EXECUTIVE SUMMARY

Before robots can be marketed for personal use, they must undergo rigorous certification to ensure they satisfy certain safety and liveness properties. These safety properties dictate things the robot must not do. In contrast, liveness properties specify things the robot must accomplish [***1***]. However, adaptive algorithms such as reinforcement learning (RL) techniques pose a problem to this certification. How can safety or liveness be demonstrated by a controller that adapts its behaviour over time?

This project was to solve this problem by incorporating safety and liveness properties into a RL algorithm. This algorithm had two tasks. First, at the end of training it must produce an action policy that maximises the agent's return on a numerical reward signal such that the probability of violating a safety or liveness property is less than some specified threshold. Secondly, the agent must choose actions during training such that the expected probability of violating a property is also below the threshold. However, this expected risk of chosen actions should be according to the agent's current knowledge. Thus, the agent's rate of violating properties should converge to the allowable thresholds as its knowledge of the environment improves.

Two algorithms are proposed to meet these requirements. Safe and Live Reinforcement Learning (SLRL) is a modification to Multiple Criteria-Adaptive Real-time Dynamic Programming (MC-ARTDP) [***2***] that includes a method of defining constraints from safety and liveness properties as well as an exploration policy which attempts to maintain them throughout training. Safe and Live G-Learning (SLGL) is an extension to SLRL which uses G-Learning [***3***] to approximate the value and risk tables with regression trees.

These two algorithms are then empirically shown to maintain safety and liveness properties during and after training by comparing them with Q-Learning, MC-ARTDP, and G-Learning on a toy grid world problem. They are then applied to an experiment using a robot that serves hot and cold drinks based on a previous study in [***4***] which examined safety and liveness using a non-adaptive controller. Similar safety and liveness properties are specified and a high level controller is developed based on the Yet Another Robotics Platform (YARP) robotics middleware library. This controller links the RL algorithms to existing low level control and sensor systems for the BERT2 robot at the Bristol Robotics Laboratory [***5***]. In this scenario, SLRL and SLGL are also shown to meet and maintain numerous specified properties while training on a simulator and then running with live subjects. In addition, the HRI scenario demonstrates SLGL's ability to improve training by using its regression trees to generalise from experience.

This project's primary accomplishments include:

- Devised a method of translating safety and liveness properties into constraints for a multiple constraint Markov Decision Problem based on a techniques from model checking [***6***] and Risk Sensitive Reinforcement Learning [***7***], see pp. 24-25.
- Developed SLRL, a modification of MC-ARTDP [***2***] which includes an action selection policy that attempts to maintain safety and liveness properties throughout exploration, see pp. 24-32.
- Applied G-Learning's regression tree function approximation techniques [***3***] to SLRL, see page 33.
- Built a high level control architecture based on YARP [***8***] to allow RL algorithms to control the BERT2 robot with existing low level control and sensor management software [***5***], see pp. 40-51.
- Conducted a Wizard-of-Oz study similar to [***9***] to train user and error models for an RL simulator, see pp. 39, 56-57.
- Trained Q-Learning, G-Learning, SLRL, and SLGL on the simulator and used their resulting policies to conduct HRI experiments on the BERT2 robot, see pp. 36-38, 58-63.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the guidance of my supervisor, Kerstin Eder. Her thoughtful suggestions continually steered me in the right direction. I would especially like to thank Corina Grigore, Tom Newton-Dines and Khurram Ghani, who were all willing and enthusiastic test subjects. Corina's initial investigation into safety and liveness with a hot drinks robot was also an indispensable starting point for this project.

I would also like to thank Tony Pipe for coordinating everything at the Bristol Robotics Lab. He was always there to make sure I had everything I needed for my research.

Alex Lenz and Sergey Skacheck's help learning the ins and outs of BERT2's hardware and control systems was invaluable to getting this project off the ground. I could never have worked on a human-robot interaction experiment in so little time without the myriad of easy to use sensory and control modules they developed for the CHRIS project.

I am also deeply grateful to my wife Jillian for encouraging me to develop a robot that serves coffee if only as an excuse to further our addiction. And that brings me to Frank, the finest coffee roaster on either side of the Atlantic. May he continue to fuel the imaginations of Bristol academics for years to come.

1 INTRODUCTION

Widespread human assistance robots have long been a dream of popular culture. However, there have been many technical issues with bringing that dream to fruition. It has only been over the last decade or so that the field of robotics has started to overcome those technical issues with the application of new algorithms and more capable computing infrastructures. However, one difficult open question is how to allow a robot to adapt to changing situations while still ensuring the safety of those interacting with it. Without the ability to prove that an adaptive robotic control algorithm adheres to specified safety constraints, industry will not adopt it no matter how useful it is. However, it is not enough to simply demonstrate a robot's safety. After all, the safest robot is one that is not turned on. The liveness of a robot must also be demonstrated. In other words, the safety of the robot should not prevent it from accomplishing its task.

This project attempts to answer this question by investigating a method of incorporating reinforcement learning techniques into a real human-robot interaction (HRI) system in such a way as to demonstrate that the system adheres to certain safety and liveness constraints. This is done through experiments with a stationary, anthropomorphic robot tasked with serving hot drinks.

A similar experiment was conducted in partnership with the Bristol Robotics Lab (BRL) in the summer of 2010 [***4***]. That project successfully implemented an initial controller which integrated information from motion capture cameras, head-gaze tracking sensors, and hand pressure sensors to hand a cup to a customer without dropping it. This HRI scenario was a complex problem in which a stationary robot must serve either a cup of coffee or a cup of water to a customer. Both drinks had a different level of risk associated with them. As such, this problem required the agent to satisfy many safety and liveness properties simultaneously. It served to stress the algorithms in a real-world scenario. However, the initial controller built in 2010 did not include any adaptation or optimisation.

The goal of this project is to develop and test a novel reinforcement learning (RL) algorithm that can learn an action policy which satisfies specified safety and liveness properties. It should also attempt to maintain those properties while learning. By studying such an algorithm in the context of the hot drinks robot, synergy between the machine learning and robotics fields can be exploited. Within the context of machine learning, the algorithm is shown to produce results in a real-world application with unpredictable dynamics, while for the robotics field, an algorithm is shown to address the problem of managing safety and liveness on an adaptive robot.

This dissertation is structured as follows. First, a review of current research in the fields of human-robot interaction, safety and liveness, and reinforcement learning in multiple constraint problems is presented. Then, Safe and Live Reinforcement Learning (SLRL), a modification of Multiple Constraint-Adaptive, Real-time Dynamic Programming (MC-ARTDP) [***2***], is discussed as a method of bringing these three fields together to impart knowledge of safety and liveness properties for an HRI task to an RL algorithm. Because the curse of dimensionality often prevents SLRL from being used on large problems, Safe and Live G-Learning (SLGL) is then proposed as a way of incorporating regression tree function approximation based on Pyeatt and Howe's version of G-Learning from [***3***] into a multiple constraint RL algorithm. Two experimental scenarios are then considered. In the first, SLRL and SLGL are compared to Q-Learning, G-Learning, and MC-ARTDP on a toy grid world problem. This test serves to highlight the capabilities of these two algorithms on a simple task. The second experiment shows how SLRL and SLGL can be used to satisfy the safety and liveness requirements of a human-robot object handover task similar to the scenario in [***4***]. In this test, a Wizard-of-Oz study was first conducted to gather real-world data on the user and robot's dynamics. From this data, a simulator was built. Q-Learning, G-Learning, SLRL and SLGL were trained on the simulator and then implemented on the BERT2 infrastructure at the

Bristol Robotics Laboratory. Trials with live users were then conducted to empirically compare algorithm performance.

Results indicate that SLRL accomplishes three things. First, SLRL includes a method of successfully translating safety and liveness properties into stochastic constraints for Markov Decision Problems (MDPs) in the form of additional value functions known as risk. Second, SLRL learns a greedy policy that maximises expected return such that the risk of violating any constraint is less than a specified threshold. Lastly, the safe ϵ-Boltzmann action selection policy used by SLRL throughout training ensures that, when possible, the expected risk of randomly chosen actions stays below the specified thresholds. This means that SLRL not only learns a safe and live policy like MC-ARTDP, but it can maintain those properties while exploring. However, SLRL uses a table format to represent the value and risk of each action in any given state resulting in huge memory requirements for complex problems. SLGL's use of regression tree approximation for the value and risk functions is shown to significantly reduce these memory requirements. It is also empirically shown to speed training on complex tasks by allowing the agent to generalise its experience to similar, unseen states. The Safe ϵ-Boltzmann action selection algorithm used by both SLRL and SLGL is also the only known exploration policy to consider expected risk. This innovative approach, which blends safety and liveness with multiple constraint reinforcement learning, goes beyond current practice. As such, these methods and results will be applicable to both robotics and machine learning researchers in projects with safety critical autonomous agents.

2 RELATED WORK

A significant amount of work has already been done in the areas of human-robot interaction (HRI), reinforcement learning with constraints, and safety verification. However, there has been little work integrating these three areas together.

2.1 HUMAN-ROBOT INTERACTION

Behavioural science studies have shown that people augment their conversations with many non-verbal queues such a nodding, gesturing, stance, and eye contact [***10***]. These non-verbal modalities allow people to colour their conversations with one another by communicating emotion or drawing attention. Robotics researchers studying human-robot interaction (HRI) attempt to harness these multimodal communications to improve the way people interact with robots. In fact, recent research has shown that people understand robots better when they use gaze to indicate spatial relationships in conversations [***11***]. Likewise, psychologists and sociologists can gain insight into human-human interactions by researching how people interact with robots.

2.1.1 Human-Human Interaction

Sociologists and psychologists studying human-human interaction (HHI) have discovered that people use gesturing and eye gaze to indicate the subject of a conversation. People will often look or point at an object they are referring to when speaking. We pick up on these queues to help disambiguate the conversation [***10***]. Children especially have been shown to rely on these non-speech modalities to communicate [***12***].

Exactly how much we use these non-verbal queues, however, is highly dependent upon the individual [***13***]. It has been shown that the amount of time people spend meeting the gaze of those they are speaking with varies from person to person. In the same way, some people have a higher tendency to gesture than others. If a robot is to successfully interact with humans, it will need to take many of these non-verbal queues into account.

2.1.2 The BERT2 Infrastructure

The Bristol Elumotion Robotic Torso 2 (BERT2) is an anthropomorphic robot developed at the Bristol Robotics Lab (BRL) explicitly for the study of multimodal HRI [***5***]. It provides researchers in the fields of robotics, psychology, and sociology with a reusable platform on which to study the ways in which people interact with both humans and robots. As such, it was built to allow simultaneous research across a number of normally separate streams of HRI including verbal and non-verbal communications, motion and object interaction, and high level cognitive reasoning [***5***].

2.1.2.1 Hardware

As shown in Figure 1 below, the BERT2 robot is a stationary, humanoid upper body including a torso, two arms, fully articulated hands, and a head [***5***]. The torso includes four joints that allow it to twist and bend forward or backward: hip rotation, hip flexion[1], neck rotation, and neck flexion. Each arm includes seven joints allowing for sophisticated movements. Joint motors for the torso and arms are directed by EPOS controllers and include both torque and absolute position sensors. These EPOS controllers are connected in a Controller Area Network (CAN) with a central controller on a Linux PC using a PCI-to-CAN interface. This central control PC provides error detection and acts as an interface to higher level control.

[1] Joint which bends the robot forward at an angle.

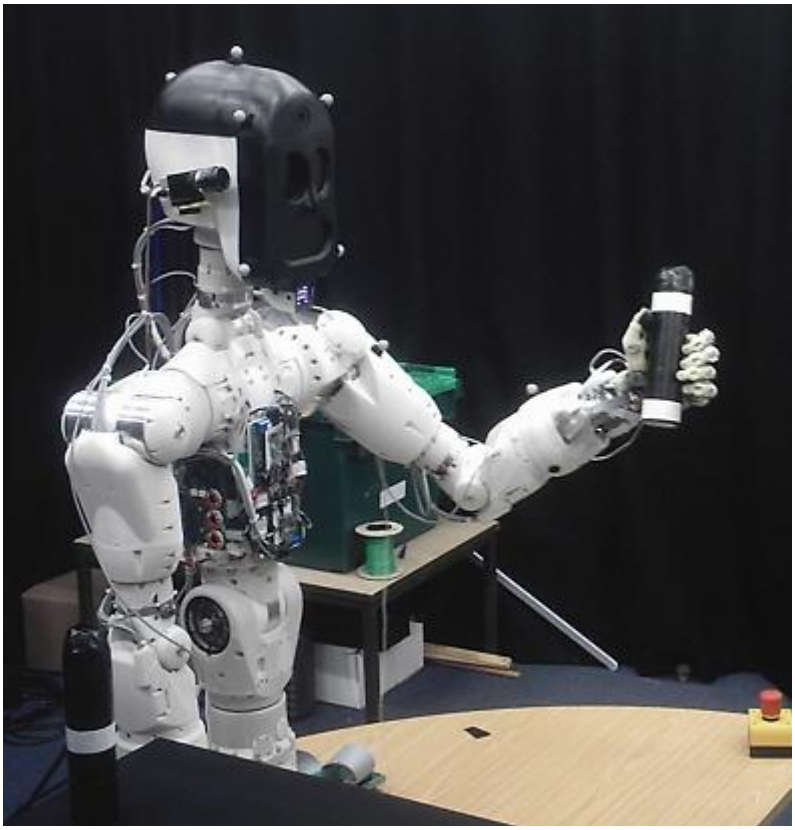


Figure 1 - The Bristol Elumotion Robotic Torso 2 (BERT2)

Each arm can be fitted with either an intricate hand capable of complex movements or a simple gripping device. Both hand options include sensors that detect pressure. Each hand is mechanically self-contained with EPOS controllers, a CAN, and a microchip controller to drive the CAN. This microchip acts as the control interface to the hand for higher layer control.

BERT2's head was specifically designed to appear only somewhat human-like. This allows researchers to avoid people adversely reacting to the robot due to it appearing nearly, but not quite human. This issue, otherwise known as the Uncanny Valley[2], has been avoided by purposefully designing BERT2's head to be flat on the front, and triangular on the top. However, the head is fully expressive with a flat panel, LCD screen which renders the robot's eyes, eyebrows, and mouth. This flat panel can display eight emotions: happy, thinking, angry, disgusted, surprised, afraid, sad, and tired.

BERT2's head is not purely for expression, however. It also includes two infrared cameras used by the faceLAB™ gaze tracking system by Seeing Machines. This system is able to detect the position of someone's face as well as determine the angle of their gaze with high accuracy. However, it has a very narrow field of view and will only work if the robot's head is pointed almost directly at the person in question. To solve this problem, a face tracking system called Facefinder was implemented with a much wider viewing angle. This system uses a small camera installed in the robot's forehead. By providing the x,y coordinate of a face, the higher level controller can move BERT2's head to a position in which the gaze tracker can see the person's eyes.

Because BERT2 is designed for primarily HRI research, an off-the-shelf, VICON motion capture (MoCap) system has been included to provide highly accurate information about the locations and

[2] If one graphed the human likeness of robots by the familiarity people feel with those robots, there would be a steadily increasing line of familiarity as robots become more humanoid. However, when robots become almost like humans but not quite, people tend to react negatively. Thus, the graph has a big trough just before robots look just like people. This trough of negative reactions is known as the Uncanny Valley [***35***].

movements of objects and individuals within the vicinity. This significantly reduces the uncertainty about the environment that comes with relying on computer vision and range finders.

2.1.2.2 Software

The BERT2 software infrastructure is based on Yet Another Robotics Platform (YARP). YARP is a middleware package written in C++ that allows robotics controllers to be developed in a highly modular fashion [***8***]. It abstracts away the communications between control modules and devices in such a way as to allow devices and controllers to be reused across projects. Control modules can be run on any machine with a network connection to the rest of the system. Modules can connect to each other via TCP, UDP, multicast, or shared memory. These connections can be established at any time, even during execution. This makes YARP controllers highly portable.

BERT2 uses YARP to connect the low level hardware controllers with the higher level cognitive controllers on multiple PCs across a local area network (LAN) [***5***]. Each low level controller connects to the cognitive control network via YARP interfaces providing sensor readings as output streams and accepting commands as input streams.

Speech recognition and synthesis is provided by CSLU Toolkit Rapid Application Development (RAD). This connects a Festival speech synthesis system and Sphinx-II speech recognition system to the cognitive control network via another YARP wrapper. The RAD toolkit allows speech interactions to be constructed via a state-based, graphical programming environment.

Knowledge about the environment is provided by two databases. The Object Property Database (OPDB) stores static information about objects known to the robot. This includes information such as the object's size, colour, and name for speech production and recognition. Any objects within the vicinity of the robot such as chairs, cups, nearby people, and the robot itself are included in this database. Transient information about objects such as current position and velocity are stored in the EgoSphere database.

The last module, Gaze-Object-Detection, sits between the low level control and perception modules and the higher level cognitive control modules. It aggregates information from the gaze detection system, the EgoSphere, and the OPDB to calculate what object a person is looking at. This information is then provided to the higher level cognitive modules via a YARP output stream.

2.1.3 Human-Robot Object Handoffs

Recently, a joint group of researchers from Carnegie Mellon University, Georgia Institute of Technology, and Intel Labs have conducted a series of research projects to study various aspects of human-robot object hand-over tasks [***14***] [***15***] [***16***].

2.1.3.1 Models of Object Handoffs

In the first study [***14***], Lee et al. observed a series of handoffs between humans and trained dogs and then between humans performing cooperative tasks. From this, they identified two types of handover models: a fixed model based on the human-dog handoffs and an adaptive model based on the interactions between people. Each model contained three different phases: the carrying, signalling, and handoff phases. In the carrying phase, one party brings the object to the other. In signalling, the party holding the object signals to the other that it is going to give it to the other. In handoff, the second party grabs the object and it is transferred.

In a fixed model, the object can be carried to an adaptive location for the receiver. However, it is always carried with the same orientation. The handoff is then signalled with a spoken cue. In the handoff phase, the object is moved close to the receiver. When the receiver grabs it, another spoken cue indicates that it is time to release the object. The only adaptation in this model is the receiver's location.

In an adaptive model, the object can be carried in many different ways including using one or two hands. In the signalling phase, the handoff is indicated through any verbal or non-verbal cue, including moving the object closer to the other party. If the receiver is busy, the handoff is paused. When the handoff is started, the object is again moved close to the receiver in an orientation that is comfortable for that person. This model has many different adaptation requirements. This includes the ability to recognise when the receiver is busy, detecting non-verbal cues such as motioning or glancing at the object, and to orient the object in a way that is comfortable for the receiver. This significantly increases the complexity of the task.

2.1.3.2 Spatial and Temporal Contrast

In the next study [***15***], Cakmak et al. examined the use of spatial and temporal contrast in a service robot to improve the fluency of object handoffs. This was motivated by a demonstration event where a robot waiter served drinks to users in a crowd. When this robot wanted to serve a drink to a person, it would verbally prompt them to grab the drink. However, they would often not hear this if they were otherwise busy. When they finally noticed the robot, they would sometimes wait for some indication that the robot wanted to hand them the drink even though it was already offering it to them.

From these observations, Cakmak et al. proposed the use of spatial and temporal contrast to signal handover events to the user. Spatial contrast was defined as the use of distinct poses when the robot was in different states. For instance, the robot should have a handover pose which is easily recognisable. On the other hand, temporal contrast is the use of transitioning from one pose to another in order to signal the handoff event. They proposed that temporal contrast would allow the user to easily predict the timing for the handoff and increase fluency.

To test these propositions, Cakmak et al. sat 24 users in front of computer terminals. A robot would then approach with a cup carried by a gripper arm. The robot would offer the cup to the user and the fluency of the cup handoff would be measured. Subjects were broken into four groups. In the first group, the robot would use neither spatial or temporal contrast to handoff the cup. In the second group, the robot would only use spatial contrast. For the third group, only temporal contrast was used. Finally, the last group experienced both spatial and temporal contrast in each handoff. To measure fluency, they recorded the amount of time the user waited to receive the object.

These tests revealed that temporal contrast significantly improved the fluency of the handoffs. When the robot moved from a carrying pose to a noticeably different handoff pose with consistent timing, the user was able to predict the best time to grab the cup to reduce waiting time. In addition, the timing of the change from carrying pose to handoff pose was more important than the spatial contrast of the handoff pose. In fact, for these trials, spatial contrast alone did not impact the fluency of the handover. However, the researchers propose this was because the users expected the handoff in these trials. Spatial contrast would mostly likely impact the fluency of handoffs if the robot was performing other actions as well. Thus, spatial contrast could be used to signal which action the robot is performing.

2.1.3.3 Mitigating Breakdowns

Finally in [***16***], Lee et al. investigated ways to manage customer's perceptions when service robots break down. They first hypothesised that managing expectations by informing the customer of a robot's abilities and limitations before starting a service will improve the customer's satisfaction when breakdowns occur. They also proposed a number of different strategies for mitigating these service errors. These included apologising for errors, presenting the user with different recovery options, and compensating the user for the error. This compensation might take the form of free goods or services or reduced prices in future transactions among other things. They hypothesised that each customer has a service orientation that predisposes them to respond better to different mitigation strategies. They identified two dimensions of customer service orientation: relational orientation and utilitarian orientation.

With the relational service orientation, the customer views the service interaction as a relationship and wish to maintain it. With the utilitarian orientation, the customer views it as a business interaction and is more concerned with the transfer of goods and services.

To test these hypotheses, they conducted a study in which 317 people watched recordings of one of two robots experiencing errors while serving drinks to customers. They found that when the customer was informed of the robot's limitations and possible errors before service began, satisfaction improved. They also found that customers with a high relational orientation responded well to the apology and options strategies. Likewise, customers with high utilitarian orientation responded best to the compensation strategy.

2.2 MANAGING SAFETY

One of the key issues in HRI involves how to ensure the safety of those interacting with the robot. Murphy and Woods argue in [***17***] that those developing an HRI system are the ones responsible for the safety of those interacting with it. This is no different from any other product. Because of this, HRI developers should adhere to the same stringent standards already in place for other safety critical industries. The automotive and aircraft industries have been dealing with the question of autonomy in safety critical systems for decades. They have developed numerous standards for dealing with these systems. While similar standards are currently lacking in the robotics industry, robotics researchers and engineers can look to the aircraft and automotive industry standards to ensure the highest level of safety in their own systems. In many ways, the same verification requirements from these industries can and should be used directly in the development of HRI systems.

These requirements are usually expressed in the form of safety and liveness properties. Safety and liveness were first introduced in the context of multi-process systems by Lamport in [***1***]. Informally, Lamport defines safety properties as undesirable conditions which the system must avoid, while liveness properties specify events that should eventually occur. Thus, safety properties can be seen as the opposite of liveness properties. In HRI the goal is to identify safety properties which specify the conditions which would result in either damage to the robot or the people it interacts with. At the same time, liveness properties are defined which ensure the robot is able to accomplish its task.

2.2.1 Safety and Liveness of a Handoff Task

In [***4***], Grigore et al. studied the specification of safety and liveness properties for a robot tasked with serving either a hot drink (coffee) or a cold drink (water) to a user. This experiment was conducted using components from a predecessor to the BERT2 robot [***5***]. This gave the robot the ability to track user's eye gaze through the faceLAB™ gaze tracking system. It also used the VICON Motion Capture system to accurately sense the user's position at all times.

In this task, a user approaches a stationary robot at a table [***4***]. The robot asks whether the user would like the hot or the cold drink. The robot is then tasked with grabbing the cup, offering it to the user, and releasing it when it is safe to do so. In order to determine when it is safe to release the cup, a number of conditions were identified in the form of the following paraphrased safety properties:

- **[S1] –** The water is not released unless the user is looking at it, their hand is next to it, and a sufficient amount of pressure is applied to it.
- **[S2] –** The coffee is not released unless the user is looking at it, their hand is next to it, a sufficient amount of pressure is applied to it, and the user confirms that they are holding it.
- **[S3] –** The user is not asked if they would like another cup unless the current cup has been handed over or the current handoff aborted.

In order to assess the robot's ability to complete its tasks while maintaining these safety properties, the following liveness properties were identified:

[L1] – If the user follows the requirements in the safety properties, the cup will eventually be released.

[L2] – If the requirements are *not* met, the robot will repeat the handover procedure at most three times before aborting and starting over.

A static controller was developed to accomplish this task using predetermined thresholds to define when the user's hand is close to the cup and how much pressure is sufficient to release it. Through experiments with five users interacting with the robot in hour-long sessions, they were able to conclude that adherence to these properties were sufficient to ensure that the cup was successfully transferred without incidence.

2.2.2 Verification

In hardware and software development, there is often a need to show that a system adheres to some specification. This specification defines the necessary behaviours of the system often including safety and liveness properties. Verification as described by Chockler et al. in [***18***] provides a suite of techniques for demonstrating that the system is correct with respect to its specification. There are two main categories of verification: static and simulated verification. In static verification, sometimes called formal methods, a mathematical model of the system is written alongside a similar description of the specification. The model is then formally demonstrated via model-checking or theorem proving to adhere to the specification. Clarke and Wing explain model-checking in [***19***] as the process of building a finite model of a system using techniques such as finite state automata or temporal logic and then proving that the model adheres to the desired properties. Theorem proving, on the other hand, uses a formal logic specification of the system to prove that the desired properties hold. These methods can be automated. However, due to computational complexity issues, they are often partially incomplete. Even if static verification is able to prove that the formalisation of the system meets the specification, its verity is limited by how well the formal description matches the actual system implementation.

On the other hand, simulated verification, sometimes known as dynamic verification, is described by Chockler et al. in [***18***] as testing a system to ensure its behaviour matches the specification. This is done by defining a reference model of the system which is known to adhere to the specification and a set of input vectors. Both the system and a reference model are run on the input vectors. If the system upholds the specification, then the outputs of the system and the reference model should match. However, it is generally infeasible to test a system against all possible input vectors. Therefore, simulated verification can only demonstrate verity for those input vectors it tests. This issue is compounded when testing adaptive systems since the system's outputs for any given input may change over time.

2.2.3 Liveness as Safety

In order to demonstrate that a system adheres to a safety property, model checkers typically search for a counter-example to that property [***6***]. If property p_s is a safety property stating event s should not occur, the model is checked for a state in which s does occur. On the other hand, liveness properties can particularly problematic for model-checking. Liveness properties typically state that something *should* eventually happen. Simply checking for counter-examples of liveness is more difficult. If property p_l is a liveness property which says event l should eventually occur, a counter-example would be a sequence of states which does not contain l. However, if time is infinite, it may not be possible to produce such a counter-example since event l might occur after the last state in the sequence. One method for dealing with this issue involves transforming liveness property p_l into a safety property with bounded time. This can be done by defining safety property p_l' as follows: it should not be the case that event l does not occur within x time steps. This can then be checked by searching for a counter-example in the same manner as

other safety properties. However, it should be noted that this is an approximation of the true liveness property. Event l still might occur after time x.

2.2.4 APT Agents

Because of the issues with simulated verification in adaptive systems, Gordon proposes a mechanism for incorporating static verification into adaptive system in [***20***] with her Adaptive, Predictable, and Timely (APT) agents. APT agents are groups of finite state automata (FSA) which act together to accomplish a specified goal. These FSAs are initially verified using model checking. As time goes on, each agent's FSA is adapted using a genetic algorithm. Before these new adaptations are implemented, they are re-verified against the required safety and liveness properties. If the verification is successful, the adapted FSAs replace the old ones. Otherwise, they are rejected.

The real innovation with APT agents is that the re-verification process is significantly quicker than the initial verification. A full model check of each adapted FSA would significantly slow down the responsiveness of agent adaptations. So, APT agents use knowledge of the adaptation process to speed up re-verification significantly. They do this through the definition of safe machine learning operators (SMLOs). A SMLO is any adaptation operator on a FSA which is proven to preserve a property of that FSA in the newly adapted FSA. More formally, if S is an FSA, P is a property defined on S, and a machine learning operator $o: S \rightarrow S'$ has adapted S to a new FSA S', then operator o is a safe machine learning operator with respect to property P if $S \models P$ implies $S' \models P$. Thus, whenever a SMLO is applied to a FSA which has already been verified to maintain a set of safety properties, the new FSA does not need to be re-verified. Its safety and liveness are already guaranteed.

Gordon then shows two operators on FSAs that are guaranteed to be SMLOs for safety and liveness properties: o_{spec} and o_{del}. The first operator, o_{spec}, is any operator which specializes the transition requirements from one state to another. In other words, an o_{spec} operator makes transitions less likely to fire. The second operator, o_{del}, deletes whole transitions. Likewise, two operators, $o_{spec+gen}$ and $o_{del+gen}$, are shown to be safe with respect to safety properties but not liveness properties. These operators move transition conditions from one edge to another. Because of these SMLOs, adapted FSAs only have to be re-verified when non-SMLOs are applied. This can significantly speed up re-verification of adapted controllers.

2.3 REINFORCEMENT LEARNING

There are many ways in which HRI systems can benefit from adaptive control algorithms. They can allow robots to react to dynamic environments, different users, new actions, or even new goals. A large body of recent research has explored various ways to refine and adapt machine learning techniques to this area. The reinforcement learning class of algorithms has proven to be highly successful in dynamic, low-level robot control, and many researchers have started to apply these techniques to HRI [***9***] [***21***] [***22***].

2.3.1 Standard Reinforcement Learning

Reinforcement learning (RL) is a class of machine learning algorithms in which an agent seeks to maximise a numerical reward passed to it by the environment [***23***]. The agent accomplishes this by choosing actions based on its current state. Thus, RL essentially optimises the agent's behaviour.

2.3.1.1 Markov Decision Processes

Effectively, the agent attempts to learn the optimal path through a Markov Decision Process (MDP) [***24***]. A MDP represents a sequential decision problem as a directed graph of possible world states connected by actions which can be taken by the agent. Each state-action transition results in a numerical reward.

MDPs may also be episodic or continuous. An episodic MDP includes a start state and a set of possible end states. In this formulation, the agent begins in the start state and chooses actions until it transitions to an end state. Then the episode is over, and the agent can then be reset. A continuous MDP only includes a start state. The agent simply continues taking actions until it is no longer able.

