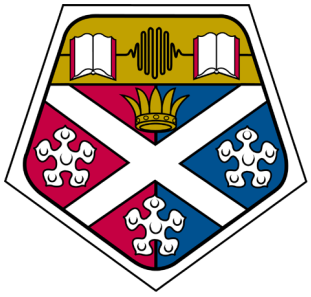
University of Strathclyde

Department of Computer and Information Sciences



An Introspective Approach to Robot Plan Execution

by

David Bell

A thesis presented in fulfilment of the requirements for the degree of

Doctor of Philosophy

2018

Declaration

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Signed:

Date:

Acknowledgements

I'd like to start by thanking my supervisors Professors' Maria Fox and Derek Long for starting me on this journey, and to the CIS department at Strathclyde University for supporting me through out. The department has been like a second home to me for many years, often with bouts of me spending more time there than my actual home, and the academic and support staff have been a great source of wisdom and assistance over the years.

To all the PhD students are researchers I've met along the way your advice, support, knowledge and friendship has been invaluable. The researchers in level 12 of the Livingstone Tower have a very special place in my heart, and Tommy, Alastair and Pete stand out as people who have made all these years bearable, more interesting, and a lot weirder than they otherwise would have been.

A very special thanks has to be extended to two of these researchers, Alan Lindsay and David Pattison, as without them there is a very real chance this thesis would never would have been completed. Their friendship has been instrumental to getting through some of the darker times on this journey, and despite going through a similarly prolonged writing up period, when they submitted rather than using their newfound free time to do something productive they instead chose to help me. Their advice, proof reading, and feedback has made this possible, and I appreciate it more than they will ever know.

I'd like to thank my Mum, Dad and brother for their love and support throughout this very long period of time. My Mum deservers a special mention here for never giving up on me, and helping me get to the end of this through a very specific type of nagging that only a mother can provide.

Finally I want to give a massive thanks to Suzanne. You met me as I was halfway through this, and therefore have only known me as a busy,

stressed, and constantly tired procrastinator. Despite this you have stuck around and provided a constant source of love, patience and distraction. For that I am eternally grateful.

Abstract

A three-tier architecture has been widely adopted for the task of controlling behaviour on board robots. The architecture acts to separate high-level deliberation over the best actions to take, from the low-level reactive task of executing those actions in a real dynamic and uncertain world. An important aspect that supports the efficiency of this architecture is that each tier acts with an increasingly abstract model. This allows the deliberation layer to construct plans for complex objectives in large domains. However when it comes to execution, the assumptions made to produce these plans can often contradict the realities of the environment they are required to operate in. This has resulted in the need for intelligent *plan execution systems*, which can monitor the execution of a plan and correct it if it an unexpected outcome is encountered.

An *introspective* approach is presented, which focuses on increasing the intelligence of a system's executive (the mediating layer between deliberation and execution). This is achieved through the use of introspective models that provide the executive with an understanding of how each of its actions should progress. An introspective execution framework is constructed around these models, extending their use from monitoring into control. This provides a method for the executive to identify anomalous behaviour within an action's execution, and intervene with a recovery action before it results in a system failure. Problems within an execution can then be overcome at an executive level, without the need to abandon an action or invoke higher-level systems.

The intelligence of a system's executive can then be further enhanced through the implementation of a stochastic controller that sits above the introspective framework. This provides additional reasoning over the executable actions, however it also provides the facility to add an element of learning to the system. By incorporating the observed outcomes from each action into the controller the executive can learn estimations of their

success, and use this to refine a plan into a more robust execution strategy. This provides the executive with an expectation how an execution should be progressing, and raises the level on introspection to plan execution.

This work is concerned with the ability to execute plans reliably on board a mobile robot operating in a dynamic environment and has therefore been implemented and deployed on a robot. Through several trials it is demonstrated that the introspective models can be used to predict action failure during action execution, which supports intervention and recovery. It is also shown that as the learning approach updates the stochastic controller, the system's behaviour changes adapting to its knowledge of the environment.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

An autonomous robot is one which can act on its own volition, able to perceive and interact with its environment without human assistance. As the field of robotics advances the application of autonomous robots to real world problems becomes a more feasible prospect. Robots are being deployed in environments that are prohibitive to humans, such as investigating the structural integrity of underground mines [67], gathering research data from under the world's oceans and lakes [64], and exploring the surface of new planets [69]. Autonomous drones and satellites can be used to survey landscapes and scout new areas providing valuable tools for both private and military application [23]. And robots have been developed with the purpose of interacting and operating alongside humans, acting as tour guides [85], helping with assisted care [65] or working as office couriers [6].

Each of these environments requires its own specific robot design, with careful positioning of sensors and actuators. However, the overall problem of controlling the robot is related for each of these environments. In particular, the sensor readings will provide a partial view of the world in which the robot exists and the actuators will, with some certainty, provide the option to manipulate the robot. The control problem involves selecting the appropriate behaviours for the current situation in order to progress towards the robot's goals. Determining the appropriate behaviour is a challenging task within a static environment and only becomes increasingly challenging in the uncertain environments that robots are often deployed.

1.1 Autonomous Control in Complex Environments

Selecting an appropriate behaviour with respect to an objective in a dynamic environment typically requires some form of deliberation. Deliberation requires the robot to consider not just its current state, but its possible future states in order to determine a sequence of actions that can realise its goals. Not all autonomous robots require deliberative capabilities. Robots operating in controlled, well modelled environments such as in manufacturing plants, or that only execute single repeatable tasks such as cleaning robots, can have their deliberation handled at the design stage and incorporated directly into their controller. However if a robot is required to execute a variety of tasks in freeform complex environments, it will require a deliberative component.

This has led to the hierarchal decomposition of robot control, constructing layers of abstraction over the uncertain and volatile sensor readings, and establishing symbolic layers, which are more amenable to reasoning at the level of goal achievement. The hierarchal approach allows the system to handle environmental interactions asynchronously, separating the strategic reasoning from the low level control. A typical approach separates the control across three communicating layers, and is the basis for many control architectures within robotics [28, 34, 72]. A traditional three layer architecture is comprised of a deliberation layer, an executive, and a reactive control/behavioural layer. An example of such an architecture can be seen in Figure 1.1.

At the top of this architecture sits the *deliberative layer*, concerned with the high level goals of the system. Operating at the highest level of abstraction, its role is to derive an execution strategy which can achieve these goals, typically within the confines of some resource constraints. AI task planning has proven a valuable tool for this, forming a core component of many robotic deliberation systems. Conventional planners use a symbolic representation to model the state of the robot and its current knowledge of the world. Actions consist of sets of preconditions which must be met before they can be executed, and effects which update the state of the world upon completion. The deliberation problem then becomes one of search, identifying sequences of actions which if applied will satisfy the goal conditions. The resulting sequence of actions from this is called a plan.

Once a plan, or other execution strategy has been identified, actions are dispatched to the *executive* for execution. The executive acts as mediator between the high level symbolic representation of the deliberative layer and the low level sensory motor func-

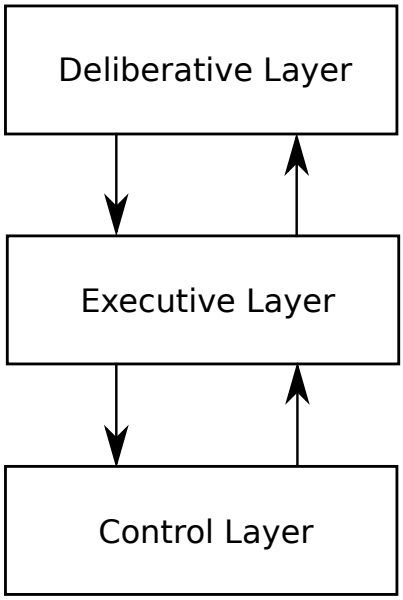


Figure 1.1: An example of a standard three tier architecture. The deliberative layer reasons about high level goals and passes actions down to the executive. The executive translates these actions into low level commands, and the control layer executes these commands.

tions of the behavioural layer. Given a high level action from above, the executive's role is to translate it into the corresponding low level behaviours, and initialise, invoke and manage these behaviours. Through the activation and synchronisation of these low level behaviours the executive can achieve more useful compound actions, and even overcome minor problems in the execution.

The low level behaviours themselves reside within the *control layer*. The control layer interfaces directly with the robot's sensors and actuators and is the lowest level of control in the architecture. It typically contains a library of hand coded behaviours, representing the building blocks of the robot's abilities. These behaviours tend to feature elements of reactive control, such as moving to a location while avoiding obstacles, to enable more robust execution.

1.2 Executing Under Uncertainty

A benefit of this layered approach is the ability for the robot to reason and operate at different levels of abstraction. It allows the system to deliberate over high level goals and problems while simultaneously executing actions and reacting to low level environmental obstacles.

A strong focus in robotics has always been on the deliberative layer, attempting to increase a robot's autonomy through more advanced reasoning. Advancements in AI planning have helped with this goal, allowing larger and more complex problems to be modelled and solved onboard a robot. However the ability for robots to execute these plans in the real world has not progressed at the same level as their ability to formulate them. When operating in dynamic environments, robots can be very prone to failure [13, 82].

A significant source of this failure can be attributed to the uncertainty a robot faces when executing an action in an unstructured environment. Nearly every aspect of this execution introduces a new element of uncertainty, which if unnoticed or unattended can accumulate and result in an execution failure. An important source of this uncertainty is the robot itself. The onboard sensors are a robot's sole means of perceiving its surroundings, and yet these sensors are notoriously noisy and limited. Unreliable sensor readings can result in localisation errors or misidentification of the robot's pose, which in turn can lead to the wrong control response being generated. Furthermore the actuators used to execute these controls can also be unreliable, introducing uncertainty into the outcome of each action. A broken motor, flat wheel, or loose gripper can lead to unexpected results during execution, and may even obstruct the real result from the system.

The environment itself can further compound these problems. Wet floors can cause wheels to slip during a navigation, and poor lighting conditions can interfere with a robot's camera as it tries to identify an object. Other people or agents acting in the area can also be troublesome, requiring critical objects for their own objectives or blocking intended paths, either of which can result in an action failing.

However while the dynamic nature of the environment can be disruptive to individual actions, it can also have a serious impact on the generation of reliable plans. Most planners use a static model to represent the robot's knowledge of the world, and deterministic representations for actions. Execution of these plans onboard the robot is

conditioned on the model accurately representing the real world, and the actions being able to reliably predict the evolution of the system through that world. One of the most critical sources of uncertainty occurs if this model becomes out of sync with the world around it, as it can lead to the generation of plans that are no longer relevant and very likely to fail. In a dynamic environment this can occur easily, with people and objects not being in their expected locations, doors changing from open to closed, or pathways being altered or obstructed.

1.3 Understanding Execution

To operate in dynamic environments deliberation is essential, and planning has proven an indispensable tool for aiding in this deliberation. However a top down approach where the majority of deliberation occurs at a high level using static, abstract models is not sufficient for environments with complex dynamics, and a greater level of reasoning needs to occur throughout the system. The pervasive nature of uncertainty that affects robots operating in these environments means that the control structures they use need to be able to anticipate and tolerate failure. One approach to this is to instil within a robot a better understanding of its own behaviour. If a robot can hold some expectation of how a plan or action should progress, it can then recognise if it strays from this expectation, and can take steps to bring the execution back in line. This is known as *introspection*, the ability to critically assess ones own behaviour.

This thesis proposes the use of introspective techniques to increase the intelligence of a robot's executive, and in turn the robustness of its execution. This work approaches the problem of execution failure from both a plan execution and action execution perspective. For action execution the executive is equipped with introspective models, learned from real sensor data, that provide it with an understanding and realistic expectation of how these actions should perform within their environment. If an action is found to be deviating from these expectations, before a failure occurs the executive can intervene and transition the system into a recovery action. The recovery action can be executed from within the current execution, and attempts to overcome the problem and leave the system in a better state from which to resume the initial action. This creates a method of action execution that can allow a greater range of problems to be tackled at the execution level without abandoning the current action or invoking higher level systems.

The introspective models allow the executive to identify failure and attempt to overcome it, however they also provide information that can be used to avoid it. If an action is repeatedly failing the system should be able to recognise this, and use the information to influence future decisions. To enable this a stochastic control structure is created within the executive which sits on top of the introspective models. These models monitor each action as it executes, and if a failure occurs they can update the controller before attempting to recover from it. This allows the executive to learn which actions are likely to succeed and which actions tend to fail, which in turn can be used to provide an expectation of how a plan is likely to progress in a real environment. The executive can then use a combination of the plan and its own experience to devise alternative execution strategies for sections of the plan that are likely to fail.

1.4 Thesis Statement

Executing high level task plans derived from static representations within dynamic environments is a difficult problem that often leads to execution failure. The thesis that this work defends is that:

"The application of introspective and experience based techniques to a robot's executive can extend the deliberative capabilities of a system and enable more robust execution of high level task plans in dynamic, unstructured domains."

1.5 Contributions

The contributions of this thesis are as follow :

- Development of an introspective framework which extends the use of behavioural models from monitoring into control. The use of this framework provides a novel method of action execution, where by the executive can reason about its own execution, determine if it is progressing according to expectations, and interject with a recovery action if it is not.
- An executive level stochastic controller that can provide additional reasoning over the introspective models, and learn directly from the robot's experience.

This experience can be used to learn realistic estimations of each action's performance, which can be used in conjunction with the plan to enable more intelligent decision making in relation to achieving the system's goals in dynamic environments.

- The implementation and evaluation of the above methods on a physical robot acting within dynamic environments. This represents the first time these introspective models have been implemented on a domain wide scale, and shows how with some additional control mechanisms they can be beneficial in real scenarios.

1.6 Thesis Overview

The remainder of this thesis is structured as follows. Chapter 2 delves deeper into the problem of executing under uncertainty, and provides an overview of the current techniques and technologies used to overcome this problem.

Chapter 3 opens with a discussion of introspection within the context of robotics, and provides the necessary background information on the introspective models used within this work. These models are used to present an introspective domain, and their use beyond monitoring and into control is explored, resulting in an initial introspective framework designed to overcome execution failure.

This framework is extended in Chapter 4 to include a stochastic controller that sits over the models and provides a greater level of control over their execution, as well as the ability to learn from them. The relationship between this controller and the planner is defined, and the introspective executive is presented.

Chapter 5 evaluates both the failure recovery and introspective executive empirically on a mobile robot through dynamic environments, and presents the results.

Finally, Chapter 6 concludes the thesis with a further discussion and final summary of the work presented.

CHAPTER 2

BACKGROUND

Developing a robot control architecture that can operate in an unstructured environment is a difficult task. Deliberation needs to occur in order to reason about goals and devise execution strategies, however this is often required to rely on incomplete or inaccurate world models. The strategies produced by the deliberator, often in the form of a plan, then need to be executed onboard the robot. Any missing or inconsistent information in the planner's reasoning model can result in an unexpected situation arising when one of its actions are applied in the real world. In order to be autonomous, a robot must be able to recognise when one of these actions is no longer performing as expected or its execution has resulted in a failure state. Furthermore the robot should be able to act on this information, and attempt to overcome the problem in order to continue to achieve its goals.

This chapter explores the core concepts surrounding executing plans on board mobile robots in dynamic, unstructured environments, and sets up the context for the rest of this work. The chapter begins with an explanation of execution monitoring, and an overview of techniques designed to determine if the execution is progressing as normal. This is followed by a review of some of the methods mobile robots have used to overcome failures within their execution once they have been detected in Section 2.2. A brief outline of the literature covering robotic learning is presented in Section 2.3, followed by an overview of a selection of dynamic execution architectures and the different aspects of the problem they focus on in Section 2.4. Finally the chapter concludes with a summary and discussion of the presented work in Section 2.5.

2.1 Recognising Failure

Plan-based deliberators need to reason over large and complex domains in order to derive a plan that can satisfy a set of goal conditions. As such, in order to reduce the scope of the problem simplifying assumptions are often employed, such as assuming the domain is fully observable, that actions have deterministic outcomes, or that the robot is currently the only agent effecting change in the environment. These assumptions rarely hold in the real world, therefore robotic architectures need to be able to enact these plans while coping with the uncertainty and environmental dynamics that may interfere with the execution. This has led to execution monitoring becoming a fundamental component of most plan-based execution architectures.

At its most basic, execution monitoring aims to compare the predicted state of the word versus the observed state and identify inconsistencies. In task planning actions are represented at an abstract, symbolic level and modelled via a series of preconditions that need met before an action is available for execution, and effects which update the system's state upon completion. This allows the deliberative component to predict the effect of executing an action, and to limit the choice of subsequent actions to those which have preconditions that align with this prediction. If given an action such as:

```
NAVIGATE (ROBOT WP1 WP2)
```

The robot's plan executor, or executive, will take this action, decompose it into a set of low level commands and begin executing it. Upon completion the robot expects to be situated at the location `WP2` represents, and in turn the following action will be predicated on the system starting from this point. If however during the execution the system tracks the action's progress, in this case through dead reckoning or some other metric, and observes that the robot's location does not match `WP2`'s, it can infer the action has terminated in an unexpected manner. An overview of a standard execution monitoring system can be observed in Figure 2.1. By being able to recognise this failure has occurred the system can then attempt to classify and recover from it, but more importantly it can stop it from executing the next action from an unsuitable, and possibly dangerous configuration.

Execution monitoring is a lively field within robotics, with a wide spectrum of techniques being developed and deployed at all levels of a robotic platform. Preconditions, invariants, and logic-based formalisms can be used to ensure plans are progressing correctly, while both model-based and model free methods can check individual actions.

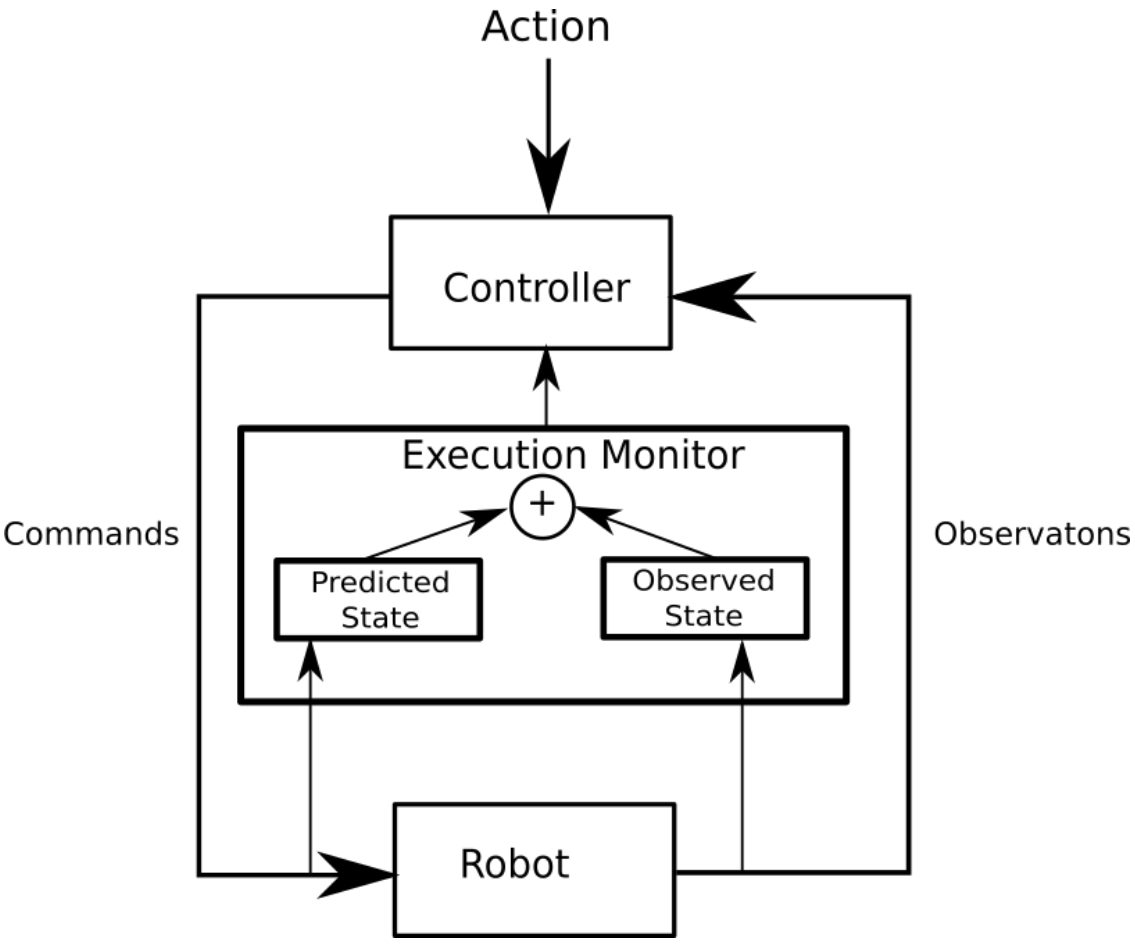


Figure 2.1: A high level representation of a traditional execution monitoring system. Each action is associated with a set of effects which allow the system to estimate the state it will be in after execution. Upon completion it can check the observed state. If the predicted and observed states are different, it can be inferred a problem has occurred.

