL, the classical RL paradigm and algorithms, QV and automated model checking tools, and abstract MDPs (AMDPs) which are a high-level representation of the RL problem's MDP and are used for generating the safety constraints.

Chapter 3 gives an exposition on the state-of-the-art for safety in RL. This includes an analysis of the various approaches so far and their strengths and shortcomings, as well as a comparison of their abilities relative to ARL.

Chapter 4 introduces ARL. First, a detailed specification of the approach is given, showing how an AMDP is constructed, how safe abstract policies are synthesised and how these safe policies are applied. Next, ARL is evaluated in two qualitatively different case studies to assess how effective the approach is. Following this is a comparison of ARL against the alternative safe RL techniques discussed in Chapter 3. Finally, there is a discussion on the abilities and limitations of ARL.

Chapter 5 details ARL with knowledge revision (ARL-KR), an extension of ARL which allows an AMDP to be updated with new information should the initial model prove to be incorrect. Additionally, a new algorithm for action reuse is presented so that actions which have previously been optimised by the agent can be reused if they are still useful after knowledge revision of the AMDP

has occurred. Once again, the approach is validated through the two case studies and its capabilities and limitations are discussed.

Lastly, Chapter 6 summarises the contributions of this thesis, the limitations of the introduced techniques and potential areas for future work. This chapter also describes, under areas of future work, a promising preliminary exploration into the runtime use of ARL, carried out by the author of this thesis at the industrial sponsor of the project. Should an RL solution become unsafe during runtime, the technique uses an automatic function known to be safe as a substitute for the autonomous RL function whilst a new, safe RL solution is learned in the background.

Chapter 2

Background

This chapter specifies the main components which form the basis of ARL. Section 2.1 introduces Markov decision processes, which are used as the foundation for the reinforcement learning paradigm and whose properties can be formally verified using quantitative verification. Section 2.2 details the core elements of the classical reinforcement learning framework. Section 2.3 outlines quantitative verification, including its capabilities and how it can be utilised. Section 2.4 discusses abstract Markov decision processes which are key to making quantitative verification a feasible technique for assuring safety in reinforcement learning. Lastly, Section 2.5 summarises the key points from this chapter.

2.1 Markov Decision Processes

Markov decision processes (MDPs) [55] are a mathematical framework for modelling sequential decision-making processes whose behaviour exhibits stochasticity. MDPs comprise states which represent a unique status of the process, actions which produce transitions between states, transition probabilities for entering a new state after performing an action and rewards which denote the cost or gain for transitioning into a new state.

A state is a vector $s = (x_0, x_1, \ldots, x_n)$ where x_i, known as a state *feature*, represents some characteristic of the process. Possible examples of a process feature are temperature, units of energy, time, location and progress. Each state contains a unique set of values for these features and there is one state for every possible configuration of the process.

Rewards are used to steer the decision-maker towards objectives within an MDP, the magnitude of which reflects the importance of the objective. Rewards can be referred to as a *cost* if entering the new state is necessary but consumes some kind of limited resource (e.g. energy). Equally, the term *punishment* is

used for a reward if entering the state is detrimental. Typically, in both cases the reward is negative.

As an example, Figure 2.1 illustrates an MDP for a simple communication protocol. In this diagram, s_0 is the initialising state where the only action is to start, this causes transitioning to state s_1 with a probability of 1. In s_1 the decision-maker can either wait, costing -1 and with probability 1 remain in state s_1, or they can attempt to send a message. Sending a message has a probability of 0.01 for entering into the fail state s_2, where the only action is to restart, costing -10, so to transition back to state s_0 with a probability of 1. Alternatively, with a probability of 0.99, the message is delivered successfully, i.e. transitioning to state s_3. In this state the process stops with a probability of 1.

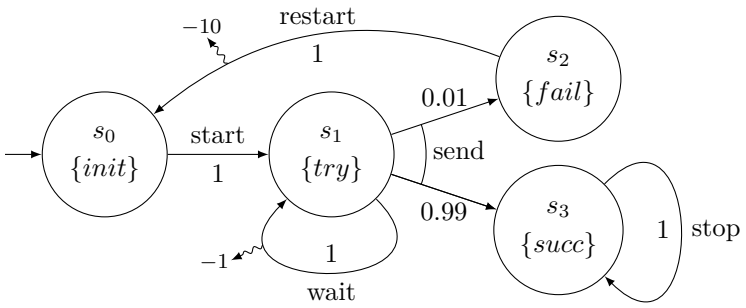


Figure 2.1: Directed graph representation of an MDP for a simple communication protocol, adapted from [1].

A solution for an MDP, known as a *policy*, defines the behaviour of the decision-maker such that for every state the policy will specify which action should be taken. The objective when solving an MDP is to identify a policy that when followed will cumulate the greatest gains/least costs possible from the process.

2.1.1 Definition

MDPs belong to the Markovian family of models which are defined by having the Markov property. This means that the next state of the process depends only on the current state and none of the preceding states [55].

Definition 2.1.1. (Markov Property): A Markov process is a stochastic process $\{X(t), 0 \leq t\}$ where at every time step $0 \leq t_0 < t_1 < \cdots < t_n < t_{n+1}$ and for the states $s_0, s_1, \ldots, s_n, s_{n+1}$, it holds that:

$$P[X(t_{n+1}) = s_{n+1} \mid X(t_n) = s_n, \ldots, X(t_0) = s_0] = P[X(t_{n+1}) = s_{n+1} \mid X(t_n) = s_n].$$

Formally, an MDP can be defined as follows [55]:

Definition 2.1.2. (Markov Decision Process): A Markov decision process M is a tuple $\langle S, A, T, R \rangle$ where:

- S is a finite set of states,
- A is a finite set of actions,
- $T : S \times A \times S \rightarrow [0, 1]$ is a state transition function such that for any $s, s' \in S$ and any action $a \in A$ allowed in state s, $T(s, a, s')$ gives the probability of transitioning to state s' when performing action a in state s,
- $R : S \times A \times S \rightarrow \mathbb{R}$ is a reward function such that $R(s, a, s') = r$ is the reward returned when action a performed in state s leads to state s'.

An MDP policy, denoted π, can take one of two forms. The first form is *deterministic* (which the remainder of this thesis will assume) meaning it will always return the same action a for a state s, according to some action selection policy, and is defined as $\pi : S \rightarrow A$. The second form is *stochastic* which maps each state s and action a to the probability that action a is taken in state s, where $\pi(s, a) = 0$ if action a is not possible in state s. A stochastic policy must satisfy $\sum_{a \in A} \pi(s, a) = 1$ for any $s \in S$.

A policy is called *optimal*, denoted, π^* if when followed will return the maximum expected cumulative reward from the process. This means that for every state, the action given by the policy is the one most beneficial with respect to achieving the overarching objectives.

2.1.2 Value Functions

The basis of algorithms for solving MDPs is the concept that states have a value which reflects the quality of transitioning to them. This brings rise to the notion of *value functions* [56]. Whilst the reward function returns the intrinsic value of entering a state, a value function returns the worth of a state when also considering the rewards of the future states reachable from that state when following a policy.

A state value function $V^\pi : S \rightarrow \mathbb{R}$ returns the expected cumulative reward when starting in a state s and following a policy π thereafter. This is defined as

$$V^\pi(s) = \sum_{i=0}^{\infty} r_{t+i} | s_t = s \tag{2.1}$$

where r_{t+i} is the reward for each individual state and i is an incremental time step. For models which have extremely large state spaces, in some cases infinite (known as *infinite horizon* models), a *discount factor* γ is incorporated into the function, where $\gamma \in [0, 1)$. This discount factor asymptotically diminishes the influence of rewards along a state path since, generally, the further into the future a reward is the less immediately important it will be in a solution. Larger values of γ add more weight to the rewards that are encountered in later states. Oppositely, smaller values will place importance on more immediate rewards. A discount factor of 1 reduces the function to Equation (2.1). Including the discount factor gives the discounted state value function

$$V^{\pi}(s) = \sum_{i=0}^{\infty} \gamma^{i} r_{t+i} | s_t = s \tag{2.2}$$

There are also state-action value functions $Q^{\pi} : S \times A \rightarrow \mathbb{R}$, known as *Q-functions*, which are similar, but differ as they return the expected reward for performing action a in state s under policy π, known as a *Q-value*. Q-functions are used for model-free algorithms, such as those discussed in Section 2.2. The Q-function is defined as

$$Q^{\pi}(s, a) = \sum_{i=0}^{\infty} r_{t+i} | s_t = s, a_t = a \tag{2.3}$$

and the discounted version as

$$Q^{\pi}(s, a) = \sum_{i=0}^{\infty} \gamma^{i} r_{t+i} | s_t = s, a_t = a \tag{2.4}$$

When all the dynamics of an MDP are known then an optimal policy can be identified using dynamic programming algorithms, such as value iteration [56] or policy iteration [57]. Instead, if the transition and/or reward functions are not known, then the *reinforcement learning* class of algorithms can be used.

2.2 Reinforcement Learning

The field of RL has been extensively researched for several decades, producing a wide variety of learning algorithms, agent exploration strategies, convergence optimisations, frameworks and model types. Therefore, this section will focus only on the classical RL approach. A detailed exposition of the numerous RL techniques can be found in [2, 25].

RL utilises an autonomous *agent*, an artificial decision-maker, to learn about the dynamics of an MDP. The agent can perceive the state it is currently in and accordingly choose an action to perform in it. The result of performing an action is the agent entering into a next state with some probability, although this may not necessarily be a different state, and also receives a reward, which may be zero. It is in this manner that the agent *explores* the MDP.

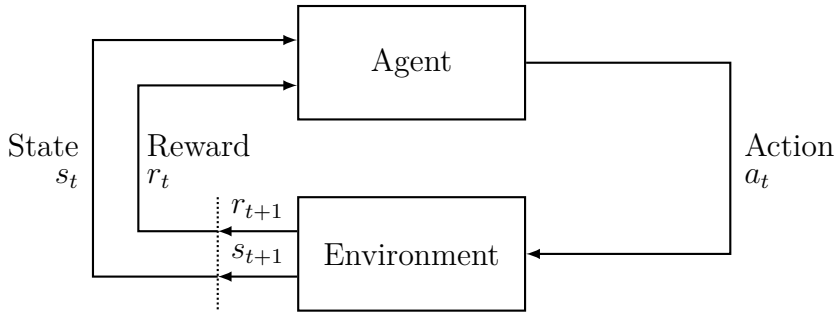


Figure 2.2: The interaction of an RL agent with an MDP environment [2].

The agent chooses an action to perform according to an action selection policy, such as the ϵ-greedy policy where the agent will select an action at random with probability ϵ and with probability $1 - \epsilon$ 'greedily' select the action associated with the highest Q-value for the state the agent is currently in, as shown in Equation (2.5).

$$a_t = \arg\max_{a \in A} Q(s_t, a) \tag{2.5}$$

The agent has no knowledge of the MDP as it begins its exploration. Therefore, its initial action choices will be arbitrary since it has not yet learned which actions are better than any others. Over time, through repeated interaction with the environment the agent will encounter rewards. Values for these rewards are stored as Q-values in a lookup table called a *Q-table*. At the start of learning Q-values are typically initialised arbitrarily, they are then iteratively updated using an update rule (detailed below) each time a state-action is sampled. With each application of the update rule the Q-values converge to those which will accurately reflect the quality of each action in a state. In this way, the agent *exploits* knowledge it gains by make increasingly better action choices when applying its action selection policy.

In episodic RL the overarching learning period is called a learning *run*. A run comprises a number of *episodes*, where an episode starts when the agent begins to explore and ends when it eventually enters into an absorbing state which it cannot transition out of (e.g. a 'success' or 'failure' state), or until a maximum

time period has elapsed. An episode itself is a series of individual *steps*, where at each step the agent can perform one action.

The number of steps and episodes required for learning to complete depends on the size of the environment, with larger environments typically requiring more learning steps and episodes than smaller ones. A recurring problem of MDPs experienced in RL is the *state explosion problem*, where the size of the model's state space increases exponentially as the number of states features increases, directly impacting the time taken to learn an optimal solution. Therefore, it is common practice in RL to terminate the learning run once the policy has become sufficiently optimal, and not necessarily completely optimal, since further learning episodes have diminishing returns [24].

Temporal-difference (TD) update algorithms are amongst the most common in RL and work by iteratively updating Q-values each time an action is performed in a state, thereby propagating knowledge of rewards across the state-actions which lead towards the reward. Two prominent TD algorithms are *Q-Learning* [58] and *SARSA* [59].

Q-learning

Q-learning finds an optimal policy through temporal differences by repeatedly sampling actions in each state in the system and updating their Q-values with the difference between the maximum expected return from future state and the Q-value of the current state-action pair. Q-learning is theoretically proven to converge to an optimal solution assuming that each state is visited an infinite number of times and the learning rate decays towards zero. The Q-learning update rule is defined as

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right], \tag{2.6}$$

where $\alpha \in (0, 1]$ is the learning rate which defines how significant a Q-value update is, r_{t+1} is the immediate reward received at time step $t+1$ after performing action a_t in state s_t and transitioning to s_{t+1}. If α is too large then learning may oscillate around an optima but not converge to it exactly. If it is too small then convergence will be inefficient. The Q-learning algorithm is shown below.

For Q-learning, the max operator is used to find the state-action pair which has the greatest expected return, this makes Q-learning an *off-policy* algorithm as it does not adhere to the current policy the agent is following, which may well be suboptimal.

Algorithm 1 Q-learning

```
Initialise Q-values
for each episode do
    Initialise state s
    while s is not terminal do
        Select action a for state s using action selection policy
        Perform a and observe reward r and new state s_{t+1}
        Q(s_t, a_t) ← Q(s_t, a_t) + α [r_{t+1} + γ max_a Q(s_{t+1}, a) − Q(s_t, a_t)]
        s ← s_{t+1}
    end while
end for
```

SARSA

The SARSA algorithm is another method for learning Q-values and is based on the Q-learning algorithm. The name stands for State-Action-Reward-State-Action, where s is the initial state, a is the selected action according to policy π, r is the reward for performing a in s, s' is the new state after performing action a and a' is the action to perform in s' according to policy π. The SARSA update formula is defined as

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right], \tag{2.7}$$

and the full algorithm is given below.

Algorithm 2 SARSA

```
Initialise Q-values
for each episode do
    Initialise state s
    Select action a for state s using action selection policy
    while s is not terminal do
        Perform a and observe reward r and new state s_{t+1}
        Select action a_{t+1} for state s_{t+1} using action selection policy
        Q(s_t, a_t) ← Q(s_t, a_t) + α [r_{t+1} + γ Q(s_{t+1}, a_{t+1}) − Q(s_t, a_t)]
        s ← s_{t+1}; a ← a_{t+1}
    end while
end for
```

SARSA is an *on-policy* algorithm since action values chosen during the update are those from the policy the agent is following. Herein lies the difference to Q-learning, which instead uses the max operator.

2.3 Quantitative Verification

This section details model checking which is a significant component of the ARL solution developed by this project. Discussion is limited to *stochastic* model checking, known as quantitative verification (QV) or, alternatively, probabilistic model checking [53], since the model to be verified (i.e. an RL MDP) is stochastic in nature.

QV is a formal technique for verifying the correctness, safety, reliability, optimality and other non-functional properties of stochastic systems. Examples of quantitative properties could include the probability of an event occurring, the cost of performing an action or the time required for a process to complete. QV has been successfully applied in a range of applications including cloud infrastructure [60] and service-based systems [61] to unmanned vehicles [62].

QV is achieved using a mathematical model of the system in the form of a Markov model and a formal specification of the system's quantitative properties to be verified in the form of a probabilistic temporal logic. Additionally, a model checker is used to automate the verification process.

A key feature of QV is that it exhaustively analyses the model's state space. This guarantees that the results of verification are accurate. When using QV to verify properties of MDPs it is necessary to first resolve the non-determinism of the MDP. This is done by a policy which represents a possible path of execution for the MDP by selecting one possible action in every state. Probabilities can only be determined for an individual policy so the combination of all the policies' probabilities is used to compute the minimum and maximum boundaries of a property holding.

2.3.1 Probabilistic Computation Tree Logic

Whilst the possible evolution of a system is represented using a Markov model, its properties are expressed using probabilistic temporal logic, a way of specifying properties over time. Examples of such properties are:

- *The probability that the system will not enter a 'retry' state before reaching a 'goal' state must be at least 0.99.*
- *What is the probability that the process completes within 100 time steps?*
- *What is the cost of entering into a 'success' state?*

Probabilistic computation tree logic (PCTL) [63], an extension of computation tree logic (CTL) [64], is one such logic. Whereas CTL is suitable for determining *qualitative* properties using the universal operator A and existential operator E, PCTL also allows *quantitative* properties to be verified since it has the probabilistic operator P. As with CTL, PCTL also uses the temporal operators X (next), F (eventually), G (globally) and U (until). Additionally, PCTL can be extended to include the R operator to calculate reward-based properties [65].

The PCTL syntax comprises state formulae Φ and path formulae ϕ, which are formally defined as:

Definition 2.3.1. (PCTL Grammar): State formulae Φ and path formulae ϕ are defined using Backus-Naur form as follows:

$$\Phi ::= \text{true} \mid \text{a} \mid \neg\Phi \mid \Phi \ \wedge \ \Phi \mid \text{A}\phi \mid \text{E}\phi \mid \text{P}_{\bowtie p}[\phi]$$
$$\phi ::= \text{X}\Phi \mid \text{F}\Phi \mid \text{G}\Phi \mid \Phi \ \text{U} \ \Phi \mid \Phi \ \text{U}^{\leq k} \ \Phi$$

where 'a' is an atomic proposition (i.e. a statement that is either true or false in each MDP state), $\bowtie \in \{<, \leq, \geq, >\}$, $p \in [0, 1]$ and $k \in \mathbb{N}$; and PCTL *reward state formulae* [53] are defined by the grammar:

$$\Phi ::= \text{R}_{\bowtie r}[\text{I}^{=k}] \mid \text{R}_{\bowtie r}[\text{C}^{\leq k}] \mid \text{R}_{\bowtie r}[\text{F}\Phi] \mid \text{R}_{\bowtie r}[\text{S}],$$

where $r \in \mathbb{R}_{\geq 0}$, I is an instantaneous reward, C is a cumulative reward and S is a steady-state reward.

State formulae include the logical operators $\wedge$ and $\neg$, which allow the formulation of disjunction ($\vee$) and implication ($\Rightarrow$). PCTL also has the following semantics [63]:

Definition 2.3.2. (PCTL Semantics): For a Markov model M, if a state formula Φ holds true for state s then $M, s \models \Phi$ is true. The following satisfaction relations can be inductively defined as:

$M, s \models \mathit{true}$ for all $s \in S$
$M, s \models a$ iff $a \in L(s)$
$M, s \models \neg\phi$ iff $M, s \not\models \phi$
$M, s \models \phi_1 \wedge \phi_2$ iff $M, s \models \phi_1$ and $M, s \models \phi_2$
$M, s \models \phi_1 \vee \phi_2$ iff $M, s \models \phi_1$ or $M, s \models \phi_2$
$M, s \models \phi_1 \Rightarrow \phi_2$ iff $M, s \models \phi_2$ whenever $M, s \models \phi_1$

where $L(s)$ is a labelling function that maps each state to the set of atomic propositions that hold in the state.

Lastly, the semantics of the PCTL operators are defined by:

Definition 2.3.3. (PCTL Operator Semantics): For a Markov model M, state s and infinite state path $\omega \in \text{Path}(s)$:

$$M, s \models \text{A}\phi \text{ iff } \omega \models \phi \text{ for all } \omega \in \text{Path}(s)$$
$$M, s \models \text{E}\phi \text{ iff } \omega \models \phi \text{ for some } \omega \in \text{Path}(s)$$
$$M, s \models \text{P}_{\bowtie \text{p}}[\phi] \text{ iff } Pr_s\{\omega \in \text{Path}(s) \mid \omega \models \phi\} \bowtie \text{p}$$
$$M, \omega \models \text{X}\Phi \text{ iff } \omega(1) \models \Phi$$
$$M, \omega \models \text{F}\Phi \text{ iff } \exists k \geq 0 \text{ such that } \omega(k) \models \Phi$$
$$M, \omega \models \text{G}\Phi \text{ iff } \forall i \geq 0 \text{ such that } \omega(i) \models \Phi$$
$$M, \omega \models \Phi_1 \text{U} \Phi_2 \text{ iff } \exists k \geq 0 \text{ such that } \omega(k) \models \Phi_2 \text{ and } \forall i < k . \omega(i) \models \Phi_1$$

where Pr_s is the probability of ϕ holding true starting from state s, $i, k \in \mathbb{N}$, and $\omega(i)$ denotes the i-th state of path ω.

This grammar allows PCTL formulae to be concise yet expressive. The three example properties listed at the beginning of this section can be formulated as PCTL respectively:

- $\text{P}_{\geq 0.99}$ [$\neg$*retry* U *goal*]
- $\text{P}_{=?}$ [$\text{F}^{\leq 100}$ *complete*]
- $\text{R}_{=?}$ [F *success*]

where *retry*, *goal*, *complete* and *success* are atomic propositions that hold true in states where it is necessary to reattempt an action, target areas have been reached, a process has finished and a process has been successful, respectively. The probability and reward bounds replaced with '=?' in the second and third properties indicate that the computation of the actual bound of the PCTL property is required. This can be ascertained using a probabilistic model checker supporting PCTL formulae.

2.3.2 PRISM Model Checker

It is infeasible to manually perform model checking on all but trivial models. Therefore, there exists a variety of model checking tools which efficiently perform

the process. Listed below are some of the more popular probabilistic model checkers with a brief overview of their capabilities:

- **MRMC** – The Markov Reward Model Checker (MRMC) [66] can verify discrete-time Markov chains (DTMCs) and continuous-time Markov chains (CTMCs) using PCTL or continuous stochastic logic (CSL) [67, 68, 69]. MRMC was developed with a focus on efficiency and dependability and achieves this through sparse transition matrices and graph analysis to reduce the number of states to check. MRMC is an explicit-state model checker and is command-line operated.

- **PRISM** – The Probabilistic Symbolic Model Checker (PRISM) [70] is primarily developed at the University of Oxford and the University of Birmingham. PRISM supports checking of DTMCs, MDPs, probabilistic automata, CTMCs, probabilistic timed automata and priced probabilistic timed automata. PRISM has several computation engines which allow models to be checked in a variety of ways (whilst not affecting the results). Supported temporal logics include PCTL, CSL and linear temporal logic [71]. PRISM features a user interface in addition to command-line usage.

- **Ymer** – Ymer [72] verifies the probabilistic transient properties of CTMCs and generalised semi-Markov processes. It uses statistical approaches and CSL to verify properties and makes use of PRISM's *hybrid* computation engine to allow numerical techniques to be used. Ymer is command-line operated.

- **VeStA** – VeStA [73] uses statistical techniques for the analysis of DTMCs and CTMCs using PCTL, CSL and quantitative temporal expressions. VeStA can be operated by both command-line and a user interface.

- **Storm** – Storm [74] is capable of analysing DTMCs, CTMCs, MDPs and Markov automata. Supported temporal logics include PCTL and CSL, extended with rewards. Storm can be run in three ways: command-line, a C++ API and a Python API.

In a comparative study done before Storm was released [75], PRISM proved to be one of the best probabilistic model checkers in terms of speed and efficiency and Ymer demonstrated as being one of the fastest for medium- to large-size models. However, since Ymer does not support MDPs it is not applicable for use in this project and, therefore, PRISM is the model checker that is used throughout

this project. Storm could be used as an alternative to PRISM; however, its development and introduction came when this project was in its mature stages so it has not been utilised.

The PRISM modelling language comprises modules and variables where a model may consist of multiple modules, which can interact with one another, and variables constitute system states. Modules are composed of commands with the form

$$\texttt{[action] guard -> } p_1\texttt{:update}_1 \texttt{ + ... + } p_n\texttt{:update}_n\texttt{;}$$

where `action` is optional to include. The inclusion of an action label allows transition rewards to be defined, as well as enabling multiple modules to synchronise by forcing transitions to occur simultaneously. `guard` is a boolean predicate over the variables, where the command whose guard matches the current state of the module can be executed. p_n represents a probability of a transition and subsequent variable $\texttt{update}_n$ to occur, where the sum of all probabilities in a command must equal to 1.

The example MDP for the simple communication protocol shown in Figure 2.1 has four states: the initial *init* state s_0, *try* s_1, *fail* s_2 and success *succ* s_3. There are five possible actions: start, wait, send, restart and stop. Also, there are two costs: -1 for waiting and -10 for restarting. This MDP can be expressed in the PRISM language as:

PRISM Language 1: Simple communication protocol

```
mdp

module communication_protocol
        s : [0..3] init 0; // 0 = init; 1 = try; 2 = fail; 3 = succ

        [start]   s=0 -> (s'=1);
        [wait]    s=1 -> (s'=1);
        [send]    s=1 -> 0.01:(s'=2) + 0.99:(s'=3);
        [restart] s=2 -> (s'=0);
        [stop]    s=3 -> (s'=3);
endmodule

rewards "process_costs"
        [wait]    true : 1;
        [restart] true : 10;
endrewards
```

Note that it is not necessary to specify an update probability in a command when the probability is equal to 1 (lines 6, 7, 9 and 10). Also note that reward structures in the PRISM language cannot contain a negative valued reward. Therefore, it is necessary to define multiple reward structures when a model contains both positive and negative rewards. This is done by giving each reward structure a label (in this example, `"process_costs"`), although, this label is not necessary to include if there is only one reward structure. In the reward structure of this code example, there is a reward (i.e. cost) of 1 whenever the `wait` transition occurs, and 10 for the `restart` transition.

Suppose for this MDP we wish to determine the cost incurred for successfully sending a message. Given the non-determinism present at state s_1, it is necessary to calculate the minimum and maximum costs; there is not a single, consistent cost incurred. Therefore, we formulate the following PCTL properties in PRISM format:

1. `R{"process_costs"}min=? [ F s=3 ]`
2. `R{"process_costs"}max=? [ F s=3 ]`

which when verified gives the minimum cost $0.\overline{10}$ (i.e. if the agent never waited and always attempt to send a message) and the maximum cost ∞ (i.e. if the agent were to constantly wait).

2.4 Abstract MDPs

Whilst using MDPs for RL problems is standard practice, the state spaces of most non-trivial problems are often many orders of magnitude in size (i.e. the state explosion problem). Although QV could be used to verify such an MDP, the time required to do so would be excessive since QV is a computationally expensive process due to its exhaustive analysis of the state space [76]. This renders it impractical to verify the RL MDP directly. Furthermore, due to the fact that an RL MDP may not be fully known, specifically its reward/transition functions, in such situations QV could not be used at all.

To overcome these problems this section introduces *abstract* MDPs (AMDPs) [77, 78] which can be constructed using limited knowledge of the problem environment and have a significant reduction in size relative to their MDP counterpart. This size reduction is achieved through state aggregation [79] which can be done by various methods, such as grouping states which have the same optimal actions and those which have similar rewards.

Furthermore, high-level policies of actions, known as *options*, can be obtained by sampling the environment dynamics [80]. Options conflate a series of individual actions into a single, all-encompassing one, further reducing the number of transitions in an MDP. For example, in a two-dimensional grid-like environment, to get from location A to location B may require several stepwise transitions (e.g. move North, North, West, West, ...), whereas the equivalent option in an AMDP would be a single transition defined as A *moveTo* B.

An AMDP functions in the same way as an ordinary MDP and can therefore be solved using the same techniques. Using Definition 2.1.2 for an MDP, an AMDP can be formally defined as:

Definition 2.4.1. (Abstract MDP): An AMDP $\bar{M}$ can be defined as a tuple $\langle \bar{S}, \bar{A}, \bar{T}, \bar{R} \rangle$:

- $\bar{S} = \bar{s}(S)$,
- $\bar{A} = \bar{a}(A)$,
- $\bar{T}(\bar{s}, \bar{a}, \bar{s}') = \sum_{s \in \bar{s}} w_s \sum_{s' \in \bar{s}'} T(s, \bar{a}, s')$,
- $\bar{R}(\bar{s}, \bar{a}) = \sum_{s \in \bar{s}} w_s | R(s, \bar{a})$,

where $\bar{s}(S)$ is the abstraction function of the state space S and w_s is the weighting of a state s based on its expected frequency of occurrence in an abstract state [77].

AMDPs have been used in [77] to provide guidance to an RL agent by solving the AMDP and using the resulting value function for reward shaping [81]. This technique allows using rewards other than those from the MDP reward function to help the agent advance towards relevant states and expend less time exploring the irrelevant ones. In [78], the use of AMDPs for reward shaping was extended to the multi-agent RL paradigm [25].

2.5 Summary

This chapter has introduced the concepts of Markov decision processes (MDPs), reinforcement learning (RL), quantitative verification (QV) and abstract MDPs (AMDPs). These technologies form the basis of the assured RL method that this project has developed. The key points of each technology are summarised below:

- MDPs are used to model sequential decision-making processes which are characterised by stochasticity. An MDP comprises states, actions, transitions and rewards. A solution to an MDP is called a policy, where a policy defines the actions to perform in each state of the system. An optimal policy is one which when followed will yield the maximum possible expected reward from the system.

- RL is a family of techniques used to solve MDPs when the reward and/or transition functions are unknown. RL uses an autonomous agent to explore a model to learn about its dynamics. Temporal difference algorithms are commonly used to propagate knowledge of rewards across all other states of the system. In this way, the agent learns which actions will lead it towards rewards in the system.