A MDP is specified by its distribution of transition probabilities and expected rewards. Each action from any given state will have an associated probability distribution for transitioning the agent to a set of possible next states. The probability that action a taken in state s will transition the system to state s' is given by:

$$\mathcal{P}^{a}_{s,s'} = \Pr(s_{t+1} = s' | s_t = s, a_t = a) \quad (1)$$

Where s_{t+1} is the state at time $t+1$, s_t is the state at time t, and a_t is the action taken at time t. The expected reward r received due to this transition is then given by:

$$\mathcal{R}^{a}_{s,s'} = \mathrm{E}[r_{t+1} = r | s_t = s, a_t = a, s_{t+1} = s'] \quad (2)$$

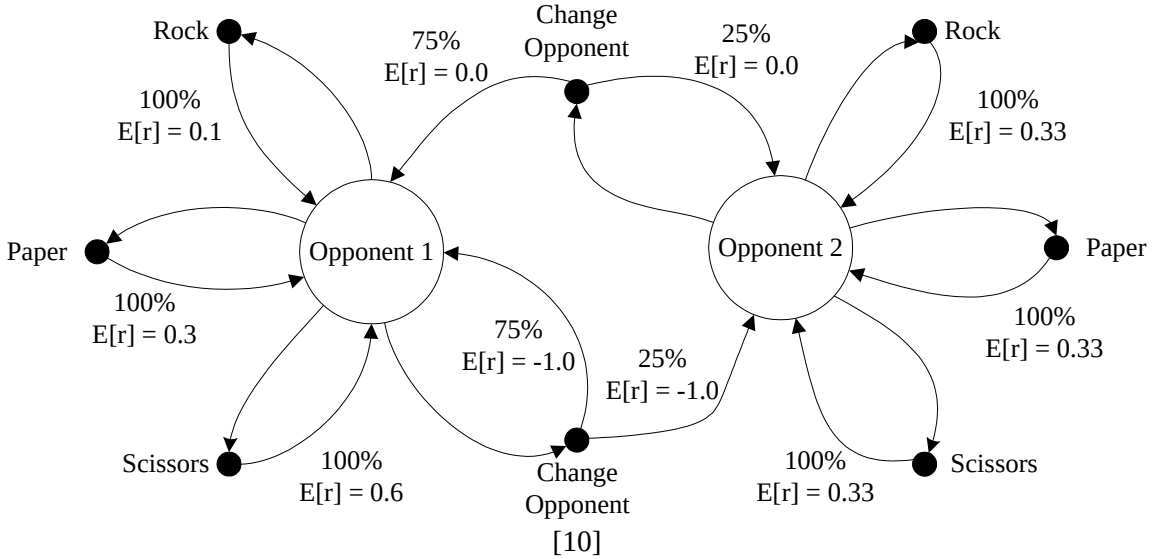
Where r_{t+1} is the reward at time $t + 1$.

By defining the reward and transition distributions this way, they satisfy the Markov property. The Markov property states that the reward r_{t+1} and state s_{t+1} at time $t+1$ can only depend on the state s_t and action a_t at time t no matter what happened before time t. Equation (3) below describes this formally [***24***]:

$$\Pr(s_{t+1} = s', r_{t+1} = r | s_t, a_t) = \Pr(s_{t+1} = s', r_{t+1} = r | s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) \quad (3)$$

An example MDP is shown in Figure 2 below. In this example, an agent is tasked with playing a series of rock-paper-scissors (also known as stone-paper-scissors) matches against two opponents. In each time step, the agent chooses between playing a round against its current opponent or trying to switch opponents. If it chooses to play a round it can choose rock, paper, or scissors. The opponent also chooses one of these options. The winner of the game is determined by the following heuristic: rock beats scissors, scissors beats paper, and paper beats rock. If the agent wins the round, it receives a reward of one. If it loses the match, it receives zero reward. Thus, the expected reward of each play directly reflects the probability of winning the round against the current opponent. The first opponent prefers to play paper. So the expected rewards against that opponent reflect a skew towards scissors. However, the second player chooses uniformly at random. So the expected reward of each play is equal. If the agent tries to switch opponents, there is a 25% chance the switch will be successful. However, there is a 75% chance that it will still face the same opponent next round. Either way, the agent receives a reward of

Figure 2 - Multi-Opponent Rock-Paper-Scissors



negative one for the cost of switching.

2.3.1.2 Policies, Expected Return, and Value Functions

The agent's goal is to learn an action policy which maximises its expected return over time [***24***]. A policy defines which action a out of the set of possible actions A to perform in each state. Policies (usually denoted by π) can either be deterministic or stochastic. Deterministic policies will always generate the same action in a given state. However, stochastic policies will choose actions from a subset of possible actions $A' \subseteq A$ based on a probability distribution defined over A' such that the probability of selecting action a in state s is given by $\pi(s, a)$.

Agents adapt their policies in order to maximise their expected return. Expected return can be defined as either discounted or undiscounted return depending upon the MDP. Undiscounted return is defined as the total reward received by the agent by taking a given action in a state and following its policy in subsequent states until it reaches an end state. Formally, the undiscounted return expected at time t is defined where T is the final time step as:

$$R_t = \sum_{i=t+1}^{T} r_i \tag{4}$$

Undiscounted expected return is particularly well suited to episodic MDPs. However, it can be applied to continuous MDPs by defining a fixed number of future actions to consider in the total reward calculation.

Discounted return, on the other hand, is a measure of the total amount of reward received by an agent by taking a given action in a state and by then following the agent's policy in each subsequent state. However, how much the reward for each subsequent action counts in the expected return depends upon how far in the future it is. In this way, the agent considers future rewards in building its policy but weighs rewards in the near future more than reward in the far future. This makes discounted returns particularly well suited to continuous MDPs. Discounted return at time t is calculated using a real valued discount parameter γ such that $0 \leq \gamma \leq 1$:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{5}$$

However, in the special case that $\gamma = 1$, Equation (5) is the same as undiscounted return in Equation (4).

This concept of expected return is used in RL algorithms to calculate value functions. Value functions describe either the quality of a state or the quality of an action performed in a state given an action policy. Therefore, the value of state s given a policy π is defined as:

$$V^{\pi}(s) = \mathrm{E}_{\pi}[R_t | s_t = s] = \mathrm{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right] \tag{6}$$

In other words, the value of state s given policy π is the expected value of the return when executing π starting from s. The function V^{π} is known as the state-value function on policy π.

Likewise, the value of action a in state s given policy π is the expected value of the return when executing action a in state s and following policy π thereafter. This is the action-value function on policy π, or Q^{π}. Mathematically, Q^{π} is defined as:

$$Q^{\pi}(s,a) = \mathrm{E}_{\pi}[R_t|s_t = s, a_t = a] = \mathrm{E}_{\pi}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\middle| s_t = s, a_t = a\right] \quad (7)$$

Expanding upon these equations allows the definition of Bellman equations for V^{π} and Q^{π}. Bellman equations are recursive function definitions which allow RL algorithms to efficiently compute the value of one state based on the value of future states. The Bellman equations for V^{π} and Q^{π} are:

$$V^{\pi}(s) = \sum_{a\in A}\pi(s,a)\sum_{s'\in S}\mathcal{P}^a_{s,s'}\left[\mathcal{R}^a_{s,s'} + \gamma V^{\pi}(s')\right] \quad (8)$$

$$Q^{\pi}(s,a) = \sum_{s'\in S}\mathcal{P}^a_{s,s'}\left[\mathcal{R}^a_{s,s'} + \gamma V^{\pi}(s')\right] \quad (9)$$

As agents conduct their search for the best policy, they are essentially trying to find the policy that maximises these value functions. This is known as an optimal policy. Optimal policies have the highest expected return possible in every state in an MDP. The value functions that correspond to these policies are optimal value functions. Optimal state-value functions are denoted V^* and are defined as:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad (10)$$

Optimal action-value functions are denoted Q^* and are formally defined by [***24***] as:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s) = \mathrm{E}[r_{t+1} + \gamma V^*(s_{t+1})|s_t = s, a_t = a] \quad (11)$$

In the augmented rock-paper-scissors MDP from Figure 2, the agent's task is to learn a policy which maximises the received reward over time. Assuming the agent is playing an infinite number of trials with a sufficiently high value for γ, an optimal policy for this MDP would be to switch to the first opponent and then only play scissors. However, this is only optimal if the opponent's reaction does not change over time.

2.3.1.3 Dynamic Programming

If a MDP's transition and reward probabilities are known, an agent can directly calculate an optimal policy using dynamic programming (DP) [***24***]. DP is an iterative process that can directly solve for the optimal value function (either V^* in Equation (10) and Q^* in Equation (11)). Once either optimal value function is known, the agent simply looks up the value of performing each possible action in its current state and chooses the one with the highest value.

While there are many algorithms for finding V^*, the main algorithm of interest here is value iteration

Figure 3 - The Value Iteration Algorithm

Initialise $V(s)$ arbitrarily for all $s \in S$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in S$
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s'\in S}\mathcal{P}^a_{s,s'}\left[\mathcal{R}^a_{s,s'} + \gamma V(s')\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
Until $\Delta < \theta$ (a small positive number)

Output deterministic policy π where:
    $\pi(s) = \arg\max_a \sum_{s'\in S}\mathcal{P}^a_{s,s'}\left[\mathcal{R}^a_{s,s'} + \gamma V(s')\right]$

shown in Figure 3 above. Value iteration maintains a table of the current value estimates for each state s. These values are first initialised to some arbitrary value. The algorithm then iterates over each table entry using a modified version of the Bellman equation for V^{π} from Equation (8) to update each entry. The update equation used is:

$$V(s) = \max_{a} \sum_{s' \in S} \mathcal{P}^{a}_{s,s'} \left[\mathcal{R}^{a}_{s,s'} + \gamma V(s') \right] \tag{12}$$

The algorithm repeatedly updates every entry in the table until the difference in values in each iteration is relatively small. By repeatedly sweeping updates through the table in this fashion, the value table V is guaranteed to converge to V^{*}. Thus, value iteration is guaranteed to return an optimal solution for a stationary MDP.

The problem with DP is that the time and space needed to calculate optimal policies makes them feasible for only small MDPs. Also, they can only be applied when the reward and transition probabilities are known.

2.3.1.4 Temporal Difference Learning

TD learning algorithms provide a way to solve MDPs without knowing the reward and transition distributions [***24***]. They differ from DP algorithms in that they use experience to calculate value functions instead of the MDP's actual reward and transition probability distributions. They do this by maintaining a table of value entries for each state (or state-action pair in the case of Q functions). These entries are initialised to arbitrary values. The agent then generates experience by choosing an action a and observing the reward r and next state s'. The (s, a, s') tuple is then used in a Bellman equation to update the value estimate for $V(s)$ (or $Q(s, a)$ if using action-value instead of state-values). This makes TD methods applicable when the MDP is not known and DP cannot be used.

TD methods are guaranteed to converge to the optimal value functions if every action in every state is tried an infinite number of times. So to adequately approximate the optimal value function, the agent must be allowed to select suboptimal actions. This gives rise to the explore-exploit dilemma. In order to maximise the amount of reward an agent receives, it must exploit its knowledge by choosing actions with the best value (the greedy action). However, the agent needs to explore its environment by picking suboptimal actions to increase the accuracy of its value estimates. It cannot do both at the same time. The method used by the agent to balance these opposing needs is its exploration policy.

There are two main types of exploration policies: ϵ-greedy policies and soft-max policies. The first, ϵ-greedy policies, choose the greedy action with a probability $1 - \epsilon + \frac{\epsilon}{|A(s)|}$ where $|A(s)|$ is the number of possible actions in state s and ϵ is between zero and one. All the remaining actions are selected with probability $\frac{\epsilon}{|A(s)|}$.

While ϵ-greedy policies choose suboptimal actions with uniform probability, soft-max policies choose actions with probabilities proportional to their value. A common soft-max policy chooses action according to a Boltzmann distribution:

$$\pi(s, a) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A(s)} e^{Q(s,a')/\tau}} \tag{13}$$

The positive, non-zero parameter τ in this equation is the temperature. As τ approaches infinity, the policy chooses actions more and more uniformly. As τ approaches 0, the policy is more likely to choose the greedy action.

2.3.1.5 Q-Learning

Q-learning is a popular TD algorithm for solving MDPs without knowing the transition and reward probabilities [***24***]. It uses the following update equation to find an optimal action-value function Q^* within the general TD algorithm:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right] \quad (14)$$

The parameter α in this function is a learning step-size parameter between zero and one that controls how fast the algorithm converges to Q^*.

By choosing actions according to either an ϵ-greedy policy or a soft-max exploration policy and then updating its action-value function using Equation (14) with the observed next state and reward values, Q-learning is able to steadily increase the performance of its policy while using it. The full Q-learning algorithm is given in Figure 4 below.

The crucial difference between Q-learning and other TD algorithms comes from the use of max in Equation (14). It updates the current estimate of the Q function towards the value of the best action a' in the next state s'. Other TD algorithms update the current estimate towards the value of the actual action a' taken in the next state s'. This is a subtle, but significant difference. By updating the value towards the best next action instead of the actual next action, Q-learning calculates the Q function for the greedy policy π^* instead of the exploration policy in use. This makes Q-learning an off-policy TD learning algorithm as opposed to an on-policy algorithm. Off-policy means that the algorithm learns about one policy while executing a different policy altogether. Thus, Q-learning agents can start with an ϵ-greedy policy or a soft-max exploration policy to learn about the MDP and then immediately switch to a greedy policy to exploit that information. This makes Q-learning a very versatile algorithm for solving MDPs.

2.3.1.6 Learning Rates

In order to ensure that Q-Learning converges to an optimal solution, the learning rate, α , must be slowly decreased towards zero. However, the rate of this convergence significantly depends on the rate at which α is decreased [***25***]. In his work presented in [***25***], Gosavi empirically studied the impact of different equations used to decrease α on convergence. He examined three schemes:

$$\alpha = {}^{1}/_{t} \quad (15)$$

$$\alpha = \frac{a}{(b+t)} \quad (16)$$

$$\alpha = \frac{\log(t)}{t} \quad (17)$$

Figure 4 - The Q-Learning Algorithm

Initialise $Q(s,a)$ arbitrarily

Repeat for each episode
    Initialise s
    Repeat for each step of the episode
        Choose a from s using policy derived from Q
        Take action a, observe r, s'
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$
Until s is terminal

Where t is the current iteration and both a and b are parameters that must be tuned. When tested on a simple MDP using Q-Learning, he found that equation (15) converged quite slowly. However, even though equation (15) is a special case of equation (16) where $a = 1$ and $b = 0$, equation (16) converged drastically faster when $a = 150$ and $b = 300$. This required extensive experimentation to find the optimal parameters, though. The log rule given in equation (17) performed almost as well as equation (16), but did not require any tuning. This suggests that the log rule is a robust method for decreasing the learning rate over time.

2.3.2 Risk-Sensitive Reinforcement Learning

The classic value function methods for solving RL problems do an excellent job of maximising reward over the long term. While many of these algorithms are guaranteed to find optimal action policies, they may intentionally pass through very undesirable states to do so [***7***]. An agent following an RL optimal action policy may perform very risky actions if the potential reward payoff is high enough. The only way to influence the amount of risk an agent will take is by modifying the reward function. However, that can have unintended consequences. The other problem with standard RL algorithms is their need to explore every state-action pair a large number of times in order to converge to an optimal action policy. Thus, a RL agent must be allowed to take very risky actions many times during learning [***26***]. Both of these factors make standard RL algorithms unsuitable on safety critical systems.

In [*7*], Geibel and Wysotzki address the first problem by considering a MDP containing a number of error states. They then define an algorithm which finds the best policy whose probability of entering an error state is bounded by a user defined maximum. To do so, they first define an extension to the standard MDP formulation. In this modified MDP, a subset of states are designated as error states $\Phi \subset S$. Where S is the set of all states in the MDP. These error states are terminal states as well. A subset of states is also designated as non-error terminal states $\Gamma \subset S$ such that $\Gamma \cap \Phi = \emptyset$.

In this context, risk is defined as the probability that an action policy will terminate in an error state. More formally, the risk of a state s with respect to policy π is defined as:

$$\rho^{\pi}(s) = \Pr(\exists i, s_i \in \Phi \mid s = s_0) \tag{18}$$

Thus, $\rho^{\pi}(s) = 1$ if $s \in \Phi$, and $\rho^{\pi}(s) = 0$ if $s \in \Gamma$.

In order to calculate risk, an absorbing final state η is added to the MDP such that all the states in $\Gamma \cup \Phi$ automatically transition to it. The rewards $r_{s,a}(s')$ for transitioning to η are set to 0. A numerical cost signal $\bar{r}_{s,a}(s')$ is then added to every transition in the MDP where state $s'$is the resulting next state when performing action a in state s and:

$$\bar{r}_{s,a}(s') = \begin{cases} 1 : if\ s \in \Phi\ and\ s' = \eta \\ 0 : otherwise \end{cases} \tag{19}$$

Thus, cost $\bar{r}$ is one when transitioning from an error state to the absorbing state and zero otherwise.

Risk can then be calculated as the discounted expected cost similar to the discounted expected reward using a cost discount factor $\bar{\gamma}$:

$$\rho^{\pi}_{\bar{\gamma}}(s) = \mathrm{E}\left[\sum_{i=0}^{\infty} \bar{\gamma}^{i}\, \bar{r}_i \,\middle|\, s = s_0\right] \tag{20}$$

This is then used to define a function $\bar{Q}$ similar to Q which gives the risk for state-action pairs:

$$\begin{aligned}\bar{Q}^{\pi}(s,a) &= \mathrm{E}\left[\bar{r}_0 + \bar{\gamma}\rho_{\bar{\gamma}}^{\pi}(s_1)\middle| s_0 = s, a_0 = a\right] \\ &= \sum_{s'} p_{s,a}(s')\left(\bar{r}_{s,a}(s') + \bar{\gamma}\rho_{\bar{\gamma}}^{\pi}(s')\right)\end{aligned} \tag{21}$$

This $\bar{Q}$-function can then be used together with the standard Q-function in an algorithm designed to find a policy which maximises expected value while constraining expected risk below a specified probability ω. To do so, a third pair of state and state-action value functions is defined which combine the reward and risk value functions with a parameter $\xi \geq 0$:

$$V_{\xi}^{\pi}(s) = \xi V^{\pi}(s) - \rho_{\bar{\gamma}}^{\pi}(s) \tag{22}$$

$$Q_{\xi}^{\pi}(s,a) = \xi Q^{\pi}(s,a) - \bar{Q}^{\pi}(s,a) \tag{23}$$

The parameter ξ dictates the relative importance of reward to risk. When $\xi = 0$, reward is not taken into consideration and the V_0^{π} and Q_0^{π} functions simply act to minimise risk. As ξ approaches infinity, the importance of risk becomes negligible. The V_{ξ}^{π} and Q_{ξ}^{π} functions act to maximise reward.

Geibel and Wysotzki's algorithm computes the maximal reward policy with constrained risk by first finding the optimal policy π_{ξ}^{*} for Q_{ξ}^{π} with $\xi = 0$. This is done with a variant of the Q-learning algorithm. The same Q-learning control loop is used as in Figure 4 above. However, the update equation updates all three Q functions in turn:

$$Q^{t+1}(s,a) = (1-\alpha_t)Q^t(s,a) + \alpha_t\left(r + \gamma Q^t(s',a^*)\right) \tag{24}$$

$$\bar{Q}^{t+1}(s,a) = (1-\alpha_t)\bar{Q}^t(s,a) + \alpha_t\left(r + \gamma \bar{Q}^t(s',a^*)\right) \tag{25}$$

$$Q_{\xi}^{t+1}(s,a) = \xi Q^{t+1}(s,a) - \bar{Q}^{t+1}(s,a) \tag{26}$$

The greedy action a^* is chosen in each iteration with respect to its Q_{ξ}^{t} value. If multiple actions have the same Q_{ξ}^{t} value, then the action with the highest Q^t value is chosen. Exploration is performed in any manner suitable in standard Q-learning such as ε-greedy or soft-max selection. The learning parameter α_t is initially set to 1, and is decreased monotonically over time. Geibel and Wysotzki argue in [7] that this algorithm will find an optimal policy on Q_{ξ}^{t} if $\gamma = \bar{\gamma}$. If $\gamma = \bar{\gamma} = 1$, then it is also necessary to ensure that all policies either terminate or result in infinite costs.

The initial policy π_0^* resulting from using this learning method with $\xi = 0$ is the minimal risk policy. This policy is then checked to ensure that it does not violate the risk constraint. A policy violates the risk constraint if any non-error state $s \in \{S - \Phi\}$ is greater than ω. In other words, the policy is feasible if:

$$\forall s \in \{S - \Phi\}, \qquad \rho_{\bar{\gamma}}^{\pi}(s) \leq \omega \tag{27}$$

If the minimum risk policy π_0^* is not feasible, then there is no solution to the MDP. If it is feasible, then ξ is increased by a small amount and Q-learning is run again to find the new optimal policy π_{ξ}^{*}. This effectively relaxes the amount of allowable risk in the new policy. As long as π_{ξ}^{*} is feasible, the ξ constraint is increased and a new policy is found. Because the amount of risk in each policy π_{ξ}^{*} increases with each iteration, an infeasible policy is eventually found. When it is, the previous iteration's feasible policy is returned as the policy with the maximum reward within the risk constraint.

To test this algorithm, Geibel and Wysotzki devised a toy grid world problem. This problem consisted of a six by six grid as shown in Figure 5 below. This grid contained two goal squares and two cliffs. One

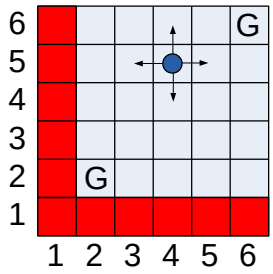


Figure 5 - Geibel and Wysotzki's Grid World Problem

cliff was located on the bottom row and the other was the leftmost column. The goals were placed in the top right and bottom left squares. In each episode, the agent was placed in a random square. At each time step, it could choose to move up, down, left or right. There was also a chance that it would be blown onto a random neighbouring square after each move. The agent's goal was to reach one of the two goal positions without falling off a cliff.

By defining error states as those in which the agent occupies a cliff square, they showed that RSRL was able to minimise the time required to reach a goal square such that the probability of falling was less than 13%. It accomplished this by learning a greedy policy that preferred the top right goal position over the riskier lower right goal.

The main advantage of this algorithm is that a user can specify a probability of failure ω for a controller. The nature of the algorithm means that the resulting policy will adhere to that constraint in practice. However, this approach will take longer than Q-learning to find a policy since it is solving the full RL problem for multiple values of ξ. Also, the learner must be allowed to violate the risk constraint during learning in order to find a suitable policy.

2.3.3 Multiple Criteria Adaptive Real-Time Dynamic Programming

Gábor et al. propose a similar solution to these multi-criteria MDPs in [***2***]. Where Geibel and Wysotzki address this issue by defining in a weighted linear combination of a risk function with a value function, Gábor el al. define an ordering between value functions. Their algorithm, Multiple Criteria Adaptive Real-Time Dynamic Programming (MC-ARTDP) assumes that reward is given in the form of a reward vector. This vector includes a set of ordered reward signals. This is in contrast to RSRL which can only handle two signals. MC-ARTDP's goal is to find an action policy that maximises return on the last reward element such that the expected return for each of the other reward components stays below a specified value. It accomplishes this goal in a manner very similar to RSRL. It learns separate value functions for each reward signal using Bellman update equations at each time step.

The primary difference between MC-ARTDP and RSRL is in their definition of the greedy action. In RSRL, the greedy action is the action with the highest weighted linear combination of value and risk, Q_{ξ}^{π}. MC-ARTDP, on the other hand defines a reverse 2nd lexigraphic ordering between the value functions. The reward vector is first ordered such that the signals that must be minimised come first and are placed in their order of importance. The signal to be maximised is then placed at the end of the vector. The ordering then considers an action's value with respect to the first reward signal. The action with the lowest value for this first component is the greedy action. If two actions have the same value, the value of the next component is compared. This continues until the last reward component is compared. If multiple actions have the same values for all but the last reward component, the action with the highest last component value is declared the greedy action.

Armed with this procedure for comparing actions, MC-ARTDP uses the following Bellman update equations for each component value function:

For $i \in [1 \dots n-1]$ $$Q_i(s,a) \leftarrow \min\left[Q_i(s,a), r_i + \gamma \operatorname{m.a.x.}_{a'} Q_i(s',a')\right] \quad (28)$$

$$Q_n(s,a) \leftarrow Q_n(s,a) + \alpha\left[r_n + \gamma \operatorname{m.a.x.}_{a'} Q_n(s',a') - Q_n(s,a)\right] \quad (29)$$

Where $\operatorname{m.a.x.}$ chooses the greedy action with respect to the reverse 2nd lexigraphic ordering discussed above and min is the normal minimum operator. Likewise, n is the number of component reward signals. Armed with these Bellman equations, MC-ARTDP is able to accomplish its goal by expressing the constraints as the first $n-1$ component reward signals in the same way RSRL defines its error signal. However, its requirement that constraints be ordered means that it will have trouble with MDPs in which the first constraint cannot be satisfied. In that situation, it might not gracefully recover by finding a solution that satisfies the other $n-2$ constraints.

2.3.4 Lyapunov Functions and Reinforcement Learning

Perkins and Barto take a different approach to safety in adaptive controllers. They point out in [***26***] that if every action available to an RL agent is intrinsically safe, not only will the optimal policy that the algorithm converges to be safe, but the agent's actions will be safe throughout the learning process. To do this, they use the concept of Lyapunov functions from control theory. A Lyapunov function is essentially a scalar function of the state of a set of differential equations describing a system. This scalar function must be monotonically decreasing over time. The existence of a Lyapunov function shows that the underlying system is stable and will approach some steady state as time progresses.

Perkins and Barto extend the idea of the Lyapunov function from systems described by sets of differential equations to the full RL problem defined on MDPs. To do so, they first define a number of desirable safety and liveness properties for an MDP. If S is the set of states in the MDP, then $T \subset S$ is defined as a set of safe states. Then the following properties are defined in relation to set T.

Property 1 - *Remain in T* – If the agent's state at time zero, $s_0 \in T$, then $\forall s_i \in S$ where $i \in [1,2,3 \dots], s_i \in T$ with probability 1.

In other words, *Remain in T* specifies that if the agent starts in a safe state, the agent is guaranteed to only enter safe states through any action in the future. This can be seen as a safety property.

Property 2 - *Reach T* – $\exists i \in [0,1,2,3 \dots]$ such that $s_i \in T$ with probability 1.

This is a formulation of a liveness property. It identifies that no matter what state the agent starts in, it will eventually reach a safe state.

Property 3 - *Reach and Remain in T* – $\exists \mathbb{i} \in [0,1,2,3 \dots]$ such that $\forall i \geq \mathbb{i}$ where $s_i \in S, s_i \in T$ with probability 1.

The third property combines the first two properties defining both safety and liveness for the MDP. However, it may not always be possible to achieve. Therefore, they provide a somewhat less restrictive property which allows the agent to leave the set of safe states as long as it always returns again:

Property 4 - *Infinite Time in T* – $\sum_{i=0}^{\infty} \varphi_T(s_i) = \infty$ with probability 1 where $\varphi_T(s_i) = 1$ if $s_i \in T$, otherwise $\varphi_T(s_i) = 0$.

The last property makes use of a function $\delta_T : S \rightarrow \mathbb{R}^+$ which describes the distance of a state to the set of safe states. The only requirements of this function are that $\delta_T(s) = 0 \; if \; s \in T$ and $\delta_T(s) > 0$ otherwise. This property, *Converge to T*, says that the agent's state will converge to T as time progresses.

Property 5 - *Converge to T* - $\lim_{i \rightarrow \infty} \delta_T(s_i) = 0$ with probability 1.

Two theorems on Lyapunov functions are then used to design controllers that maintain some or all of these properties. The first theorem assumes function $L : S \rightarrow \mathbb{R}$ is positive on all T^c, s' is the state resulting when an agent in state s performs action $a \in A(s)$, and Δ is greater than zero. The first theorem presented in [***26***] is:

Theorem 1 – If $\forall s \notin T, \forall a \in A(s)$, $\forall s'$ where s' is a possible next state of s given action a, and $L(s) - L(s') < \Delta$ whenever $s' \notin T$, then the agent will enter T in at most $\left\lceil \frac{L(s_i)}{\Delta} \right\rceil$ time steps from any starting state s_i.

If a set of actions and a Lyapunov function L can be defined which satisfy Theorem 1 for a given MDP with a set of goal states G, then the agent satisfies the *Reach* property with $G = T$ and the *Remain* property with $T = \{s: L(s) \leq L(s_0)\}$. Depending upon L, this may provide the necessary safety and liveness requirements.

However, such a function L might not always exist, especially in stochastic systems. In such systems, it may still be possible to prove that the agent will eventually *Reach* T. The first theorem presented in [***26***] is:

Theorem 2 – If L is bounded above by $U \in \mathbb{R}$ and on each time step the agent either agent chooses an action that which results in state $s' \in T$ with probability at least p or $L(s) - L(s') \geq \Delta$ with probability at least q, then the agent eventually reaches T with probability 1. Also, the probability that the agent will not reach T in at least n time steps is bounded above by a function decaying to zero exponentially in n.

This theorem can be used to establish the *Reach* property. However, there is no upper bound on how many time steps it will take to reach T.

The application of these theorems is highly dependent on the problem. As such, it may not be possible to prove these properties in any given domain. However, they present a starting point for designing safe actions for use in reinforcement learning algorithms. If they can be proven, then the need to use formal verification to ensure system safety can be mitigated.

2.3.5 Function Approximation in Reinforcement Learning

The main drawback with temporal difference reinforcement learning methods is the use of tables to store the current value for each possible state-action pair [***23***]. This results in a problem known as the curse of dimensionality. The space required to store each state-action value grows in $O(nm)$ where n is the number of states and m is the number of actions in the MDP. This format also ignores any dependencies between different states. In most MDPs, many states will have similar dynamics. Storing and updating each of these states separately wastes time and space.