There is even a wide range of monitoring tools to verify the feedback from individual sensors and actuators, designed to identify mis-calibrations or anomalies that may have adverse effects on the system's goals. This section presents an overview of a selection of execution monitoring techniques, however for a for a wider review of the literature see [75].

One of the first autonomous robots, Shakey, employed the prediction checking form of execution monitoring. Using the STRIPS planner [25], Shakey could generate plans that moved boxes from one room to another to match a given goal configuration. The execution and monitoring of these plans was handled by the PLANEX system [24]. PLANEX stores the produced plans in a triangle table, with each row in the table

corresponding to a particular action. Cells to the left of an action represent its pre-conditions, and the cell below an action store its predicted outcome. Actions are then tied directly to the procedures which execute them, and upon completion the resulting system state is compared against the triangle table for the predicted outcome. This not only allows errors to be detected, but also provides a means for the system to update its state-based on any exogenous events or opportunities that arise, though both of these features rely on the assumption of full observability for the state updates.

NASA's Remote Agent architecture [69] is one of the more famous model-based execution monitoring systems. Utilising a comprehensive and granular model of the spacecrafts's components, Livingstone [87] performs both the execution and monitoring of actions in order to transition the system into a desired goal configuration. Given a command, the Mode Identification and Recovery (MIR) component can use the spacecraft's model to predict the state of the system, and compare the current sensor values against the predicted ones. If a deviation is found it can then isolate the fault, and attempt to derive what the current configuration is.

Pettersson [76] approached execution monitoring from the perspective that a precise predictive model of a robot's behaviour may not always be available, either due to the complexity of the system or the result of emergent behaviour produced by the interaction of overlapping sub-behaviours. Rather than than trying to capture these complexities in a formal model, a model free execution monitoring variant was created using machine learning techniques to derive patterns of failure and success from historic execution data, and use them to train a neural net. This can then be applied online for future executions, to identify and classify failures directly from the sensor data.

Plan invariants are used in [33], to ensure the execution is always feasible. These invariants extend the typical plan domain by using a logical formula to describe conditions that need to hold throughout an entire plan execution. An example would be that if the final action a robot has to perform in a plan involves it passing through a doorway, an invariant would be set stipulating that the door needs to stay open throughout the duration of the execution. If at any point during the execution the robot's sensors detect the door has become closed, the invariant no longer holds and the plan can be terminated early. These invariants provide the robot with a means to monitor other agent's actions or exogenous events that might effect its own execution, without having to model or compensate for all possible interactions.

In [23], the monitoring is interwoven with planning and tested on board an UAV. A

planner, based on Temporal Action Logic (TAL [22]), uses forward chaining search to derive a plan that satisfies a set of control formulas. Subsequent execution monitoring formulas are generated from the domain's action models using preconditions, effects, and temporal constraints, and the produced plan using causal links between actions. A formula progression algorithm then checks for violations of these monitoring conditions online during plan execution. The integration of monitoring and planning through the use of a unified model allows monitoring conditions to be produced from the plan itself, and aspects of the execution reasoned about at a higher level. However due to the monitoring formulas being generated automatically it does require a user to tag which ones are relevant to the execution.

Lastly, the work in [10] takes a different approach to execution monitoring, by attempting to monitor the implicit effects of an action through semantic knowledge. If a robot executes a navigation command to move to a specific waypoint, traditional execution monitoring systems will confirm they have arrived successfully by examining the observed position of the robot within the environment. Bouguerra *et al.* create a supplementary system that utilises description logic to represent concepts within the domain, and derive additional expectations for each action. If the destination waypoint is reached, this system can query what class of room the destination belongs to, then check for evidence in the environment to support this. As an example if the robot expects to arrive in a living room, it has an expectation of the type of furniture that should be located within this room, such as couches or chairs. If a sink is discovered, this is inconsistent with those expectations and an error is generated.

2.2 Responding to Failure

Detecting the occurrence of a failure is only the first step in successfully executing a plan in a dynamic environment. The next step then has to act upon this failure, and adapt the robot's behaviour in response to the problem. Creating an appropriate response can be difficult, due to the multitude of problems that can arise during an execution. A hardware fault, such as a broken wheel, requires a very different set of actions to a software problem, such as a localisation routine failing, which in turn is different to an environmental problem such an object being moved from its location.

Often the burden of crafting the response to these problems is left to the engineer, requiring them to predict all the possible failures a robot may encounter at the design

stage. This can become a near impossible problem in a truly open and unstructured environment, compounded by the fact the often the most severe failures are uncommon and likely to be unaccounted for. It is then of the utmost importance that if a system can not overcome a problem, it can at least transition itself into a safe state.

This section will provide an overview of some of the more common techniques used in autonomous robots to continue to execute their plans with respect to failure. For the purpose of this overview these have been separated into high level response strategies, which affect the entire execution, and mid level responses which attempt to overcome the failure at a more localised level. In reality systems often employ a combination of the two.

2.2.1 High Level Response Strategies

A high level response to failure is one which involves the system having to consider the execution as a whole again, taking into consideration any new information brought forward by the failure. This type of strategy allows the system to analyse the severity of the problem with respect to the current goals, and to construct an appropriate solution based on this. For example a simple failure may just need a single action to recover the execution and continue to the goal, where as a more complex failure may result in the goals themselves needing to be reassessed due to aspects of them no longer being possible.

Replanning

Replanning is one of the most common techniques used to handle failures within plan-based control architectures. Given an initial state and a domain model, a planner will generate a sequence of actions which will transition the system towards a specified goal. In classical planning these actions are deterministic, and the assumption is made that the model associated with each action can accurately predict its effects. This allows large problems with complex numeric constraints to be solved, however it can be computationally expensive.

When executing in a dynamic environment the planner's optimistic view of both its own predictive ability and the deterministic nature of its actions can lead to fragile plans which do not align with the world. If during the execution of one of these actions

an unexpected situation or failure is detected, the system can attempt to classify it, and pass it back up through the architecture to be deliberated upon. This can involve trying to estimate the state of the system after the failure has occurred, or it can involve instigating a recovery procedure in an attempt to roll the system back to the last known plan state.

Once a new state has been determined the task planner is called in order to compute a new plan from this current state to the goal. This involves completely discarding the old plan and searching the entire domain for a new one, however it can allow the system to take into consideration any aspects of the goal that have already been achieved and to derive a solution that supports any current constraints. If a new plan is found it can be dispatched for execution, otherwise the system can be put into a safe state.

Replanning was utilised within the first plan-based robot, SHAKEY, to recover from failures. If during the execution a problem was encountered that could not be overcome with the initial plan, PLANEX would invoke STRIPS to calculate a new one. The LAAS architecture [1], uses a temporal planner IxTeT [36], to solve domains with durative actions. Failures are handled with via a layered approach, initially using hand crafted procedures and then plan repair, however if a solution can not be found within the allotted time a full replan is commissioned to derive a solution.

Replanning is an effective online execution recovery technique as it allows the system to perform a comprehensive analysis of the domain with respect to the current failure, and outline a new strategy that can continue towards the goal. However as domains grow large it can be slow to re-examine the full state space, and become prohibitive to respond quickly to errors within the execution. Furthermore it relies on a state estimator to be able to reliably and accurately predict the current state of the system after a failure, otherwise the newly created plan itself may end in failure. Similarly if the problem was caused by the model itself, for example by being out of sync with the environment, a new plan is unlikely to help.

Plan Repair

Rather than disregarding the initial plan when a failure occurs, a plan repair aims to correct the problem while continuing to execute part of the plan or sub-plan that is unaffected by the current issue. This facilitates the reuse of the original plan and can be less disruptive to the execution than replanning entirely. However attempting to repair

a plan while continuing to execute does necessitate its own considerations. The re-planned section needs to be carefully integrated back into the rest of the plan, ensuring that the portion of the plan still being executed does not invalidate the preconditions or invariants of the repaired section. Similarly the repaired section must be constructed as to not invalidate any portion of the future plan or sub-plans.

IxTeT-eXeC [60] uses plan repair in conjunction with a partial order planner. Plans are created with an element of flexibility built into them, which the system attempts to exploit by continuing to execute a valid portion of the plan while issuing a repair. The repair tries to produce a solution that is close to the original plan, however if it fails or goes over a given time limit, a full replan is issued. CASPER [14], an execution system deployed on board rovers and satellites, uses plan repair to enable continuous planning. The goals and state of the world can update incrementally during the execution, and the planner attempts to maintain a satisficing plan. If any updates result in conflict with the current plan, iterative repair techniques are employed to resolve them.

Despite not requiring the derivation of an entirely new solution, plan repair can be as difficult a problem as replanning [71]. However, this can be balanced out by the ability to continue to act and enhanced plan stability [31].

Contingency Planning

A different approach to coping with unexpected situations at a high level is to reason about them at the initial planning stage, rather than at the time of failure. This is the theory behind contingency planning, which attempts to generate plans with the appropriate responses encoded into them to deal with problems that can arise during an execution. These plans are created in a tree like structure, with branches off the main plan representing divergent solutions. This involves the system being able to both identify beforehand points of possible failure, and accurately recognise its state when it arrives at these branches so the appropriate action can be selected.

Full scale contingent planning can be problematic, as the size of the plan can grow exponentially with the number of contingencies. This can lead to large and complex plans which are difficult to execute. Just In Case [19], a planning system used on board rovers, attempts to combat this by limiting the contingencies to the most likely possible failure points within a plan. An initial plan is generated to achieve the system's goals, then approximate utilities are calculated for the possible branches off this. Only the

branches which are expected to maximise the utility of the plan are selected, and a contingency is inserted.

Petrick *et al.* [74] use a contingent planner in conjunction with sensing techniques to reason at the knowledge level in robotic domains. This represents action's preconditions as facts that must be true of the planner's knowledge state, and models effects as updates to reflect what the planner now knows. When constructing a plan the system can add in sensing actions to gather more knowledge, and then insert branches to act upon the result at run time. As an example if the robot is to clear a table of empty bottles and realises it has no knowledge for one of the object's state, it can add in a sensing action and a contingency for whether it is empty or not. At run time if that object is observed to be empty it can be cleared from the table, otherwise it is left alone.

Probabilistic Approaches

Probabilistic approaches move away from relying on plans of sequential, deterministic actions, and instead attempt to model the inherent stochastic nature of the environment. This creates actions which can have multiple outcomes, allowing failures or other unexpected situations to be captured within the system and reasoned with. The execution strategy is then typically represented by some form of policy, a mapping between states and actions which represents all the situations a robot can encounter.

One of the more common formalisms used in probabilistic approaches is a Markov Decision Process (MDP) [77]. MDPs provide a means to reason under uncertainty, utilising a transition function to represent the probabilities of how a system can evolve through each action choice. Costs and rewards can then be used to incentivise or discourage certain states or actions, and a policy can be created which attempts to maximise the utility based on these.

In [90] MDPs are used to control the actions of a planetary rover. The approach attempts to exploit the structure of the problem, and uses hierarchal reinforcement learning to generate a policy. The policy itself is computed largely offline in order to ease the burden on the rover's limited computational power. ROBEL [66] learns a MDP controller for selecting the appropriate set of skills to execute a task. Skills represent different groupings of sensory motor functions that can be used to achieve a specific goal, and different skills can produce the same results via different methods. The MDP controller produces a policy which maps the choice of skill to the current state of the

system.

Lacerda *et al.* [58] use co-safe linear temporal logic (LTL) to specify tasks a robot can execute using an MDP controller. LTL is an extension of propositional logic, and provides an effective way of defining robotic tasks while allowing concepts such as goal ordering to be captured. These tasks are then used in conjunction with an MDP to produce a cost optimal policy the robot can execute. An interesting part of this work is how new LTL tasks can be dispatched and incorporated into the current execution through dynamic recalculation of the policy onboard the robot. This is demonstrated in an office environment, using the MDP to capture the ambiguity of executing navigations actions in the real world, and allowing new tasks to be pushed to the robot as it attempts to satisfy its initial goals.

Building upon this work, [59] examines how using LTL specifications with the inclusion of a progression metric can generate policies for problems which may not be completely satisfiable. An example of this would be a robot having the goal to check several rooms in a building. If one or more of these rooms are locked, the robot cannot complete its task. However the addition of a measure of progression into the reward structure of the MDP allows the system to derive a policy which attempts to, in descending order, maximise the probability of achieving the goal, maximise the progress towards the goal, and minimise the cost of completion. This is evaluated in a navigation environment, where encountering a problem such as a closed door causes the system to move on to the next achievable part of its goal.

MDPs are an effective way to reason about uncertainty during execution, however they suffer from problems with scalability. The need to enumerate the entire domain means that as the number of state variables increase, the model can become slower and more difficult to solve. The addition of numeric variables, such as modelling battery power or resource consumption, only exasperates this problem.

Another common probabilistic approach is an extension to MDPs to operate under partial observability, called the Partially Observable MDP (POMDP)[51]. POMDP's model not just the uncertainty involved in a robot's action, but the uncertainty it has over its own state. Rather that having the state of the world known and updated after each action, the robot's state is modelled as a belief distribution over a set of possible states. Observations from the world can then be used to increase this belief.

POMDPs represent a desirable control model for robotics, due to their ability to cap-

ture more of the uncertainty an autonomous robot has to contend with. The partial observability aligns well with a robot's limited view of the world in the face of noisy sensors and unknown agents. However they can become quickly intractable in even relatively small state spaces, leading to their limited use as a control structure.

2.2.2 Mid Level Response Strategies

Mid level response strategies tend to allow a system to approach an execution failure from a more granular perspective. Rather than reduce an action to a high level symbolic representation which can register as completing successfully or not, mid level responses can look further into the specific behaviours or modalities which comprise an action and attempt to find solution at this level. This can allow actions to achieve their intended outcomes, even if it is not in the intended manner.

Procedural Execution Languages

Procedural execution languages provide the ability to represent and reason about actions at a level between the high level plan steps and the low level robot commands. It allows hand written procedures to be created which can decompose an action into its main components and handle functions such as instantiation, refinement, monitoring and exception handling. By providing a language which is more akin to a traditional programming language, with conditional statements and loops, it can allow the fast creation and deployment of these procedures onto a given system.

Firby's Reactive Action Packages (RAP) [27] is an early example of this type of system, designed to merge planning and execution. Libraries of RAPs are developed, with each individual RAP representing a selection of methods or sub plans for achieving a particular task. A RAP can then be associated with a plan action, and at execution the appropriate methods or sub plan can be invoked to achieve a goal based on the current context. Failures can be handled by changing between the different available methods, and if a previous attempt does not complete a new method can be selected.

A similar approach is taken with the Procedural Reasoning System (PRS) [48]. This also relies of the creation of a library of procedures, which are developed to achieve specific goals or react to certain situations. Each procedure contains a body describing

the steps needed to achieve the given goal, and context for when the procedure is applicable. Tests are used within the procedures to monitor conditions that are necessary for continued execution, or to fork the sequence of events based on their outcome. PRS has been developed and extended to accommodate a wide variety of applications and platforms [21, 70, 47].

Finally, Simmons creates a Task Description Language (TDL) [81] based on c++ for task level decomposition and control. TDL uses task trees to represent high level actions, with nodes on the tree corresponding to goals, subtasks, monitoring tasks and exception handlers. These trees also encode relationships between these nodes and synchronisation constraints. When a failure is detected a reason is generated from a predefined selection, and this reason is passed up through the tree until an exception handler is found which can appropriately respond to it.

Procedural languages allow high level tasks to be broken down to a more granular level, while still maintaining a level of abstraction needed for reasoning. Their main drawback is they need to be handcrafted by an engineer and can therefore only respond to issues they have predicted and accounted for.

2.3 Learning from Failure

Improving a robot's ability to execute under dynamic conditions can be achieved in a multitude of ways. The work discussed so far focuses on methods to identify and recover from failures, increasing the robot's robustness during execution. However it is also possible to use the robot's interactions with its environment to improve its ability to execute, by capturing its experiences and learning from them. This section provides a brief overview of some of the techniques used to enhance a robot's execution through learning strategies. Once again the primary focus is on plan-based systems, so the work surveyed relies on the existence of an initial control model. Learning is then either used to improve the control or to improve the actions the controller executes. Learning control strategies without a model or through human interaction are important topics outside the scope of this section, however further reading can be found in [52] and [3] respectively.

One of the main areas of application in robotic learning is to improve the robot's ability to execute its actions. At a deliberative level actions tend to be regarded as high level

abstract tasks which can be executed in various contexts and still expected to achieve the desired results. Often the person implementing the action will work from the same assumption, and actions are designed to be generic in order to operate in all situations. However in reality the same action executed in a different context can require different behaviour. A robot picking up a cup can vary from one execution to another, based on the surface the cup is resting on, the angle of the cup, the angle of the robot and how cluttered the environment is. Similarly a navigation in different areas of an environment may encounter different conditions, and something as simple as knowing to slow down at certain locations may improve the success rate of the action. Understanding this a priori and incorporating it into the action's model is a difficult and complex task, however it is these types of situations learning can assist with.

The previous point is addressed directly in [6] with XFRMLearn. In order to improve navigation throughout the environment they attempt to learn Structured Reactive Navigation Plans (SRNP) [5] from execution data. A SRNP implements a default behaviour for a navigation task, but can contain parameterized sub plans that can be activated to overwrite this behaviour. XFLRMLearn attempts to identify flaws in the execution through examining features such as instances where the robot slowed down unexpectedly, when it's velocity actually increased, or when it stopped during an action. These are correlated with environmental features, and sub-plans are inserted with rules to improve the navigation. This can mean a specific corridor is tagged as narrow, and upon entering it the robot's navigation procedure lowers the velocity and switches to a laser-based method to avoid sonar anomalies.

Infantes *et al.* [45] attempt to improve a robot's ability to execute actions by learning a model based on the system's observed behaviour, then using this model to tune the underlying parameters. A Dynamic Bayesian Network (DBN) [18] is learned from execution data through a mixture of manual identification of states and state variables, and a modified version of the Expectation Maximisation [20] algorithm to automatically learn conditional probability distributions. This provides a probabilistic model which can be used to monitor future executions, however by converting the DBN into a Dynamic Decision Network (DDN) [89] the underlying state variables can be modelled as control variables. By associating utilities with desired behaviours from the robot these control variables can be optimised to improve the execution of an action.

Learning can also be used to enhance a robot's controller and improve its ability to make decisions under uncertainty. Often the high level models used to support deliber-

ation do not capture the richness of an environment, or the reality of the robot executing within this environment. By utilising a robot's experiences, some of the gaps within these models can be bridged.