- QV is a formal technique for verifying the non-functional properties of stochastic systems. To do this the system is modelled as a Markov model (e.g. an MDP) and the properties to be verified are specified using a temporal logic (e.g. PCTL). QV exhaustively analyses the state space of the model to give results that are mathematically guaranteed to be correct.

- AMDPs are a condensed form of an MDP. Similar states are conflated to reduce the size of the model's state space and actions are represented as high-level options in order to reduce the number of transitions in the model. AMDPs can be vastly smaller than their MDP counterparts so can be solved and reasoned about proportionally faster.

Chapter 3

Related Work

This chapter discusses other significant research towards safety in RL, detailing the capabilities of each approach, how they are achieved and the practicality of them, as well as their limitations. Following this discussion is a comparison of each technique to the assured reinforcement learning with knowledge revision (ARL-KR) technique developed by this project.

Whilst research towards safe RL has been a continuous process over a number of years, this chapter limits discussion to the more distinct techniques that have emerged. An all-encompassing taxonomy of safe RL and related research throughout the years can be found in [52, 82].

3.1 Defining Safety, Risk and Optimality

Safety is a term which is often defined on an ad hoc basis in RL literature, where an RL technique is considered 'safe' with respect to some specific criteria within a specific problem environment and so may not necessarily be safe when applied in a different problem domain. Therefore, in the context of this thesis we adopt the definition of safety from formal verification/model checking of systems, that being, under certain circumstances an undesirable event will never occur [83]. Broadly speaking, these undesirable events encompass situations such as where the agent has caused damage to itself or other systems, induced harm to humans, or violated legal or ethical requirements.

Supplemental to this definition of safety is a definition for *risk*. The standard definition of risk is the *likelihood* of an unintended event occurring (e.g. to behave unsafely) combined with the *consequence* of that event taking place [84]. When discussing risk in this thesis it is implicitly assumed that the consequence of such an event is always the same, i.e. the safe RL agent has failed its mission since one of its core objectives is to *not* produce unsafe behaviour. Therefore, through-

out this thesis the term 'risk' is used to mean the probability, or likelihood, of safety violations. For example, a *high risk* solution is one where there is a high probability of the agent behaving unsafely.

A final definition relevant to discussion in this chapter is that of *optimality*. In RL, the term optimality is defined as the difference between the expected reward returned by a solution and the *maximum possible* expected reward from the RL MDP [24]. A solution is termed 'optimal' if it can expect to return the maximum possible reward from the system, otherwise it is termed 'suboptimal'.

In the context of safe RL, however, a safe solution is not necessarily an optimal one. Often, allowances must be made for reduced optimality of a solution since safety is often achieved at the expense of optimality. The most optimal behaviour with respect to completing a task may not be safe and to increase the solution optimality would be to increase the risk of being unsafe.

A tangible, real-world example of this could be when aiming to minimise the materials used in the construction of some kind of safety apparatus, such as a motorcycle crash helmet, with the goal to minimise costs. Assuming that the most appropriate material for the job is used to fabricate it, the most cost efficient solution would be to make the helmet as thin as possible for it to still retain its shape, i.e. minimising the amount of material used. However, doing so would also minimise the helmet's effectiveness at preventing injuries occurring to the wearer's head: the optimal solution is not a safe one. It is therefore necessary to allow the increased cost for extra material, decreasing the optimality of the solution, so to increase the safety provided by the helmet.

Considering this, in the context of *safe* RL we expand the standard RL definition of optimality to be the difference between the expected reward returned by a safe solution and the maximum possible expected reward that could be achieved *within the threshold of satisfying safety*. In other words, an optimal safe solution will return the maximum expected reward possible within the safety boundaries, whereas a suboptimal safe solution is one which returns an avoidably diminished expected reward, albeit whilst also being safe.

3.2 Safety in Reinforcement Learning

The mechanisms that can result in the RL agent entering into undesirable states (both during and after learning) include the criteria for how the agent optimises its action choices and how the agent explores its environment during learning.

Since actions are optimised with the goal to maximise the cumulation of

rewards defining some set of objectives, the agent is unconcerned about side-effects that may occur as a result of performing an action if they are not a part of the reward signal. As an example, if a simply defined objective for an agent driving an ambulance is to transport a critically injured patient to the hospital as quickly possible then the agent may optimise a solution to take the shortest route and drive as fast as legally possible. However, such a solution will not consider factors such as if jerky motions by the ambulance abruptly stopping and accelerating could aggravate the patient's state, or if certain roads along the chosen route are high-risk crash zones which should be traversed cautiously. Since these aspects do not feature in the reward signal and, in fact, would directly contradict the main objective, making it difficult or even infeasible to define them through a reward structure alone, the agent will not learn behaviour that is safe, despite it being optimal for quickly reaching the hospital.

The agent's exploration strategy, defined by an action selection policy such as ϵ-greedy (detailed in Chapter 2.2), can lead to entering unsafe states during the learning process. For example, the ϵ-greedy policy indiscriminately selects a random action with probability ϵ and by doing so can unwittingly enter an unsafe state when it essential that the agent must remain safe at all times (e.g. during online learning). Relating to the optimisation problem above and the traditional definition of optimality in RL as described in the previous section, this exploration strategy can also make the agent discover actions which are optimal with respect to the reward scheme but are still unsafe and should be prevented, even at the cost of some optimality of the final policy learned.

Given these two primary causes of safety violations in RL, existing approaches for safe RL generally fall into one of two categories, as outlined in [52]. One is to modify how the agent *optimises* a solution, the other is to modify how the agent *explores* the state space.

3.2.1 Safe Optimisation Techniques

An intuitive approach to instil safe behaviour into the agent is to assign a negative reward to those actions in states which will cause transitioning to unsafe states. However, this simplistic approach suffers from several problems. First, it requires knowing a priori exactly which states, or sequences of states, of the RL environment are unsafe, which may not be easy to identify. Second, assigning a cost of suitable magnitude to every unsafe state is not always obvious and can require extensive trial and error to determine them, which becomes impractical when

dealing with a large numbers of unsafe states [85]. This problem is compounded by the fact that the safety rewards may directly conflict with the optimisation objectives, meaning the agent may attempt to minimise the costs relating to safety in favour of maximising the objective rewards and consequently fail to achieve any of the required optimisation objectives. Third, it can be difficult to define complex safety requirements as a system of rewards, especially without significantly expanding the state space of the model to accommodate state features required for the safety properties. Therefore, instead of focussing on how to define rewards for unsafe behaviour, various approaches have been proposed that consider the criteria for how the accumulation of rewards is optimised.

Ergodic policies. One approach is to optimise *ergodic* solutions, i.e. a solution where every state is reachable from all other states which feature in the solution. Whilst this concept guarantees that a solution will never lead to the agent behaving unsafely, since it will not enter a state from which it cannot recover, the solution is often excessively far from being optimal since real-world problems rarely allow an ergodic solution to be useful. Despite the fact that significant rewards could be gained at *very* low risk of entering a non-recoverable state, as the risk is non-zero the rewards are not considered during optimisation.

An approach to compromise between safety and optimality is [86] which optimises a δ-*safe* policy, where with user-defined probability δ each state of the policy is ergodic. However, enforcing the ergodic safety constraints during learning is NP-hard, preventing the technique from being utilised beyond problem scenarios with small state spaces, and whilst approximate constraints can be learned more efficiently, they can result in suboptimal policies (albeit still δ-safe) [82]. Additionally, as noted in [82], an unrecoverable state does not necessarily mean that it is unsafe and preventing the agent from entering such a state to keep a policy ergodic can unnecessarily reduce the policy's optimality, or even prevent the agent from succeeding with its mission.

Permissive schedulers. In [87], probabilistic model checking is used directly on the RL MDP to identify a set of safe policies, termed *permissive schedulers*, which satisfy a set of safety requirements specified using temporal logic. These are used to constrain the RL agent so that it optimises a solution within the set of states that have been verified as safe. The technique is used to allow an RL agent to safely explore an MDP to learn its reward function. Therefore, permissive schedulers contain only conservative optimisation properties from any partial knowledge of rewards contained in the MDP.

This use of probabilistic model checking to verify that certain states of the MDP are safe to visit assumes full knowledge of the MDP's transition function, the applicability of the approach is therefore limited since RL is typically used to learn about an MDP's reward structure *and* its transition function. The approach suffers from the additional problem of scalability, since model checking the entire MDP state space is a computationally expensive process and becomes infeasible for MDPs of the size often encountered in RL.

Worst-case criterion. The *$\hat{Q}$-learning* algorithm [88] is a modification of Q-learning [58] and uses the minimax criterion to optimise a solution. The nature of the algorithm is to assume that whatever can go wrong *will* go wrong and so optimises the actions to be the best they can be for a worst-case situation. The resulting solution has the property that the worst-case outcome from it is at least as good as/better than all other possible solutions' worst-case outcomes. To this end, $\hat{\text{Q}}$-values are updated by the update rule

$$\hat{Q}(s_t, a_t) \leftarrow \max\left[\hat{Q}(s_t, a_t), r_{t+1} + \gamma \min_a \hat{Q}(s_{t+1}, a)\right], \tag{3.1}$$

which can be reformulated as [89]

$$\hat{Q}(s_t, a_t) \leftarrow \min\left[\hat{Q}(s_t, a_t), r_{t+1} + \gamma \max_a \hat{Q}(s_{t+1}, a)\right]. \tag{3.2}$$

This approach guarantees that the learned policy will always achieve some minimum reward and potentially can expect a greater reward to be returned. However, in practice, policies learned through $\hat{\text{Q}}$-learning typically yield drastically reduced expected rewards than could otherwise be achieved [52, 90]. Even though the likelihood of a worst-case scenario occurring may be *very* low, this optimisation strategy can disregard large future rewards. Furthermore, this approach relies on defining safety properties through a system of rewards, which as discussed above, is not always a feasible tactic.

Risk-sensitive criterion. Another approach is the *risk-sensitive* optimisation criterion [91]. The standard optimisation approach used by RL is to maximise the expected reward cumulated by the agent from the system. However, this does not necessarily translate well to the agent learning a solution that can be relied on to be safe, if safety features are to be defined using rewards. Even though the learned solution may cumulate the largest expected reward possible from the system, suggestive that the agent is behaving in the safest way possible, this may not necessary reflect reality, since the expected reward is *on average*.

The reality may be that there is high variance of the expected reward, where on some occasions the agent may succeed in cumulating a large reward without incurring failures, but on other occasions it *does* fail and receives a large punishment instead. Even though on average the agent may return a satisfactorily large expected reward, the variance of the reward indicates that the solution involves a high level of risk with respect to the agent behaving unsafely.

Therefore, the risk-sensitive optimisation aims to learn a solution which has a low variance for the expected reward, albeit one that potentially is less on average than what is possible to achieve from the system. This is achieved in [92] by using a parameter $k \in (-1, 1)$ to specify what amount of variability of the expected return is permissible. This parameter can be tuned so that a solution is optimised to either avoid variability or seek it by using the transformation function χ^k, defined as

$$\chi^k : x \mapsto \begin{cases} (1-k)x & \text{if } x > 0 \\ (1+k)x & \text{otherwise} \end{cases}, \tag{3.3}$$

to weight positive and negative temporal differences x appropriately. When seeking risk (k is *negative*), negative temporal differences are underweighted and positive differences are overweighted. When avoiding risk (k is *positive*), the weighting occurs oppositely.

Through this approach a solution can be found that satisfies the level of risk that the user is comfortable with. However, as with the worst-case criterion, low variability can mean excessively limiting the agent from performing profitable behaviours if there is even a low risk of unsafe behaviour [52]. Furthermore, the approach may be unable to satisfy all risk levels for multiple safety requirements.

3.2.2 Safe Exploration Strategies

Traditionally, an RL agent starts with no knowledge of the environment and must initially explore it randomly, potentially leading to the agent finding solutions which involve transitioning to unsafe states. To ameliorate this problem, the exploration strategy of the agent can be influenced so that it has some knowledge of which states to transition into and which to avoid.

Teacher knowledge. This concept involves having a teacher provide useful knowledge to the RL agent to influence its decision-making as it explores [93]. By this method, the agent can be informed how to act when in unsafe situations

where it may not be apparent to the agent how to behave safely.

One such example is the *Policy Improvement through Safe Reinforcement Learning* (PI-SRL) algorithm [94], a two-stage process. The first stage is to define a safe baseline behaviour (assumed to be suboptimal) from teacher knowledge and the second stage is for the RL agent to optimise over it. The approach uses a risk function to determine how similar the next state the agent will visit is relative to previous safe states it has visited, such as those featured in the baseline behaviour. A parameter is used to tune how the level of risk is defined, where if the similarity index of the next state is below the risk threshold then the state is considered safe, whereas it is deemed unsafe if it is above the threshold.

However, such a parameter may not accurately reflect safety across all states, where a next-state may evaluate as being below the risk threshold, and is therefore considered safe, but is actually unsafe. Furthermore, if the next state is significantly different to any previously explored states, its similarly index will exceed the threshold and is categorised as unsafe, even though the state may very well be safe.

Cautious simulation. A *cautious simulator* is used in [95, 96] to guarantee that the RL agent never explores an unsafe state during online learning for a physical system. The simulator undertakes simple physics simulations to identify safe states which correlate to states in the real-world, as well as incorporating a set of safe trajectories through the state space which are identified by an experienced human operator. Based on these simulations and safe trajectories the simulator extrapolates a *safety function* which classifies states in the real-world state space as either *safe* or *unsafe*. This process is 'cautious' since it can incorrectly identify an otherwise safe state as unsafe (which the human operator can later correct), however, it will not classify an unsafe state as safe.

A modified RL algorithm is then used to optimise within the safe states identified by the safety function. However, the authors note that the technique of enforcing the safety limits can cause RL to perform inefficiently and, furthermore, cannot guarantee that the final solution is optimal.

Safe demonstrations. The technique in [97] has an RL agent learn its behaviour from a set of demonstrations. The technique was developed as a general means to enable an RL agent to learn complex behaviour and not necessarily to instil safety. Nevertheless, the technique is well-suited to be applied when learning safe behaviour and is therefore a valuable addition to the family of safe RL techniques.

The approach uses human demonstrations of how to perform a behaviour to construct a model of the problem environment; the demonstrations are used to define the dynamics of the model and to derive a suitable reward function. From this model an RL agent can optimise a solution, thereby mimicking the behaviour from the human demonstrations.

The drawback of this technique, though, is that the optimality of the RL solution is limited to how well the human is able to perform the demonstration [82]. Furthermore, if the agent encounters states where no demonstration exists which the agent can mimic then it may resort to actions which are unsafe [52].

Backup policies. The work presented in [98] introduces the *Safety Handling Exploration with Risk Perception Algorithm* (SHERPA) which provides the RL agent with a *backup* policy, representing an 'escape route' that the agent can resort to if its exploration strays too close to predefined unsafe states.

At each state the agent enters, the next action chosen by the agent is evaluated for its risk of entering an unsafe state. This evaluation is achieved by means of a 'risk perception' function which measures the features of a state against some predefined threshold of safety. For example, if the states of a temperature control system include 'heat' as a feature, should a next state have the system reach the overheating threshold then the risk function would deem the action leading to the state as risky and prevent the agent from performing it.

If an action will cause transitioning to unsafe states, then an alternative action is chosen. If the action does not lead to an unsafe state then a search for a safe backup for the next state begins. A backup is used to take the agent back to a nearby safe state if no future states from the next state can be determined as safe. In the event that there are no safe backups in the next state then the action is once again discarded and a new action is evaluated. If all actions are assessed but none have safe backups, the backup for the current state is used, taking the agent back to a previous state where it can learn a new action for that state which will lead to a different future state.

In this way, the agent will never perform an action directly leading to an unsafe state, or enter a state from where it inevitably will enter an unsafe state. However, this relies on correctly and completely identifying the unsafe states in the system by means of an effective risk perception function. Furthermore, the approach may result in suboptimal policies if the agent behaves too cautiously by avoiding any state for which there is no backup, even though such a state may not inevitably lead to an unsafe state but could instead lead to a reward. As

with ergodic policies, an unrecoverable state is not necessarily an unsafe one.

3.3 Comparison to ARL-KR

ARL-KR can be categorised as a safe optimisation technique since the optimisation of a policy is restricted to the set of states that have been verified as safe at a high-level. In contrast to safe exploration strategies, the underlying exploration by the agent is unaffected and can be done using a policy such as ϵ-greedy.

The following sections discuss several areas of comparison between ARL-KR and the techniques described in the previous sections.

3.3.1 Requirements for Safety

The core ARL technique (Chapter 4) requires accurate abstractions of all safety- and optimality-relevant states in the form of an AMDP. Knowledge of the specific rewards and transition probabilities associated with the states is preferable as it will minimise the time for a safe solution to be reached, but the knowledge revision extension to ARL (Chapter 5) allows this information to be acquired automatically at the cost of extra computation time. In addition to the AMDP, the safety and optimisation requirements are needed in the form of PCTL formulae.

In contrast, the ergodic algorithm in [86] only requires a user-defined safety level in the form of a probability. Similarly, the risk-sensitive approach in [92] requires a user-defined scalar parameter to define the level of risk that a solution should involve. However, risk-sensitive, as well as with the worst-case criterion [88], requires the incorporation of unsafe (sequences of) states into the reward function, such that there is a cost for entering into them. In addition to identifying these states, costs of suitable magnitude must be determined which is not necessarily feasible. Identifying unsafe states is also the core requirement of the backup policies strategy [98], although, this technique does not require them to be included into the reward structure.

Similar to ARL, the use of permissive schedulers [87] requires a set of properties defined using temporal logic. Differing to ARL, though, an AMDP is not used. Instead, full knowledge of the MDP's transition function is required which is not typically available for RL tasks.

The teacher knowledge approach in [94], the cautious simulator from [95, 96] and the safety through demonstrations tactic [97] all require skilled and experi-

enced human involvement to identify safe solutions. In addition to this, [95, 96] requires a physics simulator of the problem environment and [97] requires a method to translate demonstration data into an MDP for RL.

Compared to some of the other safe optimisation techniques, ARL-KR requires a moderate amount of domain knowledge to be successful; however, it has the significant advantage over most of these approaches, as well as the backup policies exploration strategy, that it does not require knowledge of each low-level unsafe state. Furthermore, ARL-KR does not require modifying the underlying RL MDP to include unsafe states into the reward structure which, as described in Section 3.2, is not always feasible. Compared to safe exploration strategies, with the exception of backup policies, ARL-KR enjoys the major advantage that it does not require human involvement to identify and provide safe actions to the agent.

3.3.2 Knowledge Revision Capability

As is discussed in depth in Chapter 5, safety is subject to how accurate the knowledge of the problem is. Should information about features that can affect safety be incorrect or incomplete, the safety levels that are intended may not be met.

This potential problem is significant for ARL, where safety guarantees can only be provided assuming complete and correct knowledge of the features related to safety are included in the AMDP. The extension of ARL with knowledge revision (ARL-KR) resolves this problem to a large extent. Should it be the case that the AMDP is not accurate to the RL MDP, ARL-KR provides a method to update the AMDP with accurate observations of transitions and rewards in the AMDP. Furthermore, subsequent learning runs for newly generated safe abstract policies can be achieved more efficiently by reusing actions (where possible) that have previously been optimised. A limitation of ARL-KR is that it is currently not possible to update the AMDP to incorporate new states that were previously unknown, since this can require significant restructuring of the AMDP which would require manual intervention by a domain expert.

Knowledge revision is not a problem that affects [86, 87], since these approaches generate solutions directly on the RL MDP which itself does not require modification for the process. Similarly, [97] does not face the problem of knowledge revision since the MDP is induced from the demonstrations.

The approaches from [88, 92, 98] share the common task of identifying unsafe

states so that safety can be imparted, this identification is not done automatically and unsafe states must be manually identified. This is a significant limitation since if the end RL solution produces behaviour that is less safe than intended, meaning that there exists some unsafe states that have not been accounted for, it may prove difficult to identify the states if they are not obvious and hence were not identified from the beginning. This same problem is seen in [94] since safety is based around the similarity of states to those identified by the teacher knowledge: if a state is considered similarly safe to a teacher-defined safe state, but it is in fact not safe, it may not be easy to identify which aspect of the teacher knowledge produced a safe state that was similar to an unsafe state.

Whilst [95, 96] also rely on knowledge of the safety of states, the cautious nature of this approach to identify only *safe* states, as opposed to *unsafe* states, means that in a worst case the agent will behave in an overly conservative manner. In this case, the safety function produced by the simulator allows manual intervention to correct any states mislabelled as unsafe.

3.3.3 Effectiveness at Achieving Safety

A key feature of ARL-KR is its use of *formal verification* to produce RL solutions that are *guaranteed* to satisfy a strict set of safety requirements. Importantly, ARL allows a wide range of specific safety and optimisation requirements to be satisfied by defining them using PCTL, an expressive temporal logic.

By comparison, the approaches in [86, 88, 92, 98] do not guarantee that specific safety properties will be part of the RL solution. Instead, they aim to reduce the likelihood of unsafe behaviour so that the agent's behaviour is *generally* safer. Whilst it may be possible in some circumstances to learn a set of behaviours that are entirely safe, this will typically involve severe restrictions on the agent by preventing it from entering states which are perfectly safe.

The use of formal verification in [87] to identify safe constraints on the optimisation process, and temporal logic to define properties, is equivalent in effect to that of ARL; although, how the constraints are generated is done quite differently. Whilst [87] can produce verifiable safe RL solutions for a broad range of properties, as is discussed in a following section, this capability is limited to small sized problems due to the computational expense of verifying an MDP directly.

By learning from teacher knowledge [93, 94], demonstrations [97] and a cautious simulator [95, 96] it is possible to instil complex and specific safe behaviour into the agent's solution. However, with the exception of [95, 96], these techniques

do not necessarily guarantee safety; instead, they produce a solution that is only as safe as the teacher or demonstrations. Furthermore, in unfamiliar situations where the agent has no safe demonstration to learn from, or teacher knowledge is incomplete, the agent can still perform unsafely.

With the exception of [87] (which can only handle small RL problems with known transition functions), approaches for safe RL tend to suffer from one of two problems: either they lack any guarantee that specific safety levels can be met and/or do not allow complex safety requirements to be satisfied. The ARL approach presented in this thesis, however, does not have either of these limitations.

3.3.4 Impact on Optimality

With ARL it is possible to incorporate optimisation requirements in the same way as safety requirements. Therefore, a set of Pareto-optimal constraints can be generated allowing a user to choose a solution for their preferred level of compromise between optimality and safety. The level of optimality achieved through ARL can be limited by the abstract policy synthesis stage, where other than through an exhaustive search (which may not be feasible if the space of abstract policies is very large) a search heuristic is used to identify an *approximate* Pareto front of abstract policies. Therefore, it may be the case that a safe policy exists which can allow greater levels of optimisation but it was not found during the search. The underlying learning process by the agent has the standard optimality guarantee of RL that provided sufficient learning has occurred (i.e. each state-action has been sampled a sufficient number of times), the agent will learn an optimal behaviour within the safety constraints.

The techniques in [86, 88, 92] can significantly reduce the optimality of a solution, in excess of what is necessary in order to achieve safety. Since the techniques make only crude (or no) discrimination between individual safe/unsafe states and actions, it is possible for safe actions and states to be restricted. This can result in a significantly suboptimal policy relative to what could be achieved for the same, or better, levels of safety. This can mean that although a solution is safe, its usefulness is significantly limited. Similarly, [98] may cause ignoring profitable states if they are deemed unsafe, even though they may be safe, creating the same problem that significant optimality gains can be ignored.

Permissive schedulers [87] can allow specific optimisation objectives to be defined in the same way as safety objectives. Although, like ARL, the optimality

of the solution is subject to the search heuristic used to identify safe schedulers, as well as how complete the knowledge of rewards is prior to learning.

As noted by the authors in [95, 96], to optimise a solution using cautious simulations is difficult to achieve efficiently. Whilst they suggest that some level of optimality can be achieved by the agent simply by assigning a reward of $-\infty$ to unsafe states, they do not guarantee that the resulting policy will be optimal.

With [94, 97], the level of optimality that can be achieved is limited to how optimal the teacher advice/demonstrations are. This means that although the resulting solution provides satisfactory optimality, solutions with greater optimality may exist that were not known by the teacher/demonstrator.

3.3.5 Generalisability

ARL has no inherent limitations preventing it from being applied to a wide range of problem types. The potential for ARL to scale is founded on the use of an AMDP to reduce the problem to only the important features related to safety. Provided that it is possible to abstract the MDP to a degree sufficient to allow it to be verified in a timely manner then ARL can be applied effectively. Even so, this does not necessarily mean that ARL cannot be used successfully in the event that the AMDP is very large. Instead, it means that it may take a long time to verify the AMDP when generating safe abstract policies.

Ergodic policies [86], worst-case criterion [88], risk-sensitive criterion [92] and backup policies [98] have limited use in problems where safety and optimality directly influence one another. The potential impact on solution optimality (discussed above) means that their successful application to produce safe *and* useful policies is limited to problems where safety and optimality properties do not significantly overlap.

Whilst permissive schedulers [87] can be applied to a wide range of problem types, they suffer from scalability issues since the technique applies model checking directly onto the RL MDP which can become impractical beyond trivial problems with small MDPs. Furthermore, the requirement of full knowledge of the RL MDP's transition function precludes the technique from being used in the class of problems where this is not known.

Using teacher knowledge [94] and demonstrations [97] can only be done when a safe behaviour is known in advance. Therefore, these techniques cannot be used for problems where it is not obvious how to behave safely (for example, in the two case studies introduced in Chapter 4).

The cautious simulator [95, 96] is so far limited to use in physical systems, although, the use of a simulator to automatically identify safe states enables its application to larger scale systems.

3.4 Summary

Safe RL techniques generally fall into one of two categories: (i) modifying how a policy is optimised [86, 87, 88, 92]; or (ii) modifying how the agent explores the environment state space [94, 95, 96, 97, 98]. ARL-KR fits into the modified optimisation category since the agent must optimise a solution whilst restricted to a set of safe states, which it can explore using standard exploration strategies. Table 3.1 summarises the characteristics of the safe RL techniques discussed in this chapter, including ARL-KR.

Safety in RL commonly experiences several distinct obstacles, such as difficulty in expressing specific safety requirements, the inability to guarantee strict safety levels, limited applicability to real-world problems and significant reduction of possible optimality. The approaches discussed in this chapter are no exception to these problems and whilst some can successfully overcome some obstacles, it is often to the detriment of achieving others. In contrast, ARL can support a rich range of safety and optimisation requirements, can provide guarantees that safety levels will be met, and is not limited to any particular class of problem.

Of the techniques discussed, the permissive schedulers [87] technique is most similar to ARL since both use probabilistic model checking to formally verify properties when expressed as temporal logic. However, the use of an AMDP in ARL can allow probabilistic model checking to be applied to much larger problems than permissive schedulers could be used for. Furthermore, ARL-KR only requires partial knowledge of the MDP transition function, whilst the permissive schedulers approach requires complete knowledge that is not typically available for many RL problems.

Table 3.1: Summary of the characteristics of various safe RL techniques.

Approach	Assumptions and requirements	Safety assurance	Impact on optimality	Generalisability
Safe optimisation techniques				
Ergodic policies [86]	A (useful) safe solution exists which does not include non-ergodic states.	Provable probability of solution ergodicity. Cannot assure specific safety requirements.	Potentially very high reductions: non-ergodic states may be safe and profitable. Computationally efficient constraints can cause suboptimal agent exploration.	Not applicable to problems whose solutions necessarily cannot be ergodic. Difficulty of enforcing constraints limits its applicability to larger problems.
Permissive schedulers [87]	Full knowledge of RL MDP transition function and conservative knowledge of its reward function.	Formally guaranteed probability of satisfying a rich set of safety requirements.	Standard RL optimality guarantee within the limits of the constraints.	Not applicable for RL problems where the MDP transition function is unknown or has a large state space.
Worst-case [88]	All unsafe states/sequences of states are known a priori. RL MDP reward function modified to punish unsafe actions.	Solutions optimised assuming a worst-case scenario, ensuring at least a minimum level of safety is always achieved.	Potentially very high reductions: some profitable states could be entered at very low probability of safety violations but are disregarded.	Does not support strict probabilistic safety or optimality requirements. Suited to problems where all safety aspects can feasibly be incorporated into the reward function.
Risk-sensitive [92]	All unsafe states/sequences of states are known a priori. RL MDP reward function modified to punish unsafe actions.	Implied, but not guaranteed, by the consistency of returns from a solution.	Subject to the chosen value of the risk parameter. Potentially high reductions if the parameter is risk averse: agent can avoid profitable behaviour even if unsafe outcomes are rare.	Does not support strict probabilistic safety or optimality requirements. Suited to problems where all safety aspects can feasibly be incorporated into the reward function.
ARL-KR	Abstraction of all relevant RL MDP states.	Formally guaranteed probability of satisfying a rich set of safety requirements.	Standard RL optimality guarantee within the limits of the constraints. Pareto-optimal choice of solutions allowing a safety-optimality trade-off.	No inherent limitations to being applied to a range of RL problem types. Effective at ensuring strict probabilistic safety and optimality requirements from limited knowledge.
Safe exploration strategies				
Teacher knowledge [94]	A known safe policy to inform the agent during its exploration. A 'risk' function to determine if a future state is too dissimilar from known safe states.	Reduced, but not guaranteed, probability of entering unsafe states.	Subject to how optimal the teacher knowledge is and how strict the risk function is.	Limited to problems where a safe solution is already known.
Cautious simulation [95, 96]	A physics simulator. Optional domain expert knowledge.	Conservative classification of states as safe/unsafe from simulations. Unsafe states cannot be accidentally labelled as safe.	No guarantee of solution optimality.	Limited to robotics.
Safe demonstrations [97]	A domain expert who can provide a safe solution. A means of inferring the RL MDP dynamics from the demonstrations.	As safe as the expert's demonstrations. Cannot provide safe actions for states not featured in the demonstrations.	Subject to how optimal the expert's demonstrations are.	Limited to problems where a safe solution is already known.
Backup policies [98]	All fatal states must be identified a priori and a bespoke 'risk perception' function to detect risky states.	Guaranteed to avoid unsafe states assuming the risk perception function is entirely accurate.	No guarantee of solution optimality. Potentially high reduction if 'risky' states are avoided despite being profitable.	Does not support strict probabilistic safety or optimality requirements. Limited to problems where risk perception is feasible.