To solve this problem, function approximation can be used to replace value lookup tables. Numerous function approximation methods have been proposed including neural networks, linear regression, kernel methods, and even regression trees [***23***]. Neural networks in particular have received a lot of interest partially due to early success applying them to complex problem spaces such as Tesauro's master level backgammon agent, TD-Gammon [***27***]. However, linear function approximation, particularly using neural networks, quadratic regression, and locally weighted regression, does not guarantee convergence [***28***]. In fact, cyclical errors in the approximation of nearby states can cause the algorithm to diverge to poor solutions. Despite this, function approximation is still often used to allow RL algorithms to solve complex MDPs which would otherwise be too large to compute with lookup table value functions.

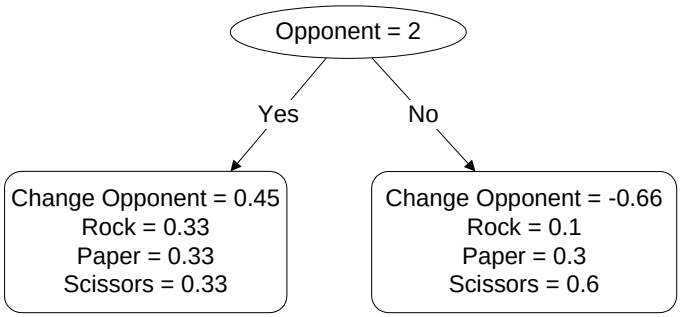


Figure 6 - Regression Tree for the Dual Opponent, Rock-Paper-Scissors Game

2.3.5.1 Regression Tree State Space Representations

Regression trees have also been proposed as a method of approximating the value functions used in temporal difference RL algorithms [***29***]. These regression trees are a type of decision tree which aggregates states with similar dynamics into the leaves of a tree. Each node of this tree represents a test against the agent's current state. Each branch represents a possible outcome from that test. The leaves of the tree contain real numerical values corresponding to the Q-value for the state-action pair. This regression tree is essentially an approximation of the true value function which partitions the state table into groups with similar values. While regression trees still do not guarantee convergence in temporal difference algorithms, they do have one major advantage over other function approximation techniques. They are human readable. Thus, an agent's behaviour can easily be predicted by inspecting the regression tree.

An example regression tree is shown in Figure 6 above for the rock, paper, scissors MDP discussed in section 2.3.1.1. In this example, the agent faces one of two opponents, one which prefers to play paper and another which plays uniformly. In this problem, the agent's regression tree tells the agent to switch opponents when it is currently facing the uniform opponent. Otherwise, it plays scissors.

In order to learn regression trees, a method known as Top-Down Induction of Decision Trees (TDIDT) is used [***30***]. TDIDT is a divide and conquer approach that builds decision trees recursively. A set of examples is presented to the algorithm. Each example contains a set of data characteristics and a value to predict. This value can be a class for classification problems or a real number for regression. Then, the algorithm considers all the possible tests that can be done on those examples to try and predict the output values. The test that produces the best split of the examples according to some measure of their performance is placed in the current node as the test. The examples are split according to the test and fed down the branches of the tree where the algorithm will run again at each child node. TDIDT continues splitting the examples with test nodes until there is no test that can adequately split them.

Many different tests have been proposed to check whether a node should be split or not. These include information gain [***30***] and the t-statistic [***29***] amongst others. Information gain measures the drop in entropy resulting from a particular split. Likewise, entropy measures how dissimilar a group of examples is. With information gain, whenever an example is added to a leaf, the number of examples in that leaf is checked to see if it is above a minimum threshold. If it is, then the information gain of each test that can be used to split those examples is calculated. The test with the highest gain is then chosen as the best split. Information gain is calculated as follows [***30***]. If S is a set of examples which is partitioned into sets S_i by test Q, then the $Gain(S, Q)$ is:

$$Gain(S, Q) = Entropy(S) - \sum_{S_i \; partitions \; of \; S \; by \; Q} \frac{|S_i|}{|S|} Entropy(S_i) \qquad (30)$$

Where $Entropy(S)$ is:

$$Entropy(S) = \sum_{j=1}^{\#\ of\ Classes} -\frac{\#\ of\ examples\ in\ class\ j}{\#\ of\ examples} \log_2 \frac{\#\ of\ examples\ in\ class\ j}{\#\ of\ examples} \quad (31)$$

2.3.5.2 G-Learning

One issue with using information gain to split regression trees is that it requires the tree to keep copies of the examples filtered into each node. Pyeatt and Howe's version of the G-Learning algorithm circumvents this issue by calculating t-statistic of each split in the place of information gain [***3***]. Their version was based on the G-Learning algorithm proposed by Chapman and Kaelbling in [***29***] which applies regression tree function approximation to the Q-Learning algorithm for MDPs where all state attributes are binary. Pyeatt and Howe removed the restriction that the attributes must be binary. When their algorithm first starts, a tree with a single leaf containing a Q value for each possible action is initialised. The normal Q-Learning algorithm from Figure 4 above is then run. However, when a leaf's Q value is updated in each iteration, the change applied to that Q value, ΔQ, is calculated. This ΔQ value is used to update a set of statistics in that leaf. These statistics include the number of times a Q value in the leaf has been updated as well as the mean and variance of the ΔQ values for all updates. These statistics are maintained both for each action and separately for each possible test that may be used to split the leaf. By updating these statistics iteratively, copies of each example do not need to be kept in the tree. Then for any action, if the number of updates is above some minimum threshold and the absolute value of the mean ΔQ is less than double the standard deviation, then the leaf is split. Formally, the following test is used to determine if a node is split:

$$\text{If } n \geq min \text{ and } |\mu| < 2\sigma \text{ then split} \quad (32)$$

Where n is the number of times the action's Q value has been updated, min is the minimum threshold, μ is the mean of the ΔQ values for that action, and σ is the standard deviation. If this test determines that the leaf should be split, then the t-statistic for all possible node tests is calculated. The test with the highest t-statistic is used to replace the leaf. Two new child leaves are then created with the same Q-values from the old leaf. However, they are created with empty statistics.

Welch's t-statistic is used to test whether or not two random variables with different underlying standard deviations have significantly different means [***31***]. In this version of the t-test, samples are drawn from two random variables. The number of samples drawn can be different for each variable. The average, $\bar{X}_1$ and $\bar{X}_2$, and unbiased sample variance, s_1^2 and s_2^2, of each group of samples in calculated. The t-statistic is then calculated as:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{s_{\bar{X}_1 - \bar{X}_2}} \quad (33)$$

If n_i is the size of sample i, $s_{\bar{X}_1 - \bar{X}_2}$ is defined as:

$$s_{\bar{X}_1 - \bar{X}_2} = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \quad (34)$$

In a normal Welch's t-test, this t-statistic is then compared to the t-distribution to determine the probability that the random variables have different means. However, G-Learning stops at calculating the t-statistic for splits resulting from each possible node test. The split with the highest t-statistic is chosen as the best.

In [***3***], Pyeatt and Howe empirically compared their version of G-Learning with regression tree function approximation using different split criterion including: information gain, Gini index, and the twoing rule. The Gini index is a measure of the likelihood that a set of examples is misclassified by a proposed split. Likewise, the twoing rule is similar to entropy in that it examines the spread of categories resulting on either side of a split [***32***]. Pyeatt and Howe also compared their algorithm to neural network approximation using back-propagation [***3***]. They found that when compared on multiple MDPs of varying complexity, the t-statistic method used by G-Learning achieved consistently higher overall returns with lower standard deviations than the other approximation methods. It also required consistently less time to converge.

2.3.6 Reinforcement Learning in HRI

Reinforcement learning has proven to be a popular and effective technique for incorporating adaptation into HRI. Many recent studies employ RL algorithms to optimize a robot's interaction policy around human behaviour. Others employ RL algorithms to adapt a robot's behaviour to individual preferences.

2.3.6.1 Learning Multimodal Dialogue Strategies

Recently, a team of researchers in the Interactive Systems Lab at the Universität Karlsruhe have developed a robotics platform called ARMAR III to study HRI [***21***]. ARMAR III shares many similarities with BERT2. ARMAR III is a humanoid robotic torso with fully articulated arms and hands and a number of sensors designed for interacting with humans including: head-pose tracking, pointing recognition, facial identification, and speech recognition. However, ARMAR III is mounted on a mobile platform and does not have access to a motion capture system for highly accurate positional information.

One of the experiments conducted on this platform involved using RL to learn a dialog strategy for interacting with a human in a simple bartender scenario [***21***] [***9***]. In this setting, ARMAR sits behind a bar with a variety of objects of different shapes and colours. A customer walks up to the bar with the intention of receiving one of the objects. The robot's task is to ask the customer a series of questions to figure out which object the customer wants. The series of questions asked by the robot both verbally and non-verbally through pointing is its dialogue strategy.

In order to ensure short and accurate interactions, Q-learning is used to optimise the dialogue strategy. However since Q-learning requires a significant amount of training experience to converge, models of the user's reactions and the robot's sensor errors are used to generate simulated experience. The user is modelled as a simple bigram probability in which the probability that the user gives a particular response (or gesture) only depends upon the last action performed by the robot. While this glosses over much of what influences a user's response, it allows the user model to be learned quickly with only a small amount of data.

A separate error model is used for each sensor system. These models depend highly on the exact system. However, they each model errors at a very high level. For instance, the speech recognition error model gives the probability that a user's intended meaning will be mistaken. Likewise, the pointing recognition error model gives the probability that a user's gesture will not be seen, the probability the intended pointing direction will be mistaken, and the standard deviation in direction error.

Both of these models are learned from a small set of sample data from an initial Wizard-of-Oz study. The goal of a Wizard-of-Oz study is to identify how a user will interact with a system before it has been developed. In this type of study, a series of users are told to interact with what is presumably a fully developed system. However, the system is really only a dummy system interface. Behind the dummy interface, a human expert manually guides the system responses to the users. This provides valuable information to system designers as to how real users will use the system before starting the development process. However, for this experiment, the Wizard-of-Oz study data is used to train the user and error

models. By using the relatively small amount of example data provided by the study to train user error models instead of directly applying Q-learning, the robot is able to learn robust dialogue strategies from relatively little data.

Experimental results in [***9***] show that when compared to hand crafted dialogue strategies, this technique identifies the correct object 6% more often with 20% shorter dialogues when trained on just 47 sample Wizard-of-Oz dialogues. However, this algorithm is not able to adapt to changing user preferences online since learning is only done on the simulated experience. In order to adapt the system based on new user data, the user and error models are updated and the system is retrained on new simulated experience.

2.3.6.2 Policy Gradient Reinforcement Learning Based HRI

Another team of researchers have recently studied ways to adapt user-robot interaction preferences to improve people's comfort level with a robot [***22***]. They identified three components to HRI that affect a user's comfort and which vary from person to person: the amount of eye contact, the length of time between the robot speaking and acting, and the speed at which the robot acts. Their goal was to rapidly adapt the robot's behaviour in these three areas while interacting with a person.

To accomplish this, they developed a method for measuring how comfortable a user is. This method assumes that the more uncomfortable a person, the more they move around and avert their eyes. So, they defined a reward function which simply returns a linear combination of the amount of time the user meets the robot's gaze and how much they move in a given time step.

This reward function is then used by policy gradient reinforcement learning to adapt the robot's behaviour parameters while interacting with the user. Policy gradient reinforcement learning (PGRL) is a form of reinforcement learning which does not rely on MDPs. Instead, PGRL directly adapts a vector of agent parameters in the direction of the highest amount of reward. It does so by first creating a number of random perturbations of the parameter vector. Each of these vectors is then evaluated. Then, the gradient of each parameter towards the nearest local reward optimum is calculated via partial derivative. Each of these parameter gradients is then added to the current parameter vector to move the agent's performance towards the highest amount of reward.

Experiments on this technique returned mixed results. A number of users were asked to interact with the robot for 30 minutes. PGRL was able to adapt the robot's behaviour within 15 minutes for most people. However, some users expressed discomfort with the robot's adapted behaviour. This was probably due to a failure of the reward function. These users tended to stare at the robot more when they became uncomfortable thereby confusing the reward function.

While this technique has shown that it is possible to quickly adapt a robot's behaviour to user preferences, it highlights a dangerous pitfall that must be avoided by HRI adaption techniques. Namely, any technique used to gauge a user's emotional state must take into account that not all users react in the same fashion, especially when this is used to guide the robot's adaptations.

3 METHODS

While multiple constraint algorithms like MC-ARTDP have been shown to perform well on toy problems and in simulations, they have not been used to address safety and liveness with live robots. In fact, it remains to be seen whether the safety and liveness properties required in HRI can even be handled as multiple constraints in RL. This project attempts to answer this question by identifying a method of translating stochastic safety and liveness properties into constraints for multiple constraint RL. It then compares the performance of Q-learning, MC-ARTDP, and a novel algorithm based on MC-ARTDP. This algorithm is designed to maximise reward while constraining risk like MC-ARTDP. However, it also tries to maintain constraints throughout exploration by ensuring that the expected risk of actions chosen during exploration is also constrained. These algorithms are compared both on a toy problem and on a HRI scenario where real users interact with a drink serving robot.

3.1 Safe and Live Reinforcement Learning

Safe and Live Reinforcement Learning (SLRL) is a multiple constraint RL algorithm based on MC-ARTDP. However, SLRL uses a different definition of greediness which is theorised to better handle problems with multiple safety and liveness constraints. It also includes a stochastic exploration policy which maintains the expected risk of chosen actions below the specified acceptable levels.

SLRL's goal is twofold. First, SLRL attempts to find a policy which maximises its expected return such that a set of safety and liveness properties are maintained. Secondly, SLRL attempts to maintain those safety and liveness properties whilst exploring the MDP for the maximal policy.

3.1.1 Safety and Liveness Properties

Before the method SLRL uses to accomplish its goals can be discussed, the formulation of the safety and liveness properties must be considered. Oftentimes in safety critical projects, safety and liveness properties are identified as part of the system's design specification. This design specification lists the tasks the system must accomplish along with the properties it must maintain. It is often written before early in the project lifecycle and used as an input to the design of the system. As such, the necessary safety and liveness properties will often be known before an RL algorithm is decided upon as the system's optimisation technique.

Safety properties describe events that should not occur [***1***]. For this project, they will take this form:

[S] – Event X should not occur more than Y% of the time.

Liveness properties, on the other hand, describe events that should eventually occur [***1***]:

[L] – If prerequisites Y are met, event X should eventually occur Z% of the time.

In order to exploit these properties in its policy search, they are first expressed as constraints similar to those used in MC-ARTDP and RSRL. To do so, a set of error states, $\Phi^{i} \subset S$, in which each property i is violated must be identified. This can be done trivially for safety properties. The set of error states for a safety property is simply the set of states in which the proscribed event occurs. The set of error states for generic safety property **[S]** above is defined by:

$$\Phi^{[S]} = \{s : s \in S \; and \; X \; is \; true \; in \; s\} \tag{35}$$

Error states for liveness properties, however, are not so trivially defined. States in which those events occur can be easily identified, and it is possible to identify when a policy *does* reach those states. However, it is difficult to show that a policy *does not* eventually reach them. Thus, liveness as defined above does not directly fit into the multiple constraint format used by RSRL and MC-ARTDP. However,

the fact that properties are expressible in logic can be exploited to solve this problem. One possible logical representation of general property [L] is:

$$Y \rightarrow \exists t, such\ that\ s_t \in S\ and\ X \in s_t\ with\ probability\ Z \tag{36}$$

This can be seen as a generalisation of Perkins and Barto's *Reach* property for Lyapunov style safety in RL which adds a precondition Y and a probability Z [***26***]. This liveness property is then transformed into a safety property with the following transformation:

$$Y \rightarrow \exists t \geq i, s_t \in S \wedge X \in s_t\ with\ probability\ Z \tag{37}$$

$$\neg Y \vee (\exists t \geq i, s_t \in S \wedge X \in s_t)\ with\ probability\ Z \tag{38}$$

$$\neg\neg\big(\neg Y \vee (\exists t \geq i, s_t \in S \wedge X \in s_t)\big)\ with\ probability\ Z \tag{39}$$

$$\neg\big(Y \wedge \neg(\exists t \geq i, s_t \in S \wedge X \in s_t)\big)\ with\ probability\ Z \tag{40}$$

$$\neg\big(Y \wedge (\forall t \geq i, s_t \notin S\ \vee X \notin s_t)\big)\ with\ probability\ Z \tag{41}$$

Where i is the current time step. The result of equation (41) above is that liveness property **[L]** equivalent to the following safety property:

[S'] – It is not the case that condition Y is met and event X does not eventually occur Z% of the time.

However, **[S']** is still infeasible since it lacks a time bound. To solve this issue, a technique is borrowed from bounded model checking [***6***]. In model checking, a logical model of a system is checked against a property such as **[S']**. Because it is only possible to model a finite number of time sequences, model checking often cannot show that **[S']** is never satisfied. Instead, a reasonable time bound is used to cut off the search. If the property has not been satisfied by that point, bounded model checking reasonable concludes that the model does not satisfy the property.

Likewise, SLRL requires that a reasonable time bound be placed on each of these unbounded properties. Adding a time bound to equation (41) results in the following mathematical safety property:

$$\neg\big(Y \wedge (\forall t \in [i \ldots i+W], s_t \notin S\ \vee X \notin s_t)\big)\ with\ probability\ Z \tag{42}$$

Where i is the current time step. This is equivalent to:

[S"] – It is not the case that condition Y is met and event X does not occur within W time steps Z% of the time.

With this version of the liveness property, there is one last issue. Most MDPs do not include the current time step as part of the state information. However, without it, a set of error states for **[S"]** cannot be defined. Therefore, timing information relevant to the liveness properties must be included in the MDP. Then, a set of error states $\Phi^{[S"]}$ can be defined for generic property **[S"]** as:

$$\Phi^{[S"]} = \begin{Bmatrix} s : s \in S\ and\ Y\ is\ true\ in\ s\ and \\ X\ is\ not\ true\ in\ s\ and\ s.time\ step = W \end{Bmatrix} \tag{43}$$

Once a set of error states for each property has been identified, a separate reward signal for each property can be defined using the same techniques from MC-ARTDP and RSRL. Error signal $\bar{r}^i$ for property i is defined as one when the agent enters an error state and zero otherwise:

$$\bar{r}^i_{s,a}(s') = \begin{cases} 1 : if\ s' \in \Phi^{\mathrm{i}} \\ 0 : otherwise \end{cases} \tag{44}$$

The reward vector, $\hat{r}$, for each time step is then defined as the vector of the reward and error signals at a single time step:

$$\hat{r}_{s,a}(s') = \langle r_{s,a}(s), \forall i\, \bar{r}^i_{s,a}(s') \rangle \tag{45}$$

As with RSRL, SLRL then defines the risk of each state for constraint i, $\rho^\pi_i(s)$, as the probability of entering a state from set Φ^{i}:

$$\rho^\pi_i(s) = \Pr\left(\exists j, s_j \in \Phi^{\mathrm{i}} \mid s = s_0\right) \tag{46}$$

The risk of constraint i can then be calculated as the discounted expected cost using a discount factor $\bar{\gamma}$:

$$\rho^\pi_{i,\bar{\gamma}}(s) = \mathrm{E}\left[\sum_{j=0}^{\infty} \bar{\gamma}^j\, \bar{r}^i_j \,\middle|\, s = s_0\right] \tag{47}$$

This is then used to define a state-action risk function $\bar{Q}_i$ for each constraint i:

$$\begin{aligned} \bar{Q}^\pi_i(s,a) &= \mathrm{E}\left[\bar{r}^i_0 + \bar{\gamma}\rho^\pi_{i,\bar{\gamma}}(s_1) \middle| s_0 = s, a_0 = a\right] \\ &= \sum_{s'} p_{s,a}(s') \left(\bar{r}^i_{s,a}(s') + \bar{\gamma}\rho^\pi_{i,\bar{\gamma}}(s')\right) \end{aligned} \tag{48}$$

Based on these risk and value functions, the goal of SLRL is to find a policy that maximises expected return such that risk of each constraint is less than the specified acceptable level:

$$\begin{aligned} &\max_\pi V^\pi \\ &such\ that\ \forall i, s \in T, \rho^\pi_i(s) \leq\ \omega_i \end{aligned} \tag{49}$$

The acceptable level of risk, ω_i, for constraint *i* is defined in the corresponding safety property. Likewise, T, is the set of states in which the constraints must be satisfied. Minimally, T must include all possible starting states, S_0. However, depending upon requirements, it can include any and all non error states:

$$S_0 \subseteq T \subseteq \left\{S - \bigcup\nolimits_i \Phi^{\mathrm{i}}\right\} \tag{50}$$

3.1.2 The Greedy Policy

Before Bellman update equations can be identified for the value and risk functions Q and Q_i, the greedy policy must be defined. In order to do so, methods for comparing reward vectors and for comparing policies are required. Where MC-ARTDP defines a reverse 2nd lexigraphical ordering between reward vectors, SLRL defines a simple, two-step ordering. SLRL first compares the number of error signals constraints that are at or below their acceptable levels. Since the error signals are binary values, this is equivalent to summing the error signals. If the error sum for two reward vectors is the same, SLRL then compares the reward signal. Therefore, reward vector $\hat{r}$ is less than or equal to ($\preccurlyeq$) vector $\hat{t}$, $\hat{r} \preccurlyeq \hat{t}$, if:

Either,
$$\sum\nolimits_i \bar{r}^i > \sum\nolimits_i \bar{t}^i \tag{51}$$

Or,
$$\sum\nolimits_i \bar{r}^i = \sum\nolimits_i \bar{t}^i \text{ and } r \leq t \tag{52}$$

The relation $\preccurlyeq$ is also extended to vectors of value and risk functions. This allows policies to be compared based on their expected return and risks. Since the risk functions are not binary like the error

signals, a simple sum is no longer applicable. Therefore, relation $\preccurlyeq$ is redefined for vectors of value and risk functions. If $\hat{Q}^1$ and $\hat{Q}^2$ are vectors of value and risk functions, then $\hat{Q}^1 \preccurlyeq \hat{Q}^2$ if:

Either,
$$\sum_i \delta\left(\bar{Q}_i^1 \leq \omega_i\right) > \sum_i \delta\left(\bar{Q}_i^2 \leq \omega_i\right) \tag{53}$$

Or,
$$\sum_i \delta\left(\bar{Q}_i^1 \leq \omega_i\right) = \sum_i \delta\left(\bar{Q}_i^2 \leq \omega_i\right) \text{ and } Q^2 \leq Q^2 \tag{54}$$

Where δ is the indicator function defined as:

$$\delta(Condition) = \begin{cases} 1 : \text{if } Condition \text{ is true} \\ 0 : \text{otherwise} \end{cases} \tag{55}$$

With this operator, a vector of value and risk functions is optimal (denoted $\hat{Q}^*$), if $\forall \hat{Q}^i \in \hat{\mathcal{Q}}$, the space of all value vectors:

$$\hat{Q}^i \preccurlyeq \hat{Q}^* \tag{56}$$

However, $\hat{Q}^*$ is not necessarily unique. All $\hat{Q}^*$ will have the same Q^* value and the same number of unviolated risks. Though, the $\hat{Q}^*$ may have varying values of each $\bar{Q}_i^*$.

Likewise, there are potentially multiple optimal policies, π^*. An optimal policy in this context is a policy whose value vector in each state is greater or equal to the value vector of all other policies. Thus, $\forall \pi \in \Pi$, the space of all action policies, and $\forall s \in S$:

$$\hat{V}^{\pi}(s) \preccurlyeq \hat{V}^{\pi^*}(s) \tag{57}$$

Similarly, a greedy policy simply chooses the action with the greatest reward vector with respect to $\preccurlyeq$ at each time step. The greedy policy, π', is defined for each $s \in S$:

$$\pi'(s) = \arg \operatorname*{m.a.x.}_{a \in \mathcal{A}} \hat{Q}(s, a) \tag{58}$$

Where $\mathcal{A}$ is the set of possible actions and m. a. x. compares value vectors using the $\preccurlyeq$ relation defined in equations (53) and (54) above.

3.1.3 Bellman Update Equations

Using the definition of the greedy policy given above, Bellman update equations can be defined. These Bellman equations allow an SLRL agent to incrementally learn the value and risk functions with experience. Just like MC-ARTDP and RSRL, they use the same form as Q-learning from equation (14) above. The only difference is that the max operator is replaced with the m. a. x. operator from equations (53) and (54).

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \operatorname*{m.a.x.}_{a'} Q(s', a') - Q(s, a) \right] \tag{59}$$

$$\bar{Q}_i(s, a) \leftarrow \bar{Q}_i(s, a) + \alpha \left[\bar{r}_i + \bar{\gamma} \operatorname*{m.a.x.}_{a'} \bar{Q}_i(s', a') - \bar{Q}_i(s, a) \right] \tag{60}$$

Because the greedy policy is defined according to the m. a. x. operator similar to MC-ARTDP, SLRL does not require a separate value function to linearly combine the value and risk functions like RSRL. This allows SLRL to be easily used with multiple constraints.

Initialise $Q(s,a)$ and $\bar{Q}_i(s,a)$ arbitrarily

Repeat for each episode
    Initialise s
    Repeat for each step of the episode
        $a \leftarrow \epsilon SafeBoltzmann(s, Q, \langle \bar{Q}_i \rangle)$
        Take action a, observe r, $\langle \bar{r}_i \rangle$, s'
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \text{ m.a.x.}_{a'} Q(s',a') - Q(s,a)]$
        For each constraint i
            $\bar{Q}_i(s,a) \leftarrow \bar{Q}_i(s,a) + \alpha[\bar{r}_i + \bar{\gamma} \text{ m.a.x.}_{a'} \bar{Q}_i(s',a') - \bar{Q}_i(s,a)]$
        $s \leftarrow s'$
    Until s is terminal

Figure 7 - The SLRL Algorithm

The algorithm for SLRL is given in Figure 7 above. The exploration policy, $\epsilon SafeBoltzmann(s)$, is described in the following section.

SLRL's primary problem is that it is fairly costly to run from a memory standpoint. Adding risk functions for each constraint compounds the curse of dimensionality. It requires $O(lnm)$ memory, where l is the number of states, n is the number of actions, and m is the number of constraints. This makes it only suitable for small to medium sized MDPs.

3.1.4 Safe ϵ-Boltzmann Exploration

In addition to finding an optimal policy with respect to $\preccurlyeq$, SLRL uses an exploration policy which attempts to maintain the constraints throughout the learning process. At each time step, safe ϵ-Boltzmann exploration ensures that, if possible, the expected risk per constraint of the action chosen non-uniformly at random is below each acceptable level, ω_i. This does not mean that only actions that satisfy the constraints are chosen. It only means that based on the agent's current knowledge, the probability of violating a risk is always reasonably low. Formally, for each constraint i, safe ϵ-Boltzmann exploration defines a probability distribution over the possible actions such that in state s, and all possible actions a:

$$\mathrm{E}[\bar{Q}_i^{\pi}(s,a)] \leq \omega_i \tag{61}$$

This is based on the current estimates for $\bar{Q}_i^{\pi}$.

In order to accomplish this, the set of possible actions, $\mathcal{A}$, is partitioned into two sets: actions that do not violate any constraints, $\mathcal{B}$, and actions that violate one or more constraints, $\mathcal{C}$. Set $\mathcal{B}$ is defined as:

$$\mathcal{B} = \{a | a \in \mathcal{A} \; and \; \forall \omega_i, \bar{Q}_i(s,a) \leq \omega_i\} \tag{62}$$

Set $\mathcal{C}$ can then be defined according to set $\mathcal{B}$:

$$\mathcal{C} = \{a | a \in \mathcal{A} \; and \; a \notin \mathcal{B}\} \tag{63}$$

If neither set is empty, safe ϵ-Boltzmann exploration calculates a threshold value, ϵ, at each time step. This threshold gives the maximum combined probability of choosing an action from set $\mathcal{C}$. The total probability of selecting an action from set $\mathcal{C}$ is then constrained such that if $\pi(s,a)$ is the probability of selecting action a in state s:

$$\sum_{a \in \mathcal{C}} \pi(s,a) \leq \epsilon \tag{64}$$

By intelligently constraining the probability of choosing actions from set $\mathcal{C}$, equation (61) above can be satisfied whenever $|\mathcal{B}| \geq 1$. The following transformations on equation (61) show how this is possible:

Linearity of Expectation,

$$(1-\epsilon_i)\mathrm{E}_{\mathcal{B}}[\bar{Q}_i^{\pi}(s,a)] + \epsilon_i \mathrm{E}_{\mathcal{C}}[\bar{Q}_i^{\pi}(s,a)] \leq \omega_i \tag{65}$$

$$\mathrm{E}_{\mathcal{B}}[\bar{Q}_i^{\pi}(s,a)] - \epsilon_i \mathrm{E}_{\mathcal{B}}[\bar{Q}_i^{\pi}(s,a)] + \epsilon_i \mathrm{E}_{\mathcal{C}}[\bar{Q}_i^{\pi}(s,a)] \leq \omega_i \tag{66}$$

$$\epsilon_i(\mathrm{E}_{\mathcal{C}}[\bar{Q}_i^{\pi}(s,a)] - \mathrm{E}_{\mathcal{B}}[\bar{Q}_i^{\pi}(s,a)]) \leq \omega_i - \mathrm{E}_{\mathcal{B}}[\bar{Q}_i^{\pi}(s,a)] \tag{67}$$

$$\epsilon_i \leq \frac{\omega_i - \mathrm{E}_{\mathcal{B}}[\bar{Q}_i^{\pi}(s,a)]}{\mathrm{E}_{\mathcal{C}}\left[\bar{Q}_i^{\pi}(s,a)\right] - \mathrm{E}_{\mathcal{B}}\left[\bar{Q}_i^{\pi}(s,a)\right]} \tag{68}$$

$$\epsilon_i \leq \frac{\omega_i - \sum_{a\in\mathcal{B}} p_{\mathcal{B}}(s,a)\bar{Q}_i^{\pi}(s,a)}{\sum_{a\in\mathcal{C}} p_{\mathcal{C}}(s,a)\bar{Q}_i^{\pi}(s,a) - \sum_{a\in\mathcal{B}} p_{\mathcal{B}}(s,a)\bar{Q}_i^{\pi}(s,a)} \tag{69}$$

Where ϵ_i is the probability threshold required to satisfy constraint i and $p_{\mathcal{D}}(s,a)$ is the probability of choosing action a assuming an action from set $\mathcal{D}$ is being chosen. This is shorthand notation for the following conditional probability where a_t is the action chosen at time t and s_t is the agent's state at time t:

$$p_{\mathcal{D}}(s,a) = \Pr\left(a_t = a | s_t = s, a \in \mathcal{D}\right) \tag{70}$$

Therefore as long as ϵ_i satisfies equation (69) for every constraint and set $\mathcal{B}$ is not empty, equation (61) will always be met according to the agent's current estimate of the risk functions. However, equation (69) does not guarantee that ϵ_i will be in the range $[0,1]$ and hence not a probability. Since $\mathcal{C}$ contains the actions that violate *any* constraint, it is possible that $\mathrm{E}_{\mathcal{C}}[\bar{Q}_i^{\pi}(s,a)] < \mathrm{E}_{\mathcal{B}}[\bar{Q}_i^{\pi}(s,a)]$ for some constraint i. By definition, $\mathrm{E}_{\mathcal{B}}[\bar{Q}_i^{\pi}(s,a)] \leq \omega_i$, so $\epsilon_i < 0$ when $\mathrm{E}_{\mathcal{C}}[\bar{Q}_i^{\pi}(s,a)] < \mathrm{E}_{\mathcal{B}}[\bar{Q}_i^{\pi}(s,a)] \leq \omega_i$. However, for those $\epsilon_i < 0$, equation (61) is trivially satisfied since $\mathrm{E}_{\mathcal{C}}[\bar{Q}_i^{\pi}(s,a)] < \omega_i$.