An early example of this can be found in the ROGUE [40] architecture. ROGUE records its execution trace as it performs a task. Upon completion it creates a set of events, which are defined as learning opportunities within the execution that relate to the success or failure of the task. Features are associated with these events and derived from the execution trace. These features provide the additional knowledge needed to describe the events, as well as a method to discriminate between them, and can represent high level concepts such as time of day or location. The learning component then associates a cost with an event for use in evaluation, and regression trees are applied to extract control rules. These rules can then be used by the planner to inform its search, and allow the system to make decisions such as avoiding a location at a specific time of the day due to crowds, or holding off on delivering a package until a after a specific time in order to allow for a person to arrive.

Another approach to learning improved control is presented in PELA (Planning, Execution, and Learning Architecture) [50], where the system starts with a standard off the shelf classical planner and a typical, deterministic PDDL action model. As it executes a plan, each action is monitored and associated with a label depending on its outcome. An action which completes as expected is labelled a success, one which fails but can be recovered with a replan is labelled a failure, and if an action fails in a manner found to be unrecoverable, it is labelled as a dead end. Relational decision trees are then used to learn the outcomes of each action, and provide a means of predicting future performance. The action model used by the planner can then be updated using these decision trees, either by enhancing it with a metric for plan fragility, or by converting the model into probabilistic PDDL [88].

2.4 Dynamic Execution Frameworks

While some of the work discussed above is presented as stand alone techniques, much of it is developed as part of a broader execution framework. These frameworks combine multiple different techniques in order to improve a robot's execution when operating within a dynamic domain. The uncertainty that surrounds executing in the real world, coupled with the range of tasks robot's are being applied to, means that many

of these frameworks specialise on different aspects of dynamic execution. This section will discuss a selection of these frameworks and the problems that they tackle.

The work in [49] focuses on creating a framework that can support a service robot interacting with people as part of its operation. This is deployed on robots operating in environments such as shopping malls or offices, and aims to assist users by performing a variety of tasks or by helping them reach selected locations. The addition of humans into the execution loop increases the uncertainty of operation significantly, as people can walk away mid task, change their mind on what should be achieved, or attempt to interrupt current behaviours. In this work tasks are defined using Progressive Reasoning Units [68], which represents an execution as a progression through a series of layers, where each layer may contain multiple modules which can alter the state of the system. These PRUs can be automatically transformed into an MDP, which can then be solved using standard techniques to produce a policy. This policy is then converted in to Petri Net Plan [91] and augmented with execution rules. These execution rules allow configurations of state variables to be associated with recovery procedures.

This approach separates the variables used for planning from the variables used for execution in order to reduce the complexity of both, and make task definition more straightforward for engineers. The framework can then takes these tasks and expand upon them using the process above into a more detailed representation for execution. This has parallels to the work in this thesis, which attempts to capture some of the complexity and uncertainty of execution within action models, helping to preserve the abstraction within the high level deliberative functions. Furthermore the output of the deliberative function in both cases is then used as the basis for augmentation to improve the robustness of execution.

Hofmann *et al.* focus on a different aspect of dynamic execution in [43]. This work looks at the problem of deliberating in an uncertain environment where the robot has to operate with incomplete knowledge. To achieve this the authors use a GOLOG [61] based deliberator which they extend to accommodate continual planning and assertions. Continual planning interweaves planning and execution through the use of sensing actions, while assertions allow placeholder actions to be used to represent sub plans which can achieve specific preconditions without specifying how. The GOLOG is translated into PDDL and solved using an ensemble of planners in order to significantly reduce deliberation time. This system is trialled on a physical robot which has to clear a table and determine on a case by case basis whether cups are clean and go in

the cupboard, or dirty and need to be placed in the dishwasher.

The STRANDS project [42] looks at a framework for supporting long term autonomy in indoor environments. Moving beyond executing single or multiple tasks in a dynamic domain, this work creates an architecture that can allow a robot to operate autonomously for a weeks at a time in human environments. To help with this longevity each task the robot will perform is scheduled to ensure it can be completed within the relevant resource allowance, including maintenance operations such as recharging the robot itself. Another important focus is on making sure navigation is robust and reliable, an aspect of the execution they have identified as a limiting factor in long term execution.

Part of the work to achieve this navigational robustness is through the use of recovery behaviours which are triggered when a fault is detected. These allow the system to iterate over a selection of recovery strategies in an attempt to overcome the issue and allow the navigation to proceed. However, as well as trying to overcome navigation faults, this system also tries to learn from them, and incorporate this learning back into the navigation system to enable it to select paths in subsequent executions which are more likely to succeed. The work in this thesis approaches execution failures from a similar perspective, developing a framework that can monitor executions and apply recovery actions in Chapter 3, and expanding it into a system that can then learn from failure to refine action selection with respect to a goal in Chapter 4.

2.5 Discussion

It is essential for a system acting autonomously in the real world to be able to recognise and respond to failures within its execution. The unstructured nature of these complex environments leads to vast amounts of uncertainty, which can be prohibitive to codify into a model or controller. To overcome this a system needs to be robust in its execution, and have the ability to reason about and adapt to unexpected situations or failures.

The first part of being able to reason about failure is being able to recognise it. The literature is rich with techniques for detecting unexpected situations, a selection of which are summarised at the start of this chapter. These range from monitoring individual sensors, to constructing large complex models which capture the interactions

of entire systems. The choice of monitoring depends on the application and platform, however it is crucial to be able to identify when the predicated state of the world no longer aligns with the real one.

However it is what to do after the identification of a failure that this thesis focuses on. There are many techniques which can be employed to recover from an unexpected situation. Passing up the problem to the high level planner is among the most common, to allow the system to initiate a replan or a repair. This allows the integrity of the execution to be maintained, as due to considering the full problem again any solution is likely to fall within the constraints of the system. These techniques tend to be applied with deterministic sequential plans, which ignore the stochastic elements of the environment and return to the planner whenever an unpredicted outcome arises. They can also be slow to find a solution, especially in more complex domains.

Contingency planning and probabilistic approaches attempt, at different levels, to model the stochastic nature of the environment within the control mechanisms. Contingency planning tends to focus on key points of uncertainty and builds alternative solutions into the plan, such that at execution time either through an observation or encountering an error, alternative actions can be chosen. Probabilistic approaches try to capture all possible states the robot can encounter, and derive a policy which can provide an action for any outcome. Both these approaches can be effective for reasoning under uncertainty, however as the complexity of the state-space grows their solutions can become unwieldy.

At the executive level procedural approaches allow the system to decompose an action into its component parts and reason at a level closer to the actual execution. This can consider multiple approaches that can achieve an action's desired outcome, and encode responses to known errors. However these techniques rely on hand coding the procedures and anticipating the possible failures that may occur, and do not take into consideration the larger scope of the execution.

There is no one solution that can be used to ensure the successful execution of autonomous robots in unstructured domains. Instead it requires multiple approaches operating together, possibly at different levels of abstraction, to ensure the system is progressing towards its goals. This requires communication between the various components and system-wide deliberation.

The concept of system-wide deliberation is important for autonomous robots, as often

deliberation can be regarded as single high level function equivalent to task planning. However in order for a robot to effectively operate under uncertain conditions deliberation has to extend beyond simply planning, and instead bring together a much richer set of functions to support its reasoning. In [46] deliberation is defined to encompass a spectrum of tasks such as planning, acting, monitoring, observing, learning and goal reasoning.

This view is expanded further in a position paper by Ghallab *et al.* [37], where they argue that not only should deliberation not be reduced to planning, but research should move away from concentrating on planners and instead refocus on actors. In this context actors are not simply plan executors, but instead deliberative components that bring together various abstract representations and reasoning techniques, which can include planners, to achieve a task. The principles outlined that define a deliberative actor are hierarchically organised deliberation in which actions can be refined and reasoned with at different levels of abstraction, and continual online planning to ensure flexible execution under dynamic conditions.

The rest of this thesis describes a robot control architecture which supports intelligent plan execution in dynamic environments. The system presented aligns with some of the core concepts laid out in [46] and [37], by focusing on developing the intelligence of the executive, as opposed to the planner. The implementation of additional reasoning at this level moves the system away from a traditional top down approach with a single point of deliberation, to a more integrated approach with deliberation occurring throughout the platform at different levels of abstraction. The presented system brings together elements of planning, acting, monitoring and learning to better achieve the system's high level goals.

What differentiates the work in this thesis from the plan execution architectures discussed above is that rather than trying to detect a failure based on specific, predefined rules, such as preconditions, invariants or sensor readings, or updating the execution strategy based on an event invalidating a plan or an action, the following work strives to provide the robot with an understanding of its own behaviour. If a system can understand its behaviour, it can then start to reason about it. This is known as introspection, and is an important concept related to intelligence. The next chapter will introduce the concepts behind introspection, and begin to explore how they can be used to support more robust execution.

CHAPTER 3

AN INTROSPECTIVE ACTION EXECUTION FRAMEWORK

Providing a robot with an introspective capability is an important step on the path to autonomy. An introspective system should be able to reason over its own behaviour and assess how well each of its actions are progressing, determining if the overall system is behaving as expected. If one of the actions is found to be executing in an abnormal manner then the system should be able to recognise this and take the appropriate actions that can start to move the execution back on course. Fox *et al* [30] began to look at this goal and aimed to provide an introspective component by creating models for individual tasks a robot could execute, allowing the system access to its own underlying behaviour.

The work in this thesis builds upon this idea, and uses these models as a means to enhance the executive by not only allowing it to monitor its own behaviour, but to recognise when this behaviour is becoming abnormal and interject at an appropriate time with a new action to prevent a serious failure. This creates an executive that can intelligently respond to disruptions within its execution, and more robustly execute actions in dynamic environments without the need to abandon or alter its current high level plan.

This chapter opens in Section 3.1 with an introduction to introspection in the context of robotics, and outlines some introspective approaches to managing failure. This is

followed by a discussion of how introspection can be applied to a typical robot architecture in Section 3.2. Section 3.3 presents the introspective models adopted for this work from Fox *et al* that provide the basis for the monitoring, and Section 3.4 outlines the techniques used in conjunction with the models for analysing behaviour. Section 3.5 advances this work to demonstrate how these models can be used to introduce an element of control into the actions, and 3.6 discusses these components in the context of an introspective framework. Section 3.7 presents an additional benefit of this work to intelligent systems, before finally Section 3.8 discusses the features and limitations of this approach.

3.1 Introspection in Autonomous Systems

Introspection is a human trait that allows people to reason about and reflect on their own conscious thoughts and feelings. It allows humans to examine their own behaviour and learn from previous experiences, and it has been said that in order for robots to operate fully within a 'common sense' world that a key component is to provide them with an introspective capability [63]. One of the motivations for this is that if a system can understand its own behaviour it can hold a realistic expectation of how that behaviour should perform, and in turn be in a better position to identify and correct problems within it. An example from the DARPA grand challenge[1] is a contestant that became stuck in a chain link fence that was not detected by its sensors [2]. From the readings provided by the sensors and actuators there was no obstruction and the wheels were continuing to turn, and as such there was nothing to alert the system a problem had occurred. If the robot had a better understanding of the action it was trying to perform, how it should be progressing, how long it should be active in a particular component, then it could reason about its behaviour and determine something had gone wrong.

Anderson and Perlis [2] state that the three components required to add introspection to an intelligent system are:

- The ability to monitor oneself
- The ability to reason about oneself

[1]The DARPA Grand Challenge is a prize competition for American autonomous vehicles, funded by the Defence Advanced Research Projects Agency (DARPA)

- The ability to correct oneself

The monitoring component is concerned with observing the incoming data from the sensors and determining the current state of the system. The reasoning component is then focused on analysing the state and ensuring that it is reasonable within the confines of the current action, and lastly the corrective module is required so that if the state is determined to be problematic then the system can initiate steps towards fixing it.

In the literature, introspection has been used to support execution robustness in a variety of situations. Morris [67] investigates an introspective approach to operating in hostile environments where sensors may become damaged, monitoring how well the current configuration is achieving the goal and reconfiguring itself if it determines it is no longer suitable. Krause *et al.* [57] use introspection to maximise the efficiency of a system, examining an algorithm's performance within the context of the current situation and switching between faster or more complex algorithms as the need arises. Cox *et al.* [17], discusses building a meta reasoning level over standard architectures using introspective techniques to reason about the actions selected and their probability of success within the current environment.

The work in this thesis builds upon a previous introspective monitoring system set out in [30], extending it to consider the problem of failure recovery. To this end in the following sections a framework is presented that realises the components outlined in Anderson and Perlis [2].

3.2 Introspection at the Executive Level

This chapter investigates an approach for execution recovery that uses introspective techniques to allow the system to monitor and understand its own behaviour, then leverage this understanding to recognise and interject when it realises an action is failing. Following the detection of a failure, this work explores transitioning into new actions from within the current execution that can allow the system to overcome the problem, allowing the initial action to resume and complete its goal. The desired result of these additional actions is that more complex problems can be resolved during an execution without the need to instigate higher levels of reasoning.

To achieve this an introspective framework has been created that can associate mod-

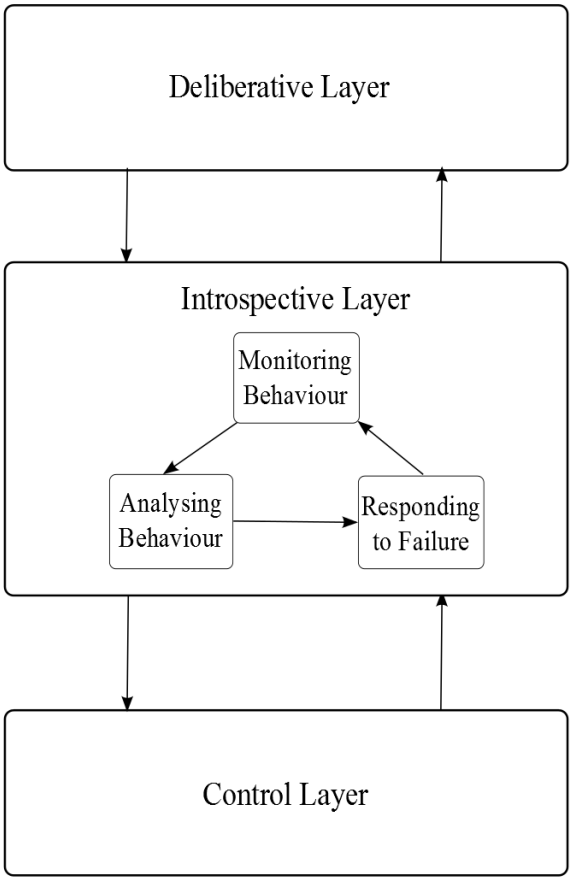


Figure 3.1: A high level view of the framework illustrating how the executive layer in the middle of the system has been enhanced to support monitoring, reasoning and control, transforming it into an introspective layer.

els with each of the actions the system can perform, and facilitate the transitioning between these models to achieve complex goals or overcome execution failures. The framework is based on the standard three tier architecture common within robotics, and a high level representation of this can be seen in Figure 3.1. At the top sits a deliberative reasoner, which takes in a symbolic representation of the domain and a set of goals and produces a plan consisting of the actions needed to achieve those goals. At the bottom is the control layer, which is responsible for the low level execution of the actions and is the main interface between the robot's sensors and actuators. In the middle sits the executive, which is the layer this work primarily focuses on.

The executive is traditionally used for taking the high level actions set by the planner and translating them into low level behaviours which the control layer can execute, as well as managing the interactions between these low level behaviours. The executive's

position between these two layers gives it a unique perspective on the tasks the robot can execute, making it an ideal candidate for increasing its operational intelligence. This work augments the executive layer with the principles of introspection discussed above to create an introspective layer that can reason over and control the execution of a task.

The introspective layer replaces the executive layer in the system and continues to operate as a go between from the deliberation to the control, translating plan actions and sequencing low level functions. However the goal of this work is to also enable it to monitor these actions, reason about their execution and step in when it detects a problem is occurring. There are many techniques in the literature for monitoring specific actions and determining their outcomes, some of which have been discussed in Chapter 2.1. This work focuses on behavioural models - models which are built from the ground up using sensor data from real executions of the task. This allows the system to capture the robot's previous experience with an action and model its complex interactions with the environment. It also allows these previous experiences to be used a basis for reasoning about the current execution, and provides the introspective foundation for this work. These models give the system an idea of how an action should progress, and if it identifies an action that is veering away from this expectation then the executive can transition into a new action that can that can help bring the execution back on track.

The focus on the executive is due to its position between the deliberative and control layers. Deliberative reasoners, such as planners, have a high level view of the world and can utilise large amounts of domain information to create complex plans that can achieve a wide variety of goals. However while their symbolic approach to reasoning can be beneficial for certain aspects of monitoring, such as providing a model of the expected effects of an action, ensuring new actions do not start until all the prerequisite requirements are met, or even checking that an action has completed within its allotted time, it can be too abstract to reason over the execution of the task itself. If a robot's navigation fails partway between two way points this can be hard for a planner to reason about as it does not have a clear concept of the current state of the robot. This can partly be solved by decomposing the problem further into more detailed substates, or using sub-planners to handle various tasks in more detail, however this can increase the complexity of the problem and in turn the time needed to reason over it.

On the other hand the low level control can be too detailed to work with in any mean-

ingful way. It deals directly with the raw sensor and actuator data, which without context can be hard to ascribe meaning to. Monitoring the sensors directly may lead to identifying a low speed or an unexpected direction as a problem, whereas in reality it may just be part of a more complex behaviour such as avoiding an obstacle in the robot's path.

Working at the executive level connects the low level control with the high level reasoning in a manner that allows more appropriate decisions to be made with regards to the current execution. The introspective models represent the actions from the deliberator, and the low level sensor data feeds into these models to estimate the current state. When viewed as a sequence these states represent the progression of an action through the environment, and it is within these states that there is an opportunity for additional reasoning that can be used to improve the execution. A direct benefit from this work is that by creating a more intelligent executive that can detect and recover from failures independently, it can allow the planner to continue to operate at a higher level. Once an action has been dispatched it can be handled almost entirely by the executive, and the planner need only be notified once it has competed regardless of the problems that may have arisen during its execution.

The next sections will discuss in more detail each of the major components in the framework.

3.3 Monitoring Behaviour

Underlying any introspective architecture is the ability to observe the system's own behaviour. This thesis follows the model based approach presented in Fox *et al.* [30], which represents actions using abstract behavioural models learned from a robot's historic sensor data. These models provide the critical introspective capability to the executive, allowing the system to compare its current behaviour to its previous experiences to determine if it fits within expectations. As these models form a core component of this thesis a brief overview of how they are constructed and used will be provided, with a focus on their application and benefit to this work [1].

[1] For a more detailed explanation on building and testing the models see [30]

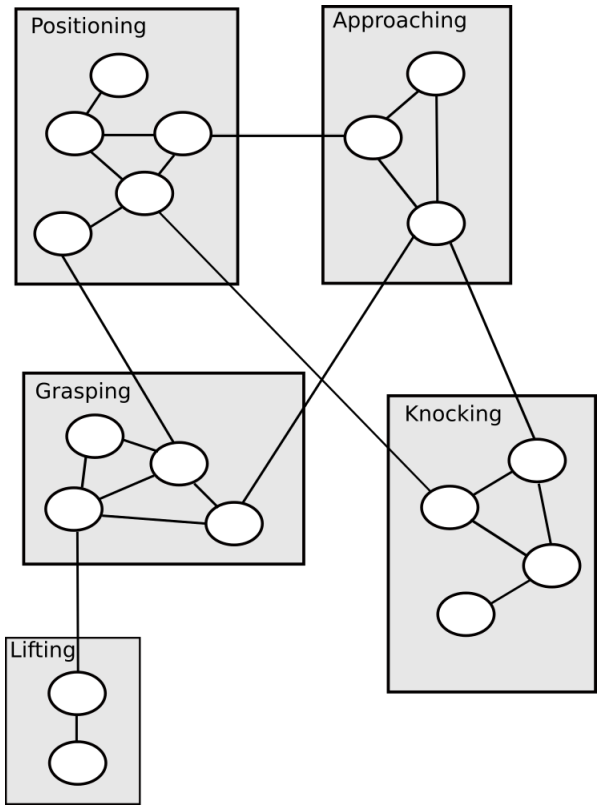


Figure 3.2: A high level state transition model representing the behaviour of picking up an object. This is not learned from real data but represents a simple example with labels highlighting similar groups of behavioural states, and the associated transitions between each of these states.