Chapter 4

Assured Reinforcement Learning

This chapter formalises the assured reinforcement learning (ARL) approach, including how it is implemented and how well it performs.

4.1 Introduction

ARL is a technique to provide assurance that an RL solution will satisfy strict safety, optimisation and other non-functional requirements. This is in contrast to traditional RL techniques where the agent will optimise a solution for some functional objectives without any regard as to *how* it achieves them.

The root of the problem with traditional RL lies in how problem objectives are expressed which consequently motivates the agent's behaviour. Objectives are defined through numerical rewards which the agent cumulates. The problem with this mechanism, though, is that it can be infeasible to express complex or subtle non-functional requirements using rewards alone. Furthermore, it can necessitate introducing more details into the underlying RL environment, which will exacerbate the state space explosion problem that affects RL and therefore will significantly reduce the rate of learning. When objectives conflict with each other, there is the additional issue of how to define a reward function to simultaneously reward the agent and punish it. The nature of RL is to maximise a reward (or minimise a cost), so traditional RL is inherently unable to satisfy two contradicting objectives. In such a situation it may be possible through trial and error to eventually identify rewards of a suitable magnitude that the agent could compromise between the two, but such a process is impractical and may not succeed at all.

ARL is a different approach to defining agent behaviour. Instead of relying on a reward scheme to shape behaviour, ARL uses formal verification techniques to identify which actions in a state will cause violations of requirements and removes

the action from the agent's action set for that specific state.

The key aspect of the ARL approach is to model the RL environment as an AMDP; to reduce the problem to only its important and relevant features, omitting superfluous details which otherwise have no impact on the safety or optimisation properties of a solution. Using this high-level model, QV is used to analyse the model and to identify a set of high-level solutions, called *abstract policies*, which when followed will satisfy all the safety and optimisation objectives. These abstract policies are then used to restrict the RL agent's action choices to those which map to the safe high-level options of the abstract policy. Once the RL agent has optimised a solution subject to the action constraints it is guaranteed that the resulting RL solution can be empirically evaluated to achieve the same safety levels that were verified for the abstract policy.

4.2 Running Example

We motivate the need for assured reinforcement learning using an extension of the benchmark RL flag collection mission from [3]. In the original flag collection mission, an agent needs to find and collect flags scattered throughout a building by learning to navigate through its rooms and hallways to collect the flags. In our extension, certain doorways between areas are provided with security cameras, as shown in Figure 4.1. Detection by a camera results in the capture of the agent and the termination of its flag collection mission.

Unknown to the agent, the detection effectiveness of the cameras decreases towards the boundary of their field of view, so that the camera-monitored doorways comprise three areas with decreasing probabilities of detection: direct view by the camera, partial view and hidden. We assume that the detection probabilities for the camera-monitored doorways from Figure 4.1 and the camera-view areas have the values from Table 4.1.

Table 4.1: Agent detection probabilities.

	Camera view		
Camera	Direct	Partial	Hidden
HallA ↔ RoomA	0.18	0.12	0.06
HallB ↔ RoomB	0.15	0.1	0.05
HallB ↔ RoomC	0.15	0.1	0.05
RoomC ↔ RoomE	0.21	0.14	0.07

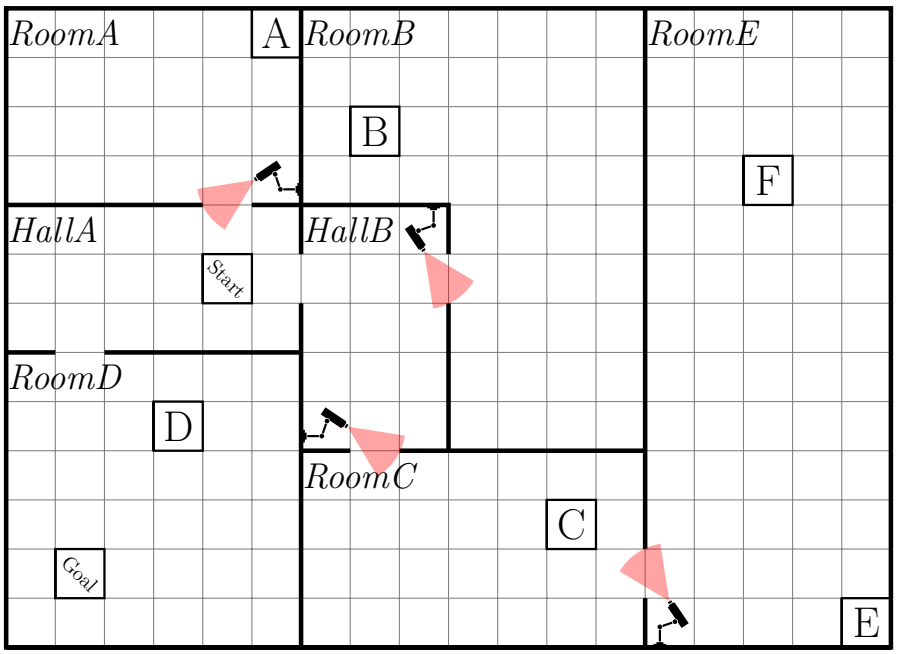


Figure 4.1: Flag collection mission from [3] extended with security cameras. The diagram shows the flag positions A–F, the start and goal positions for the agent, and the cameras and their field of view.

Consider now a real-world application where the agent is an expensive autonomous robot pursuing a surveillance mission or a search-and-rescue operation. In this scenario, its owners are interested in the safe return of the robot, but do not want it to behave 'too safely' or it will not collect enough flags. Therefore, they specify the following constraints for the agent:

C_1 The agent must reach the 'goal' area with probability at least 0.75.
C_2 The agent must expect to cumulate a reward greater than 2 before the mission terminates.

Subject to these constraints being satisfied, they are interested to maximise:

O_1 The probability that the agent reaches the 'goal' area.
O_2 The reward accumulated by the agent.

As a result, the agent owners additionally want to know the range of possible trade-offs between these two conflicting *optimisation objectives*. In this way, the right level of trade-off can be selected for each instance of the mission. Note that formulating the constraints C_1 and C_2 into a reward function and using standard RL to solve the problem cannot guarantee success because an RL agent aims to maximise its reward rather than to maintain it within a specified range.

4.3 Approach

The ARL approach takes as input the following information about the problem to solve:

1. Partial knowledge about the problem;
2. A set of constraints $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ that must be satisfied by the policy learnt by the RL agent;
3. A set of objectives $\mathcal{O} = \{O_1, O_2, \ldots, O_m\}$ that the RL policy should optimise (i.e. minimise or maximise) subject to all constraints being satisfied.

The optimisation objectives $\mathcal{O}$ can be associated with problem properties that appear in the constraints $\mathcal{C}$ (like in our running example), or also with additional problem properties (as in the assisted-living system planning problem from Section 4.4.2). The partial knowledge must contain sufficient information for the assembly of an AMDP supporting the formalisation in PCTL and the probabilistic model checking of the $n > 0$ safety constraints and $m \geq 0$ optimisation objectives.

Note that the partial knowledge about the environment assumed by ARL is necessary: no constraints could be ensured during RL exploration in the absence of any information about the environment. Additionally, it is assumed that the partial knowledge contains all necessary information for abstract safe policies to fully apply to the low-level RL model and that this information is accurate. Should these assumptions not be satisfied then an abstract policy may not necessarily provide the levels of safety in the RL solutions that it was verified to give. Furthermore, ARL has the usual RL assumption that sufficient learning is undertaken by the RL agent to find an optimal policy for safety requirements to be assured; suboptimal RL policies may not satisfy the safety requirements.

Under these assumptions, the ARL approach: (i) generates a Pareto-optimal set of safe abstract policies that satisfy the constraints $\mathcal{C}$ and are Pareto non-dominated with respect to the optimisation objectives $\mathcal{O}$; and (ii) learns a (concrete) policy that satisfies the constraints $\mathcal{C}$ and meets trade-offs between objectives $\mathcal{O}$ given by a Pareto-optimal abstract policy selected by the user.

A preliminary step for ARL is the construction of the AMDP. This step devises a *parameterised* AMDP model of the RL problem that supports the probabilistic model checking of PCTL-formalised versions of the constraints $\mathcal{C}$ and of the optimisation objectives $\mathcal{O}$. Following from this step, ARL comprises two stages:

1. **Abstract policy synthesis** – This stage generates the Pareto-optimal set of safe abstract policies.

2. **Safe learning** – This stage uses a user-selected abstract policy from the Pareto-optimal set to enforce state-action constraints for the exploration of the environment by the RL agent. Subsequently, the agent learns an optimal policy that complies with the problem constraints and meets the optimisation objective trade-offs associated with the selected abstract policy.

AMDP Construction. In this preliminary step, all features that are relevant for the problem constraints and optimisation objectives must be extracted from the available partial knowledge about the RL environment. This could include locations, events, rewards, actions or progress levels. The objective is to abstract out the features that have no impact on the solution attributes that the constraints $\mathcal{C}$ and objectives $\mathcal{O}$ refer to, whilst retaining the key features that these attributes depend on. This ensures that the AMDP is sufficiently small to be analysed using probabilistic model checking, whilst also containing the necessary details to enable the analysis of all constraints and optimisation objectives.

In our running example, the AMDP is constructed as follows: the key features are the locations and connections of rooms and halls, the detection probabilities of the cameras and the progress of the flags collected. Instead of having each Cartesian coordinate within a room or hall as a separate state, the room or hall *as a whole* is considered a single state in the AMDP. Also, we only consider the hidden-view detection probability per camera since these are the probabilities that the RL agent will learn for the optimal points to traverse the doorways. These abstractions yield a 448-state AMDP for our flag collection problem, compared to 14,976 states for the RL MDP (which is unknown to the agent). Note that the number of AMDP states is larger than the number of locations (rooms and halls) because some locations, i.e. those with more than one doorway, require different AMDP states for each possible combination of flags collected so far.

The actions of the full RL MDP are similarly abstracted. For example, instead of having the cardinal movements at each location of the building from our running example, abstract actions (i.e. options) are specified as simply the movement between locations. Thus, instead of the four possible actions for each of the 14,976 MDP states, the 448 AMDP states have only between one and four possible options each. The N options that are available for an AMDP state correspond to the $N \geq 1$ passageways that link the location associated with that

state with other locations, and can be encoded using a *state parameter* that takes one of the discrete values 1, 2, ..., N. The parameters for AMDP states with a single passageway (corresponding to rooms A, B and E from Figure 4.1) can only take the value 1 and are therefore discarded. This leaves a set of 256 parameters that correspond to approximately 4×10^{99} possible abstract policies.

Finally, this preliminary step is also responsible for labelling the AMDP with atomic propositions, enabling its probabilistic model checking and for the PCTL formalisation of the constraints $\mathcal{C}$ and optimisation objectives $\mathcal{O}$ in terms of these atomic propositions. For our running example, this involves associating an atomic proposition '*goal*' with the AMDP states corresponding to the agent reaching the 'goal' area (with any number of collected flags), and formalising the constraints and optimisation objectives as follows:

$$C_1\text{: } \mathrm{P}_{\geq 0.75}\,[\,\mathrm{F}\ \mathit{goal}\,] \qquad O_1\text{: maximise } \mathrm{P}_{=?}\,[\,\mathrm{F}\ \mathit{goal}\,]$$
$$C_2\text{: } \mathrm{R}_{>2}\,[\,\mathrm{F}\ \mathit{goal}\,] \qquad O_2\text{: maximise } \mathrm{R}_{=?}\,[\,\mathrm{F}\ \mathit{goal}\,]$$

Stage 1: Abstract Policy Synthesis. In this ARL stage, the generic heuristic from Algorithm 3 is used to find constraint-compliant abstract policies for the RL problem. Given an AMDP $\bar{M}$, a set of constraints $\mathcal{C}$ and a set of optimisation objectives $\mathcal{O}$ (all obtained in the preliminary step of ARL), the function GenAbstractPolicies from Algorithm 3 synthesises an approximate Pareto-optimal set of abstract policies that satisfy the constraints $\mathcal{C}$ and are Pareto non-dominated with respect to the optimisation objectives $\mathcal{O}$. The abstract policy set PS returned by this function in line 22 starts empty (line 2), and is assembled iteratively by the while loop in lines 3–21 until a termination criterion $\neg$Done(PS) is satisfied. This criterion (not shown in Algorithm 3) may involve ending the while loop after a fixed number of iterations, or after several consecutive iterations during which PS is left unchanged. Each iteration of the while loop first identifies a set P of 'candidate' abstract policies in line 4, and then updates the Pareto-optimal policy set in the for loop from lines 5–20. Our algorithm is not prescriptive about the method used to get new candidate policies. As such, the function GetCandidatePolicies from line 4 can be implemented using a metaheuristic such as the genetic algorithm used to synthesise Markovian models in [99], a simple heuristic like hill climbing, or just random search.

To decide how to update PS, the for loop in lines 5–20 examines each candidate abstract policy $\bar{\pi}$ as follows. First, the boolean function PMC_1 (which

Algorithm 3 Abstract policy synthesis heuristic

```
1: function GenAbstractPolicies(M̄, C, O)
2:     PS ← {}
3:     while ¬Done(PS) do
4:         P ← GetCandidatePolicies(PS, M̄)
5:         for π̄ ∈ P do
6:             if ⋀_{c∈C} PMC_1(M̄, π̄, c) then
7:                 dominated = false
8:                 for π̄′ ∈ PS do
9:                     if Dom(π̄, π̄′, M̄, O) then
10:                        PS ← PS \ {π̄′}
11:                    else if Dom(π̄′, π̄, M̄, O) then
12:                        dominated = true
13:                        break
14:                    end if
15:                end for
16:                if ¬dominated then
17:                    PS ← PS ∪ {π̄}
18:                end if
19:            end if
20:        end for
21:    end while
22:    return PS
23: end function

24: function Dom(π̄_1, π̄_2, M̄, O)
25:    return
         ∀o∈O · PMC_2(M̄,π̄_1,o)≥PMC_2(M̄,π̄_2,o)∧
         ∃o∈O · PMC_2(M̄,π̄_1,o)>PMC_2(M̄,π̄_2,o)
26: end function
```

invokes a probabilistic model checking tool) is used to establish if using policy $\bar{\pi}$ for the AMDP $\bar{M}$ satisfies every constraint $c \in \mathcal{C}$ (line 6). If it does, $\bar{\pi}$ is deemed safe and the inner for loop in lines 8–15 compares it to each of the abstract policies already in PS by using the Pareto-dominance comparison function Dom defined in lines 24–26, where the probabilistic model checking function $\mathrm{PMC}_2(\bar{M}, \bar{\pi}, o)$ computes the value of the optimisation objective $o \in \mathcal{O}$ for the policy $\bar{\pi}$ of $\bar{M}$.[1] Every policy $\bar{\pi}' \in PS$ that is Pareto dominated by $\bar{\pi}$ is removed from PS (lines 9–10). If $\bar{\pi}$ is itself Pareto-dominated (line 11), the flag *dominated* (initially false,

[1] A policy $\bar{\pi}_1$ is said to Pareto-dominate another policy $\bar{\pi}_2$ with respect to a set of objectives $\mathcal{O}$ iff $\bar{\pi}_1$ gives superior results to $\bar{\pi}_2$ for at least one objective from $\mathcal{O}$, and for all other objectives $\bar{\pi}_1$ it is at least as good as $\bar{\pi}_2$ [100]. Without loss of generality, the definition of Dom from Algorithm 3 assumes that all objectives from $\mathcal{O}$ are maximising objectives.

cf. line 7) is set to `true` in line 12 and the inner for loop is terminated early in line 13. Finally, the new abstract policy is added to the Pareto-optimal policy set if it is not dominated by any known policy (lines 16–18).

Stage 2: Safe learning. The second stage of ARL exploits the previously obtained approximate Pareto-optimal set of abstract policies. A policy is selected from this set by taking into account the trade-offs that different policies achieve for the optimisation objectives used to assemble the set. This selection is a manual step. The high-level *options* from the abstract policy are used as rules for which of the corresponding low-level MDP actions the RL agent should, or should not, perform in order to achieve the required constraints. For instance, assume that the selected abstract policy for our running example requires the agent to never enter RoomA. In this case, should the agent be at Cartesian coordinates (5,9) (i.e. the position immediately to the North of the Start position), the action to move North and thus enter RoomA is removed from the agent's action set, for this specific state.

ARL identifies that a low-level action a when in low-level state s will lead to a violation of safety requirements if undertaking the action can lead to state s', where $\bar{s}(s) \neq \bar{s}(s')$ and $\bar{T}\Big(\bar{s}(s), \bar{\pi}_{\bar{A}}\Big(\bar{s}(s)\Big), \bar{s}(s')\Big) = 0$ according to the safe abstract policy being used to constrain the RL agent's actions. Disallowing actions in a state that are not associated with the safe options of the abstract policy results in the RL agent learning low-level behaviours that are guaranteed to satisfy the safety constraints.

This restriction of actions necessarily reduces the RL agent's autonomy but it does not remove it entirely. Specifically, to ensure that the agent behaves according to the safety requirements, exploration of actions that can result in safety violations, i.e. those actions which contradict the abstract policy, are restricted. Otherwise, the agent is free to explore its environment as it normally would. For instance, in the running example, the agent's exploration is restricted only by which rooms it can enter. The agent must still explore the environment to learn the flag locations within the rooms as well as the doorway areas safest to cross, information which is unknown a priori and therefore not contained within the abstract policies.

Although abstract policy constraints may yield suboptimal RL policies with respect to the RL model in its entirety, this key feature assures safety.

4.4 Evaluation

To evaluate the efficacy and generality of ARL we applied it to two case studies from different domains. The first case study is based on the navigation task described in Section 4.2. The second case study is a planning problem adapted from [54], where a system has been designed to assist a dementia sufferer perform the task of washing their hands.

For each case study we conducted a set of four experiments. An initial experiment was first done which was a traditional RL implementation of the case study problem. This experiment serves as a baseline which we contrast with the ARL experiments in order to determine the effects of ARL. Following the baseline experiment a further three experiments were undertaken where RL in Stage 2 of ARL was applied using a different abstract policy from the Pareto-optimal set of abstract policies constructed in Stage 1, using an implementation based on random search for function GetCandidatePolicies from Algorithm 3.

For our RL implementations we used the YOrk Reinforcement Learning Library (YORLL) [101]. The experiments were carried out using an Intel Core i5-6200U 2.3 GHz CPU with 8 GB of RAM. For all experiments we use a discount factor $\gamma = 0.99$ and a learning rate $\alpha = 0.1$ which decays to 0 over the learning run. Experiment-specific parameters are shown where relevant in the remainder of this section. All parameters have been chosen empirically in line with standard RL practice. As is convention when evaluating stochastic processes, we repeated each experiment multiple times (i.e. five times) and we evaluated the final policy for each experiment many times (i.e. 10,000 times) in order to ensure that the results are suitably significant [102].

4.4.1 Guarded Flag Collection

This case study is based on the running example described in Section 4.2 and referred to throughout Section 4.3. In the interest of brevity, the details presented in these two previous sections will not be repeated here.

In our RL implementation, the reward structure was defined as follows: the agent receives a reward of 1 for each flag it collects and an additional reward of 1 for reaching the 'goal' area of the building. If the agent is captured it receives a reward of -1. We used the AMDP constructed during the first ARL stage as described in Section 4.3. Specifically, the knowledge of the problem was formulated as the AMDP shown by the code extracts in PRISM Language 2

(below). The more complete code can be found in Appendix A.

At line 1 the model type is defined to be a discrete time Markov chain (DTMC), not an MDP (more specifically, AMDP). This is because ARL uses a *parameterised* AMDP which means that each action in a state of the AMDP is fixed by a parameter, resolving the non-determinism and inducing a DTMC. Examples of these parameters are shown in lines 3, 4 and 6 as variables `w1`, `w2` and `w64`. As explained further below, there are 64 parameters for each of the 64 possible states at each location requiring an action choice. These parameters currently hold no value since the process of finding a suitable abstract policy (comprising parameters) involves searching for these values. Parameters take integer values, starting from 1, where the range of possible values depends on how many actions are possible in the location the parameter is used in.

At line 8, the example variable `p1` represents a transition probability. In particular, this transition probability is for the hidden view camera probability between HallA and RoomA.

Line 11 starts the definition of the model structure and lines 12–19 define the model variables, these combine to form individual states within the model. First, the variable `position` defines each location in the environment (i.e. 0 = HallA, 1 = RoomA, 2 = RoomB etc.) with the initial position being 0. Next, the booleans `flagA`–`flagF` define whether the respective flag has been collected or not. Initially, these are set to `false`. Last, the variable `captured`, also initialised as `false`, defines whether the agent has been captured or not.

The example commands starting on lines 21, 24, 28 and 31 show how these variables form states of the environment, they also show the potential updates to variables that can occur when transitioning out of states. The guards of the commands at lines 21 and 31 (and others not shown) represent the start of the system, i.e. its initial state where the variables specify that the the agent has not been captured, that it is in position 0 (HallA, the starting location) and Flags A–F have not been collected. Since this model is of a parameterised AMDP, equivalent to a DTMC, the actions for a state are included in a command's guard. For example, the command starting at line 21 shows the update to perform when action parameter `w1=1` and at line 31 the updates when parameter `w1=2`. In the former case, the action specifies transitioning to new `position'=1`, i.e. RoomA, which will result in collecting FlagA (`flagA'=true`) with probability `1-p1`.

PRISM Language 2: Guarded flag collection AMDP extracts

```
dtmc

const int w1;
const int w2;
...
const int w64;
...
const double p1 = 0.06;
...

module guarded_flag_collection
        position: [0..8] init 0;
        flagA: bool init false;
        flagB: bool init false;
        flagC: bool init false;
        flagD: bool init false;
        flagE: bool init false;
        flagF: bool init false;
        captured: bool init false;

        [] !captured & position=0 & w1=1 & !flagA & !flagB & !flagC
                & !flagD & !flagE & !flagF -> (1-p1):(position'=1)
                & (flagA'=true) + p1:(captured'=true);
        [] !captured & position=0 & w2=1 & !flagA & !flagB & !flagC
                & !flagD & !flagE & flagF -> (1-p1):(position'=1)
                & (flagA'=true) + p1:(captured'=true);
        ...
        [] !captured & position=0 & w64=1 & flagA & flagB & flagC
                & flagD & flagE & flagF -> (1-p1):(position'=1)
                & (flagA'=true) + p1:(captured'=true);
        [] !captured & position=0 & w1=2 & !flagA & !flagB & !flagC
                & !flagD & !flagE & !flagF -> (position'=4)&(flagD'=true);
        ...

        [end] captured | position=6 -> (position'=8);
        [] position=8 -> (position'=8);
endmodule

rewards "all_flags"
        [end] true : (flagA ? 1 : 0) + (flagB ? 1 : 0) + (flagC ? 1 : 0)
                + (flagD ? 1 : 0) + (flagE ? 1 : 0) + (flagF ? 1 : 0)
                + (position=6 ? 1 : 0);
endrewards
```

However, with probability `p1` the transition will instead result in the agent being captured (`captured'=true`). In the latter case, the new position will be 4 (corresponding to RoomD) resulting in FlagD being collected.

As there are 6 flags in total, this means there are up to $2^6 = 64$ possible combinations of flags that the agent may have collected at any time. Since the agent may hold any one of these flag combinations when in a position, there needs to be 64 parameters specifying which action must be taken for each of these flag combinations for each position. For example, the commands starting at lines 21 and 31 use the action `w1` when no flags are collected and in HallA and the command from line 24 uses `w2` when in HallA and only FlagF has been collected. Lastly, the command starting at line 28, for when all flags have been collected and in HallA, uses `w64`. The parameters `w3`–`w63` (not shown) specify the action to take for each of commands (also not shown) representing the remaining flag combinations when in position 0.

Parameters are only required in those locations from which there are 1< possible doorways to choose. When a location only has a single doorway there is only one possible route to take from that area. For example, the command for when the agent is in RoomA can simply be expressed as:

```
!captured & position=1 -> (1-p1):(position'=0) + p1:(captured'=true);
```

With this command, the guard simply requires that the agent has not been captured and that it is in position 1 (RoomA), it does not matter which flags have/haven't been collected since the agent will always leave this room for any flag combination. Hence, the update is simply the transition back into `position'=0` (HallA) with probability `1-p1` and with probability `p1` be captured. There is an equivalent command for when in RoomB and RoomE. When in RoomD the agent still has to choose between returning to HallA or proceeding to the goal, requiring a parameter to specify which action to choose.

The command on line 35 represents the state where the agent has either been captured or has entered into position 6 (i.e. the goal area). In either case, the agent's mission has ended and so the command updates the position to 8, an absorbing state (line 36). Lastly, when the action `[end]` occurs after transitioning from line 35, the rewards are given, shown in the reward structure `"all_flags"` from line 39 to line 43. Specifically, the transition reward starting at line 40 returns the sum for how many flags are collected and whether the agent has reached the goal or not.

In the first ARL stage, we generated 10,000 abstract policies with parameter

values (i.e. state to action mappings) drawn randomly from a uniform distribution. The average time required to generate and verify a policy was 272 milliseconds and the entire search was completed after 45.3 minutes. Out of these abstract policies, probabilistic model checking using the tool PRISM identified 14 policies with different quantitative properties that satisfied the two required constraints. The first safe policy found was the 47^{th} to be verified, after 12.8 seconds. The last of these 14 policies was the 1762^{nd} to be verified, after 8 minutes. Although subsequent searching identified other safe policies, their safety and optimisation properties were equal to one of the previously found 14 and so were discarded. Figure 4.2 shows the QV results obtained for these 14 abstract policies, i.e. their associated probability of reaching the 'goal' area and expected number of flags collected. The approximate Pareto front depicted in this figure was obtained using the two optimisation objectives, i.e. maximising the expected number of flags collected and the probability of reaching the 'goal' area of the building. Generating the Pareto front was achieved in just 2 milliseconds.

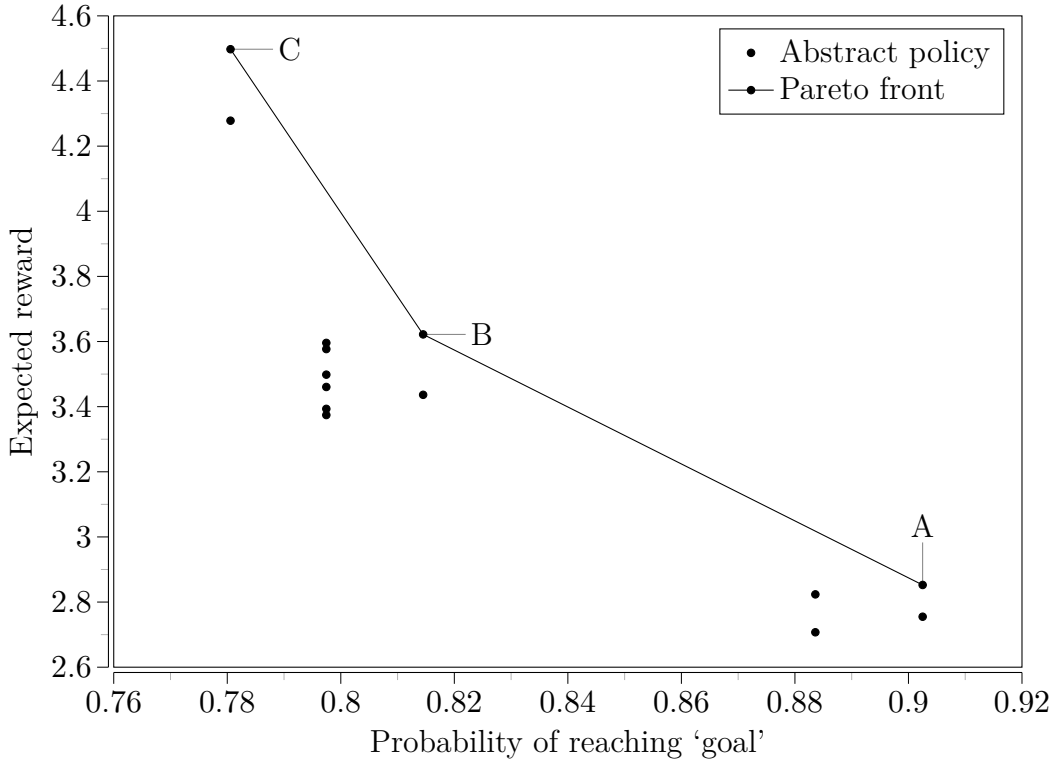


Figure 4.2: Pareto front of abstract policies that satisfy the constraints from Table 4.2. Those policies that were selected for ARL are labelled A, B and C.