So in order to calculate ϵ at each time step, the agent partitions the actions into sets $\mathcal{B}$ and $\mathcal{C}$ according to equations (62) and (63). If set $\mathcal{B}$ is empty, ϵ is trivially set to one and equation (61) is not satisfied. Otherwise, if set $\mathcal{C}$ is empty, ϵ is trivially set to zero and equation (61) is satisfied. If neither set is empty, then ϵ must be calculated.

First, the probabilities of choosing actions within each set $p_{\mathcal{B}}$ and $p_{\mathcal{C}}$ are chosen through some means. Then, equation (69) is used to calculate the maximum value ϵ_i can take for each constraint i. If any $\epsilon_i < 0$, it is trivially reset to one since it does not need to be constrained. The threshold ϵ is then set to the minimum of these ϵ_i values:

$$\epsilon = \min_i \epsilon_i \tag{71}$$

However, the probabilities $p_{\mathcal{B}}$ and $p_{\mathcal{C}}$ must be defined before the action selection policy is complete. The goal is to define a distribution that increases the probability of selecting higher value actions. At the same time, the distribution should decrease the probability selecting risky actions. The distribution should also decrease the probability of selecting suboptimal actions as time progresses. To accomplish these goals, a Boltzmann distribution has been modified to take advantage of the knowledge of each action's risk. This distribution begins by replacing the temperature parameter, τ, with an action dependent temperature, τ_a:

$$p_{\mathcal{D}}(s,a) = \frac{e^{Q(s,a)/\tau_a}}{\sum_{a\prime\in\mathcal{D}(s)} e^{Q(s,a\prime)/\tau_{a\prime}}} \tag{72}$$

Where $a \in \mathcal{D}$ and $\mathcal{D} \subseteq \mathcal{A}$. The action dependent temperature, τ_a, increases proportionately to the amount of violated risk for each constraint and inversely proportionately to the current episode number. It is defined as:

$$\tau_a = \frac{k + \sum_{i=1}^{n} \max\left(0, \frac{\bar{Q}_i^{\pi}(s,a) - \omega_i}{\omega_i}\right)}{\sqrt{e}} \tag{73}$$

Where k is a positive, non-zero constant, e is the current episode number, and n is the number of constraints. The quantity $\bar{Q}_i^{\pi}(s,a) - \omega_i$ is the amount of unacceptable risk for constraint i. It is positive when action a is riskier than constraint i allows. By dividing this quantity by ω_i, constraints with a low acceptable level of risk are weighted more than those with a high acceptable level. The max of this quantity and zero is then taken to ensure that only unacceptable risk is taken into account. This ensures that constraints whose risk is below the acceptable level cannot be used as a buffer for riskier constraints when they are all summed together. Finally, this sum is added to k and divided by $\sqrt{e}$ to slowly decrease the temperature values over time. The constant k is used to artificially influence the temperature values based on experimental results.

The effect of this modification is that actions with a higher total unacceptable risk have higher temperature values and thus a lower selection probability. On the other hand, high value actions have a higher selection probability. These two factors interact to allow the agent to choose risky, high value actions in the name of exploration, as well as choose low value, low risk actions for their safety. At the same time, k and $\sqrt{e}$ interact to make the distribution less and less uniform over time.

Figure 8 and Figure 9 below illustrate the effect on the numerator of equation (72) of varying value and risk for actions with $\omega = 0.1$ and $\omega = 0.3$ respectively. For each of these distributions, $k = 1$ and $e = 1$ and only a single risk constraint is used. These graphs show that when an action's risk, $\bar{Q}$, is at or below the threshold, ω, the relative likelihood of selecting that action increases uniformly with value, Q. When $\bar{Q}$ is greater than ω, relative likelihood decreases exponentially in $\bar{Q}$ and increases linearly in Q. In addition, the rate of exponential decrease in $\bar{Q}$ is greater when ω is smaller.

By calculating the probabilities $p_{\mathcal{B}}$ and $p_{\mathcal{C}}$ independently according to this modified Boltzmann distribution and choosing between the action sets $\mathcal{B}$ and $\mathcal{C}$ according to ϵ, safe ϵ-Boltzmann selection maintains the constraints according to the agent's current knowledge throughout exploration. Admittedly, the quality of the maintenance of these constraints is only as good as the current estimates of the risk functions. The constraints will not actually be met until the risk functions, $\bar{Q}_i$, begin to approach the real probabilities of each reaching each set of error states.

The full algorithm for safe ϵ-Boltzmann selection is given in Figure 10 below. In addition to the storage required for the value and risk functions Q and $\bar{Q}$, this algorithm requires $O(n)$ memory where n is the number of actions. It selects actions in $O(nm)$ time, where m is the number of constraints. Compared to the $O(n)$ time used by Boltzmann selection, safe ϵ-Boltzmann selection is relatively slow. However, for any reasonable MDP, the number of constraints should be relatively small. Thus, the time difference between these two algorithms should not be too costly.

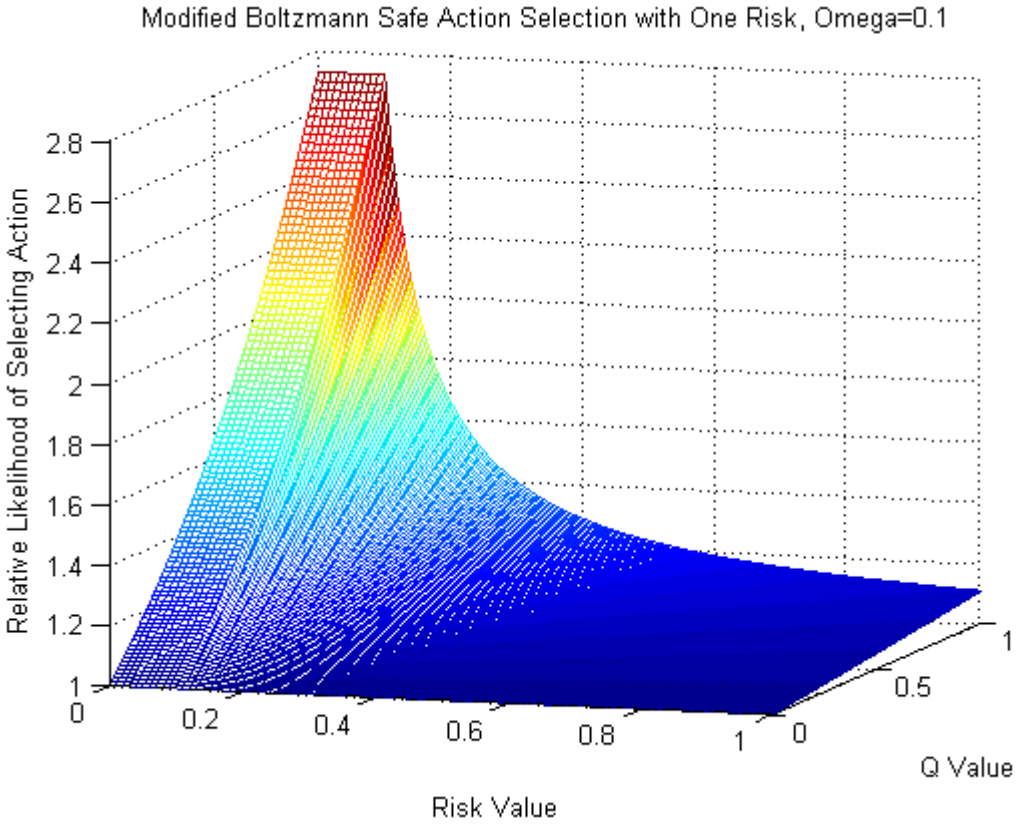


Figure 8 - Safe ϵ-Boltzmann Action Selection with One Risk, Omega=0.1

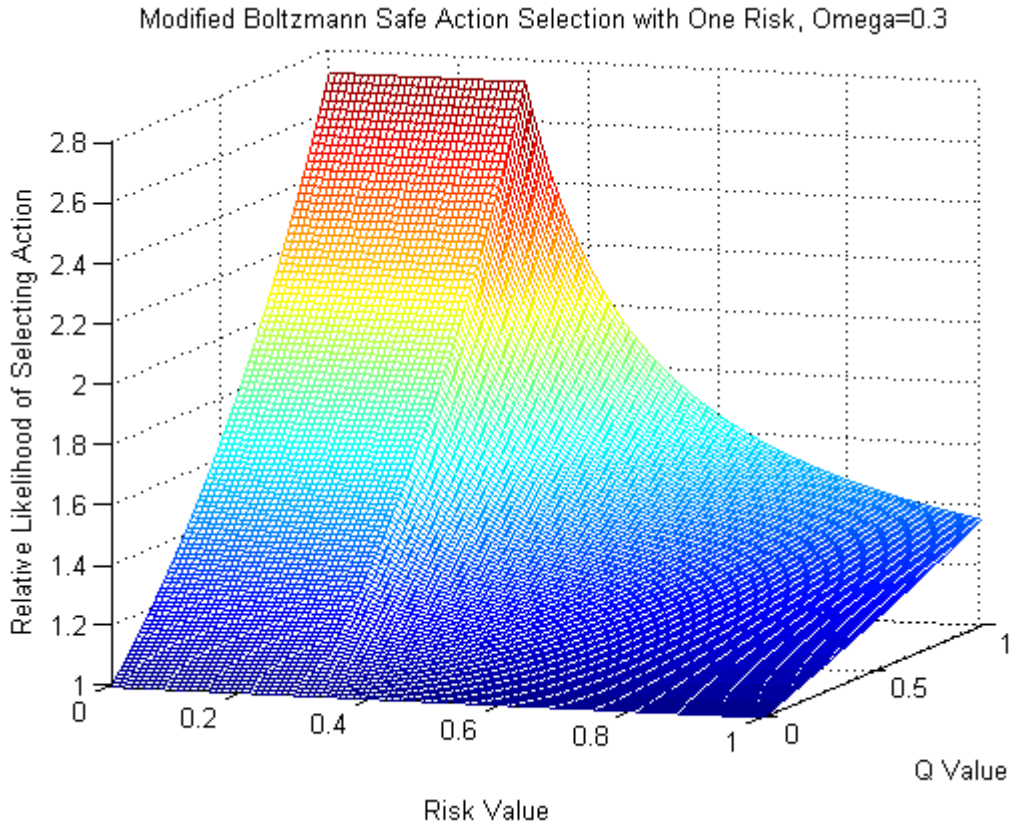


Figure 9 - Safe ϵ-Boltzmann Action Selection with One Risk, Omega=0.3

Initialise $Safe$ as an array of $true$ for each action
Initialise P to an array of zeros for each action
$safeSum \leftarrow 0$
$unsafeSum \leftarrow 0$
$\epsilon \leftarrow 1$
$threshold \leftarrow 0$
$r \leftarrow$random number between 0 and 1

Repeat for each action $a \in \mathcal{A}$
    $\tau_a \leftarrow k$
    Repeat for each constraint i
        If $\bar{Q}_i > \omega_i$
            $Safe(a) \leftarrow false$
            $\tau_a \leftarrow \tau_a + \frac{\bar{Q}_i^{\pi}(s,a) - \omega_i}{\omega_i}$
    $\tau_a \leftarrow \tau_a / \sqrt{e}$
    $P(a) \leftarrow e^{Q(s,a)/\tau_a}$
    If $Safe(a)$
        $safeSum \leftarrow P(a)$
    Else
        $unsafeSum \leftarrow P(a)$

Repeat for each constraint i
    $safeRisk \leftarrow 0$
    $unsafeRisk \leftarrow 0$
    Repeat for each action $a \in \mathcal{A}$
        If $Safe(a)$
            $safeRisk \leftarrow safeRisk + \frac{P(a)\bar{Q}_i}{safeSum}$
        Else
            $unsafeRisk \leftarrow unsafeRisk + \frac{P(a)\bar{Q}_i}{unsafeSum}$
    $\epsilon_i \leftarrow \frac{\omega_i - safeRisk}{unsafeRisk - safeRisk}$
    If $\epsilon_i \geq 0$ and $\epsilon_i < \epsilon$
        $\epsilon \leftarrow \epsilon_i$

If $safeSum > 0$ and $\epsilon > \frac{unsafeSum}{safeSum + unsafeSum}$
    $\epsilon \leftarrow \frac{unsafeSum}{safeSum + unsafeSum}$

Repeat for each action $a \in \mathcal{A}$
    If $Safe(a)$
        $threshold \leftarrow threshold + \frac{\epsilon P(a)}{unsafeSum}$
    Else
        $threshold \leftarrow threshold + \frac{(1-\epsilon)P(a)}{safeSum}$
    If $r < threshold$
Return a

Figure 10 - The Safe ϵ-Boltzmann Action Selection Algorithm

3.2 Safe and Live G-Learning

The primary issue with SLRL is that it relies on a table format for the value and risk functions. This results in high memory requirements and makes it unsuitable for large problems. Safe and Live G-Learning (SLGL) attempts to solve this issue through function approximation. In SLGL, the value and risk tables are replaced with regression trees. A separate tree is used for each value and risk function since each will have different dynamics across the MDP. Regression trees were chosen for two reasons. They significantly compress the value and risk tables by combining similar state-action pairs into the tree's leaves. They are also human readable. Readability is important for SLGL since it means resulting policies can be inspected to ensure that agents are not learning undesirable behaviour. Other than the use of regression trees, SLGL is functionally equivalent to SLRL. The same methods for defining constraints and error signals from safety and liveness properties are used. SLGL also uses safe ϵ-Boltzmann selection. Therefore, the algorithm in Figure 10 applies to SLGL as well as SLRL.

SLGL uses Pyeatt and Howe's modification to G-Learning [***3***] to build a regression tree for each value and risk function. This regression tree algorithm was chosen since their work showed that its t-statistic approach achieved consistently high rewards with very small standard deviations when compared to table formats, neural networks, and regression trees based on information gain, Gini index, and the twoing rule in multiple problem domains.

3.3 Grid World Toy Problem

The first step in evaluating safety and liveness properties in RL algorithms is to compare their performance in a contrived toy problem. For this experiment, Geibel and Wysotzki's grid world problem from [***7***] was adapted to include liveness as well as safety.

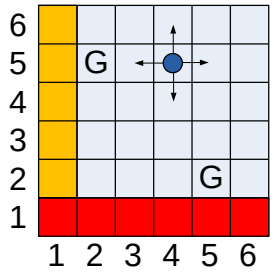
3.3.1 Algorithms

The purpose of this toy problem is to compare the ability of Q-learning, MC-ARTDP, and SLRL to maintain stochastic safety and liveness properties both during and after learning. Q-learning and MC-ARTDP are run with both ϵ-greedy and Boltzmann exploration policies. These two algorithms provide a performance baseline for comparing the novel algorithm, SLRL. Q-learning is used to illustrate the limitations of a single value function formulation. MC-ARTDP is used to show the performance of a standard multiple constraint algorithm when used with liveness properties. All three of these algorithms use lookup table value functions. Therefore, G-learning and SLGL will be used to compare decision tree function approximation methods.

3.3.2 Problem Description

In the modified problem shown in Figure 11, there is a six by six grid in which the agent can move around to reach one of two goal positions. Each goal position sits next to a cliff. The first cliff consists of the leftmost column, while the second cliff occupies the bottommost row. If the agent enters either of these cliffs, it loses the game. The agent starts in a random square, and at each time step, the agent can choose one of five actions: move left, right, up, or down, or wait in the current square. This last action is essential to winning the game. Unlike Geibel and Wysotzki's grid world, the agent must not only reach a

Figure 11- The Modified Grid World Problem



[33]

goal position, but it must wait there for a random number of turns. This requirement to wait was added to mirror the need to wait for human responses in HRI tasks. It is also this waiting aspect of the game that allows useful tests to be performed with liveness constraints.

Similar to Geibel and Wysotzki's grid world, there is a 10% chance that the agent will be blown into a random neighbouring square each turn. Because of this, the agent must learn to avoid the squares adjacent to the cliffs so as not to lose the game unintentionally. This aspect of the game allows for the testing of safety constraints.

3.3.3 Safety and Liveness Properties

In order to test the ability of the various algorithms, the following safety and liveness constraints were defined:

[S1] – The agent should not fall off of the left cliff in more than 5% of all episodes.
[S2] – The agent should not fall off of the bottom cliff in more than 10% of all episodes.
[L1] – Within 3 consecutive rounds of waiting, the agent either wins or moves to another square in 95% of all episodes.

The two safety properties above, **[S1]** and **[S2]**, constrain the agent from moving near either cliff. However, it must risk one of the cliffs in order to win due to the positions of the goals. However, **[S2]** specifies a higher allowance for falling than does **[S1]**. So, the agent should learn to prefer the bottom right goal position over the top left one.

The liveness constraint, **[L1]**, says that the agent should not wait in any position forever. This tries to ensure that the agent will not become unresponsive. However, it is not in a form which can be exploited as an error signal. Therefore, it is redefined as a safety property which specifies that it should not be the case that **[L1]** is not satisfied:

[L1] → **[S3] –** The agent should not wait for 4 consecutive turns in more than 5% of all episodes.

Logically, **[S3]** is equivalent to **[L1]**. However, it is in a form that will be easier to handle by the learning algorithms.

3.3.4 State Space Size

Each algorithm tested on this problem uses the same state space information. At each time step, agents have access to their (x,y) coordinate and the number of consecutive wait actions they have executed. This last piece of information is needed for two reasons. First, the problem would not satisfy the Markov property without it. The agent must wait at the goal position at least once before it can win. So, the probability of receiving a reward depends on the number of times the agent has waited. Secondly, the agent cannot learn to satisfy property **[S3]** without knowing how many consecutive waits it is executing.

For the purposes of defining a vector of state attributes, the number of consecutive wait actions is capped at nine for ten possible values. The agent is not prevented from executing more wait actions than this. It will simply be presented as nine for the state vector. With a six by six grid and ten possible wait values, there are 360 possible states in this MDP.

3.3.5 Reward and Error Functions

This problem makes use of two different formats for the reward function. The first format combines the error and reward signals into a single function for the Q-learning and G-Learning agents. The second format presents multiple error and reward signals for the multiple constraint algorithms. However, both formats make use of the reward and error signals defined for the safety and liveness properties.

3.3.5.1 Reward Signal

The reward signal for this problem is defined as one when the agent wins the game and zero at all other times. Since the agent must wait in a goal square at least once to win, the set of goal states, G, is defined as:

$$G = \left\{ \begin{matrix} s : s \in S \; and \; (s.x = 2 \; and \; s.y = 5 \; and \; s.wait > 0 \\ or \; s.x = 5 \; and \; s.y = 2 \; and \; s.wait > 0) \end{matrix} \right\} \tag{74}$$

Where S is the set of states. However, the reward signal is stochastic. The agent only wins in a goal state 25% of the time. Therefore, the reward signal is defined as:

If $s' \in G$

$$\Pr\left(r_{s,a}(s') = 1\right) = 0.25 \tag{75}$$

$$\Pr\left(r_{s,a}(s') = 0\right) = 0.75 \tag{76}$$

If $s' \notin G$

$$\Pr\left(r_{s,a}(s') = 0\right) = 1.0 \tag{77}$$

3.3.5.2 Error Signals

For each of the three safety properties, an error signal is defined. These error signals are one when the corresponding property is violated and zero otherwise. However, in order to define them mathematically, the set of error states for each property must first be identified. The set of error state for each property i, Φ^i, is defined below:

$$\Phi^{[S1]} = \{s : s \in S \; and \; s.x = 1\} \tag{78}$$

$$\Phi^{[S2]} = \{s : s \in S \; and \; s.y = 1\} \tag{79}$$

$$\Phi^{[S3]} = \{s : s \in S \; and \; s.wait = 4\} \tag{80}$$

Then, the error signal for each property i, $\bar{r}^i$, is defined as:

$$\bar{r}^i_{s,a}(s') = \left\{ \begin{matrix} 1 : if \; s' \in \Phi^{\mathrm{i}} \\ 0 : otherwise \end{matrix} \right. \tag{81}$$

3.3.5.3 Composite Reward Function

The reward signal format used for the Q-learning and G-Learning agents gives the agent a reward of one when the game is won, a reward of negative one for each safety or liveness property violated, and zero reward otherwise. This is just a sum of the reward and error signals at each time step. So, the composite reward function, r', is:

$$r'_{s,a}(s') = r_{s,a}(s') + \bar{r}^{[S1]}_{s,a}(s') + \bar{r}^{[S2]}_{s,a}(s') + \bar{r}^{[S3]}_{s,a}(s') \tag{82}$$

This gives a composite reward value in the range $[-3,1]$.

3.3.5.4 Multiple Constraint Reward Vector

Instead of using a composite signal like Q-learning, MC-ARTDP and SLRL use a vector of reward and error signals as a reward function. This reward vector, $\hat{r}$, is defined as:

$$\hat{r}_{s,a}(s') = \langle r_{s,a}(s'), \bar{r}^{[S1]}_{s,a}(s'), \bar{r}^{[S2]}_{s,a}(s'), \bar{r}^{[S3]}_{s,a}(s') \rangle \tag{83}$$

3.4 HRI Experimental Setup

The second stage of this project applies these algorithms to a human-robot interaction experiment. The purpose of this experiment is to determine the ability of each algorithm to adhere to specified safety and liveness properties both during and after learning in a real environment. While evaluating algorithms against toy problems can be a useful initial analysis tool, no toy problem can truly emulate the unpredictable nature of real users. Thus, a HRI scenario was developed to test the ability of these techniques to maintain constraints while interacting with a real person.

3.4.1 Algorithms

Similar to the toy problem, Q-learning and SLRL are used to compare table format algorithms in an HRI scenario. G-learning and SLGL are also used to compare decision tree function approximation methods for multiple constraint problems. MC-ARTDP is not tested on this MDP. Q-learning and G-learning are run with Boltzmann action selection. SLRL and SLGL both use the modified safe ϵ-Boltzmann action selection discussed in Section 3.1.4.

3.4.2 Experiment Description

For this experiment, BERT2 sits in front of a table with two objects representing different drinks, water and coffee. When a user approaches the table, BERT2 asks whether he or she would like either drink. At this point the RL algorithm takes over and has to execute a sequence of high level actions to grab the requested drink, offer it to the person, and release it when the person grabs it. The robot's goal is complete a successful handover of the right object without dropping it.

The learning algorithm is allowed to choose from a list of high level actions:

- Move hand to the water on the table
- Move hand to the coffee on the table
- Move hand to the offer location
- Grab whatever is at the current location
- Release what the robot is holding
- Ask the person if they have a hold of the object
- Wait in the current position

In order to choose the right action policy, the robot has access to the following state information:

- Which object has been requested
- The location of BERT2's hand
- The location of the user's hand
- Whether BERT2's hand is open or closed.
- What, if anything, the robot is holding
- How much pressure is currently being applied to the robot's hand
- How the user has responded to any questions asked
- Whether or not BERT2 has offered the cup to the user yet
- If the cup has not been offered, the amount of time since the start of the episode
- If it has been offered, the amount of time since the cup was offered

When an item is either successfully handed over or dropped, the user is asked if they would like another drink and the system is reset.

3.4.3 Safety and Liveness Properties

In this scenario, there is a danger that the robot will drop the drink. If the water is dropped, the damage to the user is not too great. However, if the robot drops the hot coffee, there is a risk of minor injury to the

human. Therefore, the robot must adhere to a few safety properties designed to limit the risk of injury. These are based on the safety properties studied by Grigore et al. in [***4***].

The following safety properties have been defined for this scenario:

[S1] – The cup should not be released if the person's hand is not near the cup in more than 5% of all episodes.
[S2] – The water cup should not be dropped in more than 10% of all episodes.
[S3] – The coffee cup should not be dropped in more than 5% of all episodes.

[S1] ensures that the robot does not randomly drop cups. **[S2]** and **[S3]** dictate the acceptable level of risk associated with the water and coffee cups.

The following liveness properties have been specified to ensure the robot's responsiveness. They are also based on the liveness properties examined by Grigore et al. in [***4***].

[L1] – If the user grabs the water cup, the robot eventually releases it in 95% of all episodes.
[L2] – If the user grabs the coffee cup and responds to the robot's questions, the robot eventually releases it in 95% of all episodes.
[L3] – Within 10 seconds of offering the cup to the user, the robot eithers release the cup or asks the user for more information in 95% of all episodes.
[L4] – The robot offers the cup to the user within 30 seconds of receiving a drink order in 95% of all episodes.

[L1] and **[L2]** dictate the sufficient conditions associated for releasing each cup. They define the actions the user must perform in order for the robot to release the cup. **[L3]** and **[L4]** bound the amount of time the robot can spend performing actions. They ensure that the robot does not sit in a waiting position forever. However, these liveness properties must be expressed as safety properties just like the liveness property in the grid world problem.

[L1] → [S4] – It should not be the case that the robot has not released the water cup 5 seconds after the user grabs it in more than 5% of all episodes.
[L2] → [S5] – It should not be the case that the robot has not released the coffee cup 5 seconds after the user grabs it and responds to the robot's questions in more than 5% of all episodes.
[L3] → [S6] – It should not be the case that the robot has neither released the cup nor asked the user for more information after 10 seconds of offering the cup to the user in 5% of all episodes.
[L4] → [S7] – It should not be the case that the robot has not offered the cup to the user within 30 seconds of receiving a drink order in 5% of all episodes.

Because the liveness properties do not include time bounds, they have to be modified slightly when converted to safety properties. Time bounds must be added. Otherwise, the property with never actually be violated as time progresses to infinity.

3.4.4 State Space Size

The state information listed in the problem description is discretised for use by the RL algorithms. As such, each state attribute has the following ranges:

- *Requested drink* – 2 values: water and coffee.
- *BERT's hand location* – 4 values: at the side, table 1 (water cup), table 2 (coffee cup), or at the offer cup position.
- *User's hand location* – 2 values: at the offer position, elsewhere.

- *BERT's hand status* – 2 values: open or closed.
- *Item held* – 3 values: water, coffee, and none.
- *Hand pressure* – 32 values: 0 through 31.
- *User responses* – 6 values: yes, no, hot, cold, none, or not asked.
- *Offered cup yet* – 2 values: yes or no.
- *Time elapsed* – 60 values: 0 through 59 half second time steps
- *Goal status* – 5 values: unfinished, successful transfer of the right cup, successful transfer of the wrong cup, dropped the water, dropped the coffee.

With this state space description, the MDP contains 11,059,200 states. This is a fairly large MDP. However, it is not so large that it cannot be solved using a lookup table value function representation. On the other hand, function approximation would significantly reduce the MDP's memory requirements. For this reason, the experiment examines both table based and function approximation methods.

3.4.5 Reward and Error Functions

Similar to the grid world toy problem, this MDP uses both a composite reward/error function for Q-learning and G-learning as well as a vector of reward/error functions for MC-ARTDP, SLRL, and SLGL. Both of these formats make use of the same reward and error signals.