3.3.1 Introspective Models

The original goal for the work described in [30] was to use sensor data from a robot to automatically learn a model of a task, such that the model could be used to monitor and explain the behaviour of the robot during subsequent executions of this task. The work approached this challenge from the point of view of a robot owner, as opposed to a robot designer. This shift in perspective meant that rather than trying to optimise the design of a task to match a specification, or incorporate monitoring into different subroutines, the assumption was made the robot could already execute the task and the model is therefore created based on the observed behaviour of the robot during its execution. By focusing on the observed behaviour this work attempts to capture a global model of the task, encompassing both the sub-behaviours that compose a task and the robot's interaction with its environment.

An example of one of the tasks this work might attempt to model is that of the robot at-

tempting to pick up an object. A human watching a robot attempt this task may be able to identify some of the different states it transitions through during an execution based on its observed behaviour. These might be states such as approaching, positioning, knocking, grasping, and lifting. It is unlikely however that the robot will consistently move through these states in a linear sequence during repeated attempts of this task. Instead it is more likely it will move through these states in different combinations depending on the current context of the execution, often going through loops where it will transition between different behaviours such as positioning and knocking multiple times before attempting to grasp the object. This underlying relationship between these behaviours is part of what this work attempts to learn.

However these high level labels that may be assigned by a human are not sufficiently granular to capture the actual behaviour a robot is exhibiting through repeated iterations of a task. In reality these behaviours can be decomposed further into states representing subtle variations of the behaviour, such as the angle of approaching, or the speed of knocking. This decomposition is further affected by the way these behaviours can interact and overlap with one another, such as the robot beginning to position itself while it is still approaching the object. A theoretical example of how such a model may look, illustrating the relationships between states and using groupings to represent high level behaviours, is shown in Figure 3.2. If a model can be learned which defines these states, along with the relationships between them, it can be used to monitor subsequent executions of the same task by comparing the states the robot transitions through against the model to determine if they make sense.

The robot however may not be able to accurately sense its own state. Equipped with noisy and limited sensors, and operating in a dynamic environment the robot can only obtain an estimate of its current state. As such Fox *et al.* chose to represent these tasks using hidden Markov models (HMMs) [79], which captures this inherent unreliability of a robot attempting to monitor its own state. States within a HMM are not directly visible, instead they are only able to be estimated through the use of observations.

Formally a HMM can be described as is a 5-tuple stochastic state transition model λ = (ψ , ε , π , δ , θ)

- $\psi = [s_1, s_2, ..., s_n]$, a finite set of states;
- $\varepsilon = [o_1, o_2, ...o_n]$ a finite set of observations;
- $\pi : \psi \rightarrow [0, 1]$, the prior probability distribution over ψ;

- $\delta : \psi^2 \rightarrow [0, 1]$, the transition model of λ such that $\delta_{i,j}$ = Prob[$q_{t+1} = s_j \,|q_t = s_i$] is the probability of transitioning from state s_i to s_j at time t (q_t is the actual state at time t);

- $\theta : \psi \times \varepsilon \rightarrow$ [0, 1], the sensor model of λ such that $\theta_{i,k}$ = Prob[$o_k \,|s_i$] is the probability of seeing observation o_k in state s_i.

These models can be learned from the robot's sensor data, creating a mapping between this sensor data and a set of discrete observations. These observations can then be used to estimate the current state of the system. By using the robot's sensor data gathered through previous executions of a task, Fox *et al.*'s models attempt to capture the complexity and uncertainty of operating within the real world and incorporate this back into the representation. This allows the executive to be able to explain certain amounts of erratic behaviour, and in turn actually reduce the uncertainty of the task with regards to the system as a whole. Modelling based on the observed behaviour of the robot also ensures the models are not tied to a particular task or platform, allowing them to be created and used on any system that can record data about its own operation.

3.3.2 Learning the Models

To begin learning one of these models, executions of the chosen task are repeated a sufficient number of times to capture the variance in behaviour of the task, and its different interactions with its environment. The number required can vary from task to task, but for the models in this work 50 - 60 executions tended to suffice. The raw data from these executions form the basis of the training set. Small variations to the environment can be introduced during these executions to prevent over learning, and to ensure the learned model represents a task within a class of environment as opposed to a specific one. A second set of execution data is also gathered for the same task, ideally in a separate but similar environment. This is the verification data, which can be used to validate the model once it has been learned.

The raw data from a robot's sensors on its own is not informative enough to convey how a task evolves throughout its execution. Knowing the robot's present position in terms of x and y coordinates, or the direction it is facing provides little insight into the robot's current behaviour. Instead more discriminatory features can be learned by combining these raw values together, and observing how these evolve over a period of time. As an example while the x and y coordinates themselves may not be useful, the

values can be used to calculate how far the robot has travelled over a short time. This is a more informative measurement, as it can be used to infer if the robot is travelling fast or slow, or even if it is stationary.

To capture these changes in behaviour over time the features are created from sub sequences of the raw data, using a sliding window approach. The size of the window is important, as it has to be large enough to allow interesting behaviour to occur, but small enough to still capture subtle changes. For this work a one second window is used across all the learned models. This is value is chosen empirically, however a more detailed evaluation on how the window size, as well as other key parameters, affect the learned models is detailed in [39]. Once the features have been selected the initial training set undergoes a processing step which results in a new training set composed of feature vectors.

These feature vectors are then used as the input to a clustering algorithm, which attempts to find a mapping from the non finite feature data to a finite set of observations. An observation represents a physical state of the system exemplified by certain values from the features. For example one observation may correlate to the robot having a relatively low velocity and a high acceleration. To derive this mapping, Kohonen network clustering [55] is used.

Kohonen network clustering was chosen as unlike in many other clustering techniques the number of clusters is not required to be specified upfront. Instead the number of clusters is derived from the data itself, which is important as the number of observations in a given task is initially unknown. The clustering works by projecting multidimensional data on to a lower dimensional space in order to reveal a cluster landscape. In this work the n-dimensional data is projected on to a 2-dimensional network. The size of this network is an important parameter, as if it is too large the clusters can degenerate into noise, while if it is too small it can end up combining interesting clusters together. As a general rule, the network size should be roughly 500 times smaller than the size of the training data set [30], [55], however selecting an appropriate value is a manual process. For the models discussed in this thesis the network size varied between 30 x 30 and 45 x 45.

To begin clustering the Kohonen network is initialised with each cell in the grid being associated with a random vector of the same dimension as the input data. Training occurs by presenting each of the input vectors to the network and determining to which cell vector they are closest. This identified vector, plus a selection of surrounding cell

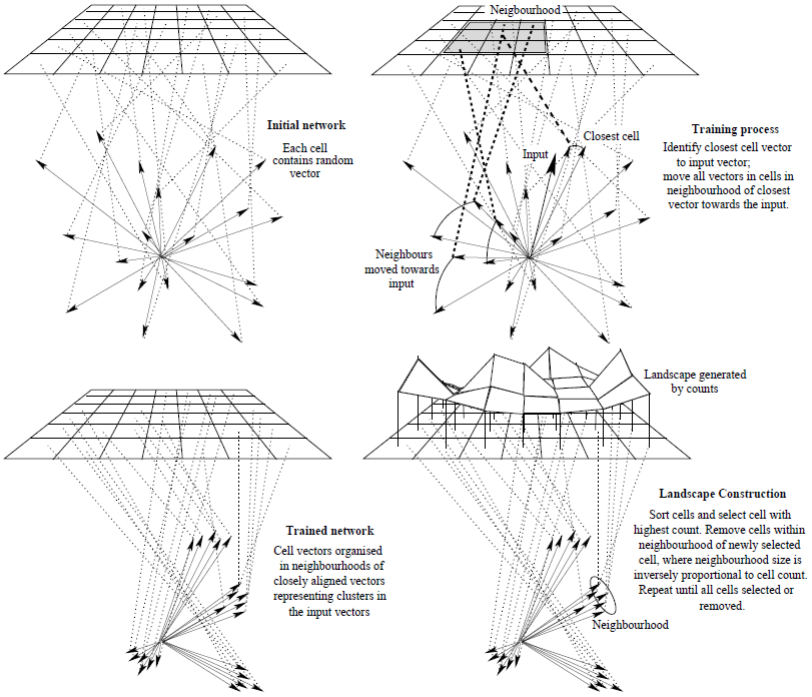


Figure 3.3: Training a Kohonen network (Figure reprinted from Fox *et al.* [30])

vectors as determined by a neighbourhood value, are then aligned with the input vector through scalar multiplication. The neighbourhood value is designed to be initially large, and decrease exponentially over time using a decay rate to reduce the impact of training. This training process occurs fifty times, then the order of the input data is randomised and the training is repeated a further fifty times. The reordering of the training data is designed to prevent over learning.

Following the initial training, each cell in the network has a number of vectors associated with it. A cluster selection strategy is applied which creates the cluster landscape under the assumption that cells with the most vectors attracted to them represent the highest peaks, and cells with few vectors are likely uninteresting. An overview of the entire cluster process is shown in Figure 3.3. The resulting peak vectors from this process become the basis for the set of observations ε. Each of the feature vectors from the initial training set can then be reintroduced to the Kohonen network and classified into an observation by using scalar multiplication to identify the peak vector it is closest to.

The cluster landscape can also be used to determine the set of states for the model through the use of a state splitting strategy. This examines the peak vectors, representing the observations, to determine if there are groups within these which may corre-

spond to larger behavioural states. This analyses the angles between these vectors, and uses the assumption that if the angle separating two vectors is sufficiently small, then they are similar and can represent the same behaviour. This compares all the vectors in the landscape, and the final result from the clustering is both a set of observations and a set of states representing the learned task.

Once the observations and states have been determined, the final stage in this process to learn the Hidden Markov Model that represents the relationship between the two. For this the Expectation Maximisation (EM) algorithm [20] is used. EM takes as its input the set of states, observations and trajectories representing the training data, along with the initial parameters of an HMM - an initial sensor model, an initial state transition model, and a prior state distribution - and iteratively re-estimates these parameters. This achieved via a two step process. Step one is the expectation phase, which calculates the likelihoods of seeing each observation based on the current model. Step two is the maximisation phase which updates the model parameters to better account for the observations. When the likelihoods are no longer increasing, EM converges.

The output from this process is a learned model which can use the robot's sensor data to estimate the current state of the system, and that can monitor and explain the state transitions that occur throughout an execution. For this thesis the underlying methods used to learn the models remains unchanged, though these techniques are applied to a larger and more sophisticated robot than in previous work, and a wider range of tasks are learned in a variety of new real world environments. The selection of features for these tasks, along with the identification of suitable parameters for the model learning, did require significant analysis and evaluation to produce accurate models. Details of this evaluation is outside the scope of this thesis, however more in depth analysis of these parameters and their affect on model quality can be found in [39].

3.3.3 Monitoring an execution

The learned models represent a class of action, and therefore should be able to explain the behaviour of all subsequent actions belonging to that class. As an example a model learned for a corridor navigation should be able to reliably represent all corridors the robot can traverse, and not just the ones featured in the training set. To verify additional executions against the learned model the Viterbi algorithm [29] is used. The Viterbi algorithm is a dynamic programming algorithm that can estimate the most likely se-

quence of states through an HMM that characterises the system's current behaviour based on a series of observations. Given an observation and the model λ, the Viterbi algorithm will return the most likely corresponding state and its associated likelihood.

The training data can provide a foundation of normal executions, whose state sequences can be used to derive an envelope representing an action's expected behaviour. New sequences can be compared against this to ensure the model can explain additional and unseen variants of an action. Viterbi traces can be visualised to provide a representation of executions and demonstrate how they compare against the model, as seen in Figure 3.4. These traces represent the estimated probabilities of observing the current state sequence, expressed in terms of log likelihood due to how small the probability of seeing any particular sequence becomes beyond a certain length. By examining these sequences and how they progress with respect to the training data, the executive can identify problems or areas of interest within the specific execution.

The main benefit of this with regards to the executive is an online version of the Viterbi algorithm that executes on board the robot. This allows the sensor readings to be processed into observations during an execution, and supplied to the online algorithm to estimate the state sequence up until the current point in real time. This allows the execution to be compared against the model as it executes, and by monitoring the probability of the states that are fitted to the model the executive can predict if an action is progressing as expected or deviating into failure.

3.4 Analysing behaviour

Once the system has the ability to monitor an action's behaviour the next step is to analyse it and determine if it is progressing as expected. The Viterbi algorithm provides the most likely trajectory through the model based on the current execution, however this alone cannot differentiate if an execution is proceeding as normal or if it is exhibiting unusual behaviour. For this another set of algorithms is required which compares the current execution against the expected behaviour and makes a judgement based on the results. From experience, abnormal behaviour has a tendency to effect the Viterbi estimates in one of two ways; either the behaviour produces an uncommon sequence of typical observations which results in the algorithm estimating unlikely state transitions causing the probability of the trajectory to drop, or the behaviour generates unlikely observations which tend to relate to unusually high self transitions within a state and the

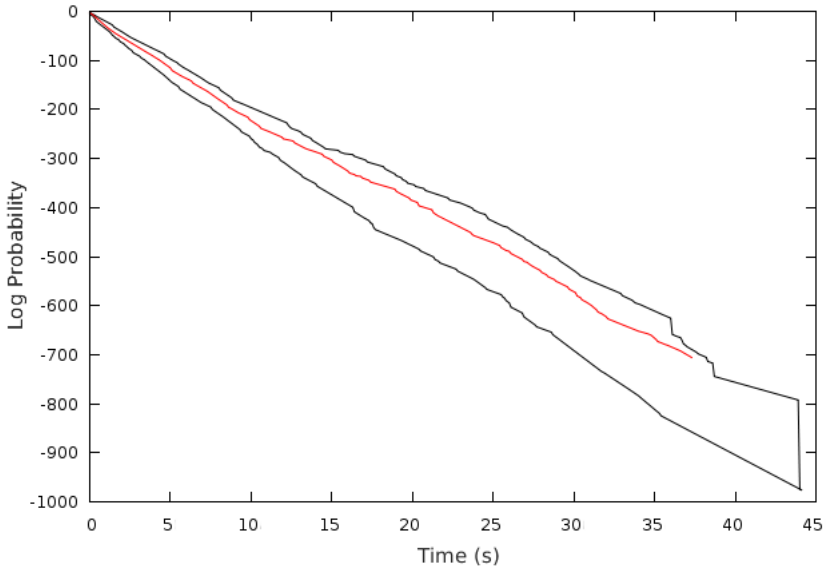


Figure 3.4: An example of a Viterbi trace through a navigation model. The two outer lines represent the highest and lowest probabilities observed from the training data, forming an envelope which encompasses the behaviour of the learned model. The red inner line through the middle represents a verification run's estimated probability within the boundaries of the model.

trajectory's probability starts to plateau. Gough [39] explores methods for identifying these behaviours from Viterbi sequences, and develops two core algorithms, Temporal State Counting, and Cumulative Log Probability Difference. These provided an initial basis for the executive's analytical routines.

3.4.1 Temporal State Counting

The first algorithm investigated is Temporal State Counting (TSC) [39]. This algorithm examines the frequency a state has been visited at each time step during an execution and compares it to the historic data. The assumption is made that a knowledge base can be constructed from the training data that represents which states should have occurred at each discrete time point during an action, and any major deviations from this represent anomalous behaviour. The occurrence of each state at a particular time is assigned an upper and lower bound derived from the training data to account for variation within runs, and anything out with these numbers is classified as an error state. Throughout the rest of this chapter an error state will refer to any state observed by the system that is determined to be abnormal within the current behaviour.

function TSC(Obs, $maxStates$, $minStates$, $states$)
// Obs $= (o_0, \ldots, o_n)$: the sequence of observations
// maxStates $= (max_0^0, .., max_i^t, .., max_p^q)$: max count of $states_i$ at time t
// minStates $= (min_0^0, .., min_i^t, .., min_p^q)$: min count of $states_i$ at time t
// $states$: ordered set of HMM states
$s_0, \ldots, s_n \leftarrow$ `getViterbiSequence`(Obs)
$tsc \leftarrow 0$
$statecount \leftarrow$ `zeros`(`size`($states$))
for $t \leftarrow 0, \ldots, n$ **do**
$i \leftarrow states$.`index`(s_t)
$statecount[i] \leftarrow statecount[i] + 1$
for $j \leftarrow 0, \ldots,$ `size`($states$) **do**
if $statecount[j] > max_j^t$ **then**
$tsc \leftarrow tsc + (statecount[j] - max_j^t)$
else if $statecount[j] < min_j^t$ **then**
$tsc \leftarrow tsc + (min_j^t - statecount[j])$
end if
end for
end for
return tsc
end function

Figure 3.5: Pseudo code for the temporal state counting algorithm

The psudeo code for the TSC algorithm is presented in Figure 3.5. This utilises the maximum and minimum expected counts for each state at each time point to derive the number of error states within the current execution. If the number of error states found reaches a threshold value, obtained empirically from the verification data, the task is deemed to be heading towards failure and the system is alerted.

Previous evaluation of this algorithm has shown it to be effective [39], which aligns with the evaluation for this thesis, as discussed in Section 5.2 . The main benefit of this error detection technique is that it adds a temporal element to the non temporal models. This allows the system to reason over how long certain sub-behaviours are taking to complete, and identify if something has gone wrong. An example would be if a robot is approaching a door it might start to reposition itself to face the entrance. If however the robot approaches from a bad angle it might have trouble aligning itself properly and continually attempt to reposition itself, only moving forward slightly before once again attempting a reposition. By noticing a certain state, or sequence of states are occurring more frequently than expected for the specific portion of the behaviour the system can recognise a problem has arisen and attempt to initiate a recovery procedure.

It can also help overcome a known limitation with these HMMs, which is their high self transition probability. If during the course of a navigation the robot becomes stuck or blocked, then the model mirrors this stationary behaviour and has the tendency to continually generate the same state. Due to the high self transition property this can often be regarded as the most likely occurrence of the system, and not immediately identified as an error. The TSC algorithm provides the robot with the capability to determine if an action has become stuck in a particular state, and flag this as a problem.

3.4.2 Cumulative Log Probability Difference

The second algorithm is Cumulative Log Probability Difference (CLPD) [39]. This algorithm also attempts to classify behaviour based on the creation of upper and lower bounds, however this time based on probability. As a task is executed the Viterbi algorithm assigns a probability to the likelihood of the current state sequence observed by the system. Due to the extremely low probability of seeing any particular sequence of states coupled with the probability only getting lower as new states are observed, the log probability is taken for each trace. The training data once again provides the maximum and minimum probabilities for each point in time, providing a comparative window within which all normal runs are hypothesised to execute.

The pseudo code for the CLPD algorithm is shown in Figure 3.6. This calculates the probability for each time point in the current execution, and compares it against the historically derived maximum and minimum values. Any deviations outside this envelope is recorded, and a cumulative total calculated. If this total reaches a threshold value, the execution is deemed to be failing.