Three abstract policies were selected to use in each of the ARL experiments during the safe learning stage, as explained in Chapter 4.3. The properties of

Table 4.2: Selected abstract policies to use for ARL in the guarded flag collection.

Abstract policy	**Probability of reaching 'goal'**	**Expected reward**
A	0.9	2.85
B	0.81	3.62
C	0.78	4.5

these three abstract policies are shown in Table 4.2. These abstract policies are translated into action constraints on the agent by blocking agent actions which do not correspond with the high-level options.

The baseline experiment, which was a standard RL implementation of the case study, used an $\epsilon = 0.8$ and performed 2×10^7 learning episodes, each with 10,000 steps. This did not, however, reach a global optimum. Even after extensive learning, in excess of 10^9 learning episodes, conventional RL did not attain a superior solution.

Since the capture probabilities for direct, partial and hidden view by the cameras at guarded doorways differ by only small amounts, the agent needs to experience being captured in each view a significant number of times before it converges on the safest area of the doorway (i.e. with the lowest capture probability). However, because the capture probabilities are relatively small at any part of the doorway, the agent is captured infrequently and so the agent must traverse the doorways a significant number of times before it is captured enough times to determine the safe areas of it. Furthermore, the agent needs to succeed with this process at all four different doorways where it can be captured *and* the agent needs to learn the safe doorway areas for each possible order that the flags are collected (i.e. the agent needs to learn the same doorway areas multiple times since each time a flag is collected the agent enters a new set of states and previous knowledge of safe areas is not carried over to them). Lastly, since the agent's probability of capture cumulatively increases with more guarded doorways that it traverses, those towards the end of a learning episode (e.g. after collecting all the flags and the agent is heading towards the goal) are encountered even less frequently as the agent is often captured before encountering them. Therefore, the agent experiences less opportunity to learn the safe areas of them than the early doorways it will encounter. Thus, the process of finding the optimal area for every doorway in every situation, and ultimately find an optimal solution, requires a vast number of learning episodes. Figure 4.3 shows the learning progress for this experiment.

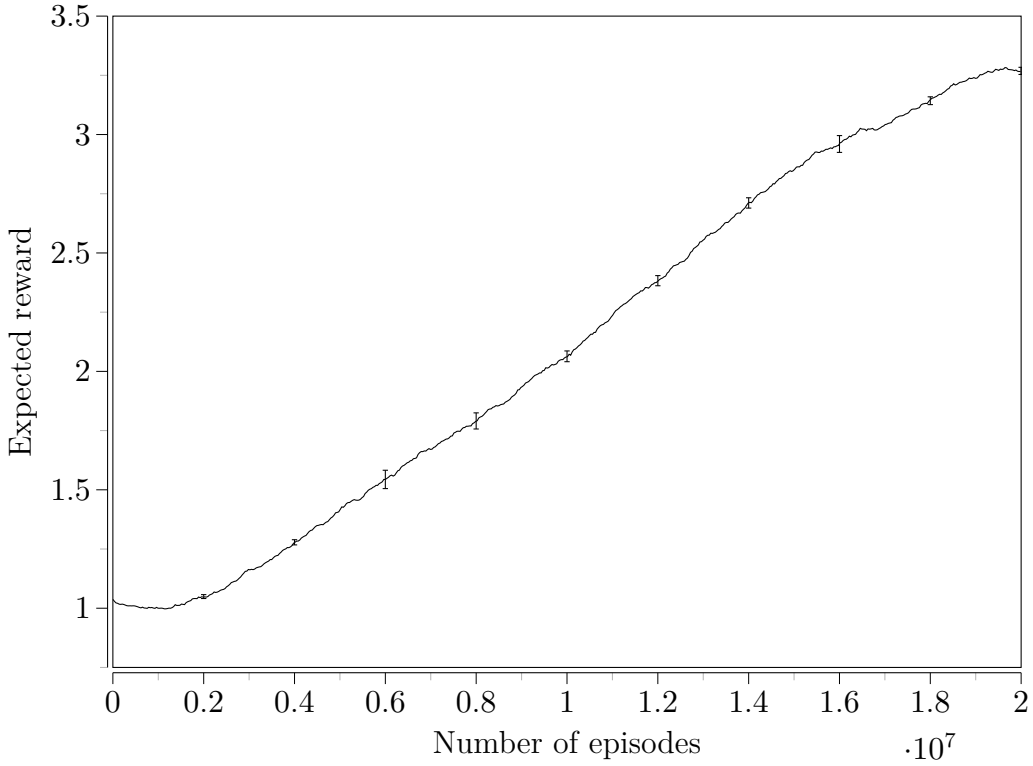


Figure 4.3: Learning progress for the guarded flag collection with no ARL applied.

Next, we present the three ARL experiments, one for each of the abstract policies from Table 4.2. Since the abstract policy had the effect of guiding the agent with regard to the locations to enter next, less exploration by the agent was required and fewer learning episodes were necessary. Therefore, we used $\epsilon = 0.6$ which decayed to zero over the learning run and 10^5 episodes were needed for the learning to converge. Figure 4.4 shows the RL learning progress for each of the abstract policies used for ARL and Figure 4.5 shows the routes through the environment learned by the agent for each of the safe abstract policies.

In contrast to the baseline RL experiment, a superior policy was learned much faster by ARL, further demonstrating the advantages of our approach. This is due to the fact that the abstract policy: (i) limits how many doorways the agent needs to learn (since it prevents the agent from ever encountering some); and (ii) restricts the order that doorways are traversed to only one possible route and so the agent doesn't have to keep relearning the same doorway for a range of possible routes. Therefore, the ARL agent has to learn significantly fewer safe areas where cameras exist and thus requires far fewer learning episodes to converge on an optimal solution when compared to standard RL.

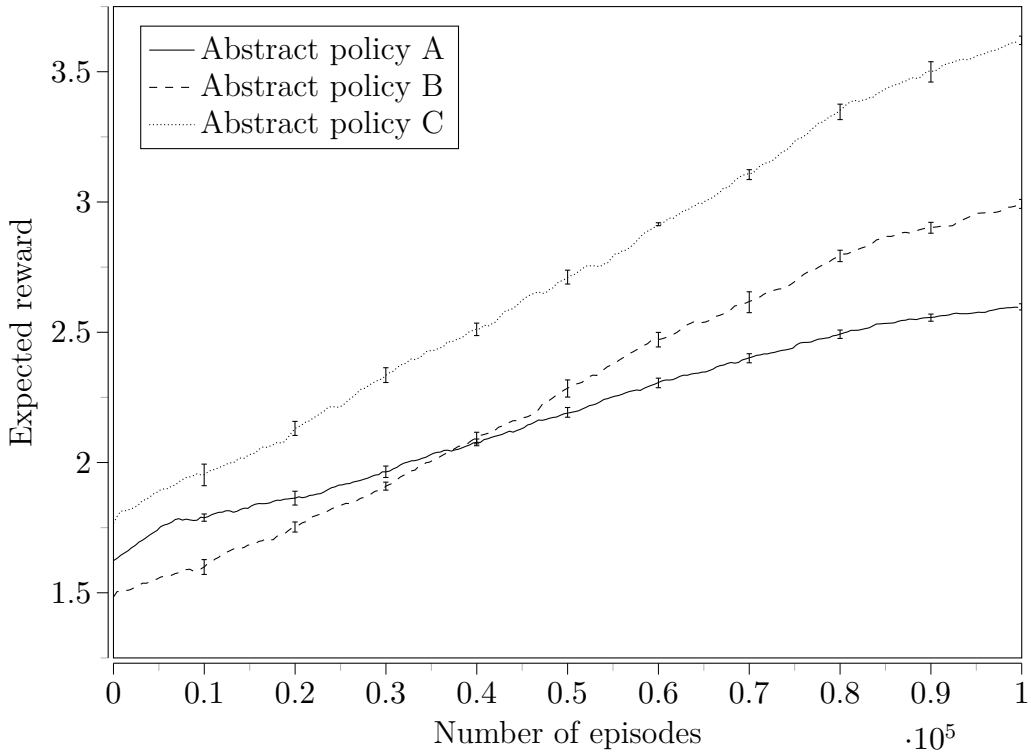
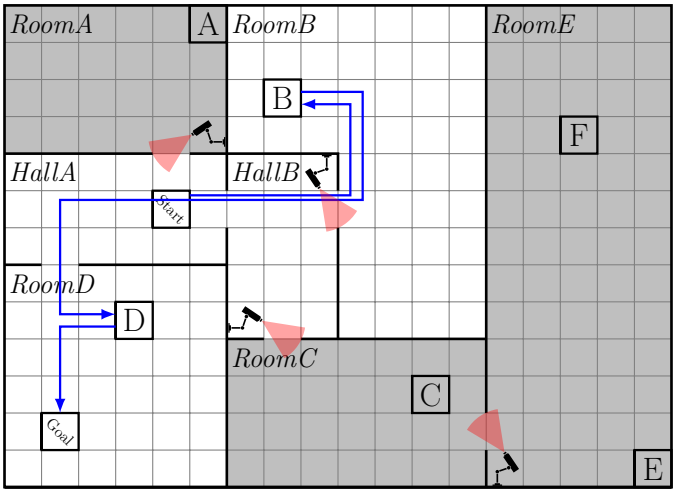
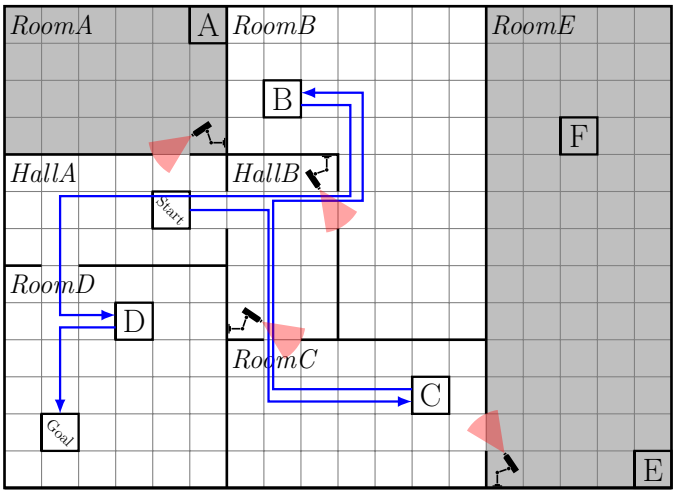


Figure 4.4: Learning progress for the guarded flag collection with ARL applied using the selected abstract policies A, B and C.

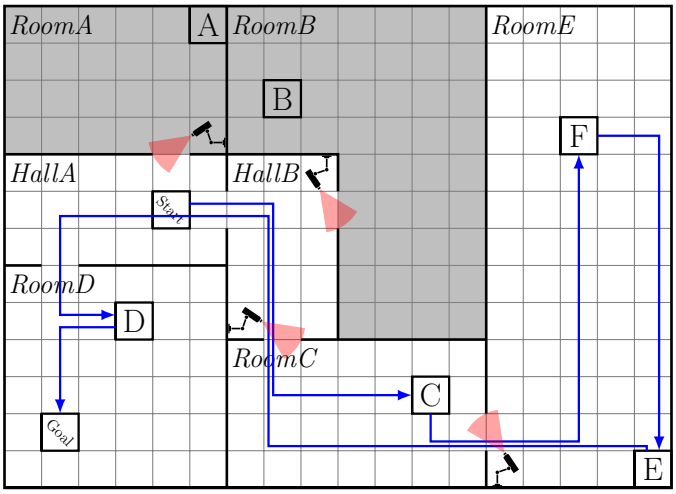
The learned policies for each of the experiments were empirically evaluated and the results summarised in Table 4.3. The experiments where an abstract policy was applied resulted in an RL policy that: (i) satisfied the problem constraints and optimisation objectives; and (ii) matched the probabilities of reaching the 'goal' area and the expected rewards of the abstract policies from Table 4.2. The baseline experiment gave results that do not satisfy our constraints; however, as discussed above it is possible that given a hugely excessive number of learning episodes the agent may optimise a solution sufficiently that it can satisfy the constraints. Even so, there are two major problems with relying on standard RL to achieve this. First, the number of learning episodes required to fully converge to an optimal solution is unknown, other than that it will be in excess of 10^9 which makes it impractical to find a solution in any reasonable amount of time. Second, any success of RL in finding a safe solution would coincidental: standard RL does not allow a range of solutions to be produced to satisfy specific requirements. The optimal solution, whatever it may be, will only ever have one unchanging set of properties. Should the safety requirements not coincide with this solution's properties, without changing the underlying reward structure of the RL MDP (which, as discussed in Chapter 3, may not be feasible), standard RL would be unable to produce a safe solution.



(a) Safe abstract policy A.



(b) Safe abstract policy B.



(c) Safe abstract policy C.

Figure 4.5: The routes optimised by the agent under abstract polices A, B and C. The areas shaded grey are those that the policy prevents the agent from entering.

Table 4.3: Results of the baseline and ARL experiments for the guarded flag collection case study.

Abstract policy	**Probability of reaching 'goal'**	**Standard error**	**Expected reward**	**Standard error**
None	0.72	0.0073	4.01	0.031
A	0.9	0.0012	2.85	0.0029
B	0.81	0.0019	3.62	0.0037
C	0.78	0.0012	4.5	0.0041

4.4.2 Assisted-Living System

Dementia is a common chronic illness with significantly debilitating consequences. As the illness progresses, it becomes increasingly difficult for the sufferer to perform even simple tasks, making it necessary for a caregiver to provide assistance with such tasks [103].

To alleviate the duties of the caregiver and the cost to healthcare, the project described in [54] has developed an automated system that helps a dementia patient perform the task of washing their hands. For our second case study we used a simulated version of this assisted-living system. For the purpose of our system, the hand-washing task can be decomposed into the subtasks listed in Table 4.4. This table also shows the atomic propositions (i.e. boolean labels, discussed in Chapter 2) that we will use in this section to indicate whether each of the subtasks has been completed.

Table 4.4: Hand-washing subtasks.

Subtask	**Atomic proposition**
Turn tap on	on
Apply soap	soaped
Wet hands under tap	wet
Rinse washed hands	rinsed
Dry hands	dried

It is possible for the dementia sufferer to regress in this task by repeating subtasks they have already performed, or by performing the wrong subtask for the stage of the hand-washing process they have reached. Figure 4.6 depicts the workflow carried out by a healthy person while progressing with the task (black, continuous-line nodes and arrows) and the possible regressions that a dementia sufferer could make (red, dashed-line nodes and arrows). For ease of reference,

each state of the workflow is labelled with a state ID (s_1 to s_{12}) and with the atomic propositions that hold in that state.

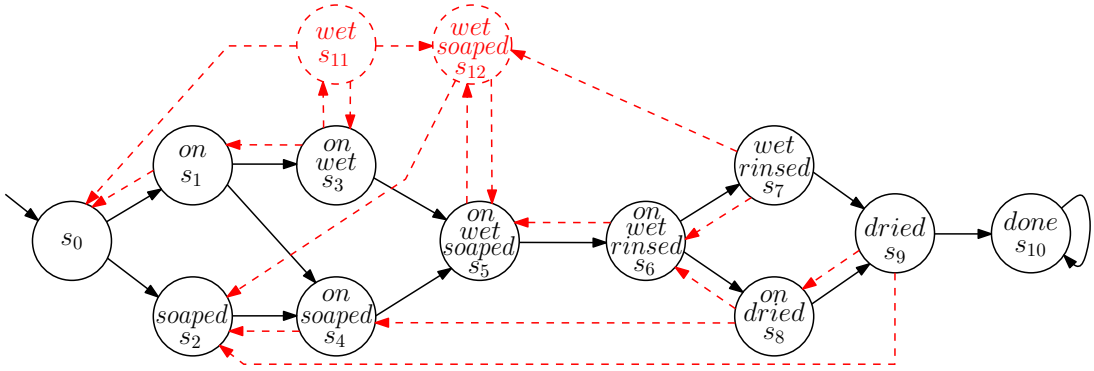


Figure 4.6: Workflow of washing hands, showing the subtasks at each stage of progress with the progression of a healthy person in black, continuous lines and the possible regressions of a dementia sufferer in red, dashed lines.

The probabilities of the dementia sufferer progressing and regressing (not shown in Figure 4.6) vary at each stage of the task and between sufferers. For the purpose of our evaluation, we decided these probabilities based on the subtask complexity, as indicated in [54].

The system is designed so that if the user fails to perform one of the next correct subtasks then it may provide a voice prompt instructing the user what subtask to do next. The system learns what style of voice is most appealing to the user based on how conducive different styles of prompt are at the user succeeding with the overall task. Voice styles vary in gender, sternness of the instructions (mild, moderate or strict) and volume (soft, medium or loud). The appeal of the voice style will induce an increase in the probability that the dementia sufferer progresses compared to no prompt being given, with the least appealing voice yielding the smallest increase and the most appealing yielding the largest increase.

For our system we wish to determine when to give a prompt to the user and when it becomes necessary to call the caregiver (i.e. the user is not making progress, despite repeated prompts). Overloading the user with prompts can become stressful and therefore each prompt has a negative reward of -1. Whilst calling the caregiver will be of relief to the user, as well as ensuring the completion of the task, doing it too frequently will become stressful to the caregiver or, in a care home, will overstretch the personnel resources available. Therefore, the caregiver should assist only when necessary, but most of the time not, and so the action to call the caregiver has a cost of -300. Completing the task results in a reward of 500. Note that the rewards for calling the caregiver and for completing

the task are only necessary in the RL simulation for learning to appropriately progress and are not necessary in the AMDP.

Finally, we desire that the caregiver be present at least once every one-to-four days, to ensure that the sufferer receives the caregiver's attention regularly. Assuming that a person washes their hands approximately five times a day, the probability that the caregiver should assist the dementia sufferer during any one hand-washing should be between $1/20$ and $1/5$, i.e. between 0.05 and 0.2. This constraint, and an additional, manually-specified optimisation objective for the abstract policy synthesis stage of ARL can be formalised in PCTL as shown in Table 4.5, where m is the number of mistakes made at any given time, *MAX_MISTAKES* is the threshold for the maximum number of mistakes that result in calling the caregiver, *distress* is the reward structure for stress to the dementia sufferer, and *done* is the atomic proposition associated with the completion of the hand-washing task by the user (see Figure 4.6).

Table 4.5: Constraints and optimisation objectives for the assisted-living system.

ID	Constraint (C) or optimisation objective (O)	PCTL
C_1	The probability that the caregiver provides assistance should be at least 0.05	$\mathrm{P}_{\geq 0.05}[\mathrm{F}\, m = \mathit{MAX\_MISTAKES}]$
C_2	The probability that the caregiver provide assistance should be at most 0.2	$\mathrm{P}_{\leq 0.2}[\mathrm{F}\, m = \mathit{MAX\_MISTAKES}]$
O_1	The level of dementia sufferer distress due to multiple voice prompts should be minimised	minimise $R^{\mathrm{distress}}_{=?}$ [F *done* $\vee$ $m =$ *MAX_MISTAKES*]
O_2	The probability of calling the caregiver should be minimised (subject to C_1 and C_2 being satisfied)	minimise $P_{=?}$ [F $m =$ *MAX_MISTAKES*]

We constructed the AMDP for this system based on the workflow shown in Figure 4.6, where each workflow stage represents a different AMDP state. To abstract the RL MDP we only used the transition probabilities for the best style of prompt which the RL agent aims to learn. Shown below in PRISM Language 3 is how the knowledge of the problem was expressed as the AMDP in PRISM. A more complete version can be found in Appendix B.

We encoded an abstract policy for this AMDP using 13 parameters, one for each stage of the task from Figure 4.6 other than stage 10 (where the task is complete), and a final one representing the threshold for the maximum number of mistakes before calling the caregiver. Lines 3 and 4 are the first two parameters for the prompts and line 6 is the parameter for the mistakes threshold. As with the guarded flag collection AMDP (PRISM Language 2), these parameters do not initially have values since the task of searching for an abstract policy involves finding the ideal values for these parameters. The parameters associated with each workflow stage, denoted by the numerical suffix of each prompt parameter name, represents the minimum number of total user mistakes that warrant giving a prompt at that stage. Each parameter can take values between zero (always give a voice prompt) and the maximum number of mistakes allowed before calling the caregiver (never give a voice prompt).

The variables at lines 8–14 hold the probabilities for transitioning between states. The name designations take the format `pXY` where `X` is the source state and `Y` is the destination state. Between lines 16–22 are the probabilities when the agent gives the best prompts (which it will converge on). Giving prompts increases the probability of progressing and decreases the probability of regressing, hence the need for this additional set of variables.

The module variable `s` on line 25 represents each stage of the workflow from Figure 4.6, which takes the initial value 0. The variable `m` on line 26 is a counter for the number of mistakes made by the user during the task, initialised as 0.

The example commands on lines 28 and 30 represent when the user is in stages 0 and 1, respectively, and does not require a prompt. As explained for the guarded flag collection PRISM AMDP, the action parameters feature in the guard of the commands. For the aforementioned commands, when the necessary number of mistakes before giving a prompt at each stage (action parameters `prompt0` and `prompt1`) is greater than the current number of mistakes made, `m`, then no prompt is given. Conversely, the commands on lines 33 and 35 specify that if the prompt threshold for each stage is less than the number of mistakes made so far then the system should give a prompt.

PRISM Language 3: Assisted-living system AMDP extracts

```
dtmc

const int prompt0;
const int prompt1;
...
const int MAX_MISTAKES;

const double p00=0.36;
const double p01=0.36;
const double p02=0.28;
const double p10=0.24;
const double p11=0.16;
const double p13=0.48;
const double p14=0.12;
...
const double pp00=0.0864;
const double pp01=0.5407;
const double pp02=0.3729;
const double pp10=0.0439;
const double pp11=0.0585;
const double pp13=0.7239;
const double pp14=0.1737;
...
module patientWorkflow
        s : [0..12] init 0;
        m : [0..MAX_MISTAKES] init 0;

        [] s=0 & m<prompt0 & m<MAX_MISTAKES -> p00:(s'=0)&(m'=m+1)
                + p01:(s'=1) + p02:(s'=2);
        [] s=1 & m<prompt1 & m<MAX_MISTAKES -> p10:(s'=0)&(m'=m+1)
                + p11:(s'=1)&(m'=m+1) + p13:(s'=3) + p14:(s'=4);
        ...
        [] s=0 & m>=prompt0 & m<MAX_MISTAKES -> pp00:(s'=0)&(m'=m+1)
                + pp01:(s'=1) + pp02:(s'=2);
        [] s=1 & m>=prompt1 & m<MAX_MISTAKES -> pp10:(s'=0)&(m'=m+1)
                + pp11:(s'=1)&(m'=m+1) + pp13:(s'=3) + pp14:(s'=4);
        ...
        [] s=10 | m=MAX_MISTAKES -> (s'=10);
endmodule
rewards "distress"
        s=0 : m >= prompt0 ? 1 : 0;
        ...
endrewards
```

Those commands where a prompt is not given use the first set of transition probabilities (such as on lines 8–14). For example, the command at line 28 updates the next state to `s'=0` and increases the mistake counter `m'=m+1`, i.e. the user had made a mistake and remains at the starting step, with probability `p00`. Alternatively, if the user does not make a mistake, with probability `p01` the user will advance to `s1` and with probability `p02` advance to `s2`. This update mechanism is shared exactly by the command on line 33, when a prompt is required, except using the respective probabilities from lines 16–22.

The absorbing state is on line 38, where either the user has successfully reached the end of the task (`s=10`) or they have made too many mistakes and the caregiver will be called for (`m=MAX_MISTAKES`).

The reward structure, starting at line 40, uses state rewards: when in a state, if the number of mistakes so far is equal to or greater than the prompt threshold, meaning a prompt will be given, then a cost of 1 is returned and 0 otherwise.

We generated 10,000 abstract policies using random search taking 114 seconds to complete, averaging at 11.4 milliseconds per policy. We used the probabilistic model checker PRISM which identified 786 abstract policies from the 10,000 generated that had unique quantitative properties and satisfied constraints C_1 and C_2 from Table 4.5. The first safe policy was found on the 4^{th} try, after searching for 45.6 milliseconds. The last unique safe policy was found on the $9{,}977^{\text{th}}$ try. Two optimisation objectives were used to assemble the approximate Pareto front and the set of Pareto-optimal abstract policies in the abstract policy synthesis stage of ARL. The first objective was O_1 from Table 4.5. The second objective, O_2 from Table 4.5, was derived from constraint C_2, i.e. we aimed to minimise the probability of calling the caregiver. The time required to generate the Pareto front was 132 milliseconds. Figure 4.7 shows the entire set of safe abstract policies, as well as the Pareto front. For the last stage of ARL (safe learning), we carried out experiments starting from three abstract policies from different areas of the Pareto front shown in Figure 4.7. Table 4.6 lists these three abstract policies with their associated attributes (i.e. the probability of calling the caregiver and the level of distress to the dementia sufferer). These policies (labelled A, B and C) are also shown in Figure 4.7.

We chose a value of $\epsilon = 0.5$ for all experiments in this case study. Figure 4.8 shows the average progress of all five learning runs for the baseline experiment (without ARL), with error bars used to show the standard error of the mean. For this experiment 10^6 episodes were necessary to reach an optimal policy and each episode had a maximum of 1,000 steps.

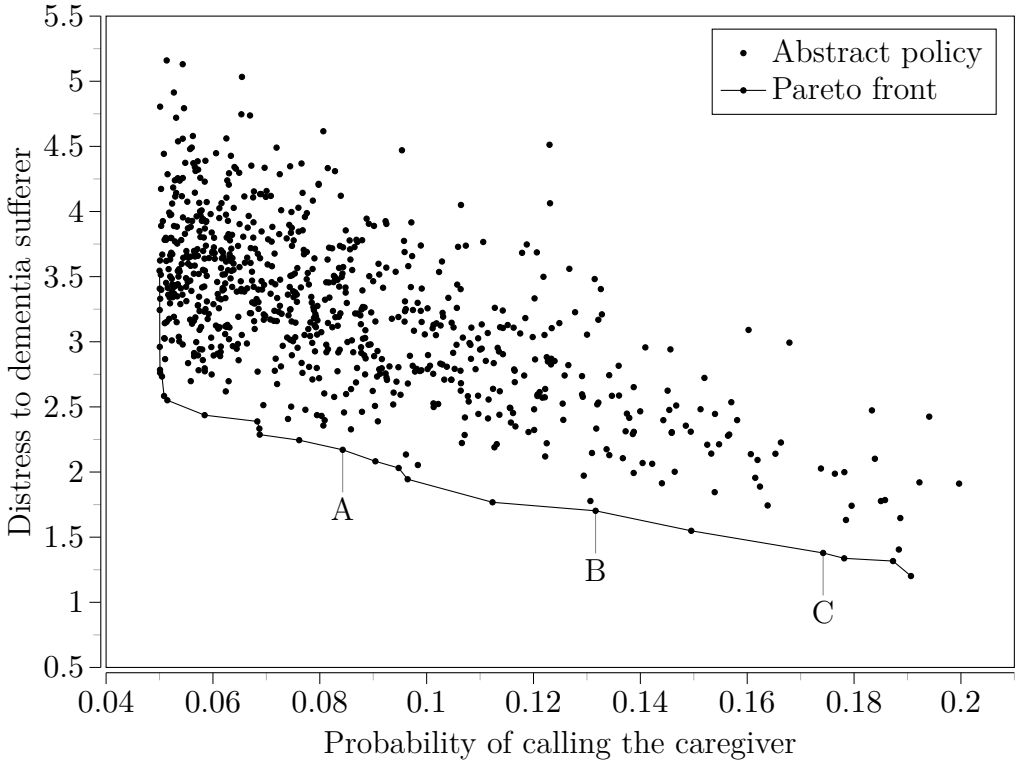


Figure 4.7: Abstract policies and Pareto front for the assisted-living system. Those policies that were selected for ARL are labelled A, B and C.

Table 4.6: Selected abstract policies used during the safe learning stage of ARL for the assisted-living system.

Abstract policy	**Probability of calling the caregiver**	**Distress to dementia sufferer**
A	0.08	2.17
B	0.13	1.70
C	0.17	1.38

Following the baseline experiment, we carried out a series of experiments for each of the three selected abstract policies from Table 4.6. The learning progress of these experiments is shown in Figure 4.9. More learning episodes were necessary for the ARL experiments since for many states the abstract policies prevented a prompt being given, delaying the agent's ability to explore and learn about different prompt styles.