3.4.5.1 Reward Signal

The reward signal is defined as one when BERT2 successfully transfers the requested drink to the user and zero at all other times. However, the robot cannot attain this reward by itself. It can only offer the drink to the user and wait. It must rely on the user to grab the cup. Without the user's cooperation, the robot will not receive any reward. Because of this, the agent must learn to wait in the offer position in order to reach the goal states. The goal states, G, are defined as:

$$G = \begin{Bmatrix} s : s \in S \; and \\ s.status = successful\; transfer\; of\; correct\; item \end{Bmatrix} \tag{84}$$

Where S is the set of all states in the MDP. Because the definition of G only includes states in which the user has grabbed the cup, the reward signal can be deterministic:

$$r_{s,a}(s') = \begin{cases} 1 : if\; s' \in G \\ 0 : otherwise \end{cases} \tag{85}$$

3.4.5.2 Error Signals

For each of the seven safety properties, an error signal is defined. These error signals are one when the corresponding property is violated and zero otherwise. However, in order to define them mathematically, the set of error states for each property must first be identified. The set of error state for each property i, Φ^i, is defined below:

$$\Phi^{[S1]} = \begin{Bmatrix} s : s \in S \; and \\ s.user\; hand\; location \neq s.robot\; hand\; location \end{Bmatrix} \tag{86}$$

$$\Phi^{[S2]} = \{s : s \in S \; and \; s.status = dropped\; water\} \tag{87}$$

$$\Phi^{[S3]} = \{s : s \in S \; and \; s.status = dropped\; coffee\} \tag{88}$$

$$\Phi^{[S4]} = \begin{Bmatrix} s : s \in S \; and \; s.offered \; and \; s.time = 10 \\ and \; s.hand\; closed \; and \; s.pressure > 0 \\ and \; s.user\; hand\; location = s.robot\; hand\; location \end{Bmatrix} \tag{89}$$

$$\Phi^{[S5]} = \begin{Bmatrix} s : s \in S \text{ and } s.offered \text{ and } s.time = 10 \\ \text{and } s.hand\ closed \text{ and } s.pressure > 0 \\ \text{and } s.user\ hand\ location = s.robot\ hand\ location \\ \text{and } s.response \neq none \end{Bmatrix} \quad (90)$$

$$\Phi^{[S6]} = \begin{Bmatrix} s : s \in S \text{ and } s.offered \text{ and } s.time = 20 \\ \text{and } s.reponse = not\ asked \\ \text{and } s.status = unfinished \end{Bmatrix} \quad (91)$$

$$\Phi^{[S7]} = \{s : s \in S \text{ and not } s.offered \text{ and } s.time = 59\} \quad (92)$$

Then, the error signal for each property i, $\bar{r}^i$, is defined as:

$$\bar{r}_{s,a}^{[S1]}(s') = \begin{cases} 1 : if\ s' \in \Phi^{[S1]} and\ a = Release \\ 0 : otherwise \end{cases} \quad (93)$$

$$\bar{r}_{s,a}^{[S2]}(s') = \begin{cases} 1 : if\ s' \in \Phi^{[S2]} \\ 0 : otherwise \end{cases} \quad (94)$$

$$\bar{r}_{s,a}^{[S3]}(s') = \begin{cases} 1 : if\ s' \in \Phi^{[S3]} \\ 0 : otherwise \end{cases} \quad (95)$$

$$\bar{r}_{s,a}^{[S4]}(s') = \begin{cases} 1 : if\ s' \in \Phi^{[S4]}\ and\ a \neq Release \\ 0 : otherwise \end{cases} \quad (96)$$

$$\bar{r}_{s,a}^{[S5]}(s') = \begin{cases} 1 : if\ s' \in \Phi^{[S5]}\ and\ a \neq Release \\ 0 : otherwise \end{cases} \quad (97)$$

$$\bar{r}_{s,a}^{[S6]}(s') = \begin{cases} 1 : if\ s' \in \Phi^{[S6]} \\ 0 : otherwise \end{cases} \quad (98)$$

$$\bar{r}_{s,a}^{[S7]}(s') = \begin{cases} 1 : if\ s' \in \Phi^{[S7]} \\ 0 : otherwise \end{cases} \quad (99)$$

3.4.5.3 Composite Reward Function

The reward signal format used for the Q-learning and G-learning agents is simply a sum of the reward and error signals at each time step. So, the composite reward function, r', is:

$$r'_{s,a}(s') = r_{s,a}(s) + \sum_{i=1}^{7} \bar{r}_{s,a}^{[Si]}(s') \quad (100)$$

This gives a composite reward value in the range $[-7,1]$.

3.4.5.4 Multiple Constraint Reward Vector

Just like the multiple constraint reward vector used in the grid world problem, the reward vector, $\hat{r}$, is defined as:

$$\hat{r}_{s,a}(s') = \langle r_{s,a}(s), \bar{r}_{s,a}^{[S1]}(s'), \bar{r}_{s,a}^{[S2]}(s'), \bar{r}_{s,a}^{[S3]}(s'), \bar{r}_{s,a}^{[S4]}(s'), \bar{r}_{s,a}^{[S5]}(s'), \bar{r}_{s,a}^{[S6]}(s'), \bar{r}_{s,a}^{[S7]}(s') \rangle \quad (101)$$

3.4.6 Wizard-of-Oz Study

Before the experiment can be run with the various RL algorithms, a simulator must be trained to match the dynamics of the real environment. This simulator is then used to train the RL algorithms. The simulator is developed by using data from a Wizard-of-Oz experiment. Like the techniques used by Prommer in [***9***], the Wizard-of-Oz trial runs the experiment as described above. However instead of using a RL algorithm to choose actions, the experimenter controls BERT2 manually. As the Wizard-of-Oz trial

progresses, a real user interacts with the robot as in the real experiment. The user requests a drink. The robot grabs the drink and offers it to them. By controlling the robot manually, the experimenter can easily handle unforeseen user actions. For instance, users might try to grab drinks before the robot is ready to release them. Identifying these unexpected events can then be used to influence the design of the experiment and the robot's controller.

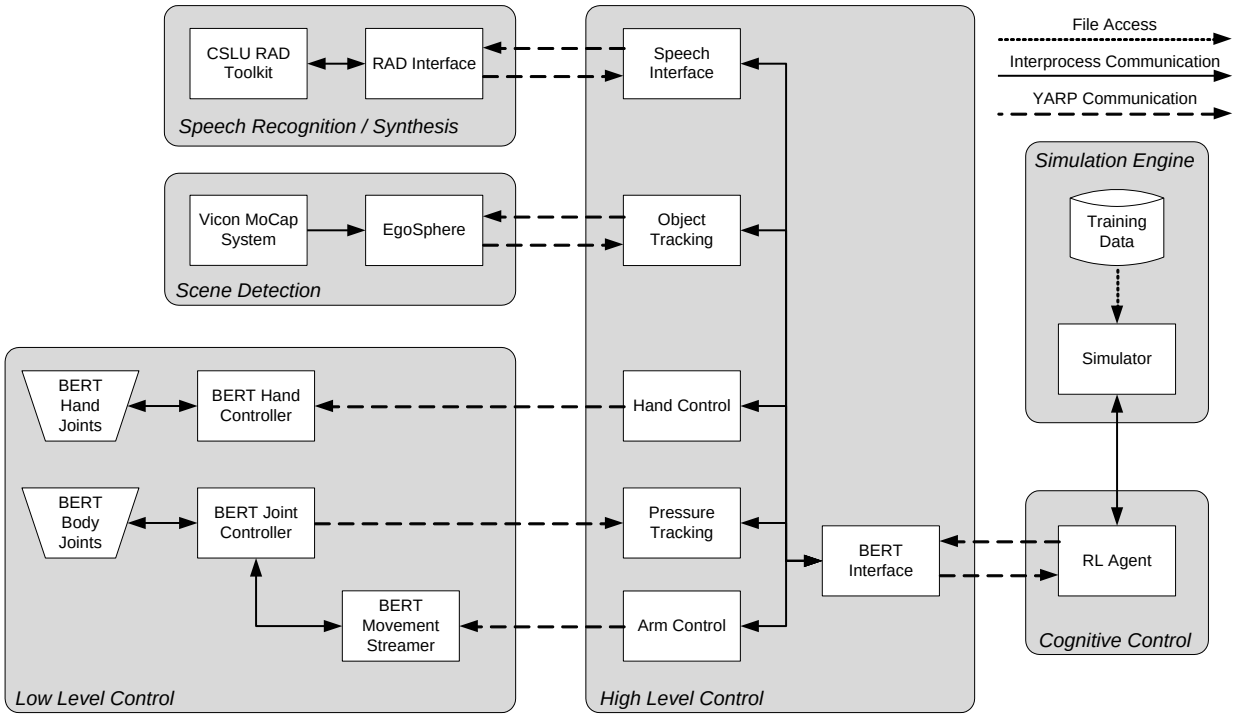
During the Wizard-of-Oz trials, all of the available sensory data is recorded along with the user's actions and the intended and actual robot actions. This data can then be used to build models of the user's responses to system actions, errors in the sensory systems' ability to accurately detect the environment, and errors in the robot's ability to perform requested actions. In general, these models use conditional probabilities to identify the probability of the robot detecting a specific state given the previous state and action. However, each of these models, the user model, the action error model, and the sensory error model handle different aspects of the resulting state. The user model is concerned with predicting user reactions based on the previous state and robot action. The action error model handles errors in the robot's ability to perform intended actions. Likewise, the sensory error model deals with the robot's inability to perfectly detect the real state of the environment. These models are also inspired by Prommer's work in [***9***].

3.5 HRI Experimental Controller Architecture

The control architecture for this project uses a modular design in order to take advantage of a number of existing sensing and control subsystems developed by researchers on the CHRIS project. Each subsystem performs a specific function and communicates with the other modules using YARP ports. The following modules are shown below in Figure 12.

- *Cognitive Control* – Houses the reinforcement learning algorithms responsible for choosing high level actions.
- *Simulation Engine* – Used to train the reinforcement learning algorithms in the Cognitive Controller. Models high level user responses to system actions as simple probabilities based on sample training data.
- *High Level Control* – Translates high level actions from the Cognitive Control module to low level actions. It also aggregates sensor information into a state vector for the Cognitive Control

Figure 12 - Controller Architecture



module.

- *Low Level Control*[3] – Translates high level motion commands from the High Level Control module to motor commands. It also provides information about the internal state of the robot.
- *Scene Detection*[4] – Uses the motion capture system to provide highly accurate positional data for all known objects in the environment (including the user and the robot).
- *Speech Recognition and Synthesis*[5] – Verbally prompts the user and translates responses into coarse values (i.e., yes, no, water, or coffee) for the High Level Control module.

3.5.1 Speech Recognition and Synthesis Module

In order to communicate with the user, the Speech Recognition and Synthesis module was programmed with a set of possible utterances and user responses. This was done with CSLU Toolkit RAD environment. This interactive dialog receives speech commands from the High Level Control module and relays back user responses via YARP.

The set of speech commands are:

- QUESTION_GOT_IT – Ask the user "Do you have it?"
- QUESTION_WHICH_DRINK – Ask the user "Would you like the hot or the cold drink?"
- OFFER – Say "Here you go."
- RESET – Reset the dialog for a new episode and ask the user "Would you like a drink?"

When asked a question, the user can give the following responses:

- RESPONSE_COFFEE – "Hot" or "Coffee"
- RESPONSE_WATER – "Cold" or "Water"
- RESPONSE_YES – "Yes"
- RESPONSE_NO – "No"

Figure 13 below illustrates the dialog implemented by this module[6]. When an episode is started, the RESET command is sent by the High Level Control module to the Speech module. This prompts BERT to ask the user which drink they would like. The module then waits for the user to respond with either coffee or water. This response is sent back to the High Level Controller. The module then waits for another speech command.

3.5.2 Scene Detection Module

The Scene Detection module is a pre-existing system developed by the CHRIS project for monitoring objects and users in the robot's vicinity. It includes the VICON Motion Capture system, the EgoSphere, and the Object Property Database. The motion capture system monitors the 3D location of objects and users tagged with reflective spheres with millimetre accuracy. These momentary locations are monitored for all known objects by the EgoSphere. This data is made available by YARP.

While this system can monitor any known object in the room, only the user's wrist was monitored for this project. To do this, the user wore a bracelet with the necessary reflective spheres on their left wrist. This allowed the High Level Control module to track the user's hand location at all times.

[3] The Low Level Control module is a set of controllers developed by Alex Lenz for the CHRIS project [***5***].
[4] The Scene Detection module was developed by Sergey Skachek for the CHRIS project [***5***].
[5] The Speech Recognition / Synthesis module's YARP and initialisation routines were developed by Sergey Skacheck for the CHRIS project [***5***]. However, the dialog used for this project was developed by myself based on Sergey's example dialogs.
[6] The initialisation steps in this dialog and the module's connection to YARP were developed by Sergey Skacheck as part of the CHRIS project [***5***]. The remaining section of the dialog was developed by myself.

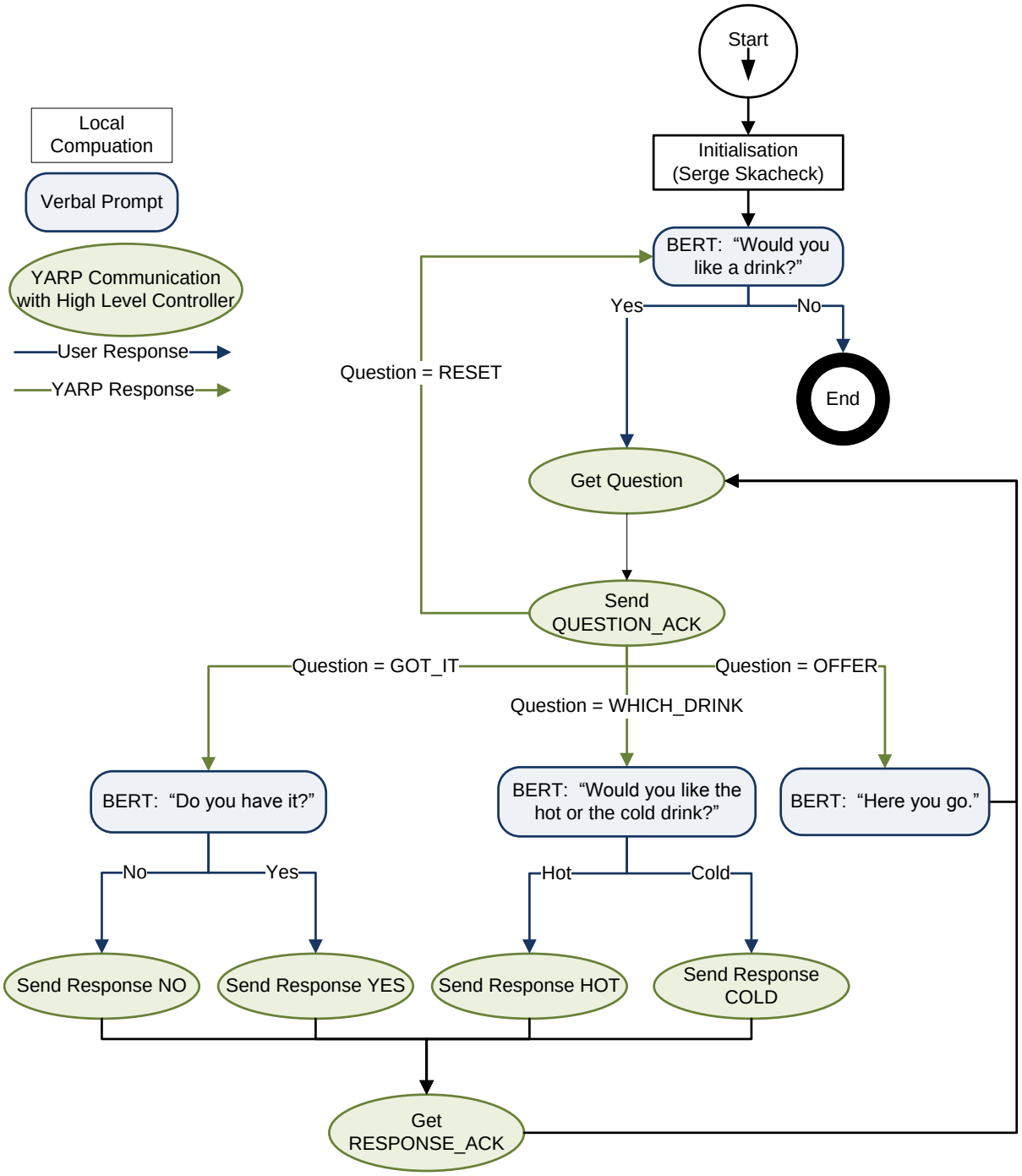


Figure 13 - Speech Dialog

3.5.3 Low Level Control Module

This low level controller contains three control subsystems shown in Figure 12 above[7]. The hand controller listens on a YARP port for high level control actions such as grasp and open hand. These high level commands are translated into actual motor commands and executed.

In a similar fashion, the joint controller listens on a YARP port for motion commands for all joints except those on the hand. However, these are low level joint velocity commands instead of high level actions like those provided by the hand controller. For this function, CHRIS project researchers provided a movement streaming controller that moves BERT2 between a set of predefined poses. This controller

[7] All subsystems in the Low Level Control Module were developed by Alex Lenz at the Bristol Robotics Laboratory as part of the CHRIS project [***5***].

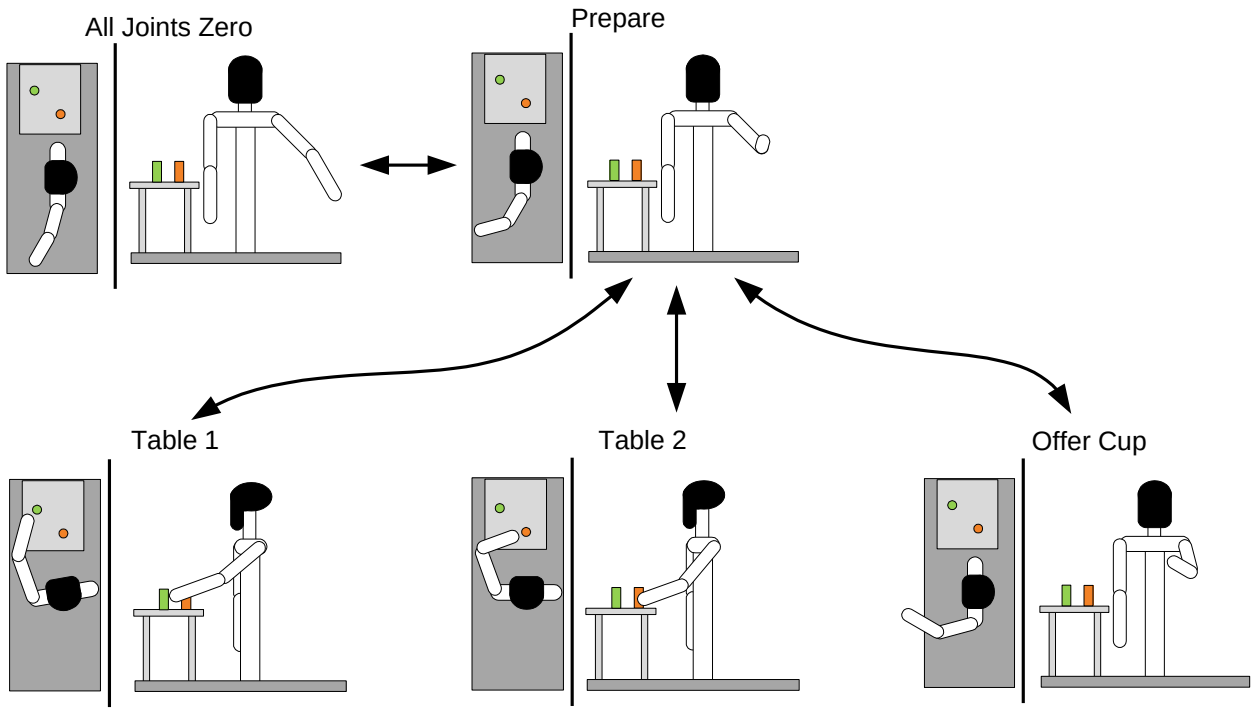


Figure 14 - BERT2 Motion Profiles

implements the motion profile state machine shown in Figure 14. Each state represents a pose the controller can move to. The edges show the possible motions between these poses.

When initialised, the robot enters the All Joints Zero pose. In this pose, every joint is positioned in the very middle of its range of motion. The controller then listens on a YARP port for pose transition commands. The only valid command in the All Joints Zero pose is to move to the Prepare to Grasp position. This is the robot's ready position. From there, it can be moved to and from the other three positions. The Table 1 position will move the arm to the water cup. In this position, a Grasp command sent to the hand controller will cause BERT2 to grab the water cup. Likewise, the Table 2 position moves the arm to the coffee cup. The Offer Cup position moves the arm to an outstretched position. These are the only five positions used by this project. However, other positions were required for similar projects being conducted at the same time. So to reduce the number of preset motions that had to be programmed by the CHRIS project researchers, all motions were constrained to pass through the Prepare to Grasp position.

The poses shown in Figure 14 were also chosen to display a high degree of spatial contrast as defined by Cakmak et al. in [***15***]. The Offer Cup position is easily distinguishable from the other positions. This allows users to easily identify the robot's intended actions. Likewise, forcing the robot to transition through Prepare to Grasp before progressing to Offer Cup increases the temporal contrast of the offer action. This allows users to better predict the timing of handoff tasks and increase fluency.

3.5.4 High Level Control Module

The High Level Control module interfaces between the Cognitive Controller and the Low Level Controller and various sensing modules. It translates the high level actions output by the RL algorithms in the Cognitive Controller to the action sequence commands required by the other modules. It also aggregates the available sensing information into the state vector required by the Cognitive Controller.

3.5.4.1 State Information Aggregation

One of the module's primary functions is to collect information from the available sensing modules and make it available to the Cognitive Controller in a format usable by the RL algorithms. In order to do this, the High Level Control module keeps track of the following information through each episode:

- Which cup the user asked for
- How the user responded to any questions via the Speech module
- The current (x,y,z) coordinate of the user's wrist via the Scene Detection module
- The current location of BERT2's hand based on the history of motion control commands
- The amount of pressure on the robot's hand via the calculation described in Section 3.5.4.3 below
- Which cup (if any) the robot is currently holding based on motion control commands
- Whether or not BERT2 has offered the cup to the user yet
- If the cup has not been offered, the amount of time since the start of the episode
- If it has been offered, the amount of time since the cup was offered
- The status of the current episode, whether it is unfinished, successfully completed, or one of the cups has been dropped.

This data is discretised and passed to the Cognitive Controller as a state vector. This state vector conforms to the specification given in Section 3.4.4 above and is written to a YARP port every tenth of a second.

3.5.4.2 Action Execution

The High Level Control module's other main function is to execute the actions requested by the Cognitive Controller. This includes not only signalling actions to the Low Level Controller and Speech module, but timing those actions so that the Cognitive Controller does not select another action before the last one is complete. To do this, the High Level Controller listens for action commands from the Cognitive Controller on a YARP port. When it receives an action command, an executing flag is set to true on the state vector that is periodically passed back to the Cognitive Controller. This signals to the Cognitive Controller that the last action is still being executed.

The High Level Control module can execute the following action commands from the Cognitive Controller:

- *Wait* – Wait for half a second
- *Grasp* – If the robot's hand is currently open, send a Grasp command to the hand controller to grab whatever is at the current location
- *Release* – If the robot's hand is currently closed, send an open command to the hand controller.
- *Move to Table 1* – Send the correct series of motion profile commands to the Low Level Controller to move the arm from its current location to the coffee cup's location on the table.
- *Move to Table 2* – Send the correct series of motion profile commands to the Low Level Controller to move the arm from its current location to the water cup's location on the table.
- *Move to Offer Position* - Send the correct series of motion profile commands to the Low Level Controller to move the arm from its current location to offer cup position. Then signal the Speech module with an Offer command to say "Here you go."
- *Ask "Got It"* – Signal the Speech module to ask the user if they have the cup with a QUESTION_GOT_IT command. "Do you have it?" Retrieves the users responses.

3.5.4.3 Pressure Tracking

While the Scene Detection Subsystem is used to determine if the user's hand is near the cup or not, it cannot tell the robot whether or not the person is actually grasping the cup. Thus, the robot's ability to sense the amount of pressure the user is putting on cup is crucial to the task. This information is provided by the Pressure Sensor Subsystem in the High Level Control module.

Originally, a pressure sensor was attached to BERT's hand. Researchers on the CHRIS project modified the hand controller to stream a single numerical pressure value on a YARP port at each 50 msec time step. However, this sensor proved to be unreliable during testing. When BERT grabbed a cup, the pressure

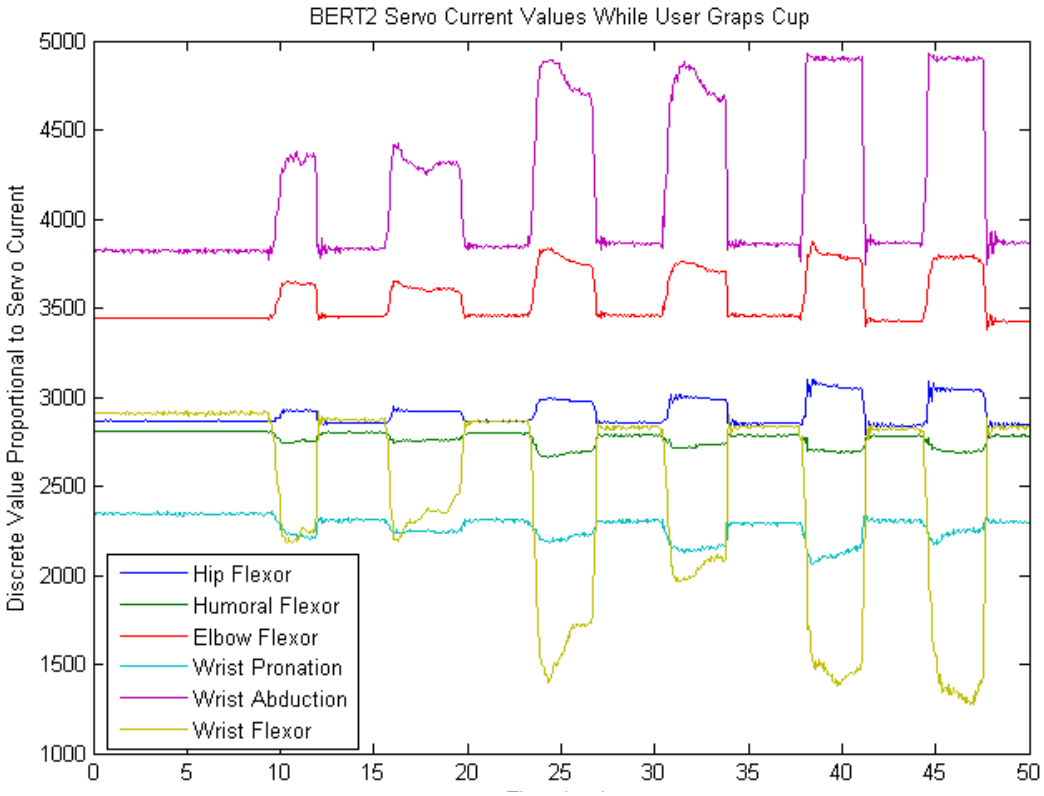


Figure 15 - Servo Current Values While User Pulls on Cup

sensor would not always contact the cup. Thus, no pressure would be registered. Faced with this mechanical issue, another solution was devised. Instead, pressure was detected by measuring the change in current used by the servos in BERT's arm.[8]

When BERT is moved to a specific pose, the Central Controller dynamically modifies the current applied to each servo to maintain that position despite any external forces applied. The effect is that when BERT's arm is outstretched, the user can push or pull on the hand, and it will not move. BERT's central controller counteracts the external force by increasing or decreasing the electrical current (and indirectly the amount of torque applied) on each joint. Thus, these current values are directly proportional to the amount of pressure being applied to each joint.

The central controller streams a vector of eleven real values on a YARP port at 50 msec intervals. Each element of this vector is a number between 0 and 10,000 that is proportional to the amount of current being applied to the corresponding servo. The Pressure Sensor Subsystem reads this vector and uses it to calculate a single discrete pressure value in the range 0 to 31. This pressure value is passed on to the Cognitive Controller by the BERT Interface along with all of the other state information at each time step.

In order to determine how to convert the vector of servo currents into a discrete pressure value, a data collection experiment was first performed. In this experiment, the YARP port streaming the current vector was recorded while BERT was manually moved through the sequence of grasping a cup, offering it, and releasing it when a user grabbed it.

An important factor was identified by visualizing this data in MATLAB. When BERT is not in motion and no external forces are being applied, each current value is roughly static. There is very little noise. However, that value depends upon the robot's pose. This is due to the amount of torque required to maintain that position. For instance, the elbow flexor requires more torque to maintain an outstretched

[8] This was suggested by Alex Lenz, the electrical engineer responsible for BERT2's low level controllers at the Bristol Robotics Lab.

arm position than when the arm is resting at the robot's side. Because of this, the Pressure Sensor Subsystem must filter out this positional dependence.

In addition, when external pressure is applied to BERT's hand, only six of the eleven available joints register change in current. These joints, the hip flexor, humeral arm flexor, elbow flexor, wrist pronation, wrist abduction, and wrist flexor, register a jump in current while the pressure is applied. When the pressure is removed, their currents return to the original value. However, because the orientation of each joint varies, the current increases in some joints and decreases in others when external pressure is applied. Figure 15 above shows the current of these six joints when a user pulls on a cup in the robot's hand six times. The user was asked to pull lightly twice, a medium amount twice, and then to pull hard twice. Each of these joints clearly shows a corresponding small, medium, and large change in current.

With these dynamics in mind, the algorithm in Figure 16 was designed to calculate a discrete pressure value from the vector of servo currents. The algorithm calculates a baseline current vector when first started by taking the mean of each servo current over a short time interval. Every time the robot moves to a new pose, the baseline vector is recalculated.