Evaluation of this algorithm, discussed further in Section 5.2, has shown it to be less reliable and highlighted some of its limitations. The nature of the algorithm means that anything within the probability window is classified as normal and anything outwith this is abnormal. However under real world conditions both these properties can become problematic. The main concern relates to the high probability of states self transitioning inherent in these HMMs. If the robot becomes blocked or stuck the probability tends to level out, which can result in significant time passing before the execution reaches the edge of the probability window, as shown in Figure 3.7. The CLPD algorithm does not begin to detect anything anomalous until the point where it leaves the probability window, which is long after after the initial problem is induced. From

function CLPD(Obs, $mMaxProb, mMinProb$)
// Obs $= (o_0, \ldots, o_n)$: the sequence of observations
// mMaxProb $= (m_0, \ldots, m_p)$: max probabilities from training data
// mMinProb $= (mn_0, \ldots, mn_p)$: min probabilities from training data
$s_0, \ldots, s_n \leftarrow$ `getViterbiSequence`(Obs)
$p_0, \ldots, p_n \leftarrow$ `getSequenceLikelihood`$(s_0, \ldots, s_n)$
$clpd = 0$
for $i \leftarrow 0, \ldots, n$ **do**
if $p_i > m_i$ **then**
$clpd = clpd + (p_i - m_i)$
else if $p_i < mn_i$ **then**
$clpd = clpd + (mn_i - p_i)$
end if
end for
return $clpd$
end function

Figure 3.6: Pseudo code for the cumulative log probability difference algorithm

both observing the task and the probability trace it is clear the run exhibits abnormalities, however these are not picked up by the CLPD algorithm until quite far into the execution.

On the other hand there can be a run such as that shown in Figure 3.8. This execution shows a normal run with no problems or interruptions, however when analysed with CLPD it detected errors from the start and accumulated a large probability difference throughout the task. While the run was normal, the execution sits just below the training data in the probability trace, leading to the algorithm constantly believing that a problem has occurred. Runs that veer far outside the window usually need further inspection, however in cases like the one above there is very little difference in quality, but the continued dips outside the window cause it to be flagged as abnormal.

3.4.3 Gradient Log Probability Difference

In order to improve the error detection and combat these short comings a new algorithm has been developed for the purpose of this work called the Gradient Log Probability Difference (GLPD). This solution focuses on the current execution's rate of decent through the model as opposed to trying capture a behaviour within a window. As described above most errors introduced into an execution effect the trace by either

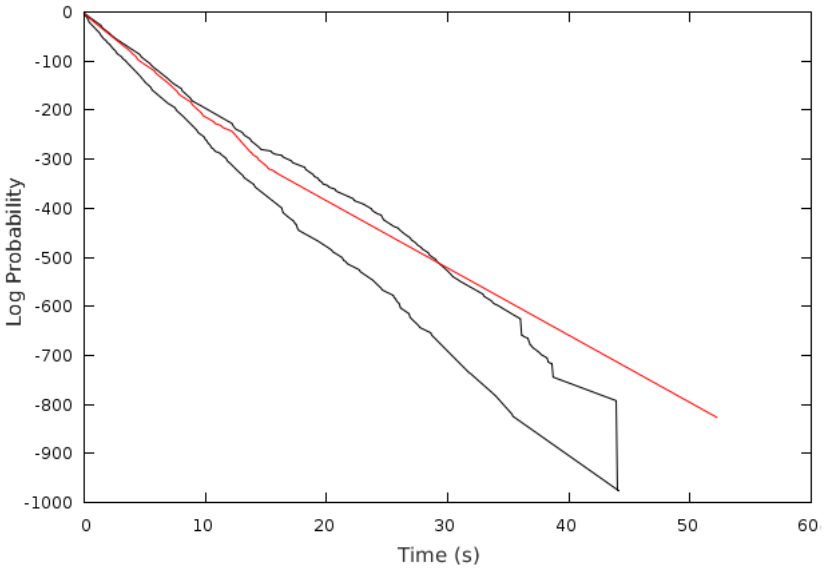


Figure 3.7: A Viterbi trace showing a corridor navigation where the robot becomes blocked and stuck continuously within the same state. Visually it can be seen the problem occurs at roughly the 15 second mark, as the run stops progressing normally and evolves into a straight diagonal line. As it takes a long time for this line to exit the envelope the CLPD algorithm is slow to respond to this fault.

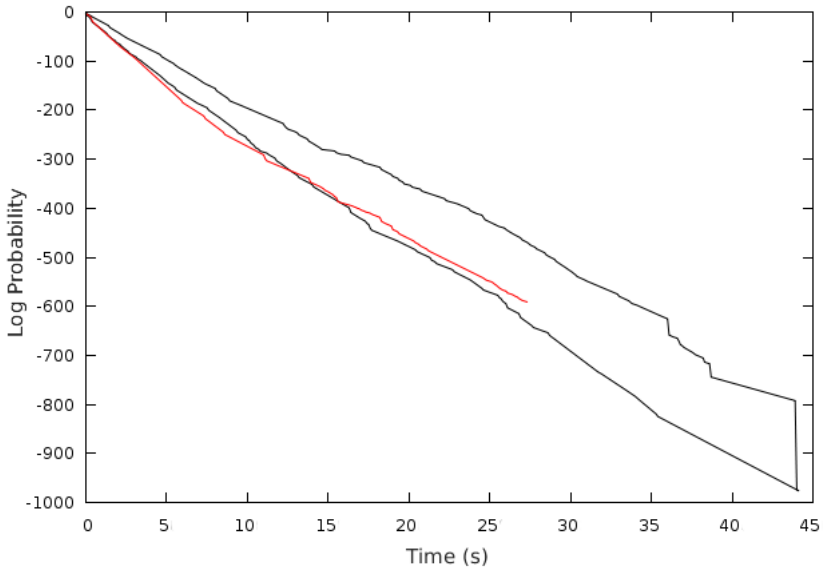


Figure 3.8: A Viterbi trace showing an execution of the corridor navigation within a new environment which is the same class as the model. The execution's progression looks normal from an observer's point of view and in comparison to other runs, such as the example in Figure 3.4, however due to the line being slightly outside the model's envelope the CLPD algorithm classifies the run as a failure

function GLPD(Obs, $mMean, mStdDev, t$)
  // Obs $= (o_0, \ldots, o_n)$: the sequence of observations
  // mMean $= (\mu_0, \ldots, \mu_p)$: the mean gradient at time point i
  // mStdDev $= (sd_0, \ldots, sd_p)$: the standard deviation at time point i
  // t : the number of time points used in the sliding window
  $s_0, \ldots, s_n \leftarrow$ `viterbi`(Obs)
  $p_0, \ldots, p_n \leftarrow$ `sequenceLikelihood`$(s_0, \ldots, s_n)$
  $glpd = 0$
  for $i \leftarrow 0, \ldots, n - t$ **do**
    $g \leftarrow (p_{i+t} - p_i)/t$
    $g_\Delta \leftarrow \mid \mu_{i+t} - g \mid$
    if $g_\Delta > sd_{i+t}$ **then**
      $glpd = glpd + (g_\Delta - sd_{i+t})$
    end if
  end for
  return $glpd$
end function

Figure 3.9: Pseudo code for the gradient log probabilty difference algorithm

levelling it out or causing a sharp decrease in its trajectory. This algorithm is designed to recognise and react to these changes quicker by focusing on how the gradient of the trajectory evolves and its relation to the previous executions.

To achieve this the training set is revisited and segmented into discrete sections of time. For each section of time the gradient of all the trajectories that pass through that point is calculated, and a mean and standard deviation is determined based on these gradients. This selection of means and standard deviations can then be used to monitor for failures during an action, by calculating the gradient of the current execution and comparing it to the corresponding time's mean gradient.

The algorithm for deriving the GLPD is outlined in Figure 3.9. This uses the probabilities calculated by the Viterbi algorithm for each state to calculate the gradient of the current trace across windows of time. Each gradient is then compared to the equivalent mean gradient from the training data for that time period, and if the current gradient falls outwith a standard deviation either above or below the mean, the difference is recorded and summed into an error margin.

A threshold limit can be learned for the GLPD by analysing the verification data gathered during the modelling process and deriving a value that can encompass all known successful runs. Future executions which go beyond this threshold can then be iden-

tified as anomalous, while states which lie out-with the standard deviation can be labelled as error states. The main benefit of this method of error detection over CLPD is that it is no longer bound by a window to dictate the value of a run, but instead focuses on its rate of progression towards the goal. This allows it to more consistently recognise and detect the same types of errors as before, but it can now more accurately classify issues such as the two described above. For the first problem identified of a state encountering a continual self transition, highlighted in Figure 3.7, the slowed progression quickly builds up a high GLPD value as it recognises the rate of the line's decent does not resemble any of the data found in the training set. Similarly it can also address the second problem whereby the trace veers slightly outside the probability window, shown in Figure 3.8, as due to the progression of the run being very similar to those encountered before the algorithm accepts that no issues have arisen and therefore does not raise any errors.

3.5 Responding to failure

At the start of this chapter three key components of introspection were discussed, which were the ability to monitor oneself, the ability to reason over one's state and the ability to correct oneself. The introspective models [30] provide this system with the first component, the ability to monitor oneself. The error detection routines [39] serve as the basis for the second component, allowing experience from previous executions to be used to reason about the current one. For the rest of this thesis the focus will now be on the last component, the ability to correct oneself.

Once the system has detected that an action is starting to deviate from its expected behaviour it should be able to make intelligent decisions that can bring the execution back in line. The control layer can handle minor errors within the execution on its own, however anything of significance such as blocked pathways, closed doors or substantial obstacles will cause it to either halt the execution or ignore the problem completely. Passing the problem up through the system to the deliberative component is often the way more complex problems are solved, however as discussed above this can introduce its own issues due to the level of abstraction at this layer.

The solution proposed is to introduce the concept of *recovery actions* into the executive. Recovery actions in this context are actions that can be launched from within the current execution, and override the original task. The position taken in this work is that

in the event of an execution failure a recovery action can be selected that upon completion will improve the state of the system in such a way that the initial action can be resumed and the original goal completed. This allows the executive to use behavioural models to monitor and control the execution of a task, augmenting it with the ability to handle failure at this intermediate level between the planner and the reactive control. This can improve the system's robustness, as by increasing the autonomy of the executive it enables the planner to dispatch an action and allow the lower levels to handle all aspects of the execution, only needing to be alerted when a task has completed or an unrecoverable failure has occurred.

Throughout the rest of this chapter, and much of this thesis, the behavioural models discussed will primarily be navigation models representing the robot traversing through corridors or more complex environments. This is merely a limitation of the platform used for developing this work, and not the work itself. The models and architectures discussed throughout the rest of this thesis are not limited to, or directly tied to navigation in any way and are entirely task independent. As such the same processes could be used to monitor and control the execution of situations such as a gripper losing its grasp on a parcel, or to observe the movements of a robotic arm and assist the execution when it detects it has knocked an object over or is approaching from a bad angle. The following sections will discuss the components that contribute to the system's failure response.

3.5.1 Transitioning between behaviours

The behavioural models discussed here have been proven in previous work to be able to monitor and detect anomalies within isolated tasks [39], however they have not been used in a system which can execute multiple sequential actions. One of the goals of this work was to create an introspective domain, where a behavioural model has been created for each of the major actions the robot can execute. This involves the system being able to handle fluidly transitioning into new behaviours upon the completion of previous ones, as well as at the point of failure.

A single action executing has a clearly defined start and end point which is learned though the model building stage, and this holds when executing that action on its own in the system. However if the action is the second step to be executed in a plan the remnants of the first action can continue into the new task's monitoring and cause com-

plications. For example the robot rarely comes to a complete stop between navigation tasks, so the later actions start monitoring from a rolling start. More problematic still is if the first action ends with conditions not expected within the subsequent action, such as the robot not being oriented in the required direction. This can cause the system to have to compensate at the beginning of the new action before before continuing to progress as expected. As this is not accounted for within the model it starts the new actions off with what is seen as a sequence of deviant behaviour, which results in an action being more likely to fail during the execution.

Initial solutions involved incorporating elements of the uncertainty possible in the start and end positions of the robot into the model, through beginning and ending training actions under different conditions. This allowed the models to transition into each other with less failures being detected, however it puts the onus back on the engineer to design training sets that cover the different possibilities surrounding a tasks starting conditions.

To help alleviate this issue after the first action of a plan has completed executing, subsequent actions are loaded into the executive and have their initial progression through the model seeded with a selection of known good states. This provides a brief transitional period that allows the system to adapt to the change in execution. Most tasks feature little variation over their first few seconds of execution, leading to relatively homogenous state traces. By using the initial state of a task model, as defined by the prior probability distribution, and selecting the most likely subsequent states based on the transition function a brief period of the robot's state trace can be generated to accommodate slight variations in the execution. For this work this was set at a 1 second window, after which the monitoring would resume generating states from observations.

3.5.2 Recovery Actions

At the heart of the system sits the recovery actions, the actions with which the system attempts to correct the execution upon detecting that a failure is occurring. The recovery actions are based on the same introspective models that represent the normal actions and as such can encompass a full range of behaviours. These can vary from simple operations such as waiting for an event to pass to more complex compound tasks such as attempting to open a door, but each has the shared target of leaving the system in a better state than from which it started. An example would be a robot deliv-

ering a package held within its gripper. If during the navigation to the goal the robot's gripper is slowly closing over time the monitoring routines can recognise this anomalous behaviour and prompt for further analysis. If the package is found to be slipping the executive can interject and a recovery model can be instigated that causes the robot to reaffirm its grip on the object, possibly by letting go of it completely, repositioning itself, and picking it up again. The recovery action is also fully monitored, and upon successful completion the system is left with the object firmly within its grasp and the initial *navigate with package* action can be resumed. Some examples of the recovery actions within the system are :

Localisation Scan : Large amounts of unexpected clutter within the environment, either from objects or people, can cause the localisation of the robot to fail and result in erratic movement within the navigation. The scan action rotates the robot 360 degrees searching for environmental landmarks, stopping at set intervals to reinitialise the robot's localisation routine. Upon completion the robot should have a better knowledge of its surroundings and its position within the environment.

Obstacle Clearance : If the robot becomes too close to an obstruction the path planning routines can struggle to find a suitable route around it, resulting in a navigation failure and the robot becoming stranded. The clearance action moves the robot away from the object blocking it, providing a greater radius to discover alternative paths.

Doorway identification : If the robot is attempting to traverse through a doorway it can often initialise from a poor angle in relation to the goal, causing repeated bouts of hesitation and searching in between incremental movement. The identification action uses the robot's laser range finder to identify the edges within the opening of the doorway, and move the robot so that it is closer and facing directly into the largest opening between these edges.

Upon detecting a deviation within the current execution the system backs up the data associated with the original action such as state trajectories and previous observations. The system then initialises the recovery action, which halts the previous action and begins executing. The data from the sensors is diverted through this new model, and the system begins to monitor its progress. This causes a smooth transition from the point of failure within the original model to the starting state of the recovery action, which can be seen in Figure 3.10. The new action is then monitored to completion as

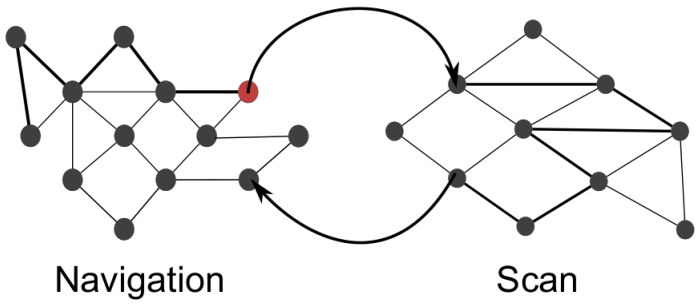


Figure 3.10: A diagram showing the transition from the failure state of one model into the initial state of another, then transitioning back into a new state once the recovery is complete. The thick lines show the path through the models

with any normal behaviour, and upon reaching a terminal state passes control back to the originating action. This attempts to put the system in a better position to resume this action, having altered its physical state via the recovery action's completion, and its internal state through the use of a state pruning algorithm described in 3.5.3.

A main aim of this chapter was to investigate the feasibility of using behavioural models to represent both actions and recovery actions. The executive should then be able to transition between the two in order to overcome failure, while preserving the original action's execution and maintaining a consistent level of monitoring throughout. The models used to monitor these executions however are observational in nature, and lack any means of formal control. Adding control to the models themselves is problematic as they do not contain a representation of failure from which to reason when an issue arises. Rather than an explicit state, failure is represented as sequences of states during the execution, many of which may be regarded as normal under a different context. In this initial system each type of task is associated with a single default recovery action which is invoked when an error is detected. Different types of task can have different recovery actions associated with them, however each task can only have one recovery action.

The execution of these tasks and their associated recovery actions can be viewed as similar to a Hierarchal Hidden Markov Model (HHMM) [26], where by at certain points within the execution it can transition control into a different model and the upper level does not regain control until the lower model completes its execution. The

main difference is that in a HHMM the drop down states are predefined, while in this system it is based on the outcome of a monitoring algorithm as to when a new model is initialised. This is due to attempting to predict and identify failure before it occurs by analysing sequences of state transitions, rather than focusing on individual states. These sequences can show signs of anomalous or unexpected behaviour before a failure has fully formed, allowing preventative measures to be taken. Another significant difference is that in HHMMs the lower models always return control back to the initiating state, while in this system the initial model resumes from a different state that represents a more suitable point in the execution in both an abstract and physical sense. The issue of additional control will be addressed in Chapter 4.

3.5.3 State Pruning

When a recovery action is activated a backup of the states and observations of the previous action is stored, representing its progression through the model up until the point of failure. When this action is resumed the previous model and its progression is loaded back into the executive, however the system is no longer in the same state that it was when the action ended. The completion of the recovery action indicates that the system has changed in some way, possibly physically such as the robot being in a new configuration, or internally such as reinitialising one of the robot's subroutines. The behavioural models are also based on the Markov assumption, that the current state can be identified by previous state and the current observation. If the previous state that is resumed from is an error state, one that represents an unusual behaviour or an unlikely state for the system to have been in, restarting from there can be problematic. Error states have a tendency to have even higher self transition probabilities than normal states, and conversely feature low transition probabilities to other states. This means that even though the robot's execution is now technically resuming from a better place, it can cause an unusual trajectory through the model while attempting to get the state sequence back on track, which can lead to further abnormalities being detected.

As a precaution a state pruning algorithm is employed to roll back the system to the last known good state within the execution. During the anomaly detection the state sequence is checked to determine if it lies within the boundaries set by the TSC and GLPD algorithms. If at any time point a state falls outside of these boundaries it is classified by the system as an error state with regards to that execution. When the failed task is resumed the state pruning algorithm looks for the last continuous sequence of

function pruneStates(Obs, $badStates_{tsc}$, $badStates_{glpd}$)
  // Obs $= (o_0, \ldots, o_n)$: the sequence of observations
  // $\text{badStates}_{\text{tsc}} = (tsc_0, \ldots, tsc_n)$: binary classification of states at time t
  // $\text{badStates}_{\text{glpd}} = (glpd_0, \ldots, glpd_n)$: binary classification of states
  // at time t
  $s_0, \ldots, s_n \leftarrow$ `getViterbiSequence`(Obs)
  $badStates \leftarrow badStates_{tsc} \wedge badStates_{glpd}$
  $badStateCount \leftarrow 0$
  $lastStateGood \leftarrow 0$
  for $i \leftarrow 0, \ldots, n$ **do**
    if $badStates[i] == 1$ & $lastStateGood == 0$ **then**
      $badStateCount \leftarrow badStateCount + 1$
    else if $badStates[i] == 0$ & $lastStateGood == 0$ **then**
      $lastStateGood \leftarrow 1$
    else if $badStates[i] == 1$ & $lastStateGood == 1$ **then**
      $badStateCount \leftarrow badStateCount + 1$
      $lastStateGood \leftarrow 0$
    else
      $break$
    end if
  end for
  $obs \leftarrow$ `subList`$(obs, 0, n - badstates)$
  return obs
end function

Figure 3.11: Pseudo code for the state pruning algorithm. This identifies sequences of states classed as bad by the anomaly detection routines and removes them from the execution trace.

error states, which as a collection is assumed to represent the deviant behaviour that led to the failure of the action.