Table 4.7 shows examples of how a user may progress through the stages of the hand-washing task and when the assisted-living system, utilising the safe RL policies learned within the constraints from abstract policies A, B and C, decides to provide a prompt and/or call for caregiver assistance. A prompt is given at

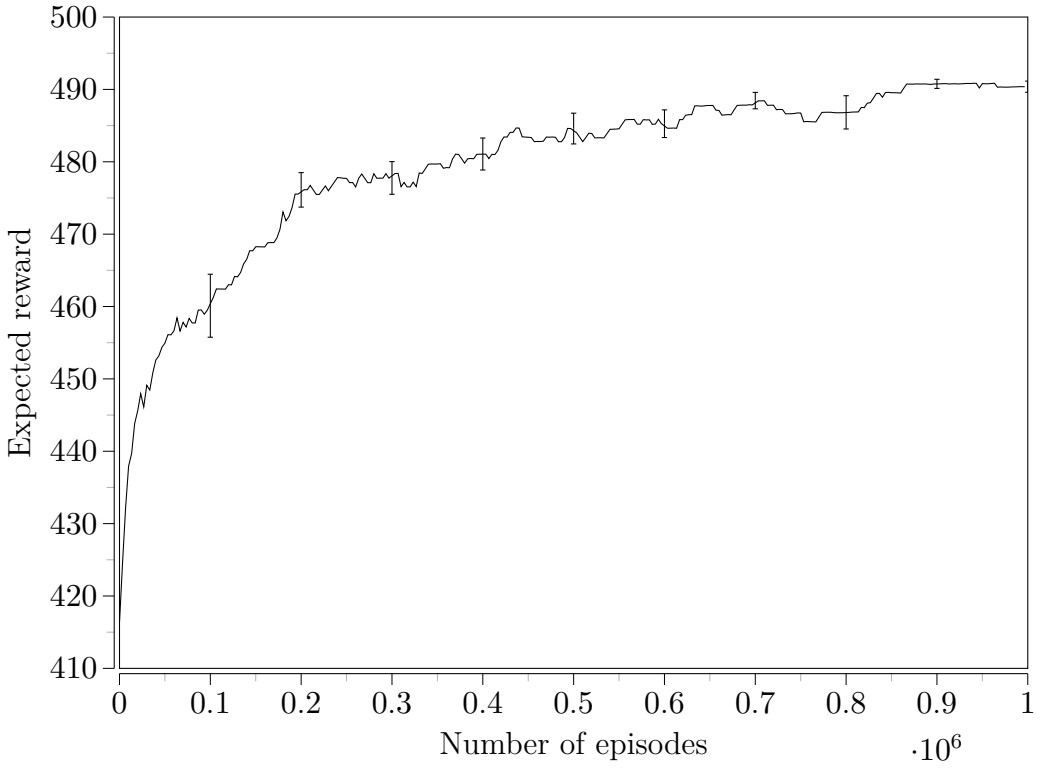


Figure 4.8: Learning progress for the assisted-living system with no ARL applied.

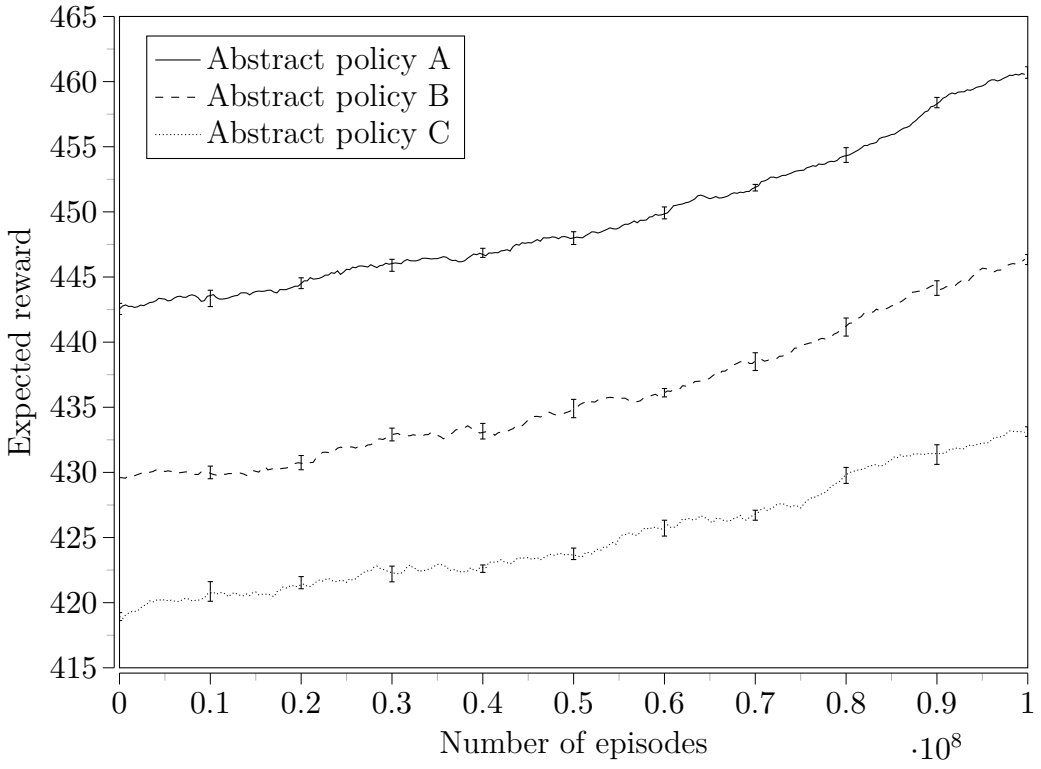


Figure 4.9: Learning progress for the assisted-living system with ARL applied using the selected abstract policies A, B and C.

a stage if the number of mistakes made by that time exceeds the threshold of safety defined by each parameter of the abstract policy. This threshold may vary for each stage of the process. Note that not every instance of policy A and B being applied will not involve calling the caregiver, nor does policy C necessarily call the caregiver every time. Instead, similar to when prompts are given, the caregiver is called only when a maximum number of mistakes has occurred, as specified by each policy.

Contrasting the results from the baseline experiment and the ARL experiments, it is clear that the action constraints are having the expected effect on the learned policy. In particular, comparing the probabilities of calling the caregiver and the level of distress to the dementia sufferer against those that were verified for the abstract policies, the action constraints are having the desired effect, with all the results being close to or matching the values shown in Table 4.6. The slight difference from abstract policy C's probability of calling the caregiver can be attributed to the learned policy not being entirely optimal and further learning should reduce the variance to zero.

These results can be explained when considering the reward structure used for this system, how the learning algorithms optimise a solution from it and, in the case of ARL, how the safety constraints shape the behaviour by the agent.

There is a large negative value associated with calling the caregiver (since the chief purpose of the system is to be a surrogate caregiver), a small negative cost for giving a prompt (since, despite being beneficial, they will increase user distress levels) and lastly there is a large positive reward, greater in magnitude than for calling the caregiver, for completing the task. Except for very rare occurrences when the user is making an excessive number of mistakes, requiring excessive prompts to rectify, the total cost of the prompts needed by the user to succeed will ordinarily be far less than the cost of calling the caregiver. Even though calling the caregiver still returns a net positive reward (as it still results in the task ending successfully), it is expected that a greater net reward will be returned by always giving prompts instead. Therefore, the baseline RL algorithm optimises a solution to always give prompts and never call the caregiver, hence the high distress level to the patient and minuscule probability of calling the caregiver. We can expect that with additional learning episodes the probability of baseline RL calling the caregiver will converge to zero.

The ARL constraints, however, *require* that the agent sometimes calls the caregiver and so for some states (i.e. where a certain number of mistakes has occurred) removes all other actions from the agent, forcing it to call the caregiver.

Table 4.7: Examples of user progression through the subtasks of the hand-washing process. 'Stage' represents the progress level the user is in, defined in Figure 4.6. 'Prompt' and 'Caregiver' indicate whether a prompt is given or the caregiver is summoned, respectively, by the agent according to the safety constraints. 'Mistakes' and 'Prompts' are the number of mistakes made and prompts given, respectively, when reaching each stage.

(a) Safe abstract policy A.

Stage	Prompt	Mistakes	Prompts	Caregiver
s_0	No	n/a	n/a	No
s_2	No	0	0	No
s_4	No	0	0	No
s_2	No	1	0	No
s_2	No	2	0	No
s_2	No	3	0	No
s_2	No	4	0	No
s_2	**Yes**	5	0	No
s_4	No	5	1	No
s_5	No	5	1	No
s_{12}	No	6	1	No
s_{12}	No	7	1	No
s_5	**Yes**	7	1	No
s_6	**Yes**	7	2	No
s_7	**Yes**	7	3	No
s_9	**Yes**	7	4	No
s_{10}	n/a	7	5	n/a

(b) Safe abstract policy B.

Stage	Prompt	Mistakes	Prompts	Caregiver
s_0	No	n/a	n/a	No
s_0	No	1	0	No
s_0	No	2	0	No
s_2	No	2	0	No
s_2	No	3	0	No
s_4	No	3	0	No
s_5	No	3	0	No
s_6	No	3	0	No
s_6	No	4	0	No
s_5	No	5	0	No
s_5	No	6	0	No
s_{12}	**Yes**	7	0	No
s_5	**Yes**	7	1	No
s_6	**Yes**	7	2	No
s_7	**Yes**	7	3	No
s_9	**Yes**	7	4	No
s_{10}	n/a	7	5	n/a

(c) Safe abstract policy C.

Stage	Prompt	Mistakes	Prompts	Caregiver
s_0	No	n/a	n/a	No
s_0	No	1	0	No
s_2	No	1	0	No
s_2	No	2	0	No
s_2	No	3	0	No
s_2	No	4	0	No
s_4	No	4	0	No
s_5	No	4	0	No
s_6	No	4	0	No
s_7	No	4	0	No
s_9	No	4	0	No
s_2	**Yes**	5	0	No
s_4	No	5	1	No
s_5	No	5	1	No
s_5	No	6	1	No
s_5	No	7	1	No
s_5	No	8	1	No
s_5	No	9	1	**Yes**

Equally, the constraints will only allow the agent to give prompts in certain states (e.g. where there is a high probability the user will make a mistake), thereby preventing the agent from optimising a solution to only give prompts. As intended, this is the reason that the ARL-based solutions have a greater probability of calling the caregiver and significantly lower distress levels to the patient.

Table 4.8: Results of the baseline and ARL experiments for the assisted-living system.

Abstract policy	**Probability of calling the caregiver**	**Standard error**	**Distress to patient**	**Standard error**
None	4.02×10^{-4}	4.28×10^{-4}	8.31	4.03×10^{-3}
A	0.08	4.95×10^{-4}	2.17	3.25×10^{-3}
B	0.13	5.17×10^{-4}	1.70	2.22×10^{-3}
C	0.18	4.27×10^{-4}	1.38	1.84×10^{-3}

4.5 Comparison to Existing Approaches

This section will compare ARL to the alternative safe RL techniques introduced in Chapter 3. Of the eight alternative techniques, though, only two can be applied to our case studies, those being the worst-case and risk-sensitive techniques (evaluated in Sections 4.5.1 and 4.5.2, respectively). The other techniques, as explained below, have assumptions or requirements (see Table 3.1) that makes them incompatible with our case studies or incapable of producing a usable solution.

Ergodic Policies. This technique assumes that the problem being solved allows an ergodic solution. More specifically, that a *useful* ergodic solution exists where all states featured in it have a probability greater than zero of allowing an agent to (re)visit any other state in the solution. For this reason, ergodic policies are not suited to the two case studies in this chapter, since a necessary part of these problem's solutions is to enter into absorbing 'goal' states—distinct from the initial states—from which the probability of returning to any other state is zero. An ergodic solution in either case study necessarily would not include the goal states, preventing the agent from ever finishing the tasks.

Permissive Schedulers. This technique requires the full transition function of the problem MDP so that it can be verified using model checking techniques.

However, in neither case study is this function known in full (hence the use of an AMDP in ARL) and so it cannot be applied.

Teacher Knowledge and Safe Demonstrations. Both of these techniques require knowing a safe solution in advance, the former to be used to guide the agent and the latter to form a baseline behaviour from which the agent can optimise over. These techniques are not intended to *identify* safe behaviour but are instead designed to *impart* it, since the behaviour may too complex or subtle for the agent to learn through a reward scheme alone. Since neither of the case studies in this chapter start with a known safe solution, these techniques cannot be applied.

Cautious Simulation. In contrast to other techniques, cautious simulation is designed specifically for robotics systems and uses a physics simulator to detect and avoid collisions/crashes etc. It is, therefore, incompatible with our case studies.

Backup Policies. This technique requires constructing a bespoke 'risk perception' function to analyse possible next-states to see if their features qualify them as 'risky'. In neither of our case studies could such a risk function be devised since individual states are not 'risky' or even unsafe. Instead, it is sequences of states (which are unknown in advance) which cumulatively determine the level of risk caused by a set of actions.

4.5.1 Worst-Case Criterion

As outlined in Chapter 3, the worst-case technique uses a modified form of Watkins' Q-learning algorithm [58] to optimise a solution assuming the worst possible outcome of a system will always occur. This technique can be applied to our case studies as its requirements and assumptions are modest. Namely, that safety aspects are encoded into the reward structure of the MDP. Whilst such rewards are sparse in our case studies (since ARL does not rely on these rewards to assure safety) they do still exist in a limited sense and so the algorithm can potentially produce a useful solution.

To enable the worst-case learning algorithm to converge correctly, the Q-table must be initialised optimistically, i.e. the initial Q-values must be set to a positive value instead of being arbitrary. This is necessary as the learning algorithm can only *decrease* Q-values, never increase them, so Q-values which are initialised arbitrarily with values much lower than they are truly worth cannot be adjusted

[89, 90]. Otherwise, all other experimental parameters and settings were set to be the same as for the baseline RL experiments for each case study.

The results of each case study can be expressed concisely. For the guarded flag collection case study, in all five experiments, without variation, the agent learned to immediately head to the goal area, collecting zero flags along the way. The probability of reaching the goal was 1 and the expected reward was 1 (i.e. for successfully reaching the goal). For the assisted-living system case study, in all five experiments, again without variation, the agent learned to immediately call for the caregiver and did not attempt any prompts or allow the user any opportunity to attempt the task unassisted. Although the distress to the patient was 0, the probability of calling the caregiver was 1. In both case studies the solutions failed to satisfy the requirements, as outlined in Section 4.2 for the guarded flag collection mission and Table 4.5 for the assisted-living task.

To understand these results, consider the nature of the worst-case optimisation algorithm and what exactly *is* the worst outcome of each case study. The algorithm assumes that if something can go wrong then it will go wrong (i.e. the 'worst-case' scenario) and therefore the solution should be that which maximises the reward under this expectation.

For the guarded flag collection case study, in terms of rewards the worst case is to fail to collect any flags and then be captured. Therefore, assuming this worst case is inevitable, the algorithm optimises a solution to head straight to the exit, avoiding the areas where it can (and therefore *will*) be captured, and ignoring all the flags (since it *won't* successfully collect any even if it tried).[2]

In terms of rewards for the assisted-living system, the worst outcome is to provide the user with a multitude of prompts only for them to still make repeated mistakes, potentially continuing for an infinite amount of time (even though the probability of this occurring approaches zero). Nevertheless, since the probability of this occurring is *not* zero, this worst-case *is* possible and so the agent assumes that it will occur. Therefore, the best solution assuming this outcome is to immediately call for the caregiver, bypassing the distress caused to the patient by the prompts.

[2]The author of [90] proposes the term 'Heinzmann's dead-is-dead conjecture' to refer to the unrealistic nature of the algorithm to treat being captured (or 'dying') without collecting any flags as being worse than to be captured after collecting some flags, since in either case the agent has irredeemably failed. Nevertheless, to the algorithm the former case is worse and so optimises a solution assuming it, hence it ignores all flags, including FlagD which it could collect at no risk.

4.5.2 Risk-Sensitive Criterion

Similar to the worst-case technique, this technique has few requirements or assumptions, primarily that safety features are encoded into the reward function of the MDP. Therefore, this algorithm can potentially succeed with the case studies.

For all experiments we use the same parameters and settings as the baseline RL experiments done for each case study. The only additional parameter required is a value k for the degree of risk seeking/aversion the algorithm will apply. To the best of this author's awareness there are no specific guidelines or rules on what value this parameter should take, with trial and error being the only approach to find a suitable value. The general principle, though, as outlined in [92], is that negative values of k encourage risk seeking, with the likely outcome being higher expected rewards but with a higher risk of failures, and positive values can expect lower expected rewards but at a decreased risk of failures. Setting k to 0 reduces the risk-sensitive algorithm to the standard, risk-*neutral* Q-learning algorithm.

For the guarded flag collection experiment, the traditional RL experiment, using risk-neutral Q-learning, produced results which had a probability of capture too high (i.e. the risk of the solution was too high). Therefore, we can expect that setting k to a positive value may decrease this risk level and produce a solution that satisfies the safety requirements. Conversely, negative values may worsen the solution's risk levels.

To confirm this expectation, two negative values of the risk parameter were first tested: an extreme of -0.9 and an intermediate of -0.45. As expected, both gave higher rewards but at greater risk than the baseline Q-learning experiment (equivalent to $k = 0$). Since negative values of k manifestly produce unsafe solutions, the search for safe parameter was focussed on positive values. Figure 4.10 shows the outcome when increasing k in 0.1 increments starting from $k = 0$ and Table 4.9 gives the exact results.

From these graphs we can see that the requirements of reaching the goal with a probability ≥ 0.75 and an expected reward > 2 can be satisfied when using a risk parameter in the approximate region $0.1 \lessapprox k \lessapprox 0.8$. This is to be expected, since, as discussed above, the baseline experiment (equivalent to $k = 0$) was too risky, and therefore a risk-averse parameter was likely to produce a satisfactory solution.

Since a positive k overweights negative temporal differences and underweights positive differences, as the parameter value increases the agent is increasingly

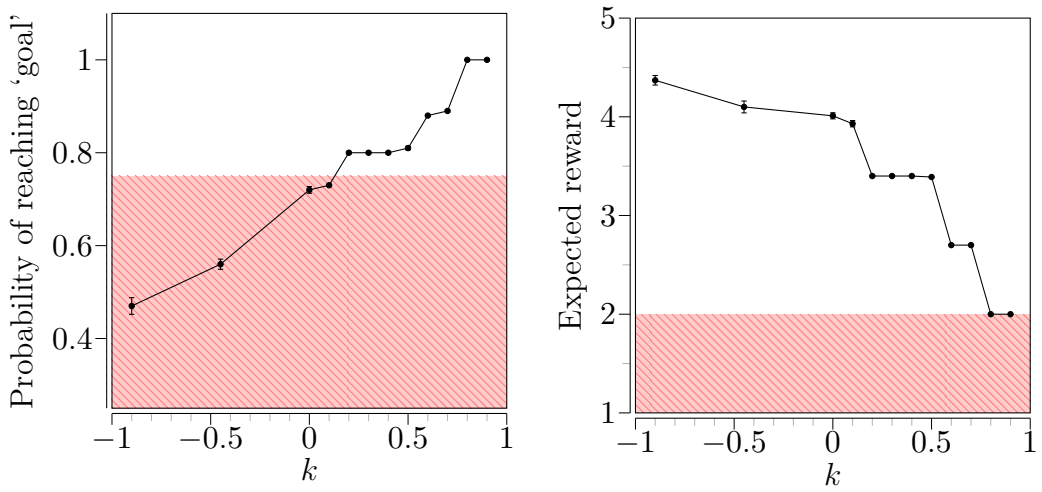


Figure 4.10: Effect of varying risk parameter k in the guarded flag collection case study. Red-shaded regions represent where solutions (black nodes) do not satisfy the safety/optimisation requirements.

dissuaded from collecting flags since they are worth less and the punishment for capture is greater. This is shown in the results, where the higher the value of k becomes, the safer, albeit less profitable, the agent's solution becomes. Where different values of k return similar results, the solutions are collecting the same flag combinations so return the same reward at the same level of risk. Since flags are discrete rewards we cannot expect a continuous change in the reward obtained. Equally, since the capture areas are associated to specific flags, the risk involved with collecting the flags will remain constant when the actions to collect them are optimised.

Although the risk-sensitive algorithm was able to produce a range of safe solutions approaching similar quality to those found by ARL, obtaining these solutions revealed two major drawbacks of the technique. First, as with the baseline RL experiment for this case study, the number of learning episodes required to approach an optimal solution was vastly more than was required by ARL. It is possible that with even more learning episodes the solutions may converge to be of equal quality to those found by ARL. However, the excessive number of episodes required to fully converge makes it impractical to do so. Second, finding an appropriate value of k is trial and error; assuming that the first choice of k is not satisfactory by coincidence, numerous experiments must be tried until a value is found that is suitable. The probable need for multiple experiments worsens the first problem of excessive learning episodes.

With the assisted-living case study, we can expect that negative values of k will produce a solution that relies on giving prompts and positive values of k will

Table 4.9: Results of the risk-sensitive learning algorithm when applied to the guarded flag collection case study.

k **parameter**	**Probability of reaching 'goal'**	**Standard error**	**Expected reward**	**Standard error**
-0.9	0.47	0.018	4.37	0.048
-0.45	0.58	0.011	4.1	0.06
0	0.72	7.3×10^{-3}	4.01	0.031
0.1	0.73	1.8×10^{-3}	3.93	0.032
0.2	0.8	3×10^{-3}	3.4	0.012
0.3	0.8	2×10^{-3}	3.4	4×10^{-3}
0.4	0.8	3×10^{-3}	3.4	3×10^{-3}
0.5	0.8	4.2×10^{-3}	3.39	3.8×10^{-3}
0.6	0.88	2×10^{-3}	2.7	3×10^{-3}
0.7	0.89	3.1×10^{-3}	2.7	2.5×10^{-3}
0.8	1	0	2	0
0.9	1	0	2	0

instead call for the caregiver immediately, or perhaps only give a small number of prompts initially. This is since calling the caregiver immediately will always return an exact, unchanging reward: the cost of calling the caregiver plus the reward for finishing. Instead, giving prompts involves the possibility of the user making an arbitrary number of mistakes, requiring an arbitrary number of prompts and returning a varied expected reward. Since positive values of k reduce the variance of the reward, this would encourage the agent to call the caregiver, returning a constant reward. Negative values of k encourage a solution with higher reward variance so will cause the agent to give prompts.

Once again, we test our expectations by starting with negative values for k to determine whether to pursue risk-seeking or risk-averse parameter values. After determining which direction to take we use the bisection method to hone in on the ideal parameter range. Figure 4.11 illustrates the resulting solutions' properties and Table 4.10 gives the results.

The graphs in Figure 4.11 show there is an abrupt effect of the parameter when set to a high, positive value; there is not a gradual change for the probability of calling the caregiver or for the distress to the patient as the parameter increases.

This abrupt change makes sense when considering the nature of the reward scheme. As discussed with the baseline RL results from Section 4.4.2, the optimal solution for standard RL is to always give prompts as, under normal circumstances, this will return a greater expected reward than for calling the caregiver.

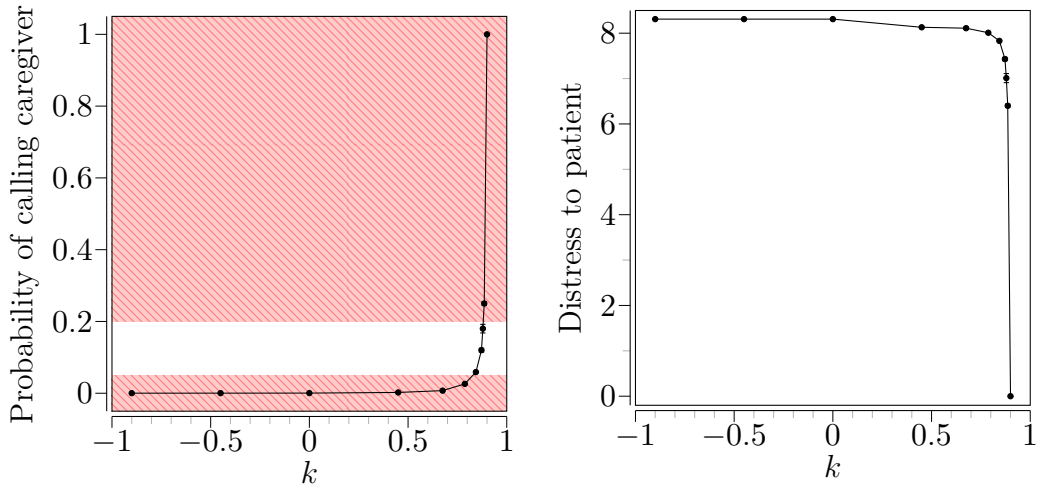


Figure 4.11: Effect of varying risk parameter k in the assisted-living system case study. Red-shaded regions represent where solutions (black nodes) do not satisfy the safety requirements.

Table 4.10: Results of the risk-sensitive learning algorithm when applied to the assisted-living system case study.

k **parameter**	**Probability of calling the caregiver**	**Standard error**	**Distress to patient**	**Standard error**
-0.9	2×10^{-4}	6.3×10^{-5}	8.31	1.2×10^{-3}
-0.45	2.4×10^{-4}	6.8×10^{-5}	8.31	3.5×10^{-3}
0	4.02×10^{-4}	4.28×10^{-4}	8.31	4.02×10^{-3}
0.45	2.2×10^{-3}	3.2×10^{-4}	8.13	7.4×10^{-3}
0.675	7.3×10^{-3}	4.2×10^{-4}	8.11	0.026
0.7875	0.026	1.9×10^{-3}	8.01	0.013
0.84375	0.059	9.2×10^{-4}	7.83	9.1×10^{-3}
0.871875	0.12	5.5×10^{-3}	7.43	0.038
0.87890625	0.18	0.012	7.01	0.1
0.8859375	0.25	5.2×10^{-3}	6.4	0.038
0.9	1	0	0	0

Instead, if the cost for calling the caregiver were much smaller and the cost for giving a prompt were much higher, then the agent would learn to always call the caregiver as this will return a greater expected reward.

A high value of k makes negative rewards significantly more punitive but also means that positive rewards are significantly diminished. This distortion of rewards can cause problems since the true long-term quality of actions is lost [52]. As Q-learning (the basis of the risk-sensitive algorithm) propagates rewards across states, a high-valued risk parameter will significantly diminish the reward at every step of the propagation. Therefore, with a high value of k,

initial actions that may ordinarily expect to yield a large reward in the future will receive a drastically diminished one when the reward is eventually propagated to them.

In this way, the net positive reward the agent receives by immediately calling the caregiver in the beginning, albeit also diminished by the risk parameter, is now greater than if the agent were to provide prompts since it has only been diminished *once*. Instead, the ordinarily larger reward returned from giving prompts is *successively* diminished to the point that the actions leading to it are updated to have less utility than to call for the caregiver immediately. This is the reason for the abrupt change in solution properties when setting k to a high value: beyond a certain value of k, always calling the caregiver becomes better than always giving prompts, it is not a gradual change.

From Table 4.10 we can see that a risk parameter value of approximately $0.844 \lessapprox k \lessapprox 0.879$ can produce solutions that satisfy the requirement that the caregiver must be called with a probability between 0.05 and 0.2. However, as discussed above, there should be a discrete change from a probability of 0 to a probability of 1, a continuous range is not to be expected. The explanation for the slight transition range is that the solutions in this region were not perfectly optimal and contained Q-values that were still in flux when learning ended and so the solution would arbitrarily call the caregiver/give prompts in some states. The sole exception to this is for the single value of k which will result in the reward for calling the caregiver to be identical to the reward for giving prompts, in which case the agent could either call the caregiver or give a prompt with equal preference and would not exclusively choose one over the other.

Even though it can be argued that the technique nevertheless did produce safe solutions satisfying the safety requirements, the distress to the patient for each solution was significantly higher than when using ARL since the solutions were still largely optimised to always give prompts and only to call the caregiver in some arbitrary states. Furthermore, as experienced with the guarded flag collection case study, numerous experiments had to be conducted to eventually find suitable safety parameters.

4.6 Summary

This chapter has introduced the assured reinforcement learning (ARL) approach and shown how it can be used to satisfy strict safety requirements for RL solutions. The ARL approach comprises two stages, with a preliminary step for

construction of the high-level AMDP.

The preliminary step of creating the AMDP involves extracting all relevant features of the problem environment which affect the safety and optimisation properties of a solution. The use of an AMDP makes it feasible to reason about the problem using QV, since typical RL MDPs can be far too vast in size for them to be verified in a reasonable amount of time. Furthermore, it may not always be possible to analyse the RL MDP since its transition and/or reward functions may not be known. Using the AMDP, the two stages of ARL follow:

1. The first stage involves searching for a set of abstract policies for the AMDP which have been verified by QV to satisfy the safety constraints and optimisation objectives. A Pareto-optimal set of these policies is generated, allowing a user to select an abstract policy which has the desired level of compromise between the safety/optimisation properties.

2. The second stage is to use the selected safe abstract policy as a set of safety constraints on the RL agent's action choices. When an abstract policy is applied during safe RL, should the agent attempt an action in a state that would cause the agent to enter into a state that does not feature in a high-level state as part of the abstract policy, this action is disallowed, forcing the agent to optimise over the remaining actions which are safe.

Through two different case studies it is demonstrated that ARL can be successfully applied and can produce RL solutions which satisfy its safety and optimisation requirements. Furthermore, these solutions will match the safety levels verified for the safety constraints. These results are in contrast to those for standard RL and other safe techniques which either failed to meet the required safety levels, or did so less effectively and with significant drawbacks.