The baseline vector, $\hat{b}$, is defined as follows:

$$\hat{b} = \langle b_i \rangle,\ i \in [1,6] \tag{102}$$

Where b_i is the baseline current value for servo i defined by:

$$b_i = \frac{\sum_{j=t'}^{t'+n-1} v_i^j}{n} \tag{103}$$

In this formula, v_i^j is the current value on servo i sampled at time j, t' is the time step in which the robot

Figure 16 - Pressure Discretisation Algorithm

$t' \leftarrow 0$ # Time step in which last move completed
$t \leftarrow 0$ # Current time step
$CurrentQ \leftarrow Empty\ Queue$
$\hat{b} \leftarrow \langle 0 \rangle$
$p \leftarrow 0$

Repeat at each time step
    Read current vector $\hat{v}$ from YARP
    Push $\hat{v}$ onto $CurrentQ$
    If $t' > t + n$
        $\hat{b} \leftarrow \hat{b} + \hat{v}$
    Else if $t' = t + n$
        $\hat{b} \leftarrow \hat{b}\,/_n$
    Else if $size(CurrentQ) = n$

$$\hat{c} \leftarrow \left(mean(CurrentQ) - \hat{b}\right) \begin{pmatrix} -1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}$$

$$p \leftarrow \max\left(31, \text{round}\left(31 * \frac{\sum_{i=1}^{6} c_i}{4000}\right)\right)$$

    Pop the last element from $CurrentQ$
Write p to the output YARP port

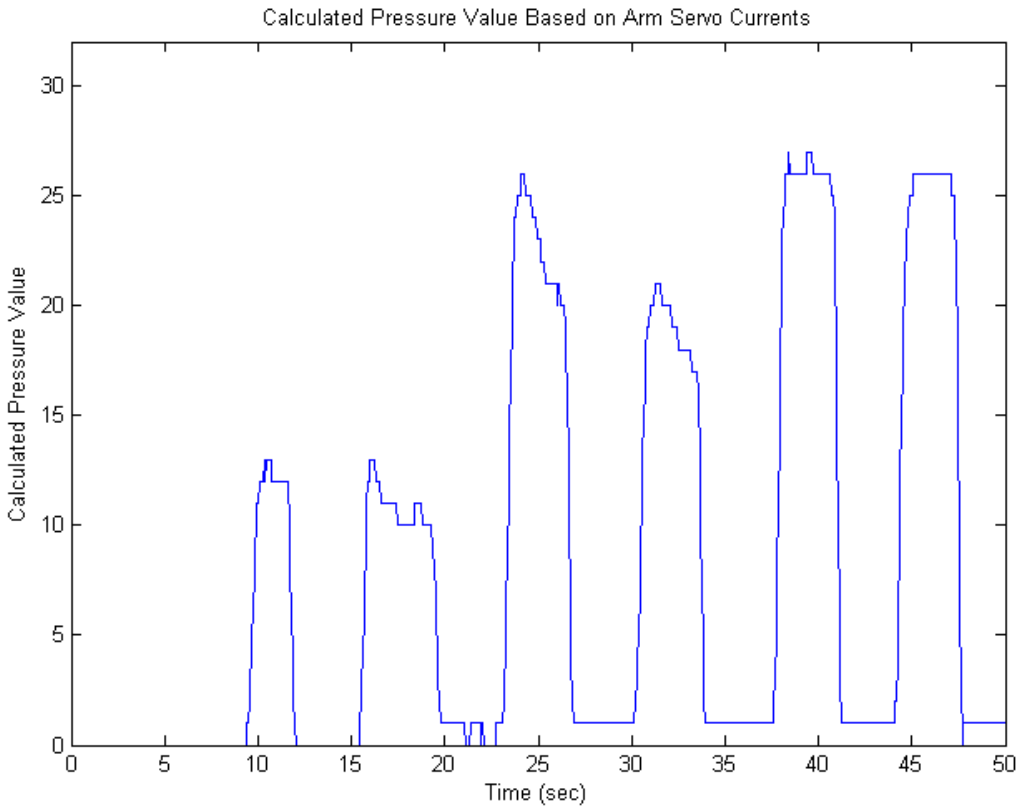


Figure 17 - Calculated Pressure Values

last stopped moving, and n is the number of consecutive time steps to sample for the baseline. Thus, $\hat{b}$ is the vector of the mean current values for each servo over the n time steps immediately after the robot last stopped moving. Recalculating the baseline every time the robot moves allows the algorithm to filter out the dependence on the present position. Taking the baseline as a mean over a short time interval helps to smooth out some of the noise in the signal.

Then at each time step, an instantaneous mean vector of the last few current values is calculated. This vector, $\hat{\mu}$, is defined as:

$$\hat{\mu} = \langle \mu_i \rangle, i \in [1,6] \tag{104}$$

Where μ_i is the mean current value for servo i over the last n time steps and t is the current time step:

$$\mu_i = \frac{\sum_{j=t}^{t+n-1} v_i^j}{n} \tag{105}$$

The baseline vector is then subtracted from the instantaneous mean vector to filter out the dependence on the robot's position. This centres the current vector about the baseline. At this point, the elements whose servo current decreases when a cup is pulled are multiplied by negative one. This ensures that if a cup is pulled, all modified vector values increase. The resulting vector, $\hat{c}$, is defined as:

$$\hat{c} = (\hat{\mu} - \hat{b}) \begin{pmatrix} -1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{pmatrix} \tag{106}$$

In order to transform this vector $\hat{c}$ into the final pressure value, each element is summed together. This value is then normalized into the range 0 to 31 and rounded to the nearest integer. The final pressure value at time t, p_t, is defined below:

$$p_t = \max\left(31, \text{round}\left(31 * \frac{\sum_{i=1}^{6} c_i}{4000}\right)\right) \quad (107)$$

This produces an integer between 0 and 31 that corresponds to how forcefully someone is pulling on the robot's hand. Figure 17 above illustrates the resulting pressure value calculated from the raw current vectors shown in Figure 15 at each time interval.

It would have been simpler to use one of the servo current values instead of a vector of six in this calculation. However, the resulting pressure value would then vary wildly depending upon the direction the user pulls. If the user only pulls in a direction orthogonal to the direction of joint motion, no pressure value would be registered. Aggregating multiple joints using the algorithm above mitigates this issue since it is difficult to pull in a direction orthogonal to six joints. In hindsight, principal components analysis could have been used to reduce the dimensionality of this data. However, it was not studied in this experiment.

3.5.5 Cognitive Controller

The Cognitive Controller contains is responsible for choosing actions to complete the robot's goal. It does so by implementing various RL algorithms. When executing an episode, it periodically receives state vectors from the High Level Controller and presents them to the currently running RL algorithm. The algorithm then chooses an action and returns it to the High Level Controller via YARP. The High Level Controller then sets the executing flag on subsequent state vectors to $true$. This flag is monitored by the Cognitive Controller. When the action is completed, it is set to $false$ and the Cognitive Controller chooses another action.

The following algorithms are implemented by the Cognitive Controller for experimentation:

- *Q-Learning*
- *SLRL*
- *G-Learning*
- *SLGL*

Q-Learning and SLRL use a table representation for the value function. However, the state space is too large to run in memory. Therefore, these algorithms store their value tables on the hard disk. A cache of the 65,535 most recently accessed states is maintained in memory. In addition, space is only allocated on the hard disk for a state when it visited. Thus, states that are never visited do not take up space. This allows medium size state spaces to be solved exactly without too much sacrifice in speed.

However, using a cached table implementation only mitigates the problem of state space explosion. Therefore, two decision tree function approximation algorithms were implemented as well: G-Learning and SLGL. Decision tree function approximation was chosen for investigation since they are easily human-readable. This allows meaningful statements to be made about the adherence of the resulting policy to the safety and liveness properties.

In addition to the RL algorithms listed above, the Cognitive Controller implements a text-based dummy interface for use in the Wizard-of-Oz trials. The dummy interface allows the experimenter to choose actions manually based on the current state information. It is also a useful debugging tool to ensure that the High Level Controller and the Simulation Engine are performing properly. The dynamics of the robot and the simulator can be compared with dummy trials.

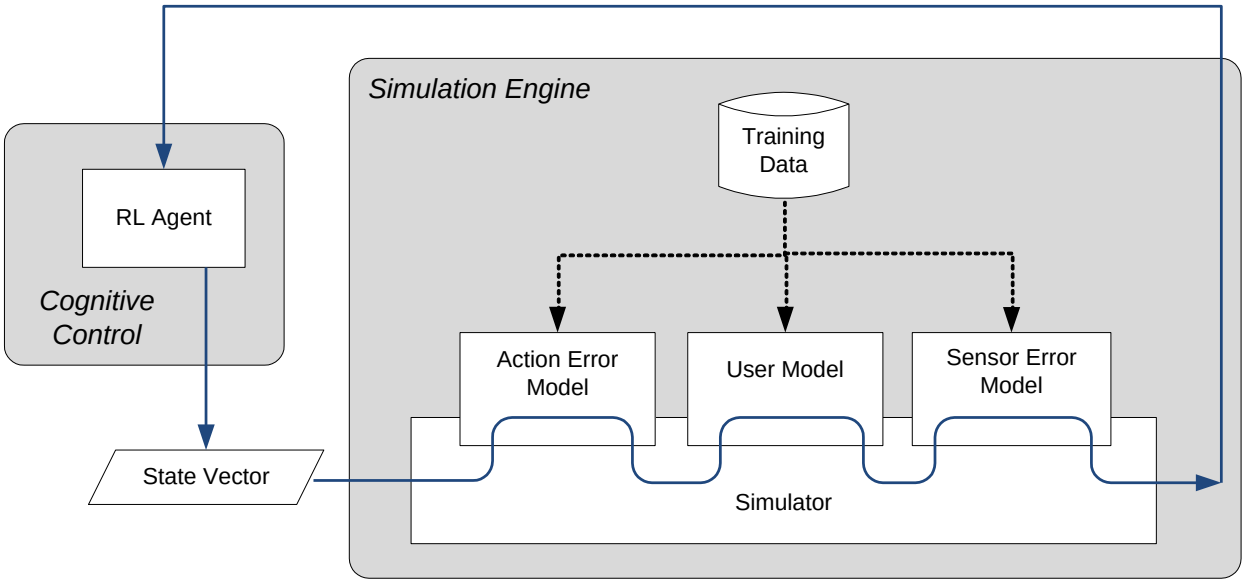


Figure 18 - Simulation Engine Workflow

3.5.6 Simulation Engine

In order to train the reinforcement learning algorithms, a simulator was developed. This simulator interacts with the Cognitive Controller in the same fashion as the High Level Controller. It executes high level action commands and returns state vectors every time step. In general, the dynamics of the environment are modelled as conditional probabilities based on the current state and system action. This abstracts many low level details behind broad probabilities. This allows the simulator to be trained on less real world data than would be required for a more detailed simulator. It follows the simulation techniques used by Prommer for the ARMAR robot discussed in Section 2.3.6.1 [***21***].

The Simulation Engine contains four components:

- *Simulator* – Calculates the state vector resulting from action commands, filters the state vector through the user and error models, and returns it to the Cognitive Controller.
- *User Model* – Models possible user responses to system actions as conditional probabilities.
- *Action Error Model* – Models errors in the action execution systems as conditional probabilities.
- *Sensor Error Model* – Models errors in the sensory systems as conditional probabilities.

Figure 18 above shows how these four components interact every time the Cognitive Controller needs to simulate an action. When an episode starts, the Simulation Engine sends the starting state vector to the Cognitive Controller. When the Cognitive Controller sends an action command back, the simulator calculates a new state vector assuming there were no errors in the sensory executing systems. It also assumes the user does not react to the action at all. This error-free state vector is then passed to the action error model.

When the state vector is passed to the action error model, it calculates faults in the execution of system actions. In particular, the likelihood of dropping a cup when grabbing it as well as the probability of simply failing to grab it is modelled. These probabilities are calculated independently for the water and coffee cups. Formally:

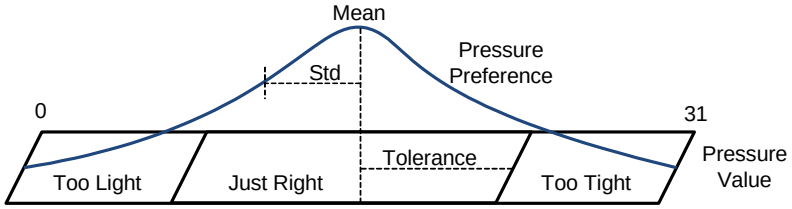


Figure 19 - Determining the Pressure Category

$$\Pr\left(s_{t+1} = dropped\ water | a_t = grab\ water, s_t = ready\ to\ grab\ water\right) \quad (108)$$

$$\Pr\left(s_{t+1} = dropped\ coffee | a_t = grab\ coffee, s_t = ready\ to\ grab\ coffee\right) \quad (109)$$

$$\Pr\left(s_{t+1} = ready\ to\ grab\ water | a_t = grab\ water, s_t = ready\ to\ grab\ water\right) \quad (110)$$

$$\Pr\left(s_{t+1} = ready\ to\ grab\ coffee | a_t = grab\ coffee, s_t = ready\ to\ grab\ coffee\right) \quad (111)$$

By modelling these errors as conditional probabilities, the error model can simply generate a uniformly distributed random number to decide the outcome of grab actions.

The state vector is then passed to the user model which uses a set of conditional probabilities to determine how the user reacts to the system's action. These probabilities model the probability that the user responds with action a when the system performs action b in state s. Formally, the user model contains the following probabilities:

$$\Pr\left(a_{t+1} = a | b_t = b, s_t = s\right) \quad (112)$$

Where a_t is the user's action response, b_t is the Cognitive Controller's action, and s_t is the state at time t.

The user's action responses are high level. They include:

- *Grab drink*
- *Respond "Yes"*
- *Respond "No"*
- *Respond "Coffee"*
- *Respond "Cold"*

The user's *release drink* action is handled different than these actions. When the user grabs the drink, the user model calculates the number of seconds to hold that drink using a Gaussian distribution. At the end of that time, the user model automatically releases the cup.

The user model uses another Gaussian distribution to set the pressure value in the state vector when the user is holding the cup. This Gaussian distribution gives the pressure value a fluctuating value that helps the RL algorithms learn to deal with momentary pressure changes.

If the user is grabbing the drink and the system chooses to release the cup, the user model decides whether or not the user drops the cup based on another set of conditional probabilities. These likelihoods are based on the current amount of pressure. Each user has a preferred pressure setting determined by another Gaussian distribution. If the pressure too much higher than that preference, it is defined as too tight. If the pressure is too much lower than that preference, it is defined as too light. Otherwise, it is labelled just right. This creates three regions in the spectrum of possible pressure values: *too light*, *just right*, and *too tight*. The size of the *just right* region is determined by a tolerance value. The centre of the *just right* region is determined by the mean of the pressure preference distribution. Figure 19 above illustrates how the pressure category is determined by the user's preference and the current pressure value.

The probability of dropping the cup is then modelled for these three categories independently. The probabilities are defined as:

$$\Pr\left(s_{t+1} = dropped\ cup | s_t.pressure = none\right) \quad (113)$$

$$\Pr\left(s_{t+1} = dropped\ cup | s_t.pressure = too\ tight\right) \quad (114)$$

$$\Pr\left(s_{t+1} = dropped\ cup | s_t.pressure = too\ light\right) \quad (115)$$

$$\Pr\left(s_{t+1} = dropped\ cup | s_t.pressure = just\ right\right) \quad (116)$$

Using the probability distributions above, the user model determines the effect the user has on the state vector before passing it back to the simulator. The state vector is then filtered through the sensor error model. The sensor error model simulates the robot's faults when sensing the true state of the environment. This includes errors detecting the pressure values and errors interpreting the user's verbal responses. Errors detecting the location of the user's hand are not modelled. It was determined that the VICON Motion Capture system was accurate and reliable enough to warrant not modelling error.

Errors detecting the pressure value are modelled as a simple Gaussian distribution with a mean of zero and a standard deviation of one. The sensor error model generates a random number from this distribution and adds it to the pressure value in the state vector.

Speech detection errors are modelled as the probability of mistaking the intended user response for a different response. These are calculated on high level intended responses instead of low level speech characteristics like phonemes. This is because user responses are only handled by the Cognitive Controller at a high level. This greatly simplifies the speech error model. When the state vector includes a new verbal user response, it replaces it with a different response based on a conditional probability distribution. For each of the four possible user responses, there is a corresponding set of four conditional probabilities in the following form:

$$\Pr\left(s_{t+1}.response = Yes | s_t.actual\ response = X\right) \quad (117)$$

$$\Pr\left(s_{t+1}.response = No | s_t.actual\ response = X\right) \quad (118)$$

$$\Pr\left(s_{t+1}.response = Hot | s_t.actual\ response = X\right) \quad (119)$$

$$\Pr\left(s_{t+1}.response = Cold | s_t.actual\ response = X\right) \quad (120)$$

After the sensor error model modifies the state vector it is passed back to the Cognitive Controller as the state of the next time step.

In order to ensure that the simulator matches the dynamics of the real environment, all of the conditional probabilities as well as the mean and standard deviations for the Gaussian distributions in the user and error models were calculated based on data recorded from the robot during the Wizard-of-Oz experiments.

	Wins	Raw Violations			Above Allowed Violations		
		S1	S2	L1	S1	S2	L1
MC-ARTDP vs Q-Learning	0.16	0.13	0.00003	~0.0	0.17	0.067	0.33
SLRL vs Q-Learning	0.00009	~0.0	~0.0	0.093	~0.0	0.18	Equal
SLRL vs MC-ARTDP	0.00020	0.00006	0.0051	~0.0	0.0061	0.061	0.33

Table 1 - T-Test Results on Table Algorithm Performance While Training

4 EXPERIMENTAL RESULTS

Two different MDPs were used to test SLRL and SLGL's ability to satisfy safety and liveness constraints both during and after training. The first MDP, the grid world problem discussed in Section 3.3, was used for initial comparisons on a simple problem. Then, the HRI scenario from Section 3.4 was used to stress each algorithm on a problem with complex safety and liveness requirements.

4.1 Grid World Results

Five algorithms, Q-learning, MC-ARTDP, SLRL, G-Learning, and SLGL were trained on the grid world problem. Q-learning, MC-ARTDP, and G-Learning were trained with Boltzmann selection. On the other hand, SLRL and SLGL used safe ϵ-Boltzmann selection. The three table algorithms, Q-Learning, MC-ARTDP, and SLRL were trained for 5,000 episodes. The two regression tree algorithms, G-Learning and SLGL were allowed to build their tree structures for 3,000 episodes. The structure of each tree was then frozen while a further 2,000 training episodes were performed.

All five algorithms used $\gamma = 1$, $\bar{\gamma} = 1$, and the log alpha rule from [***25***]. The log alpha rule decreases the learning parameter α based on the current episode number n:

$$\alpha = \frac{\log_e(n)}{n} \tag{121}$$

After training, the greedy policy for each of the five algorithms was run for another 1,000 episodes. Figures 20 through 23 on the next pages illustrate the results of these trials.

4.1.1 Q-Learning, MC-ARTDP, and SLRL While Training

The performance of the table format algorithms reveals some interesting dynamics. Throughout exploration, SLRL consistently won more games than Q-Learning and MC-ARTDP. After only 500 episodes, SLRL learned to satisfy all safety and liveness properties. At no point over the next 4,500 episodes of exploration did SLRL fail to meet any constraint. In contrast, MC-ARTP required 3,500 episodes to satisfy all constraints with Boltzmann exploration. Q-Learning failed to meet property **[S1]** at all during training.

In order to determine the statistical significance of these comparisons, the average win and constraint violation rates over each 250 consecutive episodes were calculated over the length of training. These rates were then used to calculate two-tail, paired t-tests between Q-Learning and MC-ARTDP, Q-Learning and SLRL, and between MC-ARTDP and SLRL. For the constraint violations, both the raw rate of violations and the violation rate above the allowable percentage were compared. Table 1 above shows the p-values for each pair of compared algorithms. Using a significance level of 0.05, the significant p-values are shown in either blue or green text. A blue p-value indicates the algorithm listed in blue maintained a significantly better win or violation rate throughout training. Likewise, green p-values indicate the green algorithm performed significantly better. Black p-values indicate results that were not statistically significant.

As this table shows, MC-ARTDP only performs significantly better than Q-Learning for **[S2]** while Q-Learning actually performs better on **[L1]**. This is not suprising since both MC-ARTDP and Q-Learning used Boltzmann selection which makes no attempt to meet constraints throughout training. However, SLRL performed significantly better than both Q-Learning and MC-ARTDP in almost all of the raw violation categories. The results are somewhat less impressive when only considering violation rates above the allowable thresholds. This reduces the difference between each algorithm since all three meet **[S2]** and **[L1]** early on in training.

Figure 20 - Average Number of Games Won on the Grid World

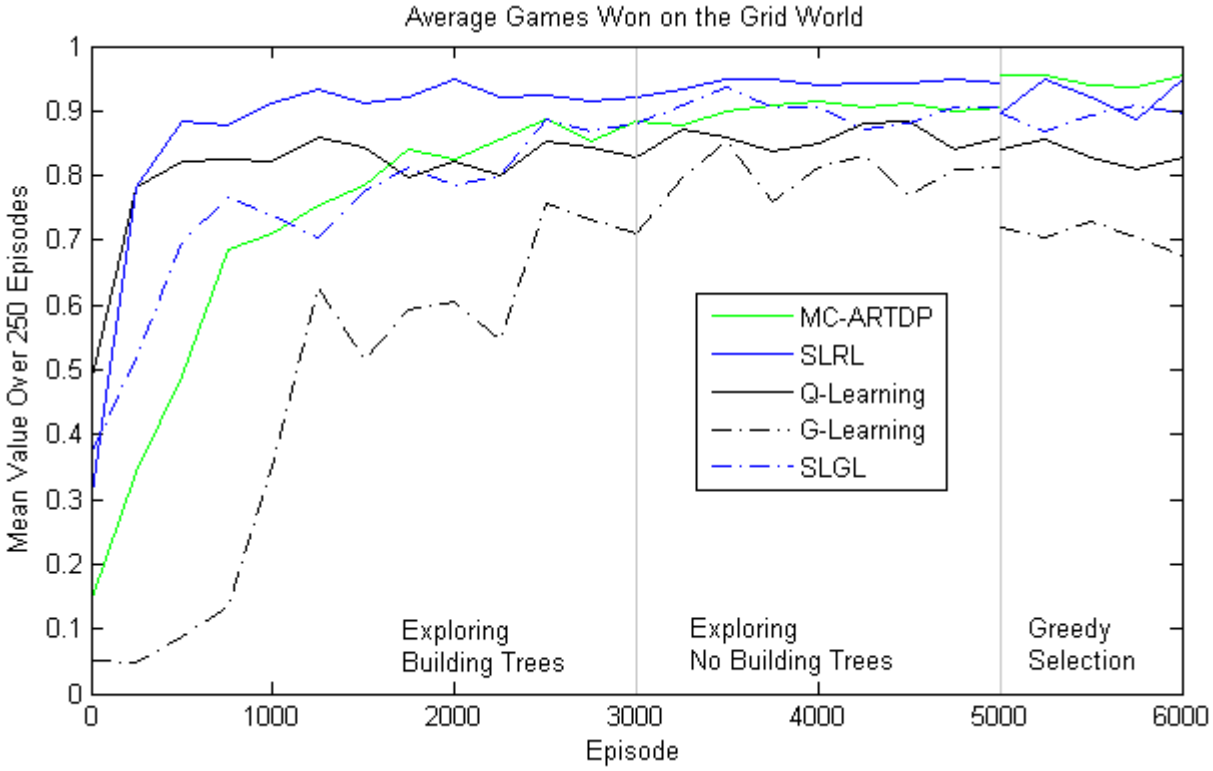
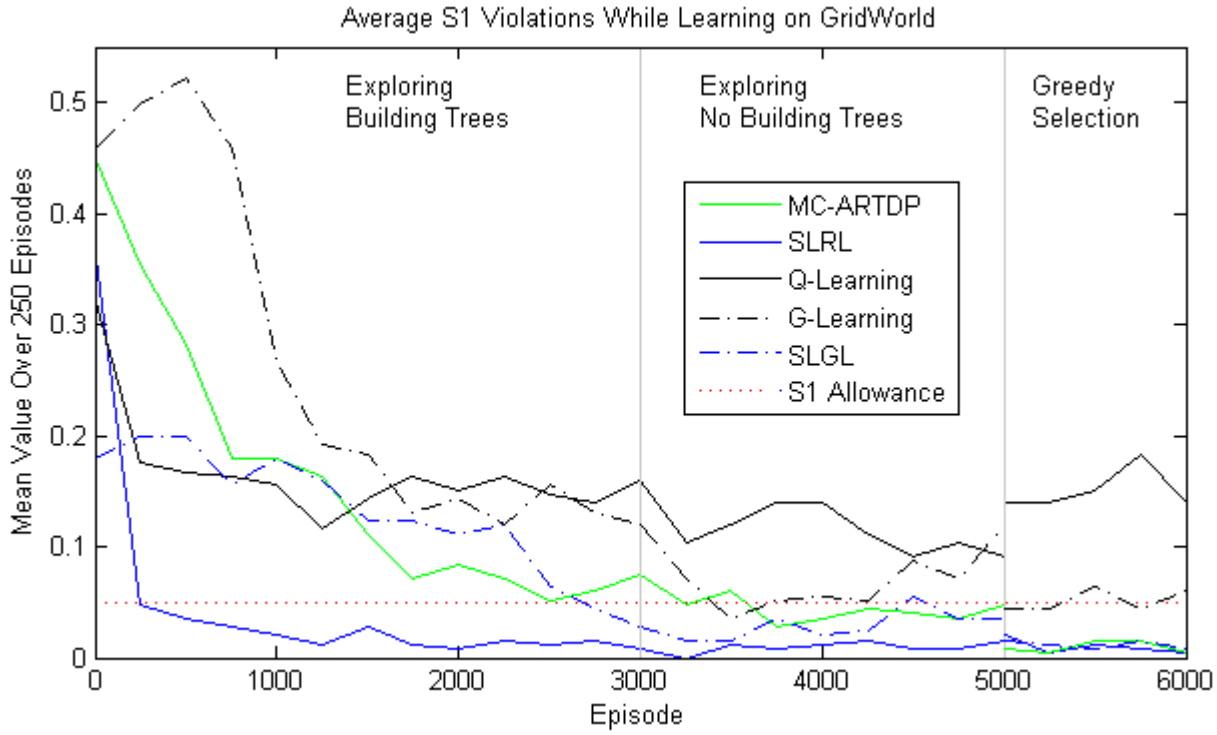


Figure 21 - Average Number of S1 Violations on the Grid World



4.1.2 G-Learning and SLGL While Training

The regression tree algorithms show similar results to the table format algorithms. Despite approximating the value and risk functions, SLGL is able to meet most constraints quite quickly. It begins to satisfy properties **[S2]** and **[L1]** after only 500 episodes of tree building. However, it takes significantly longer for **[S1]**. This suggests that the risk function approximation can slow convergence time for SLRL in simple problems. By aggregating the risk values for multiple states, some actions may be deemed more risky than they really are which may in turn slow exploration. Despite this, SLGL still manages to learn a policy which satisfies the constraints. In contrast, G-Learning wins the fewest games during exploration. It also has the most trouble meeting constraints. It is the only algorithm that fails to satisfy the liveness

Figure 22 - Average Number of S2 Violations on the Grid World

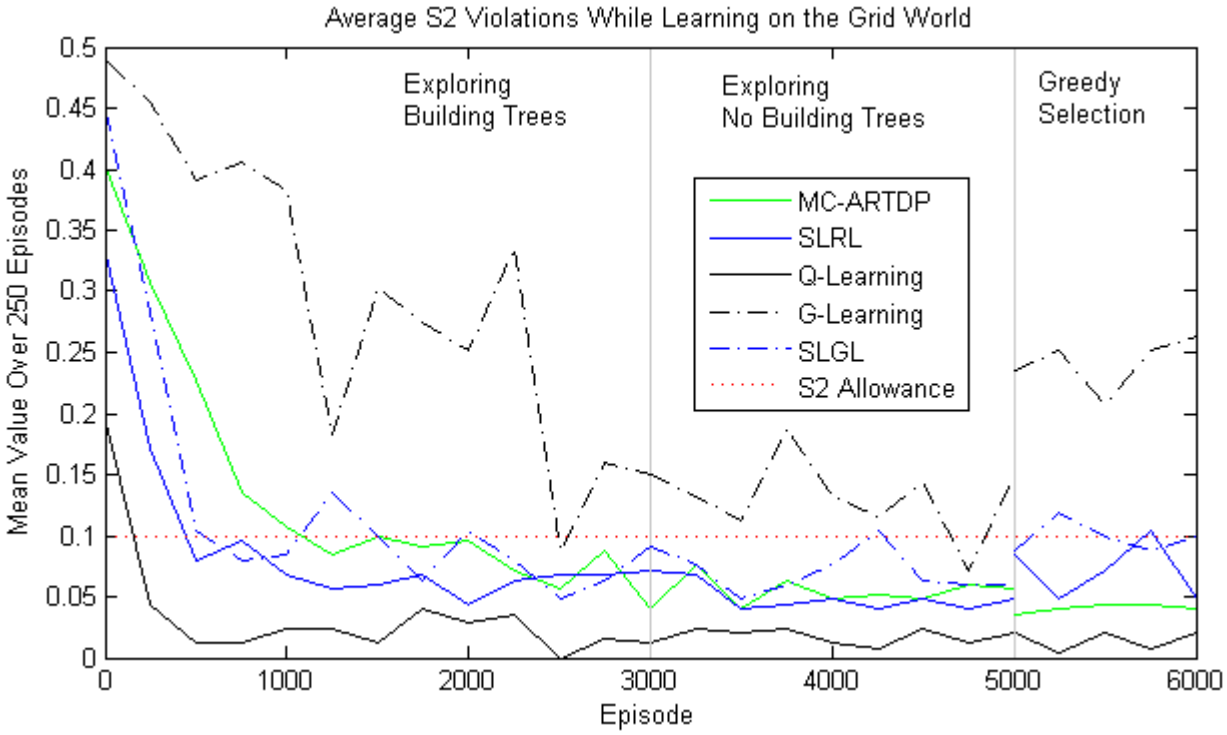
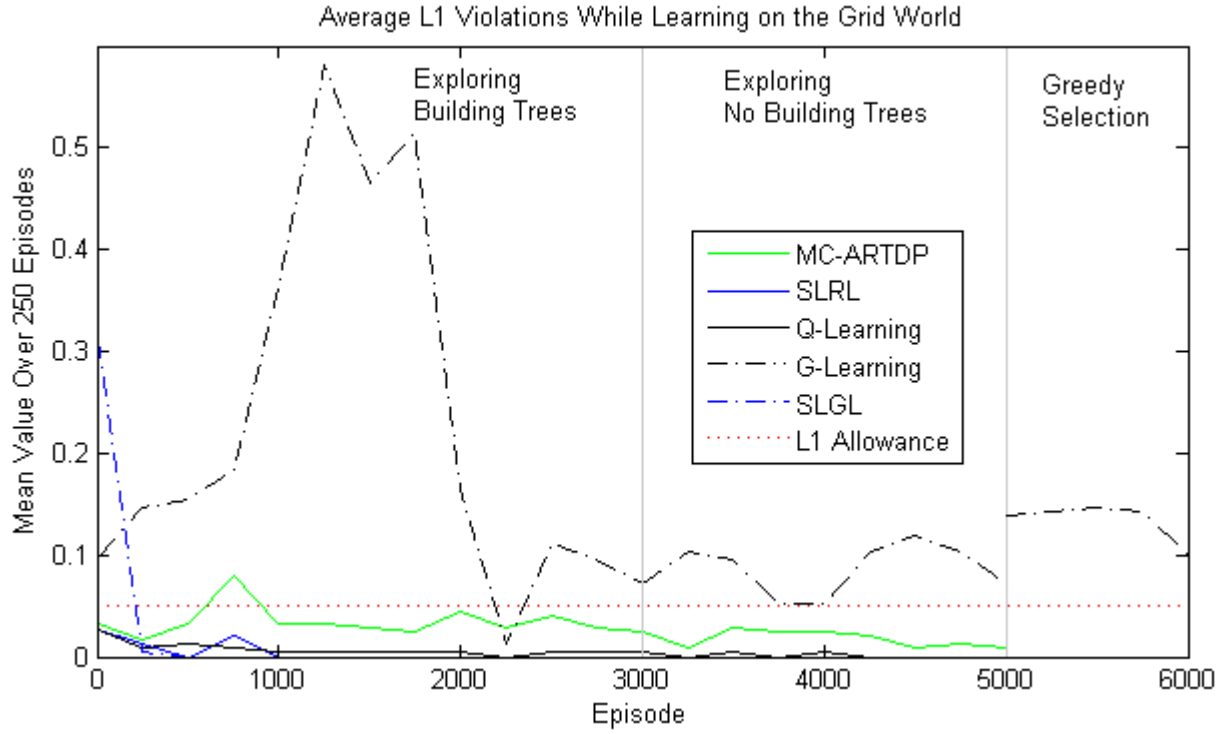


Figure 23 - Average Number of L1 Violations on the Grid World



SLGL vs G-Learning	Wins	Raw Violations			Above Allowed Violations		
		S1	S2	L1	S1	S2	L1
Building Trees	~0.0	0.00040	0.00010	~0.0	0.000041	0.000024	~0.0
Training with Fixed Trees	0.000013	0.00012	0.00093	~0.0	0.00027	0.00039	~0.0

Table 2 - T-Test Results on Approximation Algorithm Performance While Training

constraint, **[L1]**.