The algorithm for this is shown in Figure 3.11. This calculates the bad states within an execution as defined by the available anomaly section routines, and then works backwards from the last known state accumulating a count of all states that need to be pruned. A single normal state can appear within this sequence of bad states to account for any discrepancies in the state estimation, however once multiple normal states begin to occur together, the chain of abnormal behaviour is deemed to have ended. These states are stripped from the state sequence and the model resumes from the last known good state in the execution, continuing as if the failure had never occurred. Any bad states outside this final sequence are ignored in order to preserve the integrity of the execution.

3.5.4 Model Selection

One of the things discovered while experimenting with these behavioural models is that while it is desirable to create generic models for a particular class of action, if that class itself is too broad it can heavily degrade the system's monitoring performance. Models that try to encapsulate too many variable environments or executions can end up becoming overly generic, and in turn can take longer or even fail to distinguish good behaviour from bad. As such certain actions have been separated into distinct models, which represent the context they operate in. An example of this would be navigating through a corridor versus navigating a wide open space. While from a human perspective these two actions may seem quite similar, from the robot's point of view they have entirely different sensor readings, with the laser range finder and sonar having the biggest effect. Even the path itself however is often different, with the low level control plotting a straight line down a corridor between waypoints, while often favouring a shallow arc between points in an open area.

This problem can be addressed by increasing the number of actions available to the planner, allowing it to select from the various sub-types of an action then loading up the corresponding model as per above. This can increase the size and complexity of the domain, as now in the case of a navigation example the planner is required to know which areas are a corridor and which areas are open, and represent this within the relevant actions preconditions. Instead it is desirable to push this task down to the executive, allowing it to select an appropriate model for the current action and preserving the level of abstraction within the plan domain. This is achieved by grouping similar models under a single category, representing a generic action from the plan. As the Viterbi algorithm is already used online to estimate the probability of an execution's success, the same method can be applied to multiple models simultaneously to calculate their probabilities. Given a series of observations O, and M number of possible models λ, the probability of each available model m can be estimated $P(O|\lambda^m)$, and the most likely selected.

$$m^* = argmax_{1 \leq m \leq M} \left[P(O|\lambda^m) \right] \tag{3.1}$$

Most models can be differentiated by the system within a short period of time, at which point all models other than the one with the highest probability are discarded. It is important to note that a model does not have to be labelled as a failure to be

discarded. A navigate area model will accommodate corridor data without incident for a reasonable period of time, however the estimated probabilities through the respective models will vary noticeably. To ensure the system is confident of its choice, if the difference between the probabilities of competing models is marginal, it will continue to test them until a threshold difference is met. The error detection routines in the executive also ensure that the most likely model is in fact a sensible fit, and not a single state repeatedly transitioning into itself.

Currently each action has to be self contained, so if the navigation starts in a corridor it will have to end in one. It is possible to construct more complex tasks that are composed of a series of atomic actions; for example a navigation action that starts in a corridor and transitions into an open space. This was investigated using the techniques outlined above to automatically derive when an action has entered a new environment, however the preliminary results were unsatisfactory. Upon detecting a failure the executive attempted to take the last section of data from the point of deviation and process it through different models to determine if any could explain the change in behaviour. This involved the storage and manipulation of large amounts of raw sensor data, and more importantly extended the time taken from recognising a failure to initialising an action to circumvent it. As an advantage of using the HMMs is their ability to predict failure before it becomes critical, adding more time to the response was seen as a significant drawback, though this may be an interesting avenue for future applications.

3.5.5 Learning from Success

Once the system has completed a task it is possible to reincorporate the knowledge gained from the execution back into the system for use in future actions. Updating the models themselves is a time consuming and computationally expensive task that can also be detrimental under the wrong circumstances. If the robot repeatedly executed the same actions within a single environment it could result in over fitting the model, reducing its capacity to represent a class of actions. Alternatively integrating too many varying executions can over generalise and reduce the models ability to discriminate between deviant and normal behaviour. Instead an interesting focus is on the knowledge base maintained for comparison in the anomaly detection algorithms. These represent the robot's previous experiences with a task, and updating them can allow more accurate error detection.

For the TSC algorithm a maximum and minimum count of how often a state has been observed at each time point is maintained. Updating this is straightforward, as in order to compare the current execution against the previous experiences it is already necessary to derive a state count for each point in time. If at any point in the execution the current count falls out with the expected count this can be recorded, and if upon completion of the action it was deemed a success then the knowledge base at the specific times can be updated to account for the discrepancies.

The GLPD algorithm requires a record to be kept of the average gradient and the corresponding standard deviation for each of time point. In order to incorporate new knowledge into this it requires that a summed total of all the gradients at each time point is maintained, alongside the number of individual gradients that make up this sum. This allows successful executions to take the gradients they derived during the task and integrate them into the knowledge base by adding them to the summed total, allowing new mean and standard deviations to be calculated for each time slice.

Due to the nature of the recovery actions alongside the pruning of what is deemed bad states, failed executions can in theory be used in the enhancement of the knowledge base as long as they eventually complete their goal. However while the majority of executions which have encountered failures do appear normal upon completion, there can be anomalous states left throughout the trace which are outside the scope of the state pruning algorithm. If these get incorporated back into the system it can reduce the integrity of the execution monitoring, and as such the system does currently not incorporate experience from tasks with failures.

3.6 Introspective Action Execution Framework

With the main components of the system now defined, the high level framework from Section 3.2 can be revisited, and explained in more detail. The work in this thesis is set within an existing platform called the MADbot Architecture [15]. This platform was chosen as it provided an initial foundation to interpret and dispatch plan actions into low level commands, and demonstrates the ability for this work to be integrated into an existing system. A full overview of the introspective execution framework can be seen in Figure 3.12.

As before at the top sits a planner, which calculates a series of actions which move the

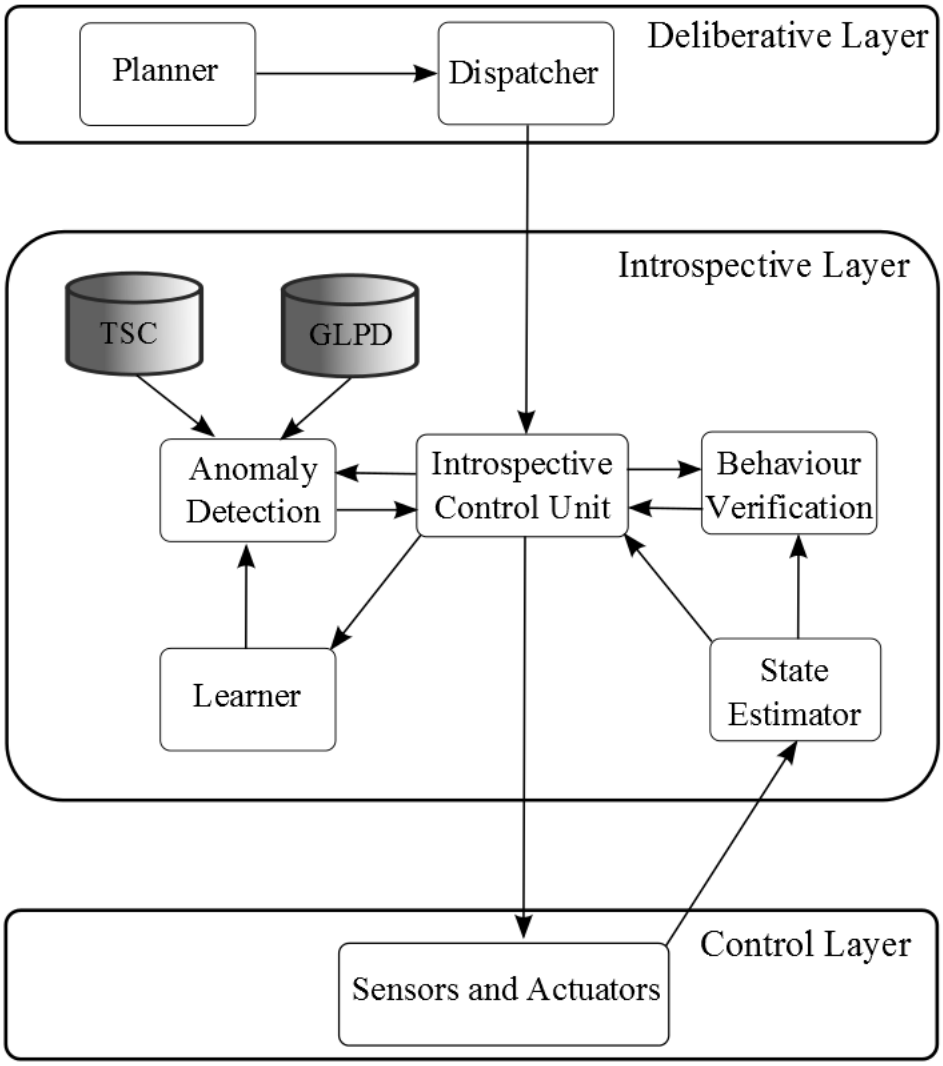


Figure 3.12: A more detailed representation of the introspective based recovery system, highlighting the connections between each of the key components.

system towards a defined goal. The dispatcher takes each action in turn and ensures the preconditions are met before handing it off to the executive to start executing. This action is passed to the introspective layer which interfaces directly with the low level control to begin the action executing. Once the execution starts the raw data from the sensors flows back into the introspective layer and is converted into observations. If the current action belongs to a class of action that encompasses multiple behaviours, the behavioural identification module examines the incoming data and selects the most appropriate model for the current action. This model is then passed to the Introspective Control Unit (ICU) which combines it with the incoming observations.

The ICU is the core of the system and is responsible for the monitoring and control of the actions. It utilises the hidden Markov models along with the Viterbi algorithm described in Section 3.3 to predict the most likely sequence of states representing the action's behaviour, based on the current data and the previous observations. Every time a new state is generated the ICU passes the current sequence to the anomaly detection module to verify the execution, which it does so by comparing the trace against a knowledge base of the system's previous experiences via the GLPD and TSC algorithms. If the action is deemed to be executing within the boundaries of the executive's expectations the system continues to monitor it until it reaches a terminal state. If however an anomaly is detected, the ICU backs up and halts the execution of the current behaviour and a recovery action is initiated.

In this case the system begins to execute the recovery action and the stream of data from the sensors flows through the new model monitoring it until it reaches completion. Once the recovery action completes the ICU switches back to the original model, loading up the previous states and observations and commencing the state pruning algorithm to allow the action to resume from the last known good state in the trace. The nature of the models coupled with the pruning algorithm means that after each failure the system resumes as if the problem had never occurred, allowing multiple faults to be tackled within each run with no significant deterioration to the execution as a whole. This provides the benefit that even if an action experiences a troubled execution with multiple failures, as long as it completes its goal the higher levels of the system need only be informed that the action has completed.

However there may be problems at this level that the introspective layer cannot handle. If an action continually fails, through either the recovery action not being able to fix the problem or even the recovery action itself failing, then there is a limit within the

system for recovery attempts which if met will cause the introspective level to abort the current action and inform the planner of a task failure. This limit is derived empirically, and is designed to provide the system with enough attempts to overcome a difficult obstacle, without allowing it to spend too much time and resources on a problem it cannot fix. For each consecutive action failure a counter is increased, and once this counter matches the attempt limit the action is cancelled, and the planner initiated. In the system described here most of the execution problems would be unlikely to be solved by the planner, however it does open the work up to be integrated into other systems that may use different levels of failure prevention to recover from problems. This scenario will be revisited in the next chapter.

This framework allows failure to be identified and recovered at the executive level. It allows a more thorough approach to tackling failure than a reactive system as it can reason over the full action, predict failure before it occurs and make an intelligent decision of when to interject and correct the execution. It also provides a faster response to a problem than many deliberative techniques such a replanning, as the executive can instantly respond to a failure the moment anomalous behaviour has been detected without having to pass it up through the system to deliberate on or disrupting the overall plan.

3.6.1 Example Usage

An interesting example of this system is the robot approaching a doorway. The low level control software which handles the robot's basic navigation will regularly encounter problems when attempting to enter doorways. Often doors are left in various states of closure which can interfere with the robot's path planning, however even when fully open the robot can have trouble proceeding. The size of the robot, angle of approach and the default setting for clearance needed for navigation often cause the robot to fail, resulting in repeated bouts of hesitation and searching. By implementing the introspective execution control and enhancing the executive this can overcome the limitation in the control software.

The action starts by the planner dispatching a `navigate_doorway` command, which is passed to the executive to begin execution. The ICU receives this command and interfaces with the control software to start the action. Upon initialising an action the data from the sensors is passed into a feature extraction algorithm, which combines

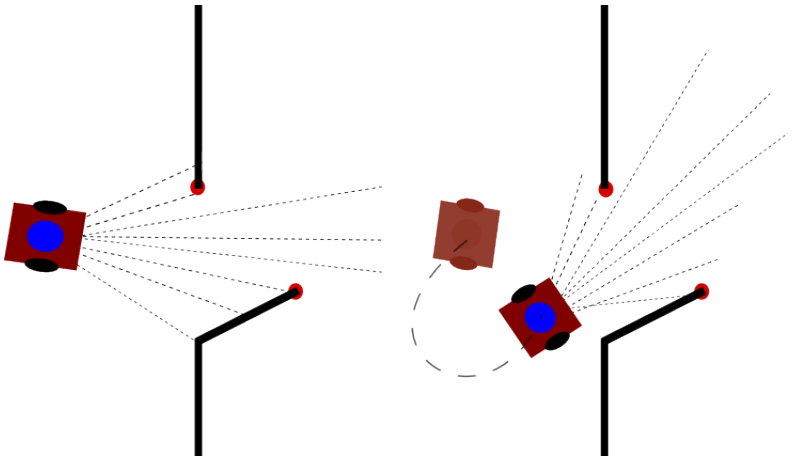


Figure 3.13: A visual representation of the doorway identification algorithm. When approaching from a bad angle the robot often gets confused attempting to traverse through doors. The recovery action attempts to identify the centre opening of the door and place the robot so that it is facing it to allow the initial enter door action to complete.

the raw data into a set of features suitable for use with the models. Initially this data is passed into the behaviour classification module to determine which of the possible sub-models is currently executing, however in the case of the doorway navigation only one model of this type exists causing the system to bypass the behaviour selection stage and instead load the appropriate model into the ICU.

In the ICU the features extracted from the sensors are classified into observations, and the Viterbi algorithm coupled with the currently selected model predicts the current state. As the action progresses the system generates a sequence of states representing the current behaviour, and this is passed to the anomaly detection module to verify it falls within expectations. In this example the robot approaches the doorway as expected, however upon getting closer begins a cycle of hesitation and incremental progress. The system's introspective ability allows it to reason over the task with an understanding of how it should be progressing, and compare this to the current behaviour. The executive knows the states should represent the robot advancing towards its goal, but instead finds the hesitating behaviour. Technically this hesitation manifests itself as a series of states that are unlikely for this particular stage in the model, which flags up as unusual state counts within TSC, and causes the probability of the trace to drop which will be identified within the GLPD algorithm.

The TSC and GLPD algorithms will continue to identify deviant behaviour until one of

the algorithms reaches its threshold value. At this point it is determined that the current behaviour is deviating too far from a normal execution and could indicate the onset of a serious failure. The anomaly detection module sends a message to the ICU which halts the execution, backs up all previous data about the current action, and transitions the system into executing the recovery model. This effectively stops the execution at the last state of the `navigate_doorway` model and restarts it from the beginning of the doorway identification model. All data from the sensors now flows into the new model and the system monitors this execution ensuring that it behaves as expected.

The completion of the recovery model leaves the robot centred and facing the direction of the largest space in the doorway. Control is then passed back to the initial model, which using the state pruning algorithm removes the last section of bad behaviour, which in this example represents the robot hesitating in front of the door. With these states removed the robot resumes the behaviour from a good state, in both an abstract and physical sense, and the robot navigates through the doorway. This final section is monitored as before, and the new states are integrated seamlessly into the execution trace.

3.7 Transparent Reasoning

A side effect of this system is the ability for the robot to convey its reasoning to an operator in an intuitive manner that permits easy monitoring. One of the goals of this thesis is to provide the robot with an introspective ability that allows it to monitor and reason about its own actions, then make intelligent decisions based on this. The models that provide this are behavioural models that attempt to represent the robot's behaviour as viewed from an outside observer. As a result this can provide an insight in to what the robot thinks is happening at each point during the execution, including identifying and reacting to failure, and display its decision process.

An example of this is during an execution the robot can inform the operator that it is starting a navigation task. At the beginning of this task it loads up the navigation models, and after a brief period of time confirms that it now believes that it is navigating within a corridor. As it is progressing it can alert the user to say that it thinks the navigation is beginning to deviate from the expected behaviour. Shortly after it updates to say that the behaviour has continued to deviate and is now classified as anomalous, and it believes it has become delocalised. It informs the operator it is going to attempt

a relocalise action, and then alerts them again when it is complete, and that it will now resume the navigation action. This can continue until the goal is reached.

This kind of transparency can be useful if the system is performing a critical or important task. Rather than a black box, it means that if someone is monitoring the progression and recognises that the system is making a poor or outright wrong decision they can intervene and correct it before it goes too far. It also means if a problem does occur someone can look back and observe at every point what the system's assumptions were and why it made the decisions it did, even including the corresponding probabilities. While this is a useful feature in robotics it can be critical if applied to other domains. The behavioural models discussed have been the basis for condition monitoring systems for electrical transformers [12], and this work can be transplanted onto similar scenarios to allow monitoring and control. In these situations having a transparent system that can communicate with the operator is a desirable quality and helps build confidence in the approach.

3.8 Discussion

Prior to this work the introspective models discussed have only been used to model and monitor a single task in a controlled environment. This chapter presents a framework that extends these models from monitoring into control, and expands the scope of their use to cover entire plans executed in dynamic environments. The framework creates an introspective layer within a system's executive, which provides it with a means to understand the actions that it is executing and determine if they are progressing as expected. If an action is not aligning with expectations, the executive can interject with a recovery action in an attempt to alter the system's behaviour and transition it into it a new state which will allow the initial action to complete. This allows the executive to tackle unexpected situations during an execution at a level between simple reactive behaviour and high level deliberation.

The example provided towards the end of this chapter describing the robot failing when entering a doorway demonstrates the utility of this system. Due to a deficiency in the low level control software the robot struggles to traverse through doors, failing more often than it succeeds. This is something, outwith rewriting the control software, that cannot be fixed when working with this platform. It is also something that passing up through the system would have little effect on. If upon failing to traverse the door

the planner is alerted, it has very little recourse. If the goal is through a door then the planner is too high level to reason about what is causing the action to fail, and there might not be an alternative action or plan that can lead to a favourable outcome.

This introspective execution framework presented can overcome this shortcoming by increasing the intelligence of the executive. Without altering the low level control or increasing the complexity of the high level reasoning this can add a greater level of robustness to the system. The executive is poised at this intermediate level between these two layers that allows it to reason over the full task in an abstract nature, but with enough detail that it can directly and intelligently alter the execution. This allows problems to be recovered and decisions to be made that are not possible at the higher or lower levels of abstraction, which coupled with the introspective models allow a potentially powerful execution architecture to be created.

The work described in this chapter is a proof of concept of an introspective executive and features some limitations, the most important of which is lack of decision making with regards to selecting between recovery actions in the event of a failure. The executive's normal role is to organise and execute actions in a plan, and with the addition of the introspective models it can reason more intelligently about the progression of these actions. However, neither the executive or the models themselves have explicit control built into them that can reason about the state of the system at this point of failure. The introspective framework allows for more intelligent behaviour to emerge from the executive with regards to executing actions, however in order to have a wider application the executive needs to contain some element of control. This will be investigated in the next chapter.