The limitations of the ARL approach are twofold: First, ARL assumes that the information used to create the AMDP is both accurate and complete (with respect to the important features of the RL MDP). If this assumption is not satisfied then ARL cannot be guaranteed to produce an RL policy which satisfies the safety requirements. This limitation is addressed in Chapter 5. The second limitation is how abstract policies are synthesised. As noted in Section 4.4.1, an AMDP can potentially have a vast number of candidate abstract policies and it may be the case that very few of them, or even none at all, satisfy the safety requirements. Therefore, the search process may fail to find a safe abstract policy in a reasonable amount of time, assuming one exists at all, or those that it does identify may not be as optimal as it is possible to achieve within the safety

requirements. This limitation can be ameliorated through the efficient design of the AMDP: by only including states and transitions that are strictly necessary, the abstract policy space can be kept to a minimum size. Additionally, using an appropriate search heuristic can maximise the probability of identifying a suitable safe abstract policy.

Chapter 5

Knowledge Revision

The previous chapter detailed the ARL approach whereby an RL solution can be learned that is guaranteed to meet a set of strict safety requirements. A key feature of the approach is the use of an AMDP to model the RL problem at a high level, which can be constructed with limited knowledge of the problem environment since it is unnecessary to know all low-level details. Furthermore, using an AMDP makes it feasible to apply QV, since the complete RL MDP is often too large to be verified in a timely manner, or may not be fully known and therefore cannot be verified at all.

Whilst it is advantageous to use an AMDP for these reasons, its successful application has two assumptions. The first is that the model contains *all* pertinent information of the problem environment, i.e. all transitions and rewards that may influence the safety and optimisation properties of an RL solution. The second is that all the information used to construct the AMDP is *accurate*.

If either of these assumptions is not satisfied then safety and/or optimisation requirements cannot be guaranteed since what the RL agent encounters in the low-level MDP may not correspond with the AMDP used to generate the safety constraints. This can result in the agent attempting to perform an action specified by the abstract policy which it cannot do (e.g. attempting to transition to a state specified by the abstract policy where no such transition exists in the RL MDP) or performing an action which has different consequences than is expected (e.g. if the agent enters into a state that is disallowed by the abstract policy). Furthermore, if the RL transition probabilities or reward magnitudes are different to what the abstract policy was verified for then the resulting RL solution may not have the same safety levels as intended.

To address this problem, this chapter introduces ARL with *knowledge revision* (ARL-KR), an approach to automatically update the AMDP model with accurate knowledge of the RL MDP.

5.1 Introduction

ARL-KR consists of two algorithms: one for *updating* the AMDP with accurate information and one for *reusing* previously optimised actions, when possible.

The update algorithm is a loop that first invokes standard ARL and attempts to learn a safe RL solution. If the produced RL solution does *not* satisfy the safety requirements as intended, meaning the AMDP used to generate the safety constraints is incomplete or inaccurate, then the AMDP is amended with up-to-date information about the RL MDP. This could be acquired by the RL agent when it explores the MDP, or it may be supplied by external sources, or both. After updating the AMDP the loop restarts, generating a new set of abstract policies for safe RL, and eventually terminates once the RL policy is safe or a maximum search time has elapsed.

Examples of where inaccuracies can arise are if the transition probabilities or rewards used to construct the AMDP were in fact approximations, or were not known to exist at all. Additionally, if these attributes change from their initial values once the system has started, so despite the AMDP being accurate at the start of the learning process, when the RL agent eventually encounters them they may no longer be the same.

The action reuse algorithm exhaustively analyses previously attempted abstract policies to identify any high-level actions which also exist in the newly generated safe abstract policy. Where they are found, meaning that these specific actions were unaffected by the knowledge update, the low-level MDP state-action pairs that were previously optimised for these specific high-level options can be reused as they may still accurately reflect which actions for those states are best. The output of this algorithm is an 'initialised' Q-table for use by the RL agent with the goal to increasing its learning speed.

ARL-KR can utilise existing techniques such as [77, 104, 105, 106] for observing transition probabilities and reward magnitudes in stochastic environments. In this way, ARL-KR is capable of updating the AMDP with new transition probabilities and reward values from the RL environment; however, it is not currently possible to update the AMDP to incorporate new *states* which were initially unknown and therefore were not included in the AMDP. The inclusion of new states into the AMDP can potentially require significant restructuring of the model for which no automatic solutions currently exist.

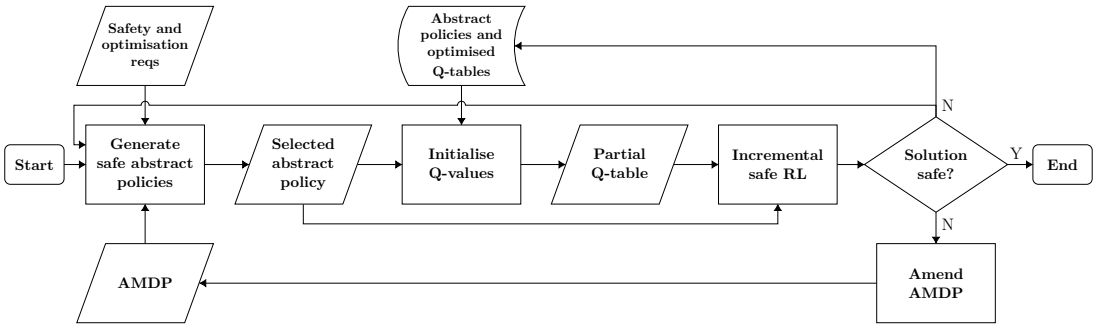


Figure 5.1: Flowchart for ARL-KR showing the stages for knowledge revision ('Amend AMDP') and Q-value reuse ('Initialise Q-values').

5.2 Approach

The ARL-KR approach is outlined in Figure 5.1. The process starts with standard ARL by generating a set of Pareto-optimal abstract policies using an AMDP that is assumed to be correct. A chosen policy is supplied to the Q-value initialisation function and also to incremental safe RL. Q-value initialisation searches all previously attempted abstract policies to see if any Q-values learned for them can be reused. The resulting partial Q-table, containing any Q-values that can be reused, is provided to the incremental safe RL process which allows the agent to start with those Q-values for actions which have previously been optimised. This provides the agent with a head start since its initial exploration is less arbitrary and can, therefore, identify a fully optimal solution faster. After RL has finished, the solution is verified empirically to assess its safety levels. If it satisfies the requirements then the solution can be deployed and the ARL-KR process terminates. Alternatively, if the solution is not safe, the AMDP is updated with up-to-date information. Simultaneously, the abstract policy and the Q-table that was optimised for it are stored in a cache so they can be reused when possible.

The approach comprises a main update loop and Q-value reuse algorithm, both of which are detailed below:

Update loop. Algorithm 4 details this process. The algorithm takes as its inputs the initial AMDP $\bar{M}$, a set of safety constraints $\mathcal{C}$ and optimisation objectives $\mathcal{O}$.

When using ARL-KR instead of core ARL, the AMDP needs to be constructed slightly differently. The AMDP for ARL assumes that all safety- and optimisation-relevant transitions and rewards are known in advance and, there-

Algorithm 4 Iterative update loop for ARL-KR

```
1: function ARL-KR(M̄, C, O)
2:     Policies ← {}
3:     do
4:         PS = GenAbstPolicies(M̄, C, O)
5:         π̄ = select from PS
6:         Q = InitQValues(Policies, π̄, M̄)
7:         Q* = IncARL(π̄, Q)
8:         isSafe = EvaluateSafety(Q*, M̄, C, O, π̄)
9:         if ¬isSafe then
10:            Policies ← Policies ∪ {(π̄, Q*)}
11:            M̄ ← Observe
12:        end if
13:    while ¬isSafe ∧ ¬maxSearchTime
14: end function
```

fore, the AMDP can be refined in such a way that impossible transitions between states and unnecessary state features can be omitted. By doing this, the model size is minimised, possibly also the number of parameters needed for an abstract policy. For ARL-KR, however, the AMDP should be constructed to include potentially redundant transitions between all states, as well as redundant rewards for transitioning to them. This is to allow updates to occur should it transpire that certain transitions or rewards *do* exist that were previously unknown. This is achieved by creating a transition and associated reward between all states, but setting their values to zero since assuming they don't exist in the MDP then they will have no effect in an abstract policy. However, as updated information about transitions and rewards in the RL MDP is acquired, if it turns out that certain transitions or rewards do exist then their respective zero-valued variables can be adjusted appropriately which may then influence an abstract policy.

As noted in the previous section, though, it is not currently possible for ARL-KR to include new states into the AMDP since it cannot be done by simply updating a variable value. This could involve potentially major restructuring of the model which would require a bespoke AMDP generator for each problem environment.

Before the loop commences, an empty set *Policies* is initialised (line 2). This will cache abstract policies and their optimised RL Q-table each time a learning run has occurred. Caching optimised Q-tables allows the potential to reuse optimised Q-values if they are applicable to subsequent abstract policies.

The loop begins by first applying the function GenAbstPolicies (line 4)

which generates a Pareto front of safe abstract policies, as outlined in Chapter 4.3. From this Pareto front, a safe abstract policy $\bar{\pi}$ is selected (line 5), either by the human user or picked according to some automated policy selection strategy, for use to constrain the RL agent in standard ARL fashion. Next, an initialised Q-table is generated by the function InitQValues (line 6). This function inspects all previously tried abstract policies to extract any reusable Q-values from their respective optimised Q-tables to produce the initialised Q-table Q. This procedure is detailed in Algorithm 5.

Using the resulting Q-table (which will be arbitrary for the initial iteration since no Q-values have previously been optimised), the function IncARL is undertaken (line 7) to produce the optimised Q-table Q^*. In contrast to standard ARL, *incremental* ARL reoptimises a Q-table each run, instead of starting each learning run with an arbitrary one. The intention of this is that previously optimised actions can be utilised again, thereby increasing the speed that an optimal policy is learned.

After a learning run has completed, the safety of Q^* is evaluated by the function EvaluateSafety (line 8). This function takes the optimal Q-table Q^*, the AMDP $\bar{M}$, the safety constraints $\mathcal{C}$ and optimisation requirements $\mathcal{O}$ and also the abstract policy $\bar{\pi}$ that was used in the learning run. The function checks if Q^* satisfies the requirements, as well as comparing its safety levels to that of $\bar{\pi}$, and returns a boolean value *isSafe* as the result. It may be the case that despite not exactly matching the properties of the abstract policy (i.e. there was incorrect knowledge in the AMDP) it may in fact still satisfy its requirements. It is therefore down to the decision of the user what the specific criteria is for being acceptable or not.

If the optimal policy is evaluated as unsafe (line 9), i.e. *isSafe* is false, then *Policies* is updated to include the attempted abstract policy $\bar{\pi}$ and its associated optimised Q-table Q^* (line 10). Additionally, the AMDP $\bar{M}$ is updated with accurate transition probabilities and rewards using the Observe function.

Model updates include transition probabilities between high-level states and their rewards. The Observe function does not dictate where new information is obtained from, as such it may be provided by external sources if applicable. Alternatively, information can be identified by analysing the effects of the learned RL policy. This can be done based on techniques such as those described in [77, 104, 105, 106]. For example, the algorithm introduced in [77] for automatically deriving a potential function [81] can be simplified so that it is only used to update the transition and reward function of the AMDP, without the further step of

solving it. This algorithm samples high-level actions in the RL environment and stores a running average of the rewards received and transition probabilities across the low-level MDP states that are encountered. Provided a suitable number of samples are taken then the observed transitions and rewards converge to their correct values. In this way, each high-level action specified by the abstract policy constraint can be sampled and the AMDP updated accordingly.

Finally, the loop restarts and continues until either the RL policy evaluates as being safe, or until a maximum search time has elapsed indicating that no safe policy may exist at all.

Q-value initialisation. It is possible that useful behaviour can be incorporated into the RL agent through Q-value initialisation [89], instead of the agent starting with initially arbitrary behaviour. Algorithm 5 details how Q-values which have been optimised in previous safe RL attempts can be reused in the form of an initialised Q-table.

Algorithm 5 Q-value initialisation using previously optimised Q-tables

```
1: function InitQValues(Policies, π̄′, M̄)
2:     Initialise Q arbitrarily
3:     for (π̄, Q*) ∈ Policies do
4:         for all transitions t allowed by π̄ do
5:             for all transitions t′ allowed by π̄′ do
6:                 if t = t′ then
7:                     identify source abstract state s̄ from t
8:                     for all low-level states s in s̄ and all actions a in A do
9:                         Q(s, a) = Q*(s, a)
10:                    end for
11:                end if
12:            end for
13:        end for
14:    end for
15:    return Q
16: end function
```

The input to the algorithm is the set *Policies* containing the abstract policies and their respective Q-tables that were cached in Algorithm 4. In addition, the newly chosen abstract policy $\bar{\pi}'$ is provided as well as the AMDP $\bar{M}$. The goal is to identify any transitions allowed by the newest policy which have also been allowed in any previous policy.

The algorithm starts by initialising a Q-table Q arbitrarily (line 2). Note that this, unaltered arbitrary Q-table will be returned in the first iteration of the

update loop (Algorithm 4, line 6) since by this point *Policies* is empty so no Q-values could be initialised with previously optimal values. Following this are a series of nested loops to identify which transitions are common to any previous abstract policy and the newest one to be tried.

The outermost loop (line 3) iterates over each previously tried abstract policy $\bar{\pi}$ and its respective optimised Q-table Q^* contained in *Policies*. This loop is necessary since it may be the case that many iterations of Algorithm 4 have occurred, producing $N > 1$ number of Q-tables, each potentially containing reusable Q-values. A second, inner loop (line 4) then iterates over each transition t allowed by the abstract policy $\bar{\pi}$, where a transition is allowed if $\bar{T}\big(\bar{s}, \bar{\pi}(\bar{s}), \bar{s}'\big) > 0$ for any $\bar{s}, \bar{s}' \in \bar{S}$. Equally, the innermost loop (line 5) iterates over each transition t' allowed by the current abstract policy $\bar{\pi}'$.

At line 6, transition t is compared to transition t' to see if they match, i.e. that the source state $\bar{s}$, target state $\bar{s}'$ and option $\bar{a}$ are equal, although the exact transition probabilities are ignored. If the transitions do not match then no Q-values can be reused. Alternatively, if the transitions are the same, then the Q-values for the low-level states which map to $\bar{s}$, the source state of both t and t', are intuitively still useful action values for the states in that transition of the new policy. Shown in lines 8 and 9, the Q-values for every RL MDP state s that is in $\bar{s}$ and for all actions a in the MDP action set A are taken from the optimal Q-table Q^*. These are used to update Q that was initialised in line 2.

Once all Q-tables have been iterated over, and all reusable Q-values found, the initialised Q-table is returned in line 15.

The Q-values that are reused for this Q-table are not necessarily optimal with respect to a fully optimised Q-table within the new constraints. However, when followed by the agent, they increase the likelihood that the agent will choose a useful action and spend less time exploring actions which are not useful.

5.3 Evaluation

We evaluated ARL-KR using our two case studies from Chapter 4.4: the guarded flag collection and assisted-living system environments. For brevity, the details of these problem environments will not be repeated here. The areas we evaluated were: (i) if Algorithm 4 can successfully be used to automatically arrive at a safe RL solution from an initially incorrect AMDP; and (ii) if the Q-value initialisation approach from Algorithm 5 can improve the speed at which an agent will learn a safe solution in subsequent learning runs.

For each case study we constructed an AMDP which contained some *incorrect* information relative to its RL MDP counterpart. The incorrect information included up to four randomly selected state transition probabilities which were adjusted using a Gaussian distribution curve. We also varied the rewards contained in each environment. However, given the bespoke nature of rewards in each RL problem, these were adjusted for each specific environment. The details of these reward adjustments are given in each case study.

For the purposes of our second evaluation criterion, i.e. the effect of the Q-value initialisation algorithm, the RL stage of the update loop was run twice. One run had standard RL occur, starting with an arbitrary Q-table, and the other utilised the initialised Q-table generated from Algorithm 5. The properties of each learning run were compared once the experiment finished.

All RL experiments started with parameters that are the same as those that were specified in Chapter 4.4; however, there was additional experimentation to see how Q-value initialisation can be optimised with different exploration and learning rates. The details of which are specified where appropriate.

ARL-KR was evaluated using this approach 30 times to ensure statistical significance of the results [102], where each experiment was configured with different knowledge. Since each experiment typically generated a unique abstract policy after knowledge revision had occurred, the learning curves, i.e. the expected rewards attained by the agent, are not necessarily the same between experiments: one abstract policy in one experiment may yield a greater or smaller reward than another abstract policy for another experiment. Therefore, so that the results can be collated, after each experiment the learning progress was normalised between the maximum and minimum expected rewards across all experiments.

5.3.1 Guarded Flag Collection

For this evaluation we have further modified the environment from that introduced in Chapter 4 so that all adjacent areas have a doorway connecting them, as well as allowing them all to potentially be guarded by cameras. This is to allow a greater ranger of changes to be made in the environment. The transition probabilities between these areas, i.e. the probability that entering into a new area of the environment would result in being detected, was varied in each experiment. Additionally, for each experiment, the value of up to two, randomly selected rewards (i.e. flags) were adjusted using a Gaussian distribution.

The modifications to the guarded flag collection AMDP, first constructed in

Chapter 4, required to accommodate changes in knowledge is shown below in PRISM Language 4.

The original doorway capture probabilities from the unadjusted AMDP appear on lines 7 to 10. To accommodate all new possible transitions for the new doorways, the probabilities on lines 11 and 12 (and more not shown) are introduced. Since these represent the probability of capture, and hence the probability of passing a doorway is 1−capture probability, setting their initial values to 1 means the probability of transitioning to the next area is actually 0, i.e. these doorways are initially assumed to not allow moving to that area. Of course, this does mean that attempting the transition will result in capture with a probability of 1, but any policy attempting this transition will be rejected during the safety verification stage of ARL and so will not interfere when generating safe abstract policies and creating safe action constraints from them.

Lines 15 to 20 are new to the PRISM model. These represent the values that each of the flags are worth. Originally, as shown in PRISM Language 2, all flags were assigned a constant value of 1 in the reward structure. Now, shown starting from line 37, the transition reward for finishing uses the reward variables for the reward values.

Contrasting with the original model, the areas which formerly only had one doorway (i.e. RoomA, RoomB and RoomE) can now all have two or more. This means that they can no longer use simple commands such as the example shown in Chapter 4.4.1. Now, they must be restructured in the same style as all the other commands shown in PRISM Language 2. Examples of these modified commands are shown starting on lines 25 and 29, where the guards for being in RoomA now include flag combinations and also the new action parameter `t` (for the specific flag combination shown: `t1`). When the parameter value is set to `1` then the transition will lead to HallA with probability `1-p1` or be captured with probability `p1`. When the value is `2` then the transition will lead to RoomB with probability `1-p5`, collecting FlagB in the process, or instead be captured with probability `p5`.

The use of variables for reward values and transition probabilities (even if they are ultimately not used or changed) makes the process of updating the AMDP with new information a simple task.

PRISM Language 4: ARL-KR guarded flag collection AMDP extracts

```
dtmc

const int t1;
const int t2;
...

const double p1 = 0.06; // RoomA <-> HallA
const double p2 = 0.05; // HallB <-> RoomB
const double p3 = 0.05; // RoomC <-> HallB
const double p4 = 0.07; // RoomC <-> RoomE
const double p5 = 1;    // RoomA <-> RoomB
const double p6 = 1;    // HallA <-> HallB
...

const double rewardA = 1;
const double rewardB = 1;
const double rewardC = 1;
const double rewardD = 1;
const double rewardE = 1;
const double rewardF = 1;

module guarded_flag_collection_modified

        ...
        [] !captured & position=1 & t1=1 & !flagA & !flagB & !flagC
                & !flagD & !flagE & !flagF -> (1-p1):(position'=0)
                + p1:(captured'=true);
        ...
        [] !captured & position=1 & t1=2 & !flagA & !flagB & !flagC
                & !flagD & !flagE & !flagF ->
                (1-p5):(position'=2)&(flagB'=true) + p5:(captured'=true);
        ...

endmodule

rewards "all_flags"
        [end] true : (flagA ? rewardA : 0) + (flagB ? rewardB : 0)
                + (flagC ? rewardC : 0) + (flagD ? rewardD : 0)
                + (flagE ? rewardE : 0) + (flagF ? rewardF : 0)
                + (position=6 ? 1 : 0);
endrewards
```

Using this modified AMDP, 30 experiments were conducted in which ARL-KR was able to update the AMDP and identify a new safe abstract policy to successfully produce a safe RL policy for 26 experiments. For the 4 experiments where ARL-KR was unable to produce a safe RL policy, after extensive policy generation and verification of the AMDP using standard ARL no safe policy could be found. This means that either no safe policy existed at all, or that the number of safe policies in the entire policy space was very small and one could not be found within the allowed search time. This is a known limitation of ARL that is discussed in Chapter 4.6. Of the 26 where ARL-KR was successful, 16 required just one update iteration, 7 required two iterations and 3 required three iterations.

A point to note is that those experiments requiring multiple update iterations generated increasingly useful Q-tables with each iteration, since there was a greater number of previously optimised Q-tables from which to extract Q-values. Therefore, the agent started with increasingly better behaviour after each iteration.

When comparing Q-value initialisation to RL with an arbitrary Q-table, we found that the best results were achieved by slightly reducing the exploration factor and learning rate compared to those experiments with an arbitrary Q-table, from 0.6 to 0.5 and 0.1 to 0.07, respectively. When using the same parameters from the standard RL experiments using arbitrary Q-values, the benefits of the initialised Q-table were reduced since the agent had a tendency to disregard this knowledge in favour of more exploration. Combined with a higher learning rate, this increased the likelihood of initialised Q-values being updated with values that did not accurately reflect the benefit of the actions in the long term. For example, when an initialised Q-value had the agent go through a doorway which risked being captured, if the agent *were* captured, receiving a reward of -1, then the Q-value could be updated to the extent that it no longer were the optimal action (at least temporarily) despite being optimal in the long run.

Across all experiments where ARL-KR succeeded in finding a safe RL solution, the average time required to generate an initialised Q-table was 6.9 seconds. The average time taken to complete a learning run with Q-value reuse was 180 seconds and the time without was 204 seconds, saving 24 seconds, i.e. an 11.8% overall speed increase. This is since in the early episodes the agent is able to reach the goal area in fewer time steps due to its actions being less arbitrary. Figure 5.2 shows the learning progress of 30 separate experiments.

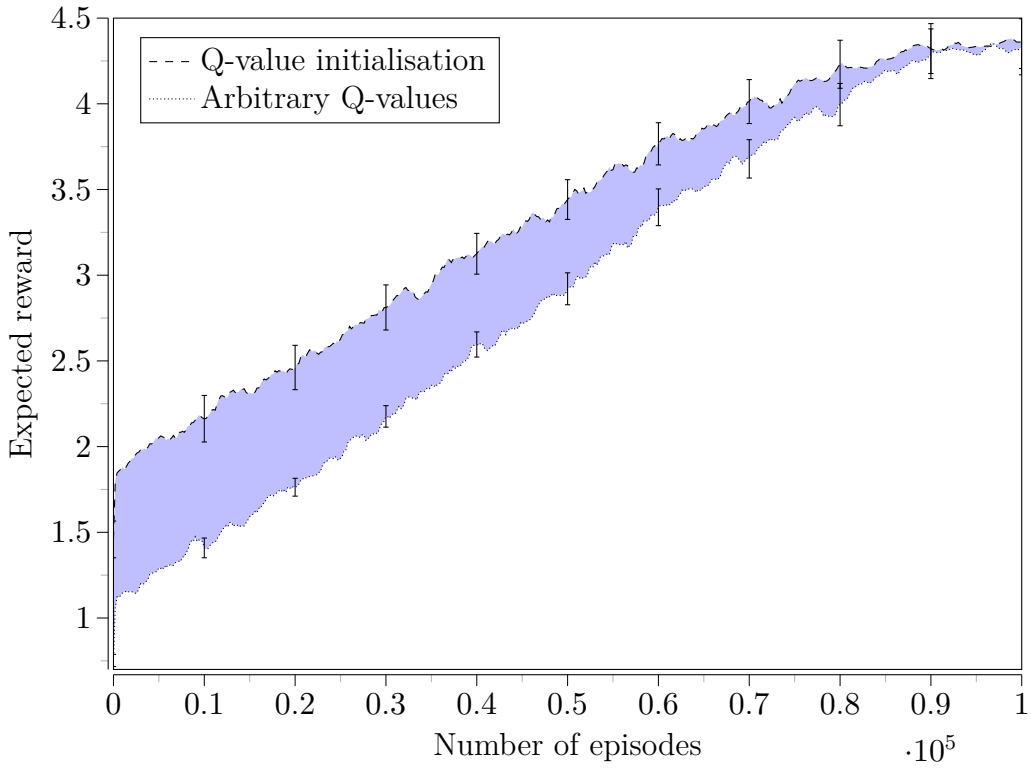


Figure 5.2: Learning progress for the guarded flag collection when using incremental ARL with Q-value initialisation and standard ARL with an initially arbitrary Q-table.

The blue-shaded area between the two plots shows the efficiency increase for learning that Q-value initialisation produces. In this case study, the initial expected reward cumulated by the agent when learning with initialised Q-values is approximately 50% greater that learned from arbitrary Q-values. Throughout the learning run, initialised Q-values result in a higher expected reward compared to arbitrary Q-values at the same number of episodes.

This increase diminishes towards the end of the learning run since the benefit of Q-value initialisation is most prominent at the start, where the agent would otherwise behave entirely randomly. As the agent converges on an optimal solution, increasingly fewer actions need optimising and the benefits of the initialised Q-values diminishes. However, the initial head start is highly advantageous in time-critical systems where it is important to have a solution as quickly as possible. If it is not essential that the solution is completely optimal, or if there is not enough time to learn completely optimal solution, Q-value initialisation allows a solution to be learned that is more optimal than what would be learned otherwise in the time allowed.

5.3.2 Assisted-Living System

For this environment, the potential transition probabilities to vary were those between progress levels, both for when a prompt was given, or not, to reach them. In this particular environment there are three rewards, however, two of them are arbitrary in their magnitude: a large punishment is needed for calling the caregiver (which the agent will otherwise learn to do immediately) and a large positive reward for finishing the task is necessary simply to motivate the agent to provide any prompts (otherwise it will do nothing at all). The magnitudes of these rewards do not need to be specific and simply need to be large. Therefore, the adjustment of these rewards has little bearing on the quality of a solution with respect to how many prompts are given and, as a result, randomly adjusting them is of no benefit to this case study. The remaining reward, -1 for each prompt given, was adjusted so that its magnitude varied at each level of progress (e.g. if the user particularly struggles with one stage of the task, so a prompt is less unwelcome, or if they are quite adept at another stage, so would be irritated by a prompt).

For this experiment, only minor modifications of the original PRISM AMDP from PRISM Language 3 were needed to allow knowledge revision to occur. Since all the possible state transitions already exist as variables, the model structure needs no alteration. The only necessary change is to the reward structure: instead of always returning a fixed value of 1, variables for each prompt reward were created and used in the reward structure. These changes are shown in PRISM Language 5. As with the modified AMDP for the guarded flag collection, the reward variables start with a value of one, since initially we can only assume these values are correct until new information is acquired. If these rewards are determined to be inaccurate they can be easily adjusted when updating the AMDP.

In this case study, ARL-KR was able to produce a safe RL policy in all 30 of the experiments. This is unsurprising when considering the number of potential safe policies that were found for this particular environment in the case study from Chapter 4.4 (Figure 4.7). In this particular case study, though, none of the experiments required more than one update iteration, where all necessary updates were achieved in a single iteration. This can be explained since the constraints on the agent do not prevent it from exploring any of the stages in the environment. This is in contrast to the guarded flag collection case study, where the constraints on the agent prevented it from exploring certain areas and therefore it did not

encounter certain rewards and transitions in the initial run, only discovering them (and therefore updating the AMDP, if necessary) in additional runs.

PRISM Language 5: ARL-KR assisted-living system AMDP extracts

```
dtmc

...

const double prompt0Reward = 1;
const double prompt1Reward = 1;
const double prompt2Reward = 1;
const double prompt3Reward = 1;
const double prompt4Reward = 1;
const double prompt5Reward = 1;
const double prompt6Reward = 1;
const double prompt7Reward = 1;
const double prompt8Reward = 1;
const double prompt9Reward = 1;
const double prompt11Reward = 1;
const double prompt12Reward = 1;

...

rewards "distress"
        s=0  : m >= prompt0 ? prompt0Reward : 0;
        s=1  : m >= prompt1 ? prompt1Reward : 0;
        s=2  : m >= prompt2 ? prompt2Reward : 0;
        s=3  : m >= prompt3 ? prompt3Reward : 0;
        s=4  : m >= prompt4 ? prompt4Reward : 0;
        s=5  : m >= prompt5 ? prompt5Reward : 0;
        s=6  : m >= prompt6 ? prompt6Reward : 0;
        s=7  : m >= prompt7 ? prompt7Reward : 0;
        s=8  : m >= prompt8 ? prompt8Reward : 0;
        s=9  : m >= prompt9 ? prompt9Reward : 0;
        s=11 : m >= prompt11 ? prompt11Reward : 0;
        s=12 : m >= prompt12 ? prompt12Reward : 0;
endrewards
```

Once again, we found that reducing the exploration and learning rates from 0.5 and 0.1, which we used in the standard RL with an arbitrary Q-table, to 0.3 and 0.05, respectively, improved the effect of the Q-value initialisation.

For the 30 experiments it took an average of 441 milliseconds to initialise the Q-values. The average length of a run with Q-value initialisation was 141 seconds and the average without was 156 seconds. This is a 15 second decrease in time, meaning a 9.6% increase in speed to reach an optimal solution. The results of learning with and without Q-value initialisation are shown in Figure 5.3.