Table 2 above shows the p-values resulting from a paired t-test comparison of SLGL and G-Learning while building trees, training with frozen tree structures, and with greedy policies. This comparison uses the same method used for the table format algorithms listed above. These tests show that on the grid world problem, SLGL performs significantly better than G-Learning in all tested categories. This is not surprising since G-Learning did not to converge to a reasonable policy.

4.1.3 All Greedy Policies

Of the lookup table algorithms, SLRL performed better in exploration. MC-ARTDP, on the other hand, achieved the highest reward greedy policy. Both SLRL and MC-ARTDP learned greedy policies which maintained all the constraints. Q-Learning's greedy policy did not satisfy property **[S1]**. It fell off of the left cliff 8.5% more often than allowed. Table 3 below lists the average reward and constraint violations for each greedy policy. These results support the expectations for Q-Learning, MC-ARTDP and SLRL. MC-ARTDP and SLRL ensure that their greedy policies satisfy the safety and liveness properties. Q-Learning does not. Likewise, SLRL's safe ϵ-Boltzmann exploration is able to meet constraints much sooner than standard Boltzmann selection with MC-ARTDP.

A two-tailed, paired t-test was used again to calculate the statistical significance of the results for the greedy policies. Table 5 below shows the p-values for each pair of algorithms when comparing the average win and violation rates for each 250 consecutive greedy episodes. Using a significance threshold of 0.05, both MC-ARTDP and SLRL's greedy policies win significantly more episodes than Q-Learning. They also violate **[S1]** significantly less. However, Q-Learning actually violates **[S2]** less than both MC-ARTDP. However, this significance doesn't hold when comparing the violation rates above the threshold level. This is because both SLRL and MC-ARTDP only try to learn a policy that is below the threshold. Once it is below the threshold, they do not try to minimise it any further. Q-Learning, however, cannot make that distinction. It simply tries to maximize return. Indirectly, it is minimising risk. It is also worth noting that difference between MC-ARTDP and SLRL's greedy policies is not statistically significant.

Similarly, SLGL's greedy policy outperformed G-Learning in every category. It achieved a higher win average while maintaining all three constraints. All of these differences were statistically significant. G-Learning was not able to maintain any of the constraints.

While the decision tree algorithms do not perform as well as the table algorithms, they significantly compress the state space. After 3,000 episodes, G-learning builds a tree with only 13 leaves. Thus, G-learning aggregates the 360 states into 13 groups for a 3.6% compression ratio. SLGL achieves similar

Table 3 - Greedy Policy Performance on 1,000 Episodes

	Win Average	S1	S2	L1
Allowable Violations	-	5%	10%	5%
Q-Learning	83.3%	15.1%	1.4%	0%
MC-ARTDP	94.9%	4.8%	4.1%	0%
SLRL	92.0%	1.0%	7.1%	0%
G-Learning	71.8%	5.1%	24.1%	13.5%
SLGL	89.2%	1.2%	9.9%	0%

	Wins	Raw Violations			Above Allowed Violations		
		S1	S2	L1	S1	S2	L1
MC-ARTDP vs Q-Learning	0.00002	0.000029	0.0030	Equal	0.00029	Equal	Equal
SLRL vs Q-Learning	0.00010	0.00010	0.076	Equal	0.00029	0.37	Equal
SLRL vs MC-ARTDP	0.054	0.99	0.044	Equal	Equal	0.37	Equal
SLGL vs G-Learning	~0.0	0.00030	0.00017	~0.0	0.00018	~0.0	~0.0

Table 5 - T-Test Results on Algorithm Performance After Training

	Q Tree	S1 Tree	S2 Tree	L1 Tree
G-Learning	13 Leaves (3.6%)	-	-	-
SLGL	37 Leaves (10.3%)	30 Leaves (8.3%)	34 Leaves (9.4%)	20 (5.6%)

Table 4 - State Space Compression Ratios on the Grid World

compression ratios when compared with table format SLRL. Table 4 above shows the number of leaves in the value and risk tree learned by G-Learning and SLGL after 3,000 episodes as well as the compression ratio when compared with a 360 state Q-table.

The results for MC-ARTDP, SLRL, and SLGL indicate that the method of defining constraints from safety and liveness properties discussed in Section 3.1.1 does in fact work well. While adding time variables to a MDP does increase its size and complexity, multiple constraint RL algorithms can exploit knowledge about the liveness properties to speed up learning and mitigate this complexity. In addition, the results show that regression tree function approximation can be used to handle the increased size if necessary.

4.2 Hot Drinks Robot

While the results above are promising, the grid world problem is only a toy problem. It does not necessarily indicate performance on complex, real-world problems. Therefore, these algorithms are also tested in both simulated and live HRI scenarios. The HRI experiment described in Section 3.4 is used to compare the RL algorithms since it includes safety and liveness constraints that can be tested with minimal risk of harm to subjects.

4.2.1 Wizard-of-Oz Study

In order to train the hot-drinks robot simulator to match the dynamics of the real scenario, the Wizard-of-Oz study described in Section 3.4.6 was performed using three subjects in 21 total trials. Due to time limitations, each of these subjects was a researcher already familiar with BERT2. For each trial, they were told not to pretend to be unfamiliar with the robot. The concern was that they would try to compensate for previous knowledge and drop cups they otherwise would not have. Because of this, the results of this study cannot predict the actions of general users. It is only directly applicable to those used to working with robots on a regular basis.

In each trial, the subject was asked to request either the hot or cold drink with no preference for either. This was done to see if users would have a natural preference for either cup. Despite this, each user requested each cup roughly the same number of times. The water cup was requested 10 out of 21 times. This is probably due to two factors. First of all, each cup was represented by a nearly identical black cylinder. Thus, the users had no incentive to prefer either cup other than being told the hot drink was "riskier". Secondly, as researchers themselves, they probably had a natural inclination to try to even out the data.

The robot's actions were manually controlled while all of the relevant information was recorded. However, the users were told that the robot was controlled by one of the RL algorithms. They were told this so that they would react the same way in the Wizard-of-Oz study and the full experiments described in the next section.

Throughout the trials, the cup was not dropped at all by a user. Three near misses were recorded, however. If the robot held onto the drink for more than a couple seconds, the user would pull their hand away. So, the cup was purposefully released as the person was drawing their hand away. Instead of allowing the cup to drop in each case, the user quickly caught it.

From this data, two distinct user pressure preference distributions were identified. These pressure preferences indicated the amount of pressure the user would use when grasping the cup. Two of the users had a firm grasp with a mean pressure value of 19.3 and a standard deviation of 2.9 on a scale between 0 and 31. The other tended to use less pressure with a mean of 8.8 and a standard deviation of 1.1.

With a pressure category threshold of 4, the pressure values for the successful handoffs transitioned from no pressure, to the just right category just before the drink was released. In the three near misses, the pressure category was allowed to transition from just right back to no pressure before releasing the drink. Since the pressure category briefly transitions back down through light pressure, the near misses were counted as drops in the light pressure category.

Because of the small quantity of data, a Laplace correction was used when calculating probabilities for the user and error models. The Laplace correction assumes a uniform prior probability distribution and is used to compensate for missing examples when calculating probabilities [***33***]. To use it, a single count is added to each observed event before calculating probabilities. So with a Laplace correction to mitigate the effect of having a small quantity of data, the following probabilities were calculated for the user model:

$$\Pr(s_{t+1} = dropped\ cup | s_t.pressure = none) = 100\% \quad (122)$$

$$\Pr(s_{t+1} = dropped\ cup | s_t.pressure = too\ tight) = 18.2\% \quad (123)$$

$$\Pr(s_{t+1} = dropped\ cup | s_t.pressure = too\ light) = 4.5\% \quad (124)$$

$$\Pr(s_{t+1} = dropped\ cup | s_t.pressure = just\ right) = 4.5\% \quad (125)$$

The Wizard-of-Oz trial was also used to calculate the probability of the user grabbing or releasing the cup based on the current state. The probability that the user would grab a cup in each time step was found to depend on whether or not it was the right one. So, this probability was added to the user model:

$$\Pr(a_{t+1} = grab\ cup | s_t.cup\ offered = s.requested\ cup) = 29\% \quad (126)$$

$$\Pr(a_{t+1} = grab\ cup | s_t.cup\ offered = s.requested\ cup) = 11\% \quad (127)$$

Likewise, the probability that the user would release the cup depended upon the amount of time the user held it without the robot releasing it. So, whenever the user grabbed the cup, the user model was set up to calculate the time it would release the cup based on a Gaussian distribution with a mean of 4 seconds and standard deviation of 1.5 seconds.

In addition to training the user model, the Wizard-of-Oz data was used to calculate probabilities for the action error model. This included the probability of dropping the cup when picking it up and of failing to pick it up without dropping it. Since the robot's low level movements were entirely scripted, the execution of grab actions was very sensitive to the exact placement of cups on the table. Cups were more frequently dropped when the robot tried to grab them than when offering them to the user. When this happened in the Wizard-of-Oz study, the episode was not stopped. The dropped cup was logged and it was manually placed into the robot's hand to continue the episode. However, at no time did the robot try to grab a cup and fail without dropping it. Therefore, a Laplace correction was used again to ensure that the RL agent would learn to handle those scenarios as well. This resulted in the following probabilities

	Responds "Yes"	Responds "No"	Responds "Hot"	Responds "Cold"
Number of Responses	21	21	11	10
Hears "Yes"	*84%*	4%	6.7%	7.1%
Hears "No"	4%	*72%*	6.7%	7.1%
Hears "Hot"	4%	16%	*80%*	7.1%
Hears "Cold"	4%	4%	6.6%	*78.7%*

Table 6 - Error Probabilities of the Speech Recognition System

$$\Pr(s_{t+1} = dropped\ water | a_t = grab\ water, s_t = ready\ to\ grab\ water) = 22.7\% \quad (128)$$

$$\Pr(s_{t+1} = dropped\ coffee | a_t = grab\ coffee, s_t = ready\ to\ grab\ coffee) = 18.2\% \quad (129)$$

$$\Pr(s_{t+1} = ready\ to\ grab\ water | a_t = grab\ water, s_t = ready\ to\ grab\ water) = 4.5\% \quad (130)$$

$$\Pr(s_{t+1} = ready\ to\ grab\ coffee | a_t = grab\ coffee, s_t = ready\ to\ grab\ coffee) = 4.5\% \quad (131)$$

Speech recognition errors were also calculated from this trial data for the sensor error model. In each trial, the user was asked "Do you have it?" once while holding the cup and once while not holding the cup. The user was also asked "Would you like the hot or cold drink" once at the beginning of each trial. Because only six words are recognised ("Hot"/"Coffee" and "Cold"/"Water" are recognized as the same intended response), the speech module was fairly accurate. The user was only misunderstood three times. Each time, the user's response of "No" to "Do you have it" was heard as "Hot" instead. Again, due to the small amount of data collected, a Laplace correction was used by adding one to count of each possible understood response. This resulted in probabilities listed in Table 6 above.

4.2.2 Simulation Results

With the user and error models calculated from the Wizard-of-Oz data, the RL algorithms were trained over 100,000 simulated episodes. Q-Learning and SLRL were compared as table format algorithms. Likewise, G-Learning and SLGL were trained as regression tree approximation algorithms. As on the grid world problem, Q-learning and G-Learning were trained with Boltzmann selection. On the other hand, SLRL and SLGL used safe ϵ-Boltzmann selection. All five algorithms used $\gamma = 1$, $\bar{\gamma} = 1$. Instead of using the log alpha rule from [***25***], a variant was devised to slow the decline of α over time. This was required since the algorithms were being trained over a significantly larger number of episodes than on the grid world problem. The modified log alpha rule decreased the learning parameter α based on the current episode number n:

$$\alpha = \min\left(0.3, \frac{\log_e(n)^3}{n}\right) \quad (132)$$

The lookup table algorithms, Q-Learning and SLRL were trained for 100,000 simulated episodes. Similar to the grid world problem, the two regression tree algorithms were allowed to build their tree structures for 60,000 episodes. The structure of each tree was then frozen while a further 40,000 training episodes were performed. After training, the greedy policy for each of the five algorithms was run for another 10,000 episodes.

4.2.2.1 Q-Learning and SLRL While Training

Figure 24 below illustrates the performance of Q-Learning on the HRI simulator. Over the course of training, Q-Learning steadily increased its rate of successful handoffs. At the same time, it steadily decreased the rate of all violations except **[L4]**. This is the property that limits the amount of time allowed before the cup must be offered. Q-Learning had difficulty satisfying this property until roughly 70,000 training episodes have passed. Despite this, only **[S1]**, **[S2]**, **[L1]**, and **[L2]** were actually satisfied during training. The others, **[S3]**, **[L3]**, and **[L4]**, were not. In contrast, SLRL was able to rapidly increase its rate of successful handoffs after 20,000 episodes as shown in Figure 25. Likewise, the

violations for each property decreased much faster than with Q-Learning. SLRL also began to satisfy all the constraints after approximately 85,000 episodes.

Table 8 below shows the paired t-test comparison between Q-Learning and SLRL's performance during training per each 250 episode increment. Using the same 0.05 significance threshold as before, there is no significant difference between the rate of successful transfers between these two algorithms throughout training. However, SLRL significantly violates **[S1]** and all four liveness constraints less than Q-Learning. Q-Learning seems to concentrate on **[S2]** and **[S3]**, the two safety properties that govern how

Figure 24 - Q-Learning Performance on the HRI Simulator

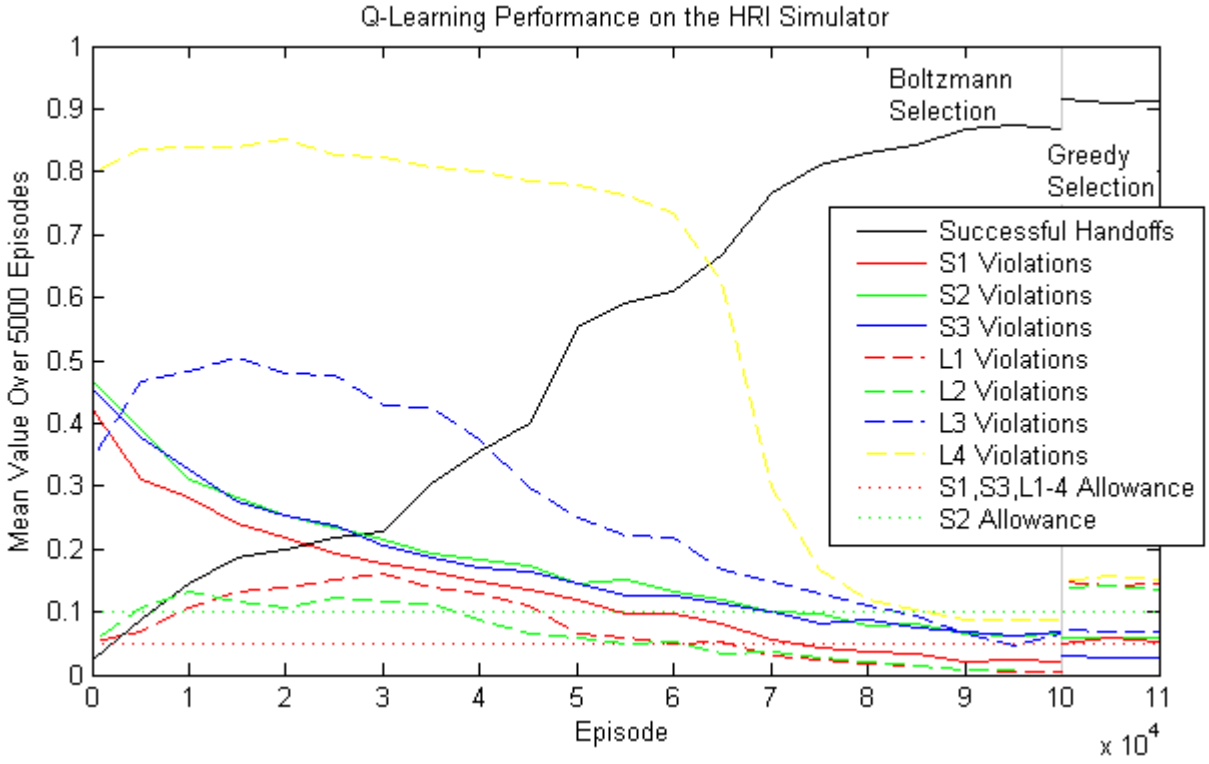
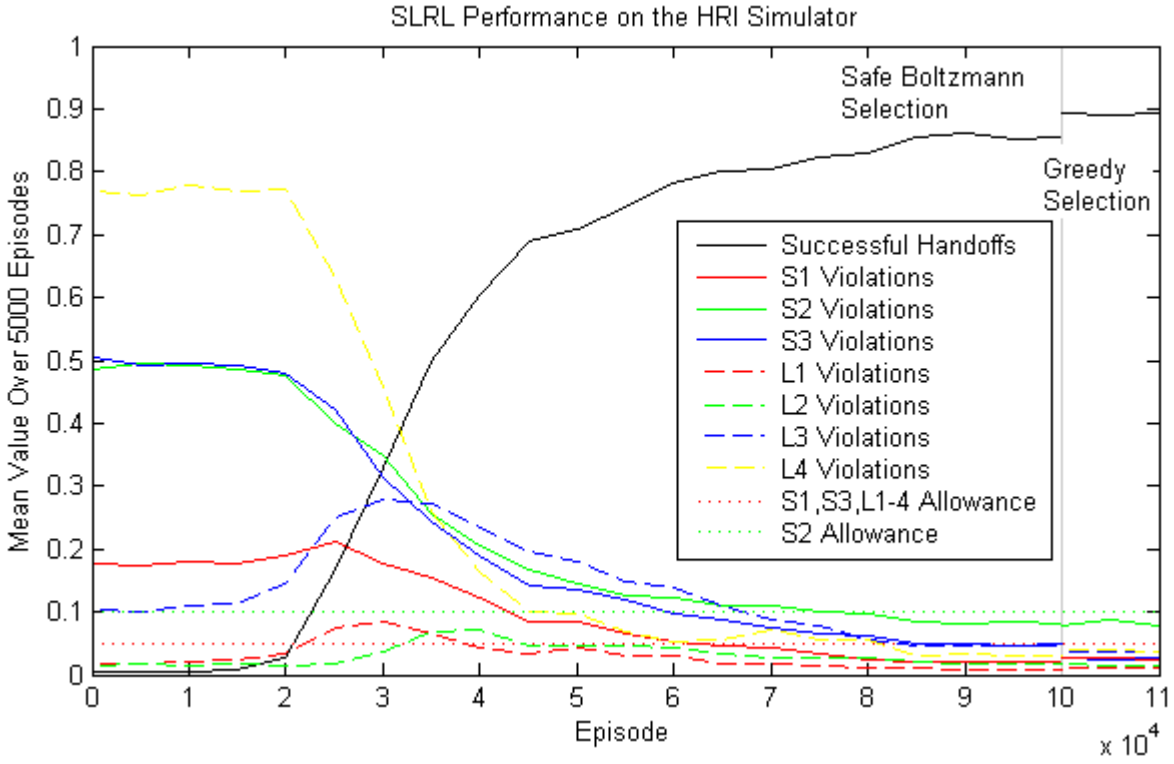


Figure 25 - SLRL Performance on the HRI Simulator



SLRL vs Q-Learning	Transfer Rate	S1	S2	S3	L1	L2	L3	L4
Raw Violations	0.19	0.0055	0.0043	0.038	~0.0	0.0029	~0.0	~0.0
Above Allowed Threshold	-	0.0096	0.0086	0.036	~0.0	0.0022	~0.0	~0.0

Table 8 - T-Test Results on Table Algorithm Performance for the HRI Simulator

SLGL vs G-Learning	Transfer Rate	S1	S2	S3	L1	L2	L3	L4
Raw Violations								
Building Trees	0.052	0.0001	0.053	0.0015	0.031	0.031	0.0005	0.16
Training with Fixed Trees	0.00013	~0.0	0.41	0.38	Equal	Equal	0.14	0.54
Above Threshold		*5%*	*10%*	*5%*	*5%*	*5%*	*5%*	*5%*
Building Trees	-	0.0009	0.049	0.002	Equal	Equal	0.0007	0.16
Training with Fixed Trees	-	0.34	Equal	Equal	Equal	Equal	0.34	0.79

Table 7 - T-Test Results for Approximation Algorithm Performance While Training

often each cup is dropped.

4.2.2.2 G-Learning and SLGL While Training

Given the complexity of the MDP, it is not surprising that the two function approximation algorithms learned suitable policies faster than SLRL and Q-Learning. Figures 26 and 27 below illustrate G-Learning and SLGL's performance through tree learning, exploring, and greedy selection. The aggregation of the 11,059,200 states into regression trees allowed both G-Learning and SLGL to generalise on experience to unvisited states. They were able to predict value and risk when encountering new states. This enabled SLGL to learn a policy that satisfies all but **[L4]** as early as 30,000 episodes. This last constraint was then met after only 60,000 episodes. Compared to SLRL's 85,000 training episodes, this was a significant improvement. Likewise, G-Learning learned a suitable policy after approximately 60,000 episodes versus Q-Learning's 80,000 episodes.

Table 7 above show the p-values from a paired t-test between G-Learning and SLGL throughout training similar to the test run for Q-Learning and SLRL. With a significance level of 0.05, SLGL violated all but **[S2]** and **[L4]** significantly less than G-Learning while tree building. However, this only holds for the raw violation rates. When considering only the violations above the acceptable threshold, SLGL only performed better on **[S1]**, **[S3]**, and **[L3]**. G-Learning, on the other hand performed significantly better on **[S2]** and on the rate of successful cup transfers. In addition, both algorithms converged to a good policy by the time the trees were fixed. So for most constraints, there was no significant difference between the two algorithms through the rest of training.

Not only do the regression tree approximation algorithms perform better throughout training, they significantly compressed the size of the state table. Instead of maintaining tables of action values and risks for each of the 11,059,200 distinct states, G-Learning and SLGL aggregated them into a small number of regression tree leaves. In fact, neither algorithm required more than 114 distinct leaves to represent all of the states in the MDP. This resulted in compression ratios as low as 0.00047%. Table 9 below shows the number of leaves and compression ratios used for each tree by the two algorithms.

Table 9 - State Space Compression Ratio for the HRI Simulator

%# Leaves	Q Tree	S1 Tree	S2 Tree	S3 Tree	L1 Tree	L2 Tree	L3 Tree	L4 Tree
G-Learning	53	-	-		-			
SLGL	130	111	114	111	109	109	106	104
Compression Ratio (%)								
G-Learning	0.00047	-	-	-	-	-	-	-
SLGL	0.0012	0.0010	0.0010	0.0010	0.00098	0.00098	0.00096	0.00094

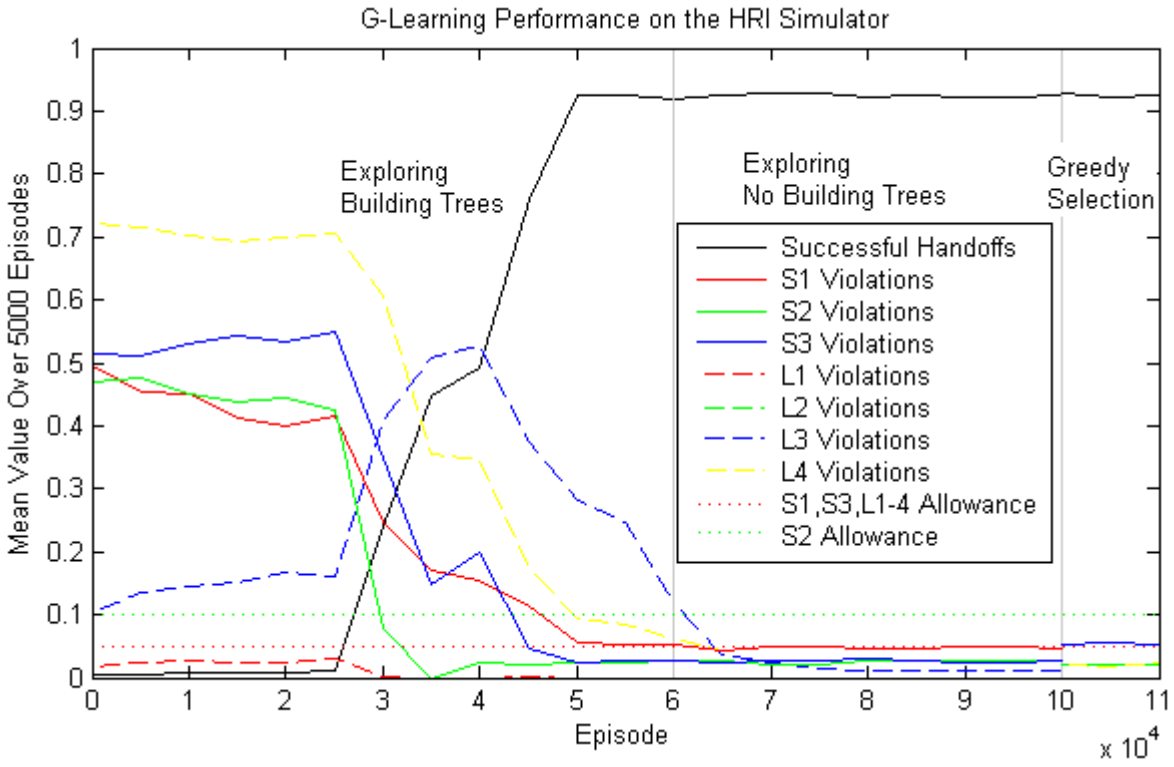


Figure 26 - G-Learning Performance on the HRI Simulator

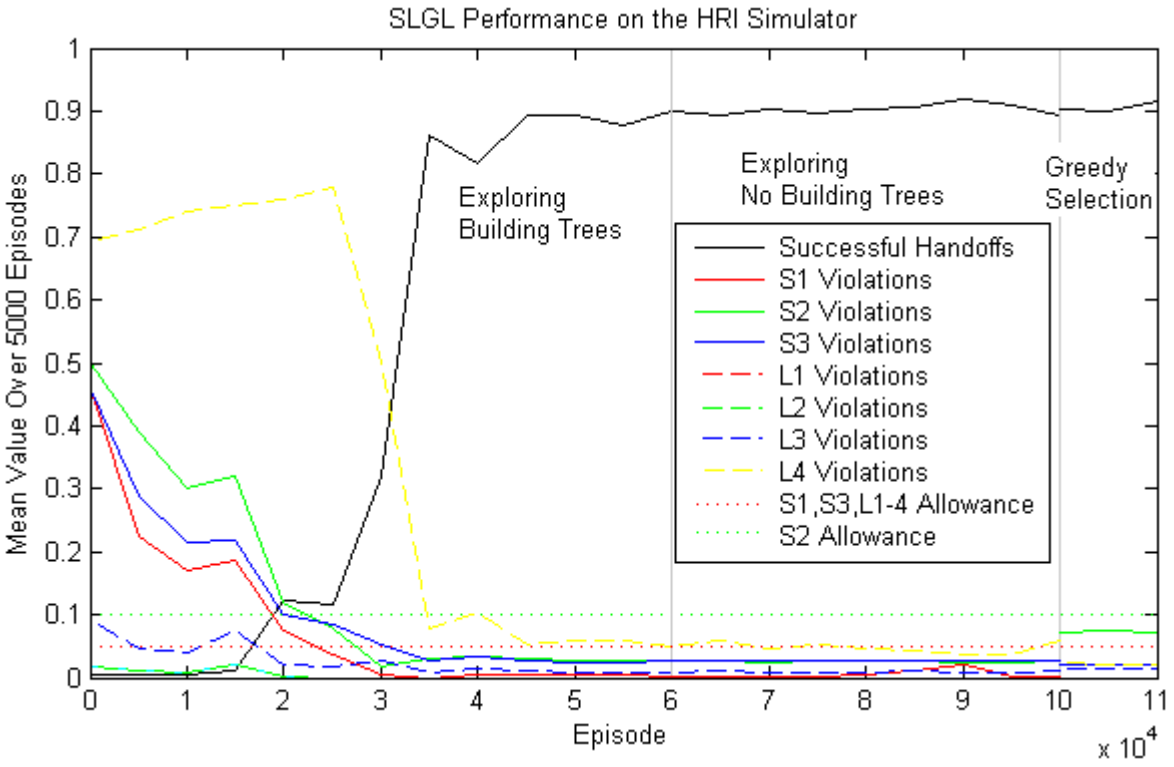


Figure 27 - SLGL Performane on the HRI Simulator

4.2.2.3 All Greedy Policies

On this MDP, G-Learning actually learns a greedy policy with the highest rate of successful transfers. However, in doing so, it violates **[S3]**, the safety property governing how often the coffee cup may be dropped. Likewise, Q-Learning achieves the second highest transfer rate, but it also violates **[S1]** and all of the liveness properties. Unlike the others, SLRL and SLGL are able to maintain all of the constraints with their greedy policy. Table 11 below shows the performance of all four algorithms on 10,000 greedy

	Successful Transfers	S1	S2	S3	L1	L2	L3	L4
Allowed Violations	-	5%	10%	5%	5%	5%	5%	5%
Q-Learning	91.2%	5.3%	5.9%	2.9%	14.6%	13.9%	6.9%	15.3%
SLRL	89.3%	0.4%	8.1%	2.5%	1.0%	1.5%	3.7%	3.8%
G-Learning	92.5%	0%	2.1%	5.4%	0%	0.5%	2.1%	2.0%
SLGL	90.6%	0%	7.3%	2.1%	0%	0%	1.5%	2.2%

Table 11 - Greedy Policy Performance on the HRI Simulator

	Transfer Rate	S1	S2	S3	L1	L2	L3	L4
Raw Violations								
SLRL vs Q-Learning	0.0016	0.0065	0.019	0.046	0.0003	0.0001	0.0008	0.0005
SLGL vs G-Learning	0.053	Equal	0.0001	0.0018	Equal	0.22	0.35	0.32
Above Threshold		*5%*	*10%*	*5%*	*5%*	*5%*	*5%*	*5%*
SLRL vs Q-Learning	-	0.26	Equal	Equal	0.0047	0.0037	0.0012	0.0004
SLGL vs G-Learning	-	Equal	Equal	0.020	Equal	Equal	Equal	Equal

Table 10 - T-Test Results for Algorithm Performance After Training

episodes after 100,000 training episodes. SLGL outperforms SLRL to a small degree. This is most likely due to the fact that SLGL is able to generalise from experience.