Another drawback is the lack of resource monitoring. One benefit of this system is that the executive can handle failure at the intermediate level without having to alert the planner to problems within the execution. An action can fail multiple times in a single run and the executive can monitor, catch and correct them, only communicating with the planner at the end when the task has a successful completion. This is an interesting addition to the system, allowing robust mid level control which can reduce some of the complexity of the deliberative reasoning component. However under certain conditions it can be unfavourable. If the resource allocation for a particular goal is highly constrained, the system failing multiple times and correcting itself could consume more resources than were anticipated, and result in a overall mission failure. There are additions to the planner that can help mitigate some of this by allocating each

action specific flexible windows of time and resources [80], however this involves increasing the complexity of the higher layers. This will also be addressed within the next chapter.

CHAPTER 4

CREATING A MORE INTELLIGENT EXECUTIVE

Chapter 3 examined the concept of a more intelligent executive through a novel method of action execution. This provided the system with the capability to introspect over its own behaviour as it executes, and to be able to recognise and respond to possible deviations within that behaviour. When combined with a symbolic task planner this created a system which could better achieve its goals through more robust and reliable actions. This work was designed to explore the use and feasibility of the introspective models as a means of identifying and recovering from abnormal behaviour within the execution of an action, and features some limitations when applied to realistic scenarios. The lack of a formal control mechanism within the models means that recovery actions are tied directly to the source action, which limits the variety of obstacles that can be overcome during an execution. Similarly the lack of resource management means that if several failures are encountered during an execution then the resource consumption of the recovery actions would be unaccounted for, which can leave the system unable to complete its goal.

This chapter introduces a control structure that sits on top on the behavioural models as a new layer within the executive. The initial aim of this control structure is to address the issues outlined above and provide explicit intermediate states for reasoning during the partial execution of an action, as well as a method for monitoring resource consumption. It also provides an opportunity to increase the introspective capabilities of

the system. Similar architectures attempt a holistic view of introspection with regards to the system, either through an introspective layer spanning all levels of operation [67], or individual introspective modules within each layer that communicate with one another [57]. By choosing a control structure that can learn from the environment as it executes it allows the system to expand from monitoring an *action's* execution and being able to alter it, to being able to monitor a *plan's* execution and being able to alter it.

Furthermore by storing the knowledge gained from executing actions within the environment, it allows the executive to learn estimations of each action's success and use this to assist in future decision making. Rather than the planner being the sole deliberative component of the system it can take on an almost advisory role, passing down a plan consisting of the ideal set of actions to accomplish a goal. The executive's role then becomes to take this plan and combine it with its own experience to make intelligent decisions on how to achieve this goal within the current environment.

The rest of this chapter will discuss how such a system can be integrated into an executive and benefits that it can provide. The first section will provide an illustrative example of a situation where a more intelligent executive could improve a robot's performance. Section 4.2 will explain a high level overview of the system, and expands upon the functionality the executive control layer can contribute to the robot. Section 4.3 will discuss the details of the control model itself in detail, explaining the main components, while Section 4.4 will outline the algorithms used in conjunction with the model to provide the additional intelligence. This is brought together in a detailed overview of the system in Section 4.5, before finishing up with a discussion of the work presented.

4.1 Motivating Scenario

In the previous chapter the robot's inability to reliably traverse through doors was discussed. Often during the execution of this type of action the robot will stall in front of the door and refuse to go any further, causing the action to fail. A scenario was presented which demonstrated that with the introspective execution control the probability of an action succeeding could be increased by allowing the robot to use a recovery action to attempt to reposition and align itself, before resuming the navigation. The scenario focussed entirely on the individual action, and was only concerned with the

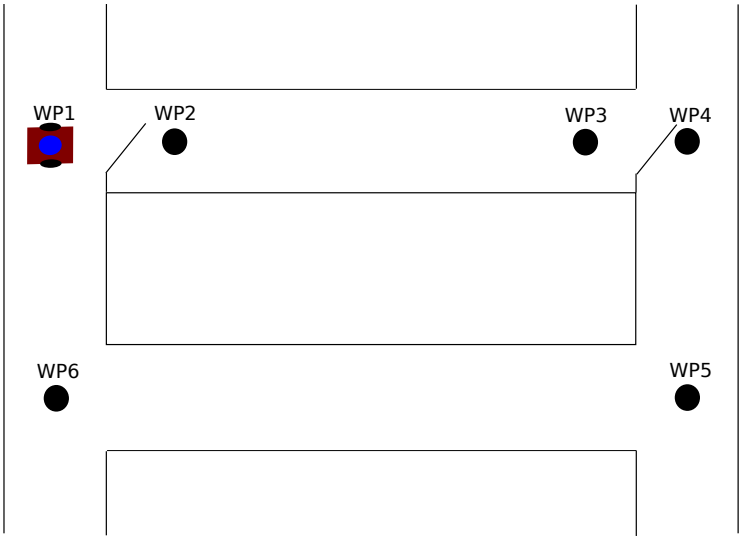


Figure 4.1: An example environment in which the robot is tasked with navigating from WP1 to WP4.

success of that action.

For this chapter a similar scenario is presented, however the focus will take a step back to examine the execution as a whole. If presented with an office environment such as the one in Figure 4.1, the robot (`BOT1`) can be given a goal of delivering a package to a person at waypoint 4. The planner can deliberate over this problem, and return a set of actions that achieve this as shown in Figure 4.2.

```
(NAVIGATE_DOOR BOT1 WP1 WP2) [20.0000]
(NAVIGATE BOT1 WP2 WP3) [40.0000]
(NAVIGATE_DOOR BOT1 WP3 WP4) [20.0000]
(DROP_OBJECT PACKAGE1 BOT1 WP4) [20.0000]
```

Figure 4.2: A simple plan to move the robot to WP4 to deliver a package. Each action is shown with its associated cost in square brackets.

While these actions represent the theoretical best route to the goal, in practice when the robot begins to execute them it will encounter a very high chance of the plan failing upon attempting to navigate through each of the doorways. If the planner is invoked it can reassess the domain and attempt to replan, which depending on the size of the problem can be time consuming but does ensure that the new actions are valid within the context of the available resources. A problematic scenario can arise if the robot

reaches waypoint 3 and fails to traverse through the next door, as any new plan to the goal will require a NAVIGATE_DOOR action regardless of which path it chooses, resulting in a possible plan-fail-replan loop.

If the introspective recovery is activated the probability of each action succeeding increases, however attempting each doorway navigation can now take longer than anticipated due to one or more failures requiring the robot to reposition itself. Each recovery action consumes extra resources that have not been accounted for by the deliberative component, and if the above scenario is then taken as a section of a much larger plan the depletion of these resources can lead to later actions being unable to complete and the possibility of jeopardising the goal.

An important drawback common to both of the approaches outlined above is that if the system does overcome the problem, either through replanning or the use of a recovery action, it has no way of capturing this knowledge for use in future executions. If at the end of this task the robot is repositioned to its initial state and given the same goal, it will view it as the same problem and attempt the same solution. If the deliberative component is based on a deterministic planner then no matter how many times the plan or actions fail, or how many times replanning or recovering the action helps, if posed the same problem it will generate the same solution, unless an engineer specifically intervenes to update its domain knowledge.

Ideally a robot should be able to attempt a range of solutions to overcome a problem and improve its execution. If an action fails it should be able to recover it, and attempt to continue towards the goal. However if the action is repeatedly failing it should be able to recognise and adapt to this, and attempt to update its execution strategy to compensate. But importantly, when attempting either of these solutions it should be able to learn from what it is doing, and be able to use this knowledge in the future.

4.2 Introspective Controller

One way to address the issues highlighted above is to continue to increase the intelligence of the executive, providing it with a greater ability to reason over the execution as a whole. The goal of this is to move the executive away from being a component primarily designed to sequence and execute plan actions, and towards being an intelligent entity that can utilise its own knowledge and experience in conjunction with a plan

to make informed decisions about the execution. This splits the deliberation over the two components, while still maintaining a close collaboration between them to more robustly achieve the system's goals.

The top level deliberative component continues to operate with a high level deterministic view of the problem, containing numerical constraints and resource consumption. Its role is to use well established planning techniques to search through large state spaces and synthesise a plan that can transition the robot from its initial state to its goal state. The executive's role then moves from trying to ensure the correct execution of individual plan actions, to trying to ensure the correct execution of the plan itself.

This changes the relationship between the planner and the executive to that more akin to a human following instructions. If a person arrives in a new city and has to travel to work, they are likely to look up directions. As they are new to the city and it is their first time driving to this location, they will probably follow these directions step by step. Even if a problem arises during the journey, such as bad traffic or someone breaking down, most people will attempt to overcome or circumvent the issue, but will be hesitant to deviate too far from the instructions. However as time goes on and the person becomes more familiar with the city they incorporate that knowledge into their decision making. Now when travelling to work if they know traffic is bad at a certain time, or come across an accident they are more capable to adapt and may attempt a different route for that part of the journey. Furthermore now when travelling to a new location they are still likely seek out directions, but will use their experience and knowledge on how best to apply these directions. If they know particular roads are busy or closed, they may devise alternative paths without needing to attempt them.

To achieve this type of system a new layer has been added to the executive. This layer consists of a stochastic control structure which sits on top of the previous introspective execution system. This enables greater reasoning over the actions, allowing more intelligent decisions to be made about the execution during operation. Rather than transitioning straight into a recovery action it allows the system to quickly deliberate and choose an appropriate solution, either from a selection of recovery actions or abandoning the action altogether. It also allows an element of learning to be introduced to the executive, giving it the ability to learn from its environment and apply this knowledge in order to increase the level of deliberation across the execution as a whole.

The structure chosen for this new layer is a Markov Decision Process (MDP) ([44], [7], [77]), a control mechanism offering a well understood framework for handling

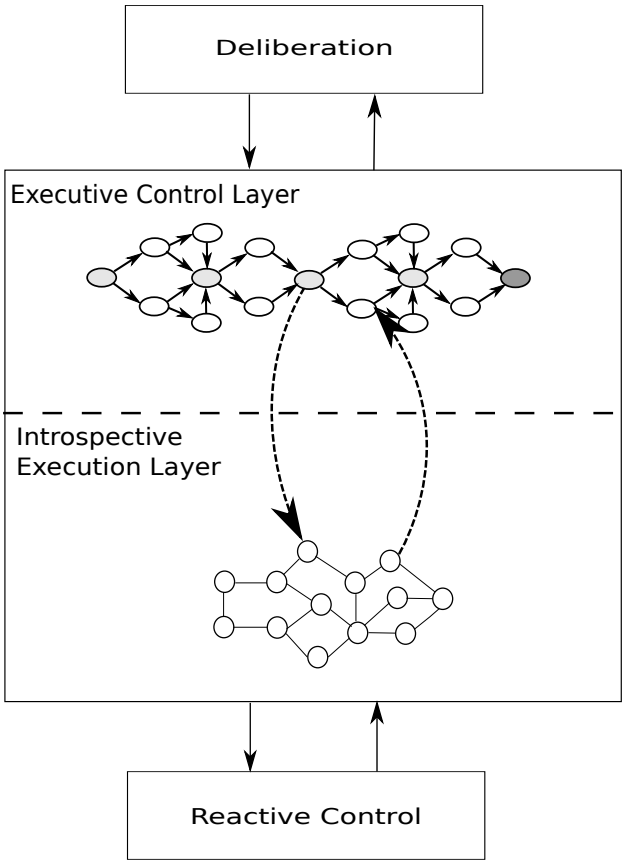


Figure 4.3: A high level view of the system. The plan is taken and overlaid on to the control model, represented by the grey states in the diagram, serving as a foundation for the policy. Each action executed is associated with an introspective model which is monitored until completion, at which point the result is passed to the upper layer to update the state of the control model and the next appropriate action is dispatched.

uncertainty and actions with probabilistic outcomes. The MDP in this work is based on an abstracted version of the planner's state space, removing all numeric constraints from the domain while adding a small number of support states to assist with action execution. The similarity between the state spaces is designed to facilitate communication between the two components, allowing the high level plan to be passed into the executive and used as the basis for its reasoning.

A high level representation of this system can be seen in Figure 4.3. The deliberative component now passes the entire plan over to the executive, rather than dispatching an action at a time. The executive takes the plan, and exploiting the similarity between the planner and the MDP's representations, incorporates it into the model producing

a policy. The policy is a mapping from states to actions, and is calculated based on a combination of the actions passed in from the plan and the executive's experience within the environment. After determining an initial state of the system, the first action is selected and dispatched to the introspective execution module. This executes the action and monitors its progress. If it completes successfully it updates the executive controller and a new action is dispatched. If the action fails it alerts the controller and it selects an appropriate response that will continue the execution.

The relationship between the planner, the executive controller and the introspective execution module allow for more robust execution of deterministic plans within non-deterministic environments. The planner operates on the largest domain, and searches for actions which will achieve the system's goals while attempting fulfil some optimisation criteria, such as plan length, resource usage, etc. However the plans produced are based on an idealised view of the world, operating on the assumption that the planner has complete knowledge of the environment and that it can reliably predict the outcome of its actions. In dynamic domains both these assumptions rarely hold.

The executive controller attempts to overcome the planner's problem of prediction by taking the plan and mapping its actions onto a MDP. This provides the system with a level of stochastic reasoning that accounts for the actions having multiple outcomes. The executive then attempts to learn these outcomes, in order to better understand how these actions will perform when executed in a real environment. It can then use this knowledge to influence future executions, and make more intelligent decisions that help achieve its goals. This executive features the limitation of assuming the environment it operates in is fully observable.

To help mitigate this last problem, the introspective execution module is used. It continues to be the foundation through this work, with the introspective models encapsulating the uncertainty faced by these actions as they operate in the real world. These models capture the dynamics at play between the robot and its environment as it executes each action, and the monitoring allows the system to determine if the action has completed within the boundaries of nominal behaviour. This allows the executive controller to reliably know if an action has completed successfully or not, and update its state accordingly.

The next sections will go into more detail of the benefits such a system can provide.

4.2.1 Reasoning with partial execution

One of the initial goals of this system was to build upon the ideas from the previous chapter and better utilise the introspective models in overcoming problems within an execution. Previously these models were tied to actions within the planner, which upon being dispatched the executive would load the corresponding model and monitor it to completion or failure. In the event of a failure the system would automatically transition into a preselected recovery action, as the planner is too abstracted to intervene at an execution level and the models themselves feature no form of control logic.

With the addition of the executive control layer the introspective models now correspond to the actions of the MDP. The MDP actions in the model are based on the plan actions, so an introspective model tied to a navigate plan action will now be associated with a navigate MDP action. This change brings the level of reasoning closer to the action being executed, and allows more detailed control over it. This is achieved by an extension to the executive's control model to include explicit states that represent intermediate points within an action's execution from which additional reasoning can be undertaken. A section of the executive controller representing an action and its possible transitions can be seen in Figure 4.4.

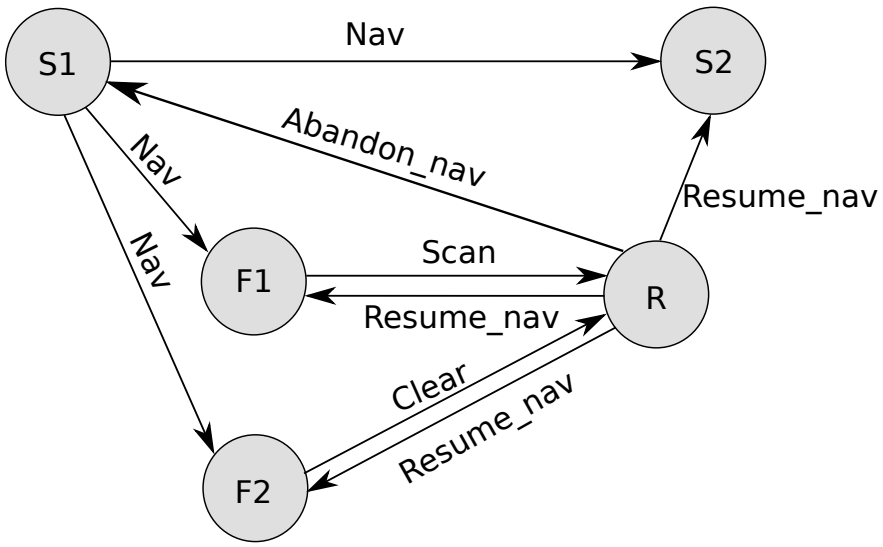


Figure 4.4: An example of a navigate action within the executive controller. Each action now has stochastic outcomes, allowing for the explicit representation of intermediate states within the execution from which additional reasoning can be undertaken.

This diagram highlights the recovery state (R) and multiple failure states ($F1$, $F2$) that can now be associated with each action the executive can execute. As before when an action is dispatched, now from the MDP, the introspective execution module loads up the corresponding model and monitors the execution. If the execution is successful it updates the controller which transitions into the next state ($S2$) and dispatches the next action. If it is unsuccessful and a failure is detected it pauses the execution and alerts the controller to this issue. The controller can then classify the cause of the failure, discussed further in Section 4.4.4, and updates the system into the relevant failure state.

The failure states represent a set of abstract problems that can arise during the execution of an action. They provide the system with a point to reason about and alter the execution without having to abandon the current action, or pass control away from the executive. As before when a failure is detected the current action is paused, and its progress through the introspective model is backed up. However instead of automatically transitioning into a recovery action, the executive can now deliberate on the situation and choose an appropriate response. In order to not disrupt the execution and minimise the transition time between models the types of failure an action can encounter are represented within the state space of the MDP. Recovery actions are also explicit within the control model, with each failure state having a corresponding recovery action associated with it. This allows these actions to be built into the policy, such that when a failure is diagnosed it will update the current state of the executive into the relevant fail state, such as F1 in the example above. While an action may now have many different failure states associated with it, each failure state itself only has one recovery action. This means upon entering a failure state the policy will immediately instigate the specified recovery action, persevering the smooth transition between models.

Upon successful completion of a recovery action the controller transitions into a recovered state for the initiating action. The recovered state represents a partially complete point within the execution from which a failure has been overcome, but the previous action has not been resumed. This provides a further deliberation point within the partial execution of an action for the executive to decide on the best progression. The previous action can be resumed, which as before loads up the partially completed introspective model and resumes the execution from a more suitable point with which to complete the action. Alternatively the previous action can be abandoned, and the system can attempt a new recovery action to transition it back to the original state from

which the action was selected.

The addition of a controller within the executive on top of the introspective execution system allows the robot to be able to operate more robustly in dynamic environments. It allows the executive to ensure actions are operating within acceptable bounds, and interject with an appropriate response once it feels it is veering too far out with these bounds. This allows more intelligent execution of actions, which in turn increases the system's ability to achieve its goals.

4.2.2 Learning From the Environment

Another important component of this system is for the executive controller to be able to learn from its environment, in order to identify and avoid possible sources of failure. One of the aspects that sets the controller apart from the planner is the treatment of its actions as probabilistic, and the use of these probabilities to reason over how effective the current plan is likely to be when operating in the observed world. If the probabilities of these actions are static they are likely to go out of sync with the dynamic aspects of the environment they are supposed to represent, greatly diminishing their use.

As such whenever an action is executed within the world the system is designed to take that knowledge and incorporate it back into the model. If an action is successful this should be noted, and the probability of that action being successful in the future increased. If an action is unsuccessful then that should also be noted, and the probability of that action failing alongside the probability of recovering from that failure should be learned for use in the future. This allows the executive to develop a realistic estimate of which actions are likely to succeed and which actions are likely to fail, essentially learning how to operate effectively within its environment as opposed to how to better achieve a specific goal.

Similarly, as it is important to try and learn the dynamics of the environment it is important not to adhere to this knowledge too rigidly. A common problem when operating in uncertain domains is the dilemma of *exploration versus exploitation*, or when to use the acquired knowledge over when to attempt to acquire more. If the robot constantly fails at an action its probability of success will continually decrease indicating that the action could be avoided in future executions. While on the surface this is desirable, if the failure was caused by a transient environmental condition, such as an obstruction in a corridor or an object not being in the correct position to be picked up, over time

these actions may be achievable again. As such a degradation function is added to the learning component so that over time actions which are rarely chosen move back to their original probability.