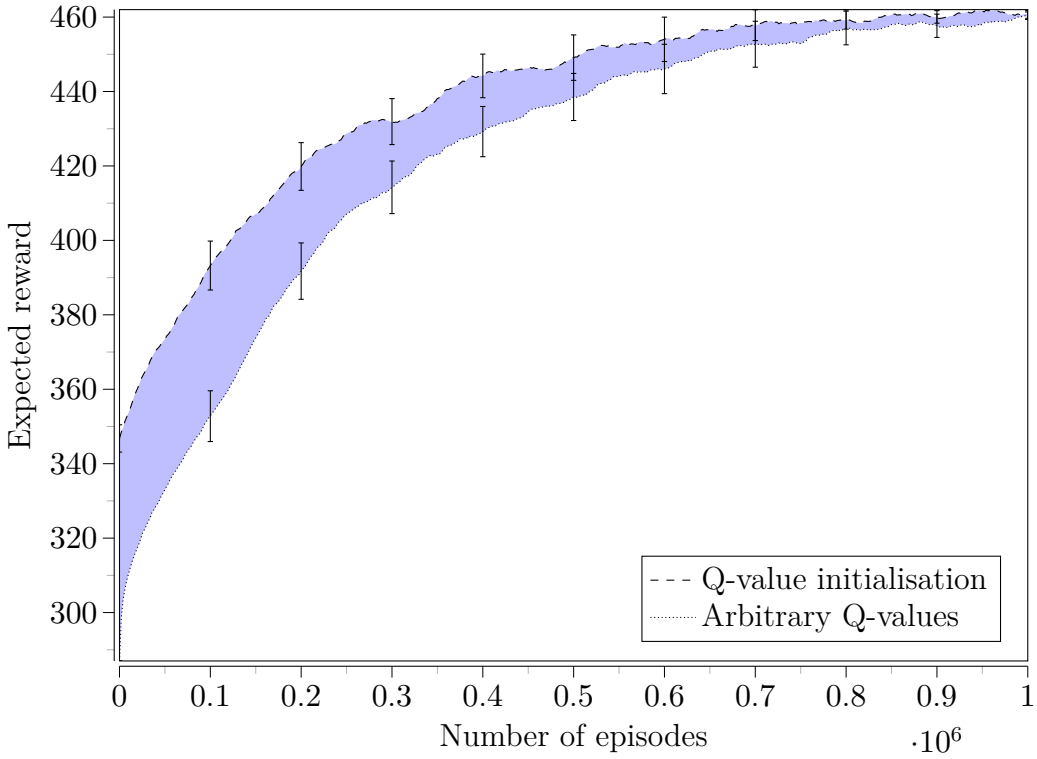


Figure 5.3: Learning progress for the assisted-living system when using incremental ARL with Q-value initialisation and standard ARL with an initially arbitrary Q-table.

As seen in the blue-shaded area, initialising Q-values improves the quality of the agent behaviour from the start and at all stages of the learning run, most notably in earlier episodes. Although the improvement diminishes towards the end of the learning run, the initial gains are significant for a time-critical system.

5.4 Summary

ARL with knowledge revision (ARL-KR) extends ARL by enabling ARL to automatically update its AMDP should it contain inaccuracies, either included at design time or having developed over time as the system changes from its initial state, which can cause an ARL solution to fail to satisfy its requirements. Furthermore, ARL-KR utilises a novel algorithm for Q-value initialisation to extract reusable actions from previously optimised Q-tables to improve the speed of subsequent learning runs through incremental ARL. ARL-KR decreases the need for manual intervention when model inaccuracies are present and also reduces the time required to recover from them, increasing the applicability of ARL in real-time systems.

In two case studies we show how:

1. The ARL-KR approach can be used successfully to automatically recover from an inaccurate AMDP which can cause an RL solution to violate its intended safety requirements;

2. Q-value initialisation, by extracting previously optimised Q-values, can speed the learning process, which for time-critical systems where a solution is needed as quickly as possible, even if not entirely optimal, is a significant advantage.

However, as experienced in the guarded flag collection case study, ARL cannot always identify a safe solution if the existence of one is rare or absent.

For best results when using ARL-KR we found that decreasing the exploration and learning rates can motivate the agent to rely more on the actions provided to it in the partial Q-table. However, the magnitude of which should be determined empirically for each problem domain, since some domains may allow significantly more or less Q-values to be reused, i.e. if safe abstract policies differ a little or a lot. In the former case, smaller values for the exploration and learning rates may prove beneficial, and the opposite in the latter case.

The ARL-KR algorithm has the limitation that updates can only occur when they are related to states which are known, or assumed, to exist and therefore feature in the AMDP. To update an AMDP to include new states currently must be done manually, since it can mean changes to the model as a whole and at present no automatic techniques to do this exist. Therefore, updates are limited to changes in rewards and transition probabilities of entering states that

are at least assumed to exist. A limitation of the Q-value reuse algorithm is that it is only applicable to learning techniques where Q-values have a tabular representation (i.e. a Q-table); it is not applicable with function approximation learning algorithms.

Chapter 6

Conclusion

This thesis has addressed the significant limitation of RL that the AI technique cannot provide assurances that a solution it produces can be trusted as safe. Specifically, the behaviour learned by a traditional RL agent can lead to violations of strict safety, regulatory, legal or ethical requirements. This limitation has prevented RL from being utilised in the domain of safety- and mission-critical applications where it is essential that agent behaviour will not risk harm to humans or damage to itself and other systems.

Whilst several techniques have been proposed to mitigate this limitation (as discussed in Chapter 3), these techniques are mostly unable to provide guarantees that strict safety requirements will be satisfied and instead produce a solution which is 'generally safer'. Those techniques which are able to assure safety typically involve significant and unnecessary reductions to the optimality of the solution.

The ARL technique developed by this project overcomes these obstacles by using formal verification techniques to provide RL solutions which will satisfy a broad range of strict safety requirements, as well as achieve desired levels of solution optimality in the same manner. Furthermore, the extension ARL-KR allows new and up-to-date information to be incorporated so that RL safety can be assured if initial information used to generate the safety constraints proves to be inaccurate, incomplete or has changed since it was first observed. Finally, the ARL technique is highly generalisable; its use is not limited to any specific problem domains since its only requirement is partial knowledge of the environment so that a high-level model can be constructed.

The following sections outline the contributions by this project, the limitations of the ARL technique and areas of future work to be done.

6.1 Contributions

The main contribution of this thesis is the ARL safe reinforcement learning approach, detailed in Chapter 4. ARL is a technique for providing assurance that an RL solution will satisfy strict safety requirements.

To this end, ARL builds an abstract representation of the RL environment in the form of an AMDP. This requires only partial knowledge of the problem environment. Specifically, this knowledge is the transition probabilities and rewards that influence the safety and optimality of a solution. This model is then formally analysed using QV to identify a Pareto-optimal set of abstract policies which satisfy a set of safety and optimisation requirements. These requirements are specified using PCTL temporal logic. A user manually selects an abstract policy from this set which suitably compromises between optimisation and safety levels. Safe abstract policies are used to constrain the RL optimisation process so that the low-level policies learned by the agent will not involve visiting states identified as unsafe at a high-level. This is achieved by disallowing actions by the RL agent which will result in transitioning to MDP states that do not map to high-level safe states from the abstract policy. These constraints are enforced during learning and become part of the final policy learned by the agent. When evaluated, the learned policy matches the levels of safety that were verified for the abstract policy.

The use of QV gives ARL the significant advantage over most other safe RL techniques that safety is *formally* assured and therefore results are guaranteed to be correct. The use of temporal logic to express safety and optimisation requirements is a further advantage of ARL over other techniques. This feature allows ARL to support a broad range of complex and specific requirements, in contrast to other safe RL techniques where safety is loosely defined in terms of rewards, or where MDP states must be explicitly labelled as 'safe' or 'unsafe'.

The second contribution of this thesis is ARL-KR (Chapter 5), an extension of the core ARL technique to allow knowledge revision for the AMDP if the RL solution has unintended safety levels, demonstrating that the AMDP was constructed using inaccurate or incomplete information. ARL-KR allows the AMDP to be updated with up-to-date knowledge and subsequently a safe RL policy can be identified. This important extension allows ARL to be used for problems where it may not be possible to identify all pertinent information of the problem environment prior to learning, or for problems whose environment is subject to change from what it was initially. In this way, ARL-KR allows

the potential for ARL to be used in systems where it may become necessary to generate new constraints as the system evolves.

Third, a new algorithm for Q-value initialisation is introduced in Chapter 5 to allow Q-value reuse. Should it be necessary that ARL-KR must be employed to update the AMDP, this requires that at least one subsequent learning run be undertaken to optimise an RL solution within the updated safety constraints. In some situations, the previously tried abstract policy may overlap with the new safe abstract policy, i.e. there are common transitions allowed by each. When this is the case, it is inefficient that the RL agent must completely relearn the actions which correlate with these abstract transitions, if it has previously learned them. Therefore, the Q-value reuse algorithm iterates over all previously attempted safe abstract policies and their respective optimised Q-tables to identify any Q-values for state-actions which are safe according to the newest abstract policy. The Q-values that are identified to still be usable can be copied into an unoptimised Q-table to be used to initiate the RL agent. This reuse of previously optimised Q-values increases the probability that the agent will choose the optimal safe actions during the subsequent learning process, thereby increasing the speed which it can arrive at an optimal solution.

Fourth, two new case studies are introduced, each from a different class of problems commonly used to evaluate ARL. These case studies are designed to expose the agent to risk so that they can be used when evaluating safe RL techniques. The first case study is a navigation task, based on the RL benchmark flag collection environment [3], where an agent must navigate a two-dimensional environment to collect flags contained in certain areas. We have modified the environment so that entering and exiting certain areas involves a risk of being detected by an adversary, which results in failure. The goal of the agent is to collect as many flags as possible whilst minimising the risk of being detected, goals which oppose one another. The second case study is a planning problem, based on an assisted-living system [54], where an RL agent must adapt to the preferences of a person suffering from dementia so that it gives helpful prompts to guide the person through the everyday task of washing their hands. The system's goal is to minimise the need for a human caregiver to assist, in order to alleviate their duties, but also minimise the number of prompts it provides to guide the user, which could become stressful to them.

Finally, we demonstrate through extensive evaluation using the two case studies the effectiveness of ARL and ARL-KR. We show that ARL is able to generate and successfully enforce safety constraints so that a safe RL policy is learned that

will match the safety levels verified for the constraints. For ARL-KR we demonstrate how an AMDP can be updated to accurately reflect the RL environment to produce a safe RL policy and that the use of Q-value initialisation to reuse previously optimised actions can speed up the learning process by the RL agent.

6.2 Limitations

For ARL to be used successfully, some knowledge of the problem must be known in advance. If no knowledge of the problem is known at all then it is not possible for ARL to assure any safety requirements. Furthermore, some effort is required to construct an AMDP from this partial knowledge. Whilst the use of ARL-KR can ameliorate the task of ascertaining the necessary knowledge of the problem, and the introduction of an automatic AMDP generator (discussed under future work) could minimise the effort required to construct the AMDP, it is unavoidable that some knowledge and effort will be required for ARL to be applied successfully.

Depending on how the AMDP is constructed, ARL can be limited by the number of possible abstract policies since the size of the abstract policy space increases with the size of the AMDP. For example, an abstract policy for our guarded flag collection AMDP contains a parameter for each state of the AMDP and the number of values each parameter can have is equal to the number of possible transitions out of that state. Therefore, the number of possible policies (i.e. for all possible combinations of parameter values) increases exponentially as the number of states and transitions increases. This becomes a problem if only a small proportion of abstract policies are safe, which can mean that the search process spends significant time verifying policies which are unsafe. If the number of possible policies is very large and the number of safe policies is very small, this can result in the search process reaching a maximum search time without finding any safe policies. In this situation it is not possible to know whether there are no safe policies at all, or if one could be found after extensive searching. Furthermore, even though a safe policy may be found, it may be that there are other optimal policies which have significantly better levels of optimality than what was found within the allowed search time. This problem can be mitigated by using metaheuristics such as genetic algorithms like in EvoChecker [99]; they have a proven track record of supporting effective search within extremely large search spaces.

The knowledge that can be revised during ARL-KR is limited to transition

probabilities and rewards for states that are known, or at least assumed, to exist. It is not currently possible for ARL-KR to update the AMDP with new states that may be discovered after the AMDP has been constructed. Since ARL does not automatically build the AMDP, the inclusion of new states requires manual intervention. Introducing new states to the model can require significant restructuring of the AMDP and as a direct consequence the format of the abstract policies may change. Therefore, this is not something that can be automatically achieved by ARL-KR at present. As discussed below as a direction of future work, this problem could be resolved by the use of an automatic AMDP generator.

6.3 Future Work

There are several areas where research into ARL can be extended and continued.

6.3.1 Extending the use of ARL to Runtime

ARL and ARL-KR are used under the assumption that the problem environment and requirements do not change after a safe RL solution has been produced. However, in many real-world problems these assumptions are not the case. Therefore, in the preliminary research carried out by the author of this thesis, in [5] an architecture is proposed (Figure 6.1) to enable ARL to be used at runtime to accommodate environment or requirement changes.

The architecture comprises two main functions: an intelligent autonomous function, utilising an RL solution, and a (potentially suboptimal) automatic function as its backup. Each time an autonomous system based on this architecture takes an input from the environment, requiring an action output, the current safety constraints are verified against the requirements and AMDP. This reverification is necessary since the safety or optimisation requirements of the system may have been redefined, or the AMDP may have been updated. Should verification determine that the safety constraints hold safe then the autonomous function can be allowed to provide an action to perform. Alternatively, if the constraints are verified as unsafe, the autonomous function is substituted with the automatic function which provides an action that is known to be safe, albeit one that is suboptimal.

Whilst the automatic function is being utilised, ARL with Q-value initialisation can run in the background to learn a new RL policy for the autonomous function. When a safe RL policy has been learned it can be deployed and the autonomous function can be reinstated.

An assumption is that the automatic function is capable of producing an

action in every state which is known to be safe. For example, in the assisted-living system case study, a safe behaviour in any state would be to call the caregiver. In the guarded flag collection case study, a safe behaviour may simply to wait until the autonomous function is usable again, or, if in HallA, HallB or RoomD, the agent can safely proceed to the exit.

The architecture is based on the monitor-analyse-plan-execute control loop from [107] which is designed to allow computer systems to automatically manage themselves. For ARL, this can be applied in the following manner:

1. **Monitor** – In this stage, information about the environment is continuously received and the AMDP is updated when necessary. Examples of such changes include transition probabilities (e.g. due to changes in the environment, impacting the performance of the system) or rewards involved with certain actions (e.g. if energy reserves are low and certain behaviour should be limited, incurring a lesser reward than usual, or maybe a punishment). In addition, safety requirements are monitored in case changes to them affect the safety levels of the system.

2. **Analyse** – Using the potentially updated AMDP and PCTL requirements from the monitoring stage, and the currently employed safety constraints, the properties of the AMDP are verified using QV to determine if the safety requirements are still satisfied. If safety violations are detected then the autonomous function (which operates under the now unsafe safety constraints) is disabled and control of the system temporarily transferred to the automatic function.

3. **Plan** – When violations of the safety requirements are detected then it is necessary to identify new safety constraints. This is done in the planning stage, where new safe abstract policies are synthesised. Depending on the urgency of utilising up-to-date safety constraints, either a Pareto front of safe policies can be generated over some period of time, allowing a domain expert to choose the policy most appropriate, or in a time-critical situation the first identified safe policy can be automatically chosen.

4. **Execute** – Once a safe abstract policy has been selected, the system can then learn a new RL policy for the constraints. After learning has completed, the automatic function can be disabled and the now-safe RL policy can be supplied to the autonomous function so its use can be resumed.

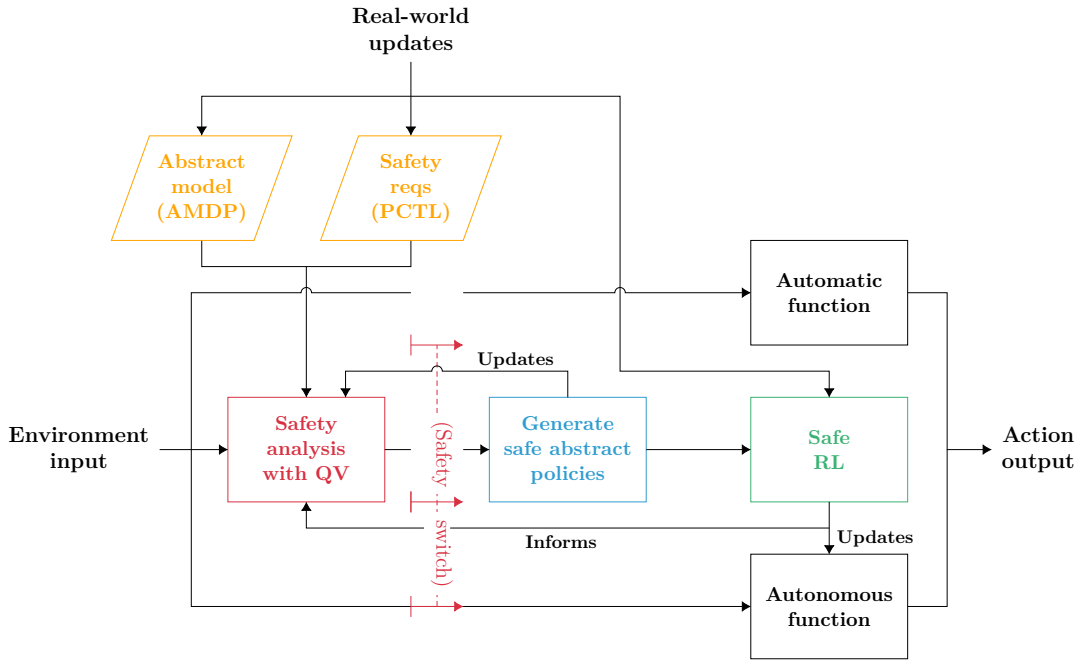


Figure 6.1: Architecture for an autonomous system using ARL at runtime.

The autonomous function and QV (red box) are initialised with safety constraints which have been identified before the system is operational. Once operational, real-world information continuously updates the AMDP, the PCTL requirements (amber boxes) and the safe RL environment (green box)—this forms the *monitor* stage of the architecture.

The *analyse* stage utilises a safety switch to alternate control between the autonomous and automatic functions. The operation of the switch is based on the result of verification. If QV verifies that the autonomous function is now unsafe then it triggers the switch over to the automatic function, simultaneously invoking the process of searching for new safety constraints for ARL (blue box).

Once a safe policy has been selected it is relayed back to the safety analyser so that it knows what actions the autonomous function is capable of. Additionally, it is also *executed* to constrain the safe RL process. When the new safe policy has been fully optimised it is used to update the autonomous function. Furthermore, the safety analyser is informed that the autonomous function can now be reinstated, discontinuing the automatic function.

This work on a preliminary runtime architecture for ARL, carried out to initiate the adoption of the project results in the autonomous systems domain, needs further work. First, a means of creating an automatic function is required, possibly based on an ergodic solution as discussed in Chapter 3.2.1. Second, a

technique is required for switching agent behaviours whilst it is operating in the environment. Lastly, an evaluation of the architecture in a case study to see if the agent can maintain safety when its environment and/or requirements regularly change.

6.3.2 Other Future Work Directions

Automating AMDP generation. This functionality would be particularly useful to incorporate into ARL since a significant step of ARL is to construct an AMDP from limited details of the problem. Currently, this requires a domain expert who is skilled with the PRISM modelling language (or an equivalent language which supports quantitative verification). Depending on how much detail of the problem is required to ensure that safety constraints are effective, and how the problem is structured, the resulting AMDP can become cumbersome (in terms of lines of code) and/or complex if commands use multiple variables and produce multiple updates. Furthermore, depending on how abstract policy parameters are defined, the number of commands in the model can significantly increase the space of possible abstract policies. Therefore, it is beneficial to design the AMDP with the goal to keep it as small as possible to minimise the number of possible policies.

To automatically generate this AMDP would significantly simplify this step, removing the need for an expert to construct the AMDP as well as to ensure the model is designed efficiently to keep the abstract policy space as small as possible. Additionally, this would increase the abilities of ARL-KR, since it is currently limited to updating transitions and rewards for states that are assumed to exist. ARL-KR cannot currently include new states into the model since this could require significant restructuring of the AMDP.

Existing research in this direction appears in [108, 109, 110, 111] which introduce techniques for automatically abstracting actions and states from an RL environment. These techniques could form the basis of an AMDP generator, although, additional work is necessary to express the abstractions in a PRISM compatible format.

Requirement synthesis from plain English descriptions. In addition to building the AMDP, the domain expert is also needed to translate the safety requirements to PCTL formulae. This necessary process potentially risks the expert misinterpreting requirements from the user and is also a bottleneck in

the usability of ARL. Even though the domain expert may not be required to modify the AMDP once it has been constructed, they are still needed afterwards should the user wish to experiment with different safety requirements beyond those defined at the initial use of the ARL system.

A significant step towards automatic formulae synthesis that can be adapted for this ARL extension appears in [112], which introduces the tool ProProST to synthesise commonly occurring PCTL formulae from plain-English descriptions. Research is ongoing towards an all-encompassing catalogue of property patterns [113] which potentially could be used to automate ARL property specification.

Abstract policy reuse. Depending on the proportion of safe abstract policies in the entire policy space, searching for them can be a time consuming process. This is most significant in the ARL-KR algorithm, which may require multiple searches for safe abstract policies.

If an AMDP update is only minor, it would be beneficial to make accordingly minor adjustments to an existing abstract policy to accommodate the change, instead of restarting the potentially expensive process of searching for new safe abstract policies from scratch.

A possible solution could involve identifying which parameters of a previously generated abstract policy are affected by updates and which are not. Then, in a similar fashion to the Q-value reuse algorithm, it could be possible to fix those parameters which are unaffected and to focus the search process on only those parameters which will now cause safety violations.

Large scale evaluation. An evaluation of ARL in larger environments with more complex safety requirements would be beneficial to assess the extent to which ARL can feasibly be scaled. Although ARL is not limited to any specific problem domains, if a problem has a large state dimensionality, even if the AMDP is significantly smaller than the RL MDP, it may still be large enough that it becomes impractical to verify in a reasonable space of time.

Recent benchmark experiments devised for deep learning, such those discussed in [114], could be useful for establishing the capabilities of ARL.

Appendix A

PRISM AMDP for the Guarded Flag Collection

```
1   dtmc
2
3   //// Indexes of action to take:
4   // w = action in HallA (1, 2 or 3): 1 = go to RoomA and get FlagA;
        2 = go to RoomD and get FlagD; 3 = go to HallB.
5   // x = action in RoomC (1 or 2): 1 = go to HallB; 2 = go to RoomE
        and get Flags E and F.
6   // y = action in RoomD (1 or 2): 1 = go to Goal; 2 = go to HallA.
7   // z = action in HallB (1, 2 or 3): 1 = go to HallA; 2 = go to
        RoomB and get FlagB; 3 = go to RoomC and get FlagC.
8   const int w1;
9   const int w2;
    ...
71  const int w64;
72  const int x1;
    ...
135 const int x64;
136 const int y1;
    ...
199 const int y64;
200 const int z1;
    ...
263 const int z64;
264
265 //// Camera probabilities:
266 const double p1 = 0.06; // HallA <-> RoomA
267 const double p2 = 0.05; // HallB <-> RoomB
```

```
268 const double p3 = 0.05; // RoomB <-> HallC
269 const double p4 = 0.07; // RoomC <-> RoomE
270
271 module guarded_flag_collection
272     position: [0..8] init 0; // 0 = HallA (Start); 1 = RoomA;
          2 = RoomB; 3 = RoomC; 4 = RoomD; 5 = RoomE; 6 = Goal;
          7 = HallB; 8 = Absorbing state.
273     // Flag variables (false = not collected; true = collected)
274     flagA: bool init false;
275     flagB: bool init false;
        ...
279     flagF: bool init false;
280     captured: bool init false; // agent captured
281
282     // Transitions from HallA
283     [] !captured & position=0 & w1=1 & !flagA & !flagB & !flagC
          & !flagD & !flagE & !flagF -> (1-p1):(position'=1)
          & (flagA'=true) + p1:(captured'=true);
284     [] !captured & position=0 & w2=1 & !flagA & !flagB & !flagC
          & !flagD & !flagE & flagF -> (1-p1):(position'=1)
          & (flagA'=true) + p1:(captured'=true);
285     [] !captured & position=0 & w3=1 & !flagA & !flagB & !flagC
          & !flagD & flagE & !flagF -> (1-p1):(position'=1)
          & (flagA'=true) + p1:(captured'=true);
        ...
346     [] !captured & position=0 & w64=1 & flagA & flagB & flagC
          & flagD & flagE & flagF -> (1-p1):(position'=1)
          & (flagA'=true) + p1:(captured'=true);
347     [] !captured & position=0 & w1=2 & !flagA & !flagB & !flagC
          & !flagD & !flagE & !flagF -> (1-p5):(position'=4)
          & (flagD'=true) + p5:(captured'=true);
        ...
411     [] !captured & position=0 & w1=3 & !flagA & !flagB & !flagC
          & !flagD & !flagE & !flagF -> (1-p6):(position'=7)
          + p6:(captured'=true);
        ...
475
476     // Transitions from RoomA
477     [] !captured & position=1 -> (1-p1):(position'=0)
```

```
            + p1:(captured'=true);
478
479     // Transitions from RoomB
480     [] !captured & position=2 -> (1-p2):(position'=7)
            + p2:(captured'=true);
481
482     // Transitions from RoomC
483     [] !captured & position=3 & x1=1 & !flagA & !flagB & !flagC
            & !flagD & !flagE & !flagF -> (1-p3):(position'=7)
            + p3:(captured'=true);
        ...
547     [] !captured & position=3 & x1=2 & !flagA & !flagB & !flagC
            & !flagD & !flagE & !flagF -> (1-p4):(position'=5)
            & (flagE'=true) & (flagF'=true) + p4:(captured'=true);
        ...
611
612     // Transitions from RoomD
613     [] !captured & position=4 & y1=1 & !flagA & !flagB & !flagC
            & !flagD & !flagE & !flagF -> (position'=6);
        ...
677     [] !captured & position=4 & y1=2 & !flagA & !flagB & !flagC
            & !flagD & !flagE & !flagF -> (1-p5):(position'=0)
            + p5:(captured'=true);
        ...
742
743     // Transitions from RoomE
744     [] !captured & position=5 -> (1-p4):(position'=3)
            & (flagC'=true) + p4:(captured'=true);
745
746     // Transitions from HallB
747     [] !captured & position=7 & z1=1 & !flagA & !flagB & !flagC
            & !flagD & !flagE & !flagF -> (1-p6):(position'=0) +
            p6:(captured'=true);
        ...
811     [] !captured & position=7 & z1=2 & !flagA & !flagB & !flagC
            & !flagD & !flagE & !flagF -> (1-p2):(position'=2)
            & (flagB'=true) + p2:(captured'=true);
        ...
875     [] !captured & position=7 & z1=3 & !flagA & !flagB & !flagC
```

```
            & !flagD & !flagE & !flagF -> (1-p3):(position'=3)
            & (flagC'=true) + p3:(captured'=true);
        ...
939
940     // Final states
941     [end] captured | position=6 -> (position'=8);
942     [] position=8 -> (position'=8);
943 endmodule
944
945 rewards "all_flags"
946     [end] true : (flagA ? 1 : 0) + (flagB ? 1 : 0)
            + (flagC ? 1 : 0)+ (flagD ? 1 : 0)
            + (flagE ? 1 : 0) + (flagF ? 1 : 0)
            + (position=6 ? 1 : 0);
947 endrewards
```