To determine the statistical significance of these results, another set of paired t-tests was performed in the same manner that was used for the training episodes. Table 10 above shows the results of these tests using a significance level of 0.05. While Q-Learning's rate of successful cup transfers is significantly higher than SLRL's, almost all of SLRL's raw violation rates are significantly lower than those of Q-Learning. Likewise, both SLGL and G-Learning learned good greedy policies with very little significant difference. While SLGL violates **[S3]** significantly less, G-Learning violates **[S2]** less. This highlights the main difference between these two pairs of algorithms. Q-Learning and G-Learning are simply trying to maximise return and indirectly minimise total error. However, SLRL and SLGL have more information available in the form of the reward/error vector. Thus, they are able to maximise reward such that the constraint violation rates are below the specified thresholds. This is more apparent when comparing the p-values for the violation rates above the thresholds. The only significant differences between SLRL and Q-Learning are for the liveness constraints since they are the only ones violated by Q-Learning. Likewise, the only significant difference between G-Learning and SLGL is for **[S3]** which is also the only constraint G-Learning violates.

4.2.3 Live Results

While the simulation results are promising, they do not necessarily translate to performance on live interactions. Therefore, the greedy policies that were trained on the HRI simulator for Q-Learning, SLRL, G-Learning, and SLGL were used to control BERT2 in a series of live trials in the hot-drinks robot scenario. In this experiment, three subjects interacted with BERT2 in ten episodes each for each of the four algorithms. Two of the subjects partook in the Wizard-of-Oz study. The third subject was myself. Thus, all of the subjects already knew what to expect from the robot. Because of this fact, these results are not applicable to interactions with the general population. People unfamiliar with robotics may be more hesitant or less forgiving of the robot's errors.

Although, in each trial, the subjects were free to choose either coffee or water, all subjects, including myself, still asked for each cup in equal ratios for each algorithm. So, each algorithm was tested in five trials for each cup. Table 12 below lists the transfer and violation rates for each algorithm in the live trials. In all of these trials, none of the safety properties were violated more than their allowable rates. However, Q-Learning failed to satisfy liveness constraints **[L1]**, **[L3]**, and **[L4]** due to slow response times throughout each trial. It had a tendency to execute wait actions at inappropriate times. The robot

	Successful Transfers	S1	S2	S3	L1	L2	L3	L4
Allowed Violations	-	5%	10%	5%	5%	5%	5%	5%
Q-Learning	96.7%	3.3%	3.3%	0%	13.3%	3.3%	20%	10%
SLRL	96.7%	0%	3.3%	0%	0.0%	0.0%	3.3%	0.0%
G-Learning	96.7%	0%	0%	3.3%	0.0%	0.0%	0.0%	0.0%
SLGL	100%	0%	0%	0%	0.0%	0.0%	0.0%	0.0%

Table 12 - Algorithm Performance on Live Trials

would reach to pick up a cup and wait in the grab position for a time before continuing. When offering the cup, the robot would also sometimes pull the cup away when the user grabbed for it. While this did not result in violated safety constraints, it caused many liveness violations and increased user frustration. Subjects commented that this algorithm felt sluggish and unresponsive.

The other three algorithms did not suffer from this failure. They did not execute wait actions inappropriately or pull cups away when the user grabbed for them. However, subjects still felt they were less responsive than they should be. This was due to two factors. First, the BERT2's joint speed was set low to keep subjects from getting injured. Secondly, the High Level Control module introduced delay between executing actions and sending the resulting states back to the Cognitive Controller. This delay could be reduced through additional engineering of the control architecture.

Q-Learning was also the only algorithm to release a cup when the user was not reaching for it. However, both SLRL and G-Learning experienced near misses with cup transfers. When the user grabbed the cup, the robot held onto it longer than the user expected. Only when the subject started to pull their hand away did the robot release the cup. However, in these incidents the user caught the cup as it fell. If they were real cups, the liquid would have spilled. Therefore, these were counted as dropped cups.

While the number of live trials performed was relatively small, these results support the dynamics revealed by the greedy policies in simulation. This indicates that the simulator's dynamics match the dynamics of the live trials to a good degree. However, stronger comparisons cannot be made without significantly more live data which would be possible in future research.

5 CONCLUSIONS

In this dissertation, a reinforcement learning (RL) algorithm designed to address both safety and liveness in human-robot interaction (HRI) systems was presented. This algorithm, Safe and Live Reinforcement Learning (SLRL), includes a method to translate stochastic safety and liveness properties into error constraints of the form used in Geibel and Wysotzki's Risk-Sensitive Reinforcement Learning [***7***]. SLRL uses an approach similar to Multiple Criteria Adaptive Real-Time Dynamic Programming (MC-ARTDP) [***2***] to find a greedy policy that maximises an agent's expected reward such that the risk of violating one of the safety or liveness properties stays below a set of specified thresholds. It does so by learning separate risk functions for each constraint. However, where MC-ARTDP uses a reverse 2nd lexigraphic ordering to compare the value and risks of actions, SLRL uses a simple two step comparison which does not require the constraints to be ordered. SLRL also uses safe ϵ-Boltzmann selection to manage risk while explore the state space. This exploration strategy attempts to keep the expected risk of chosen actions below the allowable thresholds when choosing actions according to a modified Boltzmann distribution.

Because SLRL compounds the curse of dimensionality through its use of multiple lookup tables, Safe and Live G-Learning (SLGL) was presented as the combination of SLRL with Pyeatt and Howe's G-Learning [***3***]. SLGL uses regression tree function approximation to significantly compress the value and risk state tables and give the agent a method to generalise from its experience.

The experimental results on both the grid world problem and the hot-drinks HRI scenario support the claim that safety and liveness properties can indeed be expressed as error signals in an MDP and solved with multiple constraint reinforcement learning algorithm such as MC-ARTDP, SLRL, or SLGL. In addition, SLRL produces greedy policies that maximise return with constrained risk. With safe ϵ-Boltzmann selection, SLRL is also able to explore an MDP while maintaining the expected risk of randomly chosen actions below specified thresholds. In other words, SLRL has been shown to meet the safety and liveness properties as soon as its estimates of the risk functions begin to approach the true risk functions. Through its exploration policy, SLRL exploits its knowledge of risk and value to significantly speed up learning on multiple constraint MDPs.

Results also indicate that Pyeatt and Howe's G-Learning extension from [***3***] can be used to approximate the value and risk functions in a multiple constraint RL algorithm such as SLRL. SLGL's performance on the hot drinks robot scenario indicate that using regression tree approximation for the risk functions allows the algorithm to generalise its experience with errors. This results in shorter convergence times and better performing greedy policies compared to using table format risk and value functions on complex MDPs.

This project has shown that HRI and machine learning research can be brought together to solve problems with complex safety and liveness requirements. Incorporating safety and liveness into a RL algorithm allows them to be considered directly by the learner instead of attempting to verify those properties against the algorithm's output. Both the fields of machine learning and robotics benefit from this technique. Bringing safety and liveness into the realm of the learner allows it to reason about constraints it will be required to satisfy. This keeps the learner from producing invalid results. An optimisation algorithm that incorporates safety and liveness reduces the burden of certification. By drawing from current research in safety and liveness, robotics, and machine learning, the methods in this project advance the state of the art in all three.

6 SUGGESTIONS FOR FUTURE WORK

This project was primarily concerned with managing safety and liveness on stationary MDPs with multiple constraints throughout learning[9]. However, safe ϵ-Boltzmann selection's ability to manage the risk of violating constraints while performing exploratory actions makes it a prime candidate for managing risk in non-stationary MDPs. As such, SLRL and SLGL could next be applied to a human-robot interaction scenario with changing dynamics as a possible follow-on project. For example, a scenario in which BERT2 must learn to adapt online to people who are more or less clumsy could be used. This would test each algorithm's ability to maintain safety and liveness constraints despite changing drop probabilities.

Because this project focused on adherence to safety and liveness properties, the reward signal used did not include any measure of the quality of the handoff. As such, the algorithms involved were not able to truly optimise fluency. A useful extension to this project would examine alternate formulations of the reward signal to increase fluency in handoff tasks. Possibilities include measuring the amount of time the user waits while holding the cup before the robot releases it similar to Cakmak et al.'s work in [***15***] or some subjective measure of user satisfaction akin to Mitsunaga et al.'s work in [***22***]

In addition, both SLRL and SLGL rely on attribute vector state space formulations. However, these algorithms could be extended to use first-order logic state space formulations. SLRL and SLGL could be applied to relational MDPs with multiple constraints as a relational reinforcement learning algorithm [***34***]. This would be a huge benefit to HRI since it would allow robots to reason about the risk of possible actions in complex environments using the power of first order logic.

Another useful project would further explore using knowledge of the electrical current of a humanoid robot's servos to predict the robot's position and the amount of outside force applied to each joint. Hidden Markov models could be used to predict this proprioceptive state. This was recommended to a fellow researcher at the Bristol Robotics Lab as a follow-on project using BERT2. The resultant research is currently ongoing.

[9] The results of this project are currently being prepared for publication.

7 REFERENCES

[1] L. Lamport, "Proving the Correctness of Multiprocess Programs," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125-143, March 1977.

[2] Z. Gábor, Z. Kalmár, and C. Szepesvári, "Multi-Criteria Reinforcement Learning," in *Proceedings of the 15th International Conference on Machine Learning*, 1998, pp. 197-205.

[3] L. Pyeatt and A. Howe, "Decision Tree Function Approximation in Reinforcement Learning," in *Proceedings of the 3rd International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models*, 2001, pp. 70-77.

[4] E. C. Grigore et al., "Towards Safe Human-Robot Interaction," in *Towards Autonomous Robotic Systems, Lecture Notes in Computer Science*. Berlin / Heidelberg: Springer, 2011, vol. 6856, pp. 323-335.

[5] A. Lenz et al., "The BERT2 Infrastructure: An Integrated System for the Study of Human-Robot Interaction," in *IEEE-RAS International Conference on Humanoid Robots*, Nashville, TN, 2010, pp. 346-351.

[6] A. Biere, C. Artho, and V. Schuppan, "Liveness Checking as Safety Checking," *Electronic Notes on Theoretical Computer Science*, vol. 66, no. 2, 2002, http://www.elsevier.nl/locate/entcs/volume66.html.

[7] P. Geibel and F. Wysotzki, "Risk-Sensitive Reinforcement Learning Applied to Control under Constraints," *Journal of Artificial Intelligence Research*, vol. 24, pp. 81-108, 2005.

[8] P. Fitzpatrick, G. Metta, and L. Natale, "Towards Long-Lived Robot Genes," *Robotics and Autonomous Systems*, vol. 56, no. 1, pp. 29-45, January 2008.

[9] T. Prommer, "Rapid Simulation-Driven Reinforcement Learning of Multimodal Dialog Strategies for Human-Robot Interaction," Interactive Systems Laboratories, Carnegie Mellon University, PhD Thesis 2006.

[10] A. Kendon, *Gesture: Visible Action as Utterance*.: Cambridge University Press, 2004.

[11] M. Staudte and M. Crocker, "Visual Attention in Spoken Human-Robot Interaction," in *Proceedings of the 4th ACM/IEEE International Conference on Human Robot Interaction*, 2009, pp. 77-84.

[12] M. Tomasello, M. Carpenter, J. Call, T. Benhe, and H. Moll, "Understanding and Sharing Intentions: The Origins of Cultural Cognition," *Behavioral and Brain Sciences*, vol. 28, no. 5, pp. 675-691, 2005.

[13] S. Duncan and D. W. Fiske, *Face-to-Face Interaction: Research, Methods, and Theory*.: Lawrence Erlbaum Associates, Inc., 1977.

[14] M. K. Lee, J. Forlizzi, M. Cakmak, and S. Srinivasa, "Predictability or Adaptivity? Designing Robot Handoffs Modeled from Trained Dogs and People," in *Proceedings of the 6th International Conference on Human-Robot Interaction*, 2011, pp. 179-180.

[15] M. Cakmak, S. Srinivasa, M. K. Lee, and J. Forlizzi, "Using Spatial and Temporal Contrast for Fluent Robot-Human Hand-overs," in *Proceedings of the 6th International Conference on Human-Robot Interaction*, 2011, pp. 489-496.

[16] M. K. Lee, S. Kiesler, J. Forlizzi, S. Srinivasa, and P. Rybski, "Gracefully Mitigating Breakdowns in Robotic Services," in *Proceedings of the 5th ACM/IEEE International Conference on Human-Robot Interaction*, 2010, pp. 203-210.

[17] R. Murphy and D. Woods, "Beyond Asimov: The Three Laws of Responsible Robotics," *Intelligent Systems, IEEE*, vol. 24, no. 4, pp. 14-20, 2009.

[18] H. Chockler, O. Kupferman, and M. Vardi, "Coverage Metrics for Formal Verification," in *Lecture Notes in Computer Science: Correct Hardware Design and Verification Methods*, D. Geist and E. Tronci, Eds. Berlin: Springer, 2003, vol. 2860, pp. 111-125.

[19] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626-643, 1996.

[20] D. Gordon, "APT Agents: Agents That Are Adaptive, Predictable and Timely," in *Proceedings of the First Goddard Workshop on Formal Approaches to Agent-Based Systems*, 2000, pp. 278-293.

[21] R. Stiefelhagen et al., "Enabling Multimodal Human-Robot Interaction for the Karlsruhe Humanoid Robot," *IEEE Transactions on Robotics*, vol. 23, no. 5, pp. 840-851, October 2007.

[22] N. Mitsunaga, C. Smith, T. Kanda, H. Ishiguro, and N. Hagita, "Robot Behaviour Adaptation for Human-Robot Interaction based on Policy Gradient Reinforcement Learning," in *International Conference on Intelligent Robots and Systems 2005*, 2005, pp. 218-225.

[23] A. Gosavi, "Reinforcement Learning: A Tutorial Survey and Recent Advances," *INFORMS Journal on Computing*, vol. 21, 2009.

[24] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: The MIT Press, 1998.

[25] A. Gosavi, "On Step Sizes, Stochastic Shortest Paths, and Survival Probabilities in Reinforcement Learning," in *Proceedings of the 2008 Winter Simulation Conference*, 2008, pp. 525-531.

[26] T. Perkins and A. Barto, "Lyapunov Design for Safe Reinforcement Learning," *Journal of Machine Learning Research*, vol. 3, pp. 803-832, 2002.

[27] G. Tesauro, "TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master Level Play," *Neural Computation*, vol. 6, no. 2, pp. 215-219, March 1994.

[28] J. Boyan and A. Moore, "Generalization in Reinforcement Learning: Safely Approximating the Value Function," in *Advances in Neural Information Processing Systems 7*. Cambridge, MA: MIT Press, 1995, pp. 369-376.

[29] D. Chapman and L. Kaelbling, "Input generalization in delayed reinforce-," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1991, pp. 726-731.

[30] J. R. Quinlan, "Induction of Decision Trees," *Machine Learning*, vol. 1, pp. 81-106, 1986.

[31] G. Snecdecor and W. Cochran, "Comparison of Two Samples," in *Statistical Methods, 8th Edition*. Ames, Iowa: Iowa State University Press, 1989, ch. 6, pp. 83-106.

[32] S. Murthy, S. Kasif, and S. Salzberg, "A System for Induction of Oblique Decision Trees," *Journal of Artificial Intelligence Research*, vol. 2, pp. 1-32, 1994.

[33] E. Frank and R. Bouckaert, "Naive Bayes for Text Classification with Unbalanced Classes," in *Proceedings of the 10th European Conference on Principles and Practices of Knowledge Discovery in Databases*, Springer Berlin, 2006, pp. 503-510.

[34] S. Dzeroski, L. De Raedt, and K. Driessens, "Relational Reinforcement Learning," *Machine Learning*, vol. 43, pp. 7-52, 2001.

[35] M. Mori, "The Uncanny Valley," *Energy*, vol. 7, no. 4, pp. 33-35, 1970, Translated by Karl F. MacDorman and Takashi Minato.

```
//*********************************************************************
// safeExploreAction - Choose a random action based on
//                     the Safe e-Boltzmann exploration strategy.
//                     If possible, the expected risk of the randomly
//                     chosen action is below allowable thresholds.
//
//                     state - int representation of the current state
//                     return - a random action
//*********************************************************************
Action CacheSLQLearner::safeExploreAction(__int64 state) {
        Action action = ACTION_NONE;
        unsigned __int16 cacheIndex = this->getCacheIndex(state);
        double *probs = new double[NUM_ACTIONS];
        double safeSum = 0.0;
        double unsafeSum = 0.0;
        bool *safeAction = new bool[NUM_ACTIONS];
        int numUnsafe = 0;
        for (int a = 0; a < NUM_ACTIONS; a++) {
                safeAction[a] = true;
                probs[a] = this->getSafeBoltzNum(cacheIndex, a,
                                safeAction + a, false);
                if (probs[a] > 0.0) {
                        if (safeAction[a]) {
                                safeSum += probs[a];
                        } else {
                                unsafeSum += probs[a];
                        }
                }
                if (!safeAction[a]) {
                        numUnsafe++;
                }
        }
        double safeEpsilon = 0.0;
        if (numUnsafe == NUM_ACTIONS) {
                safeEpsilon = 1.0;
        } else if (numUnsafe > 0) {
                safeEpsilon = 1.0;
                for (int r = 0; r < 7; r++) {
                        double safeRisk = 0.0;
                        double unsafeRisk = 0.0;
                        for (int a = 0; a < NUM_ACTIONS; a++) {
                                if (safeAction[a] && probs[a] > 0.0
                                                && safeSum > 0.0) {
                                        safeRisk += (probs[a] *this->
                                                riskCache[r][cacheIndex][a])
                                                / safeSum;
                                } else if (probs[a] > 0.0
                                                && unsafeSum > 0.0) {
                                        unsafeRisk += (probs[a] *
                                                this->riskCache[r]
                                                [cacheIndex][a])/unsafeSum;
                                }
                        }
                        if (safeRisk > 0.0 && unsafeRisk > 0.0) {
                                double e = (this->omega[r]-safeRisk)
                                        / (unsafeRisk - safeRisk);
                                if (e < safeEpsilon && e >= 0.0) {
                                        safeEpsilon = e;
                                } else if (e < 0.0) {
                                        safeEpsilon = 0;
                                }
                        }
                }
                if (safeSum > 0.0 && safeEpsilon > unsafeSum /
                        (unsafeSum + safeSum)) {
                        safeEpsilon = unsafeSum / (unsafeSum + safeSum);
                }
        }
        double num = Random::uniform();
        double threshold = 0.0;
        for (int a = 0; a < NUM_ACTIONS; a++) {
                if (probs[a] >= 0.0) {
                        if (safeAction[a])          {
                                probs[a] /= safeSum;
                                probs[a] *= (1 - safeEpsilon);
                        } else {
                                probs[a] /= unsafeSum;
                                probs[a] *= safeEpsilon;
                        }
                        threshold += probs[a];
                        if (num <= threshold || threshold > 0.99999) {
                                action = (Action)a;
```

```
                                        break;
                                }
                        }
                }

        if (action == ACTION_NONE) {
                action = this->promptForAction();
                FILE *errorFile = fopen("actions.txt","a+");
                if (errorFile != NULL) {
                        char str[64];
                        sprintf(str, "%d\n", this->episode);
                        if (fwrite(str, 1, strlen(str), errorFile) < strlen(str)) {
                                cerr << endl << "ERROR-getErrors –
                                        unable to write to file: errors.txt: "
                                        << str << endl << endl;
                        }
                } else {
                        cerr << endl << "ERROR-getErrors - unable to
                                open file: errors.txt" << endl << endl;
                }
                fclose(errorFile);
        }

        delete probs;
        return action;
}



//*********************************************************************
// getSafeBoltzNum - Calculates the numerator of the
//                   Safe e-Boltzmann probability distribution for
//                   one action.
//
//                   cacheIndex - index of the state in the cache table
//                   action - the action to get the numerator of
//                   safeAction - a pointer to a bool which will be set
//                                to true if the action does not violate
//                                any constraints
//                   return - relative probability of action
//*********************************************************************
```

```
double CacheSLQLearner::getSafeBoltzNum(unsigned __int16 cacheIndex,
        int action, bool *safeAction) {
        double numerator = -1.0;
        double t = 0.5;//1.0;//0.05;;
        double temp = 0.0;
        *safeAction = true;
        for (int r = 0; r < this->numRisks; r++) {
                temp = (this->riskCache[r][cacheIndex][action]
                                - this->omega[r]) / this->omega[r];
                if (temp < 0.0) {
                        temp = 0.0;
                }
                if (this->riskCache[r][cacheIndex][action] > this->omega[r]) {
                        *safeAction = false;
                }
                t += this->k * temp;
        }
        if (valid && t <= 0.0) {
                cerr << endl << "ERROR-getSafeBolzNum - t <= 0.0 - " << t
                                << endl << endl;
                return -1;
        } else {
                double mod = sqrt((double)episode);
                if (mod < 1.0) {
                        mod = 1.0;
                }
                t /= mod;
                if (t < 0.0000001) {
                        t = 0.0000001;
                }
                double val = (this->qCache[cacheIndex][action])/t;
                numerator = exp(val);
        }
        return numerator;
}
```

```
//*********************************************************************
// greedyAction - Returns the highest value action based
//                on the ordering of value and risk functions.
//                If this data structure is set to use MC-ARTDP,
//                then a reverse 2nd lexigraphic ordering is used.
//                Otherwise, the SLRL ordering is used.
//
//                state - an int representation of the state
//                return - the greedy action
//*********************************************************************
Action CacheSLQLearner::greedyAction(__int64 state) {
        __int64 action = 0;
        unsigned __int16 cacheIndex = this->getCacheIndex(state);
        if (this->MCARTDP) {
                // MC-ARTDP Greedy Action
                list<Action> actions = actionList;
                for (int r = 0; r < this->numRisks; r++) {
                        if (actions.size() < 1) {
                                cerr << endl << "ERROR-greedyAction –
                                        Empty Action Queue" <<
                                        endl << endl;
                                return this->randomAction(state);
                        }
                        actions = this->maxActions(cacheIndex, actions, r);
                        if (actions.size() == 1) {
                                break;
                        }
                }
                if (actions.size() > 1) {
                        list<Action>::iterator iter = actions.begin();
                        Action maxAction = *iter;
                        while (++iter != actions.end()) {
                                if (this->qCache[cacheIndex][(int)maxAction]
                                        < this->qCache
                                        [cacheIndex][(int)*iter]) {
                                        maxAction = *iter;
                                }
                        }
                        return maxAction;
                } else if (actions.size() == 1) {
                        return actions.front();
                }
                return ACTION_NONE;
        } else {
                // SLR Greedy Action
                int maxAction = 0;
                int maxValid = -1;
                double maxQ = -10000000000000.0;
                for (int a = 0; a < NUM_ACTIONS; a++) {
                        int numValid = 0;
                        double Q = this->qCache[cacheIndex][a];
                        for (int r = 0; r < this->numRisks; r++) {
                                if (this->riskCache[r][cacheIndex][a]
                                                <= this->omega[r]) {
                                        numValid++;
                                } else {
                                        Q -= (this->riskCache[r][cacheIndex]
                                                        [a] - this->omega[r]);
                                }
                        }
                        if (numValid > maxValid || (numValid == maxValid
                                        && Q > maxQ))
                        {
                                maxAction = a;
                                maxValid = numValid;
                                maxQ = Q;
                        }
                }
                action = maxAction;
        }
        return (Action)action;
}

//*********************************************************************
// maxActions - Finds all of the maximum actions at a
//              specific risk level using MC-ARTDP's reverse
//              2nd lexigraphic ordering.
//
//              cacheIndex - the state's cache table entry index
//              actions - list of max actions at the previous risk level.
//              riskLevel - the level in the lexigraphic ordering
//              return - list of max actions at riskLevel
//*********************************************************************
```

```
list<Action> CacheSLQLearner::maxActions(unsigned __int16 cacheIndex,
                list<Action> actions, int riskLevel) {
        if (actions.size() < 1) {
                actions.clear();
                return actions;
        }
        if (riskLevel >= this->numRisks) {
                list<Action>::iterator iter = actions.begin();
                double max = this->qCache[cacheIndex][(int)*iter];
                while (++iter != actions.end()) {
                        if (this->qCache[cacheIndex][(int)*iter] > max) {
                                max = this->qCache[cacheIndex][(int)*iter];
                        }
                }
                iter = actions.begin();
                while (iter != actions.end()) {
                        if (this->qCache[cacheIndex][(int)*iter] < max) {
                                iter = actions.erase(iter);
                        } else {
                                iter++;
                        }
                }
                return actions;
        } else {
                list<Action>::iterator iter = actions.begin();
                double min = this->riskCache[riskLevel][cacheIndex]
                                                        [(int)*iter];
                while (++iter != actions.end()) {
                        if (this->riskCache[riskLevel][cacheIndex][(int)*iter]
                                        < min) {
                                min = this->riskCache[riskLevel][cacheIndex]
                                                                [(int)*iter];
                        }
                }
                iter = actions.begin();
                while (iter != actions.end()) {
                        if (this->riskCache[riskLevel][cacheIndex][(int)*iter]
                                > min) {
                                iter = actions.erase(iter);
                        } else {
                                iter++;
                        }
                }
                return actions;
        }

        return actions;
}

//*********************************************************************
// updateQLearning - Updates the current Q and Risk values
//                  based on observed reward and error signals
//
//                  currState - a pointer to the the current state object
//                  action - the action performed
//                  nextState - a pointer to the resulting state object
//                  reward - the numerical reward observed
//                  return - relative probability of action
//*********************************************************************
void CacheSLQLearner::updateQLearning(State *currState, Action action,
                State *nextState, double reward) {
        __int64 cState = this->getStateNum(currState);
        unsigned __int16 cStateI = this->getCacheIndex(cState);
        double *errors = this->getErrors(currState, action, nextState);
        __int64 nState = this->getStateNum(nextState);
        unsigned __int16 nStateI = this->getCacheIndex(nState);
        __int64 maxAction = this->greedyAction(nState);
        int act = (int)action;
        if (MCARTDP) {
                // MC-ARTDP updates
                list<Action> actions = this->actionList;
                for (int r = 0; r < this->numRisks; r++) {
                        actions = this->maxActions(nStateI, actions, r);
                        double min = errors[r] + this->gamma *
                                riskCache[r][nStateI][(int)actions.front()];
                        if (min < riskCache[r][cStateI][act]) {
                                riskCache[r][cStateI][act] = max;
                        }
                }
                actions = this->maxActions(nStateI, actions,
                        this->numRisks);
```

```
                qCache[cStateI][act] += (float)(this->alpha * (reward
                        + this->gamma * qCache[nStateI][(int)actions.front()]
                        - qCache[cStateI][act]));
        } else {
                // SLRL updates
                for (int r = 0; r < this->numRisks; r++) {
                        riskCache[r][cStateI][act] += (float)(this->alpha
                                * (errors[r] + this->gamma *
                                riskCache[r][nStateI][maxAction] –
                                riskCache[r][cStateI][act]));
                }
                qCache[cStateI][act] += (float)(this->alpha * (reward
                        + this->gamma * qCache[nStateI][maxAction] –
                        qCache[cStateI][act]));
        }
        this->cacheChanged[cStateI] = true;
        if (this->logAlpha) {
                double logEp = log((double)episode+1);
                this->alpha = logEp * logEp * logEp / ((double)episode);
                if (this->alpha > 0.3) {
                        this->alpha = 0.3;
                }
        }
        delete errors;
}

//*********************************************************************
// boltzmannAction - Returns the actions according to a boltzmann
//                distribution calculated using their values.
//
//                state - an int representation of the state
//                return - the greedy action
//*********************************************************************
Action CacheSLQLearner::boltzmannAction(__int64 state){
        Action action = ACTION_NONE;
        unsigned __int16 cacheIndex = this->getCacheIndex(state);
        double *probs = new double[NUM_ACTIONS];
        double sum = 0.0;
        double temp = 10.0 / (sqrt((double)episode+1.0));
        for (int a = 0; a < NUM_ACTIONS; a++) {
                probs[a] = exp(this->qCache[cacheIndex][a] / temp);
                if (probs[a] > 0.0) {
                        sum += probs[a];
                }
        }
        double num = Random::uniform();
        double threshold = 0.0;
        for (int a = 0; a < NUM_ACTIONS; a++) {
                if (probs[a] >= 0.0) {
                        probs[a] /= sum;
                        threshold += probs[a];
                        if (num <= threshold || threshold > 0.99999) {
                                action = (Action)a;
                                break;
                        }
                }
        }
        delete probs;
        return action;
}
```