Appendix B

PRISM AMDP for the Assisted-Living System

```
1   dtmc
2
3   //Give prompt in states s=0, s=1... of the dementia workflow
        after prompt0, prompt1... number of mistakes. Note that
        state=10 of the workflow is the end and does not require
        prompts.
4   const int prompt0;
5   const int prompt1;
6   const int prompt2;
7   const int prompt3;
8   const int prompt4;
9   const int prompt5;
10  const int prompt6;
11  const int prompt7;
12  const int prompt8;
13  const int prompt9;
14  const int prompt11;
15  const int prompt12;
16
17  // Maximum number of mistakes in total allowed before the
        agent calls the carer.
18  const int MAX_MISTAKES;
19
20  // Transition probabilities without prompt.
21  // Key: pAB, where A = source state and B = destination state.
22  const double p00=0.36;
23  const double p01=0.36;
```

```
24  const double p02=0.28;
25  const double p10=0.24;
26  const double p11=0.16;
27  const double p13=0.48;
28  const double p14=0.12;
29  const double p22=0.432;
30  const double p24=0.568;
    ...
64
65  // Transition probabilities with prompt.
66  const double pp00=0.0864;
67  const double pp01=0.5407;
68  const double pp02=0.3729;
    ...
108
109 module patient_workflow
110     s: [0..12] init 0; // Stages of task workflow
111     m: [0..MAX_MISTAKES] init 0; // Cumulative number of mistakes
112
113     // Transitions without prompt
114     [] s=0 & m<prompt0 & m<MAX_MISTAKES -> p00:(s'=0)
            & (m'=m+1) + p01:(s'=1) + p02:(s'=2);
115     [] s=1 & m<prompt1 & m<MAX_MISTAKES -> p10:(s'=0)
            & (m'=m+1) + p11:(s'=1) & (m'=m+1) + p13:(s'=3)
            + p14:(s'=4);
116     [] s=2 & m<prompt2 & m<MAX_MISTAKES -> p22:(s'=2)
            & (m'=m+1) + p24:(s'=4);
117     [] s=3 & m<prompt3 & m<MAX_MISTAKES -> p31:(s'=1)
            & (m'=m+1) + p33:(s'=3) & (m'=m+1) + p35:(s'=5)
            + p3_11:(s'=11) & (m'=m+1);
118     [] s=4 & m<prompt4 & m<MAX_MISTAKES -> p42:(s'=2)
            & (m'=m+1) + p44:(s'=4) & (m'=m+1) + p45:(s'=5);
119     [] s=5 & m<prompt5 & m<MAX_MISTAKES -> p55:(s'=5)
            & (m'=m+1) + p56:(s'=6) + p5_12:(s'=12) & (m'=m+1);
120     [] s=6 & m<prompt6 & m<MAX_MISTAKES -> p65:(s'=5)
            & (m'=m+1) + p66:(s'=6) & (m'=m+1) + p67:(s'=7)
            + p68:(s'=8);
121     [] s=7 & m<prompt7 & m<MAX_MISTAKES -> p76:(s'=6)
            & (m'=m+1) + p77:(s'=7) & (m'=m+1) + p79:(s'=9)
```

```
            + p7_12:(s'=12)&(m'=m+1);
122     [] s=8 & m<prompt8 & m<MAX_MISTAKES -> p84:(s'=4)
            & (m'=m+1) + p86:(s'=6) & (m'=m+1) + p88:(s'=8)
            & (m'=m+1) + p89:(s'=9);
123     [] s=9 & m<prompt9 & m<MAX_MISTAKES -> p92:(s'=2)
            & (m'=m+1) + p98:(s'=8) & (m'=m+1) + p99:(s'=9)
            & (m'=m+1) + p9_10:(s'=10);
124     [] s=11 & m<prompt11 & m<MAX_MISTAKES -> p11_0:(s'=0)
            & (m'=m+1) + p11_3:(s'=3) + p11_11:(s'=11)
            & (m'=m+1) + p11_12:(s'=12);
125     [] s=12 & m<prompt12 & m<MAX_MISTAKES -> p12_2:(s'=2)
            & (m'=m+1) + p12_5:(s'=5) + p12_12:(s'=12) & (m'=m+1);
126
127     // Transitions with prompt
128     [] s=0 & m>=prompt0 & m<MAX_MISTAKES -> pp00:(s'=0)
            & (m'=m+1) + pp01:(s'=1) + pp02:(s'=2);
129     [] s=1 & m>=prompt1 & m<MAX_MISTAKES -> pp10:(s'=0)
            & (m'=m+1) + pp11:(s'=1) & (m'=m+1) + pp13:(s'=3)
            + pp14:(s'=4);
130     [] s=2 & m>=prompt2 & m<MAX_MISTAKES -> pp22:(s'=2)
            & (m'=m+1) + pp24:(s'=4);
131     [] s=3 & m>=prompt3 & m<MAX_MISTAKES -> pp31:(s'=1)
            & (m'=m+1) + pp33:(s'=3) & (m'=m+1) + pp35:(s'=5)
            + pp3_11:(s'=11) & (m'=m+1);
132     [] s=4 & m>=prompt4 & m<MAX_MISTAKES -> pp42:(s'=2)
            & (m'=m+1) + pp44:(s'=4) & (m'=m+1) + pp45:(s'=5);
133     [] s=5 & m>=prompt5 & m<MAX_MISTAKES -> pp55:(s'=5)
            & (m'=m+1) + pp56:(s'=6) + pp5_12:(s'=12) & (m'=m+1);
134     [] s=6 & m>=prompt6 & m<MAX_MISTAKES -> pp65:(s'=5)
            & (m'=m+1) + pp66:(s'=6) & (m'=m+1) + pp67:(s'=7)
            + pp68:(s'=8);
135     [] s=7 & m>=prompt7 & m<MAX_MISTAKES -> pp76:(s'=6)
            & (m'=m+1) + pp77:(s'=7) & (m'=m+1) + pp79:(s'=9)
            + pp7_12:(s'=12) & (m'=m+1);
136     [] s=8 & m>=prompt8 & m<MAX_MISTAKES -> pp84:(s'=4)
            & (m'=m+1) + pp86:(s'=6) & (m'=m+1) + pp88:(s'=8)
            & (m'=m+1) + pp89:(s'=9);
137     [] s=9 & m>=prompt9 & m<MAX_MISTAKES -> pp92:(s'=2)
            & (m'=m+1) + pp98:(s'=8) & (m'=m+1) + pp99:(s'=9)
```

```
            & (m'=m+1) + pp9_10:(s'=10);
138     [] s=11 & m>=prompt11 & m<MAX_MISTAKES -> pp11_0:(s'=0)
            & (m'=m+1) + pp11_3:(s'=3) + pp11_11:(s'=11)
            & (m'=m+1) + pp11_12:(s'=12);
139     [] s=12 & m>=prompt12 & m<MAX_MISTAKES -> pp12_2:(s'=2)
            & (m'=m+1) + pp12_5:(s'=5) + pp12_12:(s'=12) & (m'=m+1);
140
141     // Final states: either with reaching state 10
            (carer not called) or with MAX_MISTAKES (carer called)
142     [] s=10 | m=MAX_MISTAKES -> (s'=10);
143 endmodule
144
145 // When in state i, the patient's level of distress increases
        by 1 if number of mistakes m > prompt_i
146 rewards "distress"
147     s=0 : m>=prompt0 ? 1 : 0;
148     s=1 : m>=prompt1 ? 1 : 0;
149     s=2 : m>=prompt2 ? 1 : 0;
150     s=3 : m>=prompt3 ? 1 : 0;
151     s=4 : m>=prompt4 ? 1 : 0;
152     s=5 : m>=prompt5 ? 1 : 0;
153     s=6 : m>=prompt6 ? 1 : 0;
154     s=7 : m>=prompt7 ? 1 : 0;
155     s=8 : m>=prompt8 ? 1 : 0;
156     s=9 : m>=prompt9 ? 1 : 0;
157     s=11 : m>=prompt11 ? 1 : 0;
158     s=12 : m>=prompt12 ? 1 : 0;
159 endrewards
```

References

[1] M. Kwiatkowska, G. Norman, and D. Parker, 'Probabilistic model checking: Part 4 – Markov decision processes', http://www.prismmodelchecker.org/lectures/biss07/04-mdps.pdf, March 2007, accessed: 19th December 2017.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts, USA: MIT Press, 1998.

[3] R. Dearden, N. Friedman, and S. Russell, 'Bayesian Q-learning', in *Proceedings of the Fifteenth National Conference on Artificial Intelligence*. AAAI Press, July 1998, pp. 761–768.

[4] G. Mason, R. Calinescu, D. Kudenko, and A. Banks, 'Assurance in reinforcement learning using quantitative verification', in *Advances in Hybridization of Intelligent Methods: Models, Systems and Applications*, ser. Smart Innovation, Systems and Technologies, I. Hatzilygeroudis and V. Palade, Eds. Springer International Publishing AG, 2018, vol. 85, pp. 71–96.

[5] G. R. Mason, 'An autonomous system safety-monitor architecture using reinforcement learning and runtime quantitative verification', Defence Science and Technology Laboratory (Dstl), Platform Systems, Porton Down, Salisbury, Wiltshire, SP4 0JQ, Tech. Rep. DSTL/TR106266, December 2017, version 1.0.

[6] G. Mason, R. Calinescu, D. Kudenko, and A. Banks, 'Assured reinforcement learning for safety-critical applications', in *Doctoral Consortium on Agents and Artificial Intelligence (DCAART 2017)*. Porto, Portugal: SciTePress, February 2017, pp. 9–16, best PhD project award.

[7] G. Mason, R. Calinescu, D. Kudenko, and A. Banks, 'Assured reinforcement learning with formally verified abstract policies', in *Proceedings of the 9th International Conference on Agents and Artificial Intelligence (ICAART 2017)*, vol. 2. Porto, Portugal: SciTePress, February 2017, pp. 105–117, best student paper award.

[8] G. Mason, R. Calinescu, D. Kudenko, and A. Banks, 'Combining reinforcement learning and quantitative verification for agent policy assurance', in *Proceedings of the 6th International Workshop on Combinations of Intelligent Methods and Applications (CIMA 2016)*, I. Hatzilygeroudis and V. Palade, Eds., The Hague, Holland, August 2016, pp. 45–52.

[9] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, New Jersey, USA: Prentice Hall, 2009.

[10] N. J. Nilsson, *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., 1980.

[11] J. Levinson, J. Askeland, J. Becker *et al.*, 'Towards fully autonomous driving: Systems and algorithms', in *Intelligent Vehicles Symposium (IV)*. IEEE, June 2011, pp. 163–168.

[12] D. Nguyen-Tuong and J. Peters, 'Model learning for robot control: A survey', *Cognitive Processing*, vol. 12, no. 4, pp. 319–340, November 2011.

[13] S. D. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. H. Eng, D. Rus, and M. H. Ang, 'Perception, planning, control, and coordination for autonomous vehicles', *Machines*, vol. 5, no. 1, 2017.

[14] A. Vieira, 'Business applications of deep learning', in *Ubiquitous Machine Learning and Its Applications*, ser. Advances in Computational Intelligence and Robotics, P. Kumar and A. Tiwari, Eds. IGI Global, March 2017, pp. 39–67.

[15] Y. Li, W. Jiang, L. Yang, and W. Tian, 'On neural networks and learning systems for business computing', *Neurocomputing*, vol. 275, pp. 1150–1159, January 2018.

[16] S. E. Dilsizian and E. L. Siegel, 'Artificial intelligence in medicine and cardiac imaging: Harnessing big data and advanced computing to provide personalized medical diagnosis and treatment', *Current Cardiology Reports*, vol. 16, no. 1, pp. 1–8, January 2014, article 441.

[17] D. D. Luxton, Ed., *Artificial Intelligence in Behavioral and Mental Health Care*. Academic Press, 2016.

[18] H. K. Palo, M. Narayana, and M. Chandra, 'Design of neural network model for emotional speech recognition', in *Proceedings of ICAEES 2014 Artificial Intelligence and Evolutionary Algorithms in Engineering Systems*, ser. Advances in Intelligent Systems and Computing. New Delhi, India: Springer, 2014, pp. 291–300.

[19] D. Amodei, S. Ananthanarayanan, R. Anubhai *et al.*, 'Deep speech 2 : End-to-end speech recognition in English and Mandarin', in *Proceedings of the 33rd International Conference on Machine Learning*, vol. 48. New York, NY, USA: JMLR, June 2016, pp. 173–182.

[20] I. Millington, *Artificial Intelligence For Games*, 2nd ed. Burlington, Massachusetts, USA: Morgan Kaufmann Publishers, 2009.

[21] D. Silver, A. Huang, C. J. Maddison *et al.*, 'Mastering the game of Go with deep neural networks and tree search', *Nature*, vol. 529, pp. 484–489, 2016.

[22] J. Fox and S. Das, *Safe and Sound: Artificial Intelligence in Hazardous Applications*. AAAI Press/MIT Press, 2000.

[23] M. O. Riedl and B. Harrison, 'Using stories to teach human values to artificial agents', in *Proceedings of the 2nd International Workshop on AI, Ethics and Society*. AAAI Press, 2015, pp. 105–112.

[24] L. P. Kaelbling, M. L. Littman, and A. W. Moore, 'Reinforcement learning: A survey', *Journal of Artificial Intelligence Research*, vol. 4, no. 1, pp. 237–285, 1996.

[25] M. Wiering and M. van Otterlo, Eds., *Reinforcement Learning: State-of-the-Art*, ser. Adaption, Learning, and Optimization. Berlin/Heidelberg, Germany: Springer-Verlag, 2012, vol. 12.

[26] S. B. Thrun, 'Efficient exploration in reinforcement learning', Carnegie-Mellon University, Tech. Rep. CMU-CS-92-102, January 1992.

[27] D. E. Moriarty and R. Mikkulainen, 'Efficient reinforcement learning through symbiotic evolution', *Machine Learning*, vol. 22, no. 1–3, pp. 11–32, 1996.

[28] M. Kearns and D. Koller, 'Efficient reinforcement learning in factored MDPs', in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, vol. 2. Morgan Kaufmann Publishers, Inc., 1999, pp. 740–747.

[29] K. O. Stanley and R. Miikkulainen, 'Efficient reinforcement learning through evolving neural network topologies', in *Proceedings of the Genetic and Evolutionary Computation Conference*. San Francisco, California, USA: Morgan Kaufmann Publishers, Inc., 2002, pp. 569–577.

[30] A. A. Sherstov and P. Stone, 'Function approximation via tile coding: Automating parameter choice', in *Proceedings of the 6th International Symposium on Abstraction, Reformulation and Approximation*. Springer-Verlag, 2005, pp. 194–205.

[31] G. Dulac-Arnold, L. Denoyer, P. Preux, and P. Gallinari, 'Fast reinforcement learning with large action sets using error-correcting output codes for MDP factorization', in *Proceedings of the 2012 European Conference on Machine Learning and Knowledge Discovery in Databases*, vol. 2. Springer, 2012, pp. 180–194.

[32] V. Mnih, K. Kavukcuoglu, D. Silver *et al.*, 'Human-level control through deep reinforcement learning', *Nature*, vol. 518, no. 7540, pp. 529–533, February 2015.

[33] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, 'Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation', in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 3675–3683.

[34] I. Arel, C. Liu, T. Urbanik, and A. G. Kohls, 'Reinforcement learning-based multi-agent system for network traffic signal control', *IET Intelligent Transport Systems*, vol. 4, no. 2, pp. 128–135, June 2010.

[35] W. M. Hinojosa, S. Nefti, and U. Kaymak, 'Systems control with generalized probabilistic fuzzy-reinforcement learning', *IEEE Transactions on Fuzzy Systems*, vol. 19, no. 1, pp. 51–64, February 2011.

[36] S. Lange, M. Riedmiller, and A. Voigtländer, 'Autonomous reinforcement learning on raw visual input data in a real world application', in *Ninth*

International Joint Conference on Computer Science and Software Engineering. IEEE, 2012, pp. 1–8.

[37] J. Baxter, A. Tridgell, and L. Weaver, 'KnightCap: A chess program that learns by combining TD(λ) with minimax search', in *Proceedings of the 15th International Conference on Machine Learning*. Morgan Kaufmann, 1998, pp. 28–36.

[38] B. Tastan and G. Sukthankar, 'Learning policies for first person shooter games using inverse reinforcement learning', in *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2011, pp. 85–90.

[39] C. Kwok and D. Fox, 'Reinforcement learning for sensing strategies', in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2004, pp. 3158–3163.

[40] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, 'Reinforcement learning for robot soccer', *Autonomous Robots*, vol. 27, no. 1, pp. 55–73, July 2009.

[41] T. Hester, M. Quinlan, and P. Stone, 'Generalized model learning for reinforcement learning on a humanoid robot', in *IEEE International Conference on Robotics and Automation*. IEEE, 2010, pp. 2369–2374.

[42] J. Kober, E. Oztop, and J. Peters, 'Reinforcement learning to adjust robot movements to new situations', in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*. AAAI Press, 2011, pp. 2650–2655.

[43] N. Abe, P. Melville, C. Pendus *et al.*, 'Optimizing debt collections using constrained reinforcement learning', in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Washington, District of Columbia, USA: ACM, 2010, pp. 75–84.

[44] O. Teboul, I. Kokkinos, L. Simon, P. Koutsourakis, and N. Paragios, 'Shape grammar parsing via reinforcement learning', in *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2011, pp. 2273–2280.

[45] M. Peters, W. Ketter, M. Saar-Tsechansky, and J. Collins, 'A reinforcement learning approach to autonomous decision-making in smart electricity markets', *Machine Learning*, vol. 92, no. 1, July 2013.

[46] S. Hore, L. Tyrvainen, J. Pyykko, and D. Glowacka, 'A reinforcement learning approach to query-less image retrieval', in *Proceedings of the Third International Workshop on Symbiotic Interaction*, ser. Lecture Notes in Computer Science, vol. 8820. Springer, Cham, 2014, pp. 121–126.

[47] H. Cuayáhuitl, 'SimpleDS: A simple deep reinforcement learning dialogue system', in *Dialogues with Social Robots: Enablements, Analyses, and Evaluation*, ser. Lecture Notes in Electrical Engineering, K. J. Wilcock, Ed. Singapore: Springer, 2017, vol. 427, pp. 109–118.

[48] 'Nuclear power plants – Instrumentation and control important for safety – Software aspects for computer-based systems performing category B or C functions', International Electrotechnical Commission, Geneva, Switzerland, Standard IEC 62138, 2004.

[49] 'Medical device software – Software life cycle processes', International Electrotechnical Commission, Geneva, Switzerland, Standard IEC 62304, 2006.

[50] 'Road vehicles – Functional safety', International Organization for Standardization, Geneva, Switzerland, Standard ISO 26262, 2011.

[51] 'Software considerations in airborne systems and equipment certification', Radio Technical Commission for Aeronautics, Washington, District of Columbia, USA, Standard DO-178C, 2012.

[52] J. García and F. Fernández, 'A comprehensive survey on safe reinforcement learning', *Journal of Machine Learning Research*, vol. 16, no. 1, pp. 1437–1480, August 2015.

[53] M. Kwiatkowska, 'Quantitative verification: Models, techniques and tools', in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2007, pp. 449–458.

[54] J. Boger, J. Hoey, P. Poupart *et al.*, 'A planning system based on Markov decision processes to guide people with dementia through activities of daily living', *IEEE Transactions on Information Technology in Biomedicine*, vol. 10, no. 2, pp. 323–333, April 2006.

[55] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. New York, NY, USA: John Wiley & Sons, Inc., 1994.

[56] R. Bellman, *Dynamic Programming*. Princeton, New Jersey, USA: Princeton University Press, 1957.

[57] R. A. Howard, *Dynamic Programming and Markov Processes*. Cambridge, Massachusetts, USA: MIT Press, 1960.

[58] C. J. Watkins and P. Dayan, 'Q-learning', *Machine Learning*, vol. 8, no. 3–4, pp. 279–292, May 1992.

[59] G. Rummery and M. Niranjan, 'On-line Q-learning using connectionist systems', Cambridge University, England, Tech. Rep. CUED/F-INFENG-TR 166, September 1994.

[60] R. Calinescu, S. Kikuchi, and K. Johnson, 'Compositional reverification of probabilistic safety properties for large-scale complex IT systems', in *Large-Scale Complex IT Systems: Development, Operation and Management*, ser. Lecture Notes in Computer Science, R. Calinescu and D. Garlan, Eds., vol. 7539. Springer, 2012, pp. 303–329.

[61] R. Calinescu, K. Johnson, and Y. Rafiq, 'Developing self-verifying service-based systems', in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 734–737.

[62] S. Gerasimou, R. Calinescu, and A. Banks, 'Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration', in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2014, pp. 115–124.

[63] H. Hansson and B. Jonsson, 'A logic for reasoning about time and reliability', *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, September 1994.

[64] E. M. Clarke and E. A. Emerson, 'Design and synthesis of synchronization skeletons using branching time temporal logic', in *Logics of Programs: Workshop*, ser. Lecture Notes in Computer Science, D. Kozen, Ed., vol. 131. Berlin/Heidelberg, Germany: Springer-Verlag, May 1981, pp. 52–71.

[65] S. Andova, H. Hermanns, and J.-P. Katoen, 'Discrete-time rewards model-checked', in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, K. G. Larsen and P. Niebert, Eds., vol. 2791. Berlin, Heidelberg: Springer, 2004, pp. 88–104.

[66] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen, 'The ins and outs of the probabilistic model checker MRMC', *Performance Evaluation*, vol. 68(2), pp. 90–104, 2011.

[67] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton, 'Verifying continuous time Markov chains', in *Proceedings of the 8th International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science, R. Alur and T. A. Henzinger, Eds., vol. 1102. Springer, 1996, pp. 269–276.

[68] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton, 'Model-checking continuous-time Markov chains', *ACM Transactions on Computational Logic*, vol. 1, no. 1, pp. 162–170, July 2000.

[69] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, 'Model-checking algorithms for continuous-time Markov chains', *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 524–541, June 2003.

[70] M. Kwiatkowska, G. Norman, and D. Parker, 'PRISM 4.0: Verification of probabilistic real-time systems', in *Proceedings of the 23rd International Conference on Computer Aided Verification*, vol. 6806. Springer, 2011, pp. 585–591.

[71] A. Pnueli, 'The temporal logic of programs', in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. IEEE, 1977, pp. 46–57.

[72] H. Younes, 'Ymer: A statistical model checker', in *Proceedings of the 17th International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 3576. Springer, 2005, pp. 429–433.

[73] K. Sen, M. Viswanathan, and G. Agha, 'VeStA: a statistical model-checker and analyzer for probabilistic systems', in *Second International Conference on the Quantitative Evaluation of Systems*. IEEE, 2005, pp. 251–252.

[74] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk, 'A storm is coming: A modern probabilistic model checker', in *Proceedings of the 29th International Conference on Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds., 2017, pp. 592–600.

[75] D. N. Jansen, J.-P. Katoen, M. Oldenkamp, M. Stoelinga, and I. Zapreev, 'How fast and fat is your probabilistic model checker? An experimental

comparison', *Hardware and Software: Verification and Testing*, vol. 4899, pp. 69–85, 2008.

[76] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, 'Model checking and the state explosion problem', in *Tools for Practical Software Verification*, ser. Lecture Notes in Computer Science, B. Meyer and M. Nordio, Eds. Berlin/Heidelberg, Germany: Springer-Verlag, 2012, vol. 7682, pp. 1–30.

[77] B. Marthi, 'Automatic shaping and decomposition of reward functions', in *Proceedings of the 24th International Conference on Machine learning*, June 2007, pp. 601–608.

[78] K. Efthymiadis, S. Devlin, and D. Kudenko, 'Abstract MDP reward shaping for multi-agent reinforcement learning', in *11th European Workshop on Multi-Agent Systems*, 2013.

[79] L. Li, T. J. Walsh, and M. L. Littman, 'Towards a unified theory of state abstraction for MDPs', in *9th International Symposium on Artificial Intelligence and Mathematics*, January 2006, pp. 531–539.

[80] R. S. Sutton, D. Precup, and S. Singh, 'Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning', *Artificial Intelligence*, vol. 112, no. 1–2, pp. 181–211, August 1999.

[81] A. Y. Ng, D. Harada, and S. J. Russell, 'Policy invariance under reward transformations: Theory and application to reward shaping', in *Proceedings of the Sixteenth International Conference on Machine Learning*, I. Bratko and S. Dzeroski, Eds. San Francisco, California, USA: Morgan Kaufmann Publishers, Inc., 1999, pp. 278–287.

[82] M. Pecka and T. Svoboda, 'Safe exploration techniques for reinforcement learning – an overview', in *Modelling and Simulation for Autonomous Systems*, ser. Lecture Notes in Computer Science, J. Hodicky, Ed., vol. 8906. Cham, Switzerland: Springer International Publishing, 2014, pp. 357–375.

[83] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen, *Systems and Software Verification.* Berlin/Heidelberg Germany: Springer-Verlag, 2001.

[84] 'Information technology – Security techniques – Information security risk management', International Organization for Standardization and the In-

ternational Electrotechnical Commission, New York, Standard ISO/IEC 27005, 2011.

[85] M. J. Matarić, 'Interaction and intelligent behaviour', PhD thesis, Massachusetts Institute of Technology, May 1994.

[86] T. M. Moldovan and P. Abbeel, 'Safe exploration in Markov decision processes', in *Proceedings of the 29th International Conference on Machine Learning*, 2012, pp. 1711–1718.

[87] S. Junges, N. Jansen, C. Dehnert *et al.*, 'Safety-constrained reinforcement learning for MDPs', in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, M. Chechik and J.-F. Raskin, Eds., vol. 9636, 2016, pp. 130–146.

[88] M. Heger, 'Consideration of risk in reinforcement learning', in *Proceedings of the 11th International Conference on Machine Learning*, 1994, pp. 105–111.

[89] S. Koenig and R. G. Simmons, 'The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms', *Machine Learning*, vol. 22, no. 1–3, pp. 227–250, March 1996.

[90] C. Gaskett, 'Reinforcement learning under circumstances beyond its control', in *Proceedings of the International Conference on Computational Intelligence for Modelling Control and Automation*, 2003.

[91] R. A. Howard and J. E. Matheson, 'Risk-sensitive Markov decision processes', *Management Science*, vol. 18, no. 7, pp. 356–369, March 1972.

[92] O. Mihatsch and R. Neuneier, 'Risk-sensitive reinforcement learning', *Machine Learning*, vol. 49, no. 2, pp. 267–290, November 2002.

[93] J. A. Clouse and P. E. Utgoff, 'A teaching method for reinforcement learning', in *Proceedings of the Ninth International Workshop on Machine Learning*, D. Sleeman and P. Edwards, Eds. San Francisco, California, USA: Morgan Kaufmann Publishers, Inc., 1992, pp. 92–110.

[94] J. García and F. Fernández, 'Safe exploration of state and action spaces in reinforcement learning', *Journal of Artificial Intelligence Research*, vol. 45, no. 1, pp. 515–564, September 2012.

[95] M. Pecka, K. Zimmermann, and T. Svoboda, 'Safe exploration for reinforcement learning in real unstructured environments', in *Proceedings of the 20th Computer Vision Winter Workshop*, P. Wohlhart and V. Lepetit, Eds., 2015, pp. 85–93.

[96] M. Pecka, V. Šalanský, K. Zimmermann, and T. Svoboda, 'Autonomous flipper control with safety constraints', in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2016, pp. 2889–2894.

[97] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, 'An application of reinforcement learning to aerobatic helicopter flight', in *Proceedings of the 19th International Conference on Neural Information Processing Systems*. MIT Press, 2006, pp. 1–8.

[98] T. Mannucci, E.-J. van Kampen, C. de Visser, and Q. Chu, 'Safe exploration algorithms for reinforcement learning controllers', *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1–13, 2017.

[99] S. Gerasimou, G. Tamburrelli, and R. Calinescu, 'Search-based synthesis of probabilistic models for quality-of-service software engineering', in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 319–330.

[100] C. Liu, X. Xu, and D. Hu, 'Multiobjective reinforcement learning: A comprehensive overview', *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 45, no. 3, pp. 385–398, March 2015.

[101] P. Scopes, V. Agarwal, S. Devlin *et al.*, *YOrk Reinforcement Learning Library (YORLL)*, Reinforcement Learning Group, Department of Computer Science, University of York, 2012.

[102] A. Arcuri and L. Briand, 'A practical guide for using statistical tests to assess randomized algorithms in software engineering', in *Proceedings of the 33rd International Conference on Software Engineering*, May 2011, pp. 1–10.

[103] J. P. W. Bynum, P. V. Rabins, W. Weller *et al.*, 'The relationship between a dementia diagnosis, chronic illness, medicare expenditures, and hospital use', *Journal of the American Geriatrics Society*, vol. 52, no. 2, pp. 187–194, February 2004.

[104] R. Calinescu, K. Johnson, and Y. Rafiq, 'Using observation ageing to improve Markovian model learning in QoS engineering', in *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*. ACM, 2011, pp. 505–510.

[105] R. Calinescu, Y. Rafiq, K. Johnson, and M. E. Bakir, 'Adaptive model learning for continual verification of non-functional properties', in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ACM, 2014, pp. 87–98.

[106] K. Efthymiadis and D. Kudenko, 'Knowledge revision for reinforcement learning with abstract MDPs', in *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems*, 2015, pp. 763–770.

[107] J. O. Kephart and D. M. Chess, 'The vision of autonomic computing', *Computer*, vol. 36, no. 1, pp. 41–50, January 2003.

[108] S. Mannor, I. Menache, A. Hoze, and U. Klein, 'Dynamic abstraction in reinforcement learning via clustering', in *Proceedings of the Twenty-First International Conference on Machine learning*. ACM Press, July 2004, pp. 560–567.

[109] G. Kheradmandian and M. Rahmati, 'Automatic abstraction in reinforcement learning using data mining techniques', *Robotics and Autonomous Systems*, vol. 57, no. 11, pp. 1119–1128, November 2009.

[110] M. Ghafoorian, N. Taghizadeh, and H. Beigy, 'Automatic abstraction in reinforcement learning using ant system algorithm', in *Lifelong Machine Learning Papers from the 2013 AAAI Spring Symposium*, 2013, pp. 9–14.

[111] N. Taghizadeh and H. Beigy, 'A novel graphical approach to automatic abstraction in reinforcement learning', *Robotics and Autonomous Systems*, vol. 61, no. 8, pp. 821–835, August 2013.

[112] L. Grunske, 'Specification patterns for probabilistic quality properties', in *Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 31–40.

[113] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, 'Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar', *IEEE Transactions on Software Engineering*, vol. 41, pp. 620–638, July 2015.

[114] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, 'Benchmarking deep reinforcement learning for continuous control', in *Proceedings of the 33rd International Conference on Machine Learning*, vol. 48, 2016, pp. 1329–1338.