4.2.3 Intelligent Plan Execution

One of the most important aspects of this executive is the ability for it to intelligently execute the plans passed to it by the deliberative component. The purpose of adding a separate controller into the executive is not to replace the planner – there are large bodies of work concerned with decision theoretic robot control – but to augment it. Symbolic planning has had a great deal of research focused on improving search through complex domains with numerical components [35, 54, 16]. While it is possible to add numeric elements to an MDP it causes a large state space explosion that can make typical manipulation difficult. Techniques have been developed to represent these large state spaces in more efficient manners, such as hierarchal MDPs [41] or factored MDPs [11], but these can still be slow to solve and produce less accurate solutions.

The combination of the planner and the MDP allows for intelligent execution within dynamic environments. The planner is designed to search for a set of actions to achieve the goal under known conditions. The executive is encouraged to follow these actions whenever possible, however real situations can often arise which cause disruptions to the execution and can invalidate the plan. The previous section discussed how the executive's controller allowed it to better handle these disruptions, by being able to choose a recovery action which could bring the execution back within expectations. However sometimes recovering and completing an action is not possible and other steps need to be taken.

If an action continually fails, it might not be possible for the executive to recover and overcome the problem. If for example the robot is trying to traverse a corridor and the path is completely blocked, there is little the robot can do in this situation to successfully complete its action. This is where the learning and control intersect to help make an informed decision. As the executive attempts to execute the task and fails, it updates the model and the probability of success of the action lowers. If the action fails enough the control policy will update and attempt to locate an alternative action (or set of actions) which can achieve the same result, if any are available. The recovered states within the actions are where the executive can decide midway through

an execution if it is worth attempting to complete an action, or to abandon it if another solution has been found which can achieve the goal with a higher success.

The executive's knowledge from the environment is also used to reason over a plan before it executes. Similar to the manner in which the introspective models allow the executive to use its past experiences to analyse its current actions, the MDP allows it to use its previous experience in an environment to reflect upon the current plan. It allows the introspective reasoning to expand from an action level to a plan level, and provides the system with a means to reason about how likely a plan is to succeed.

If as in the example above the robot has attempted a particular path and has failed due to a blockage, aside from recognising this and looking for an alternative, the executive's control model can store this information for use in future executions. If a new plan is passed to the executive which features the same path, rather than attempting it again and encountering the same obstruction, the system can now exploit its knowledge and attempt to find a new set of actions which can achieve this part of the plan before it begins execution. This means that if the robot does have a consistent problem, such as difficulty passing through doors, the executive can learn this and attempt alternative actions it feels are more likely to succeed.

Even in scenarios that do not directly impede the success of the goal, the executive's ability to learn can help improve the quality of the execution. If the planner has a choice of two similar routes, it will select the one which best fits its optimisation criteria such as the shortest path or the least likely power consumption. However from an execution perspective these two paths may not be equal. The one thought to be the better choice may be a busier pathway, causing the robot to have to regularly attempt a recovery action due to people obstructing it. This does not make this path a bad choice or indeed one which should be avoided, but if there is an alternative which is more likely to succeed this may be a better selection from a pragmatic point of view. In certain scenarios it can even improve the efficiency of the execution, due to the reduced failures. The executive can learn this environmental knowledge and use it to improve decision making.

4.3 Constructing the Control Model

The previous section describes the concept of the executive control system, the relationship between its major components and the benefits it can produce. This section will focus on the details of the implementation. The executive controller is based on a MDP, which is a decision network with stochastic actions and modelled utility. The probabilistic nature of these models allows the inherent uncertainty of dynamic environments to be incorporated into the model, and the utility function provides a measure of performance which can be used to optimise over. This makes MDPs a natural formulation for many robot control problems, and has lead to extensive use of them within the literature [90, 73, 62, 84].

The initial direction for this work involved taking the fully constructed plan and converting it into a MDP. This would take the initial state and the set of plan actions and apply them sequentially until the goal state was reached. These actions would be augmented with the stochastic outcomes leading to possible failure states, and the failure states would have the recovery actions associated with them to move the system back onto the main path through the plan. This would quickly create and solve small control models based on the plan that explicitly represent the intermediate states within an execution. This approach provided the executive with a greater ability to reason over an action's failure, however if a recovery action could not overcome a problem then the policy would fail and the system would need to instigate a replan. The disposable nature of plans also limited the system's ability to learn, as each control model would be relevant only to the current goal.

In order to build upon this, and increase the capabilities of the executive such that it can reason about the execution as a whole, the domain based MDP was created. This involves constructing the model based on an abstracted version of the planner's domain, rather than the plan itself. A plan domain is a set of predicates and uninstantiated operators that reflect the concepts in a domain and how the planner can manipulate them. In a traditional planning approach this is combined with a problem file, which provides the initial grounded state from which to plan. The MDP requires a similar initial state, however this is for the generation of a generic model for the specified domain, and not for a specific instance of a problem.

This initial state for the MDP is referred to as a seed state, and is a grounded state from which the rest of the model is based. Actions are applied to this state in a sequence,

and the resulting states are stored in a queue. Control knowledge is used to derive the probabilistic effects of actions, such that if a specific action is applied, it will automatically generate the possible failure states associated with that type of action. The MDP is then grown by taking a state from the queue, applying a set of actions to it and storing the results, then removing another state from the queue. This is repeated until no states are left and the domain has been fully expanded. The seed state remains unchanged as long as the domain does, ensuring the model stays consistent through out different goals and executions.

Using similar representations to model the problem allows both components to easily communicate and pass information between each other. The domain based representation also allows the executive's control model to be separated from a generated plan, allowing a consistent representation to be used for as long as the domain itself stays the same. This consistency is what allows the model to learn from each action executed within it, and provides the main source of the executive's intelligence. The downside of this representation is that it involves the enumeration of the state space, which for a complicated domain can get very large. As the domain increases in complexity, the state space can face an exponential increase in size, commonly referred to as the *curse of dimensionality*. This can at best make these models slow to solve and at worst make them completely intractable.

This problem can be somewhat mitigated within this work through careful design of the domain and the use of abstraction. The introspective execution module allows the simplification of the domain for both the planner and the MDP, by allowing the actions to represent high level compound tasks. These actions can reliably achieve complex goals which may previously have required multiple individual actions to represent. The use of recovery actions during execution can also reduce the need for certain aspects of the environment to be represented within the model. As an example if the robot has a recovery action which can open a door, or alert someone nearby to do it for them, it can allow the status of doors to be stripped from the model. This allows the executive to handle the problem during execution and prevents the need to represent the status of multiple objects, which can be particularly useful if these objects are dynamic and their status is likely to become outdated.

Using an abstracted version of the planning domain for the basis of the MDP also allows a more compact and tractable state space. By stripping out all numerical factors from the domain it reduces the size and complexity of model significantly, resulting

in a solvable state space. Furthermore, using a shared state representation between the planner and the executive controller allows the created plan to be overlaid onto the MDP states and used as the basis for the policy, allowing standard decision theoretic techniques to achieve a quick convergence.

The model used in this work is a discrete time, infinite horizon Markov Decision process with static transition and reward functions. This can be formally defined as a tuple of the form (S, A, T, R) where

- S, a finite set of states representing the environment;
- A, a finite set of actions;
- $T : S \times A \times S' \rightarrow$ [0,1], the transition function representing the probabilistic effect of taking actions. $T(s, a, s')$ is the probability of moving to state s' after taking action a in state s.
- R, a reward function used to encourage the robot towards desirable states and discourage it from possible instances of failure. $R(s)$ is the reward for moving into state s upon completion of an action.

Each of these components will be briefly discussed.

4.3.1 States

Before discussing the MDP states, it is worth defining the plan states from which they are based. Plan states represent the system's knowledge of the world at a specific time, such as the current location of the robot; the location of important objects in the environment; the power level of the robot etc. These facts are represented as grounded propositions in the form `(room r1)` and `(at robot room1)`, and the states are composed of collections of these propositions.

The states used in the MDP are abstracted versions of the plan states. They use the same grounded propositions as the planner to represent the state of the robot and its knowledge of the environment. The main difference between these two representations is that the MDP states feature an additional set of propositions relating to possible failures, whilst also lacking any numerical fluents. The similarity between the two representations is designed to allow easy communication between the two components. States derived from the plan can have their numerical component stripped and and

mapped directly onto the MDP for use in its solution, and the reverse is true where the current state of the MDP can have the numerical propositions calculated and added to allow use within the planner.

For each action the possible types of failure are identified, discussed in Section 4.4.4, and these failures are represented in the model through additional propositions. This extends the state space and creates intermediate states within each action which represent abstract failures and post failure recovery, as shown in Figure 4.4. These extra states provide the executive with the facility to reason within the partial execution of an action, allowing it greater control and influence over an action's outcome.

4.3.2 Actions

The actions used within the executive's control model are based on versions of the planner's actions, such that every action in the plan will have a corresponding executive action. These represent the ways the robot can interact with its environment, and update the current state of the executive upon completion. The difference between the two representations is that the MDP actions are stochastic, and are extended to encompass multiple outcomes based on the performance of the action. The set of actions available to the executive has also been expanded beyond that of the plan actions, in order to accommodate these new outcomes.

In order to determine the result of an action when executed in a dynamic environment, each action in the executive is associated with an introspective model. These models monitor the progression of an action as before. However previously when the introspective models were associated with plan actions, if they encountered a failure they tried to shield it from the planner and attempt to overcome it at the execution level. This was due to the planner being too high level to reason over execution problems within individual actions. Now when a failure is encountered it can be recovered as before, without the need to abandon the current action. However with the addition of the MDP controller and explicit intermediate execution states the outcome of an action can be represented regardless of success or failure. This allows the executive to select an appropriate response to overcome the current failure.

4.3.3 Transition Function

The transition function represents the stochastic nature of the robot interacting with its environment, and defines the probability of moving from one state to another via a specific action. Deriving sensible probabilities for each action which represents a realistic estimation of the system is a difficult task, and one often attempted through the use of simulation techniques, long term observation or specialist knowledge. In this work the transition function is initialised using empirical data gathered during the learning stage for each of the action models. During this process executions that contain failures are separated from the main learning library, manually labeled with the cause of failure and set aside for later use in verification. This can be used to give an idea of the type of failure each action can encounter, and when combined with the full set of verification data, the frequency of these encounters. These frequencies can then be used to derive a probability of an action succeeding or entering a fail state.

More formally, for each action modelled a, a verification set is created recording each of the possible observed outcomes o, where $o_a = \langle o_a^1, o_a^2, ..., o_a^n \rangle$. For each o_a^i there is an associated count $c(o_a^i), \forall i \in [1, |o|]$ which represents the number of times an outcome has been observed during training, $c(o_a^i) \in \mathbb{Z}^+$.

A probability, ϕ_a^i, over each of the action's possible outcomes o_a^i, can then be derived through:

$$\phi_a^{o^i} = \frac{c(o_a^i)}{\sum_{x=1}^{n} c(o_a^x)} \tag{4.1}$$

These estimations provide a set of priors for each class of action, and can be used to initialise the transition function for the grounded actions belonging to each class.

Due to the relatively small and homogenous nature of the sample set used in verifying the action models, this starts the MDP with a generic transition function with a predisposition to actions completing successfully. This is a deliberate choice designed to encourage the executive to follow the plan actions when operating in new environments in which it has no prior knowledge. As the system executes it can learn the underlying interactions between specific actions and the environment, and update the transition function to better represent these.

4.3.4 Rewards

Within an MDP the reward function specifies the utility of the system. This provides an optimisation criteria for use when deriving a policy, with most techniques designed to solve MDPs attempting to find solutions which maximise the expected reward through a system. In this work the reward function is based on the plan passed down from the deliberative component.

As previously discussed the planner in this architecture has a broader view of the domain, and searches through large state spaces which model resource consumption, distances between objects, and other numerical constraints that need to be considered when producing a plan. This plan, while not representing any of the uncertainty inherent in the execution, is the best set of actions to reach the goal taking into consideration all other aspects of the domain. As such the executive's role is to execute this plan to the best of its ability and only deviate from it if a section of the plan becomes infeasible.

To emphasise this and support cooperation between the planner and the executive, the plan is used as the basis for the reward function. Using the initial state given to the planner, each action of the plan is applied in sequence, producing a set of states representing the trajectory of the plan to the goal. These states are then transformed into their corresponding states within the MDP, and assigned a reward. The goal state receives the largest reward, to ensure all policies converge towards it. States that are determined to occur within the plan trajectory receive a partial reward, to encourage the policy to follow the plan without rigidly enforcing it, and failure states receive a negative reward to dissuade paths that are likely to fail. These rewards are assigned to the states themselves, rather than the traditional state-action combination. This is designed to not only encourage the executive to follow the plan, but to ensure any deviations from it to realign with the plan with as little disruption as possible.

4.4 Executing Plans in Dynamic Environments

The main goal of this work was to improve the executive's ability to execute plans in dynamic environments. The previous section discussed the model used to represent the executive controller, and each component's relationship to the plan. This section explores the algorithms and modules within the executive which use this model to improve the system's execution.

4.4.1 Deriving Polices for Execution

With the model initialised and a plan produced, the executive's role is to derive a control policy based on a combination of the two that achieves the system's goals in a robust manner. A policy is a mapping from states to actions such that every state within the domain has an action associated with it, and each action will move the system towards its goal. Unlike the plan which produces a sequence of actions, this prepares the system to act from any state from which it can transition into.

Within the literature there is large bodies of work dedicated to generating polices through large complex domains, however many of these focus on calculating the policy offline [83, 86] before deploying it to a robot for execution. The nature of this work instead requires that the policy is created onboard the robot itself prior to the execution commencing. As the plan is the basis for the reward function, each new plan requires a new policy. Furthermore these policies can be revised online during the execution, based on the robot's current experience. Due to the abstraction techniques and domain design discussed above, coupled with the size and type of problems the robot has currently been applied to, the state space in the executive is typically in the range of hundreds of states, rather than the hundreds of thousands.

Due to the relatively small size of the domains, value iteration [7] is used to generate the polices. Value iteration is an iterative process which calculates the utility of each state based on its successor states, and continues to revise these utilities until a stopping criteria is met. Typically this is when the maximal difference between two value functions falls below a threshold ϵ (known as the Bellman residual), the value of which can be used to create approximate solutions. Given a sufficient value of ϵ and a discount factor $\gamma \in (0, 1)$ value iteration is guaranteed to converge and produce optimal policies. The function designed to calculate the expected discounted reward based on the Bellman equation:

$$V^*(s) = max_{a \in A} \left[R(s) + \gamma \sum_{s' \in S} T(s, a, s') V(s') \right] \tag{4.2}$$

Once the utilities converge, the policy π^* can be extracted via:

$$\pi^*(s) = argmax_a \left[R(s) + \gamma \sum_{s' \in S} T(s, a, s') V(s') \right] \tag{4.3}$$

The initial policy is created during the initialisation of the robot, however during the execution the policy does not necessarily remain static. As the robot interacts with its environment it is constantly gaining knowledge, and this knowledge can be incorporated back into the policy. One of the important aspects of this system is the ability to use learning not just to improve future executions, but to improve the current one. This continues the theme of introspective control by allowing the system to reflect on the progression of the system towards the goal and determine if it is still feasible. If an action continues to fail it can jeopardise the rest of the plan, and alternative actions may need to be taken to ensure the completion of the goal.

With respect to the policy this means that every time an action completes, regardless of success, the transition function is updated and the policy is revised. These updates involve rerunning the value iteration algorithm with the updated transition function, which due to the pre-existing value function coupled with the incremental nature of the change, results in a quick convergence to a new policy. To further ensure no disruption the updates happen during the execution of the subsequent actions. If an action completes successfully, the updated transition will have little impact on the current policy and the revision can occur after the next action has been dispatched. If the action fails, the system transitions into a failure state, from which the only option is a recovery action. The execution of the recovery actions masks any time needed for the policy to update, and upon completion moves the system into one of the recovered states from which any change to the policy can be executed.

4.4.2 Incorporating Experience

The transition function that the robot is initialised with represents a generic but optimistic view of the environment. The transitions are derived from the verification data used to evaluate the associated action's behavioural model, and each grounded action of the same class is represented with the same probability of success. This means the probability of successfully traversing one corridor is the same as any other, and each doorway is treated alike. Realistically the same action executed in a different part of the environment can produce drastically different results. Some corridors may be quiet

and easy to navigate, while others can be busy or cluttered. Similarly different doorways can be different widths, changing how easy it is for the robot to pass through them on a first attempt. Even if these probabilities were representative for each action, over time the environment can change and they would become outdated. This can result in the executive facing a similar problem to the one its trying to solve, making predications on a model that no longer necessarily represents the current state of the world.

Being able to learn from each action as it executes provides the system with an insight into the dynamics of the environment. It equips the executive with an understanding of which actions are more likely to succeed and which will likely fail, enabling it to make more appropriate decisions with regards to successfully achieving a goal. A problem with this however is that gaining experience is expensive, especially as the number of states and actions within a domain grow. For even a single action a system would require a wealth of execution data to reliably estimate its outcome, and even then its important to ensure that just because a possible outcome did not occur, it is not disregarded in the model.

This motivates the transition function being initialised with values from the respective action's training data. While these probabilities do not represent specific actions within the environment they do represent a realistic estimate of an action's performance out with any additional information. They provide a base case for the model to reason with, and a preliminary probability distribution over each action's possible outcomes. As the robot executes it begins to gather experience on how each action actually interacts with the environment, and can combine the empirical data with the prior knowledge to make more accurate predictions about an actions outcome.

To formalise this, if in state s action a is applied, the system upon completion can transition into one of the possible successor states o, where $o_a = \langle o_a^1, o_a^2, ..., o_a^n \rangle$ and $o \subseteq S$. A prior distribution over these outcomes can be derived from the action classes verification data as described in Section 4.3.3, providing an initial optimistic estimate ϕ_a^o:

$$\phi_a^o = \langle \phi_a^{o^1}, ..., \phi_a^{o^n} \rangle \tag{4.4}$$

After each action executes the system gains an observation $z_{s,a}^{o^i}$ as to which outcome o^i was reached via the application of action a to state s. Each observation increases an

outcome's associated count $c(z_{s,a}^{o^i})$ by 1, and provides an empirical component to the data. The probability σ of an outcome o^i can then be calculated as:

$$\sigma_{s,a}^{o^i} = \frac{c(z_{s,a}^{o^i})}{\sum_{x=1}^{n} c(z_{s,a}^{o^x})} \tag{4.5}$$

The posterior probability over an action's outcomes can then be calculated using a combination of the empirical observation and the prior:

$$T(s, a, o^i) = \alpha(\lambda^t \phi_a^{o^i} + \sigma_{s,a}^{o^i}) \tag{4.6}$$

Where α is a normalisation factor represented by:

$$\alpha = \frac{1}{\sum_{x=1}^{n} T(s, a, o^x)} \tag{4.7}$$

Equation 4.6 estimates the probability of reaching a specific outcome from an action based on the discounted prior estimate and the current experience. The experience of reaching a state from an action is derived from the frequency of reaching that state over the total number of times the action has been executed. The prior probability $\phi_a^{o^i}$ acts as a stabilising influence over the estimates, until such a time as the robot has gathered enough experience $\sigma_{s,a}^{o^i}$ to make reliable predictions over a specific action on its own.

A discount factor, $\lambda \in [0, 1]$, is applied to the prior so that that over time as the system gains a greater number of observations t, where $t_a = \sum_{x=1}^{n} c(z_{s,a}^{o^x})$, its effect on the estimates is diminished and the system can smoothly transition from prior knowledge to learned knowledge. Higher values of λ preserve the prior's influence over a greater range of executions, while lower values quickly put the onus onto the newly acquired experience.

4.4.3 Forgetting Failures

The goal of knowledge acquisition in this work is improve the executive's understanding on how the robot can interact with its environment, and in turn improve its ability

to reason about future executions. Learning to accurately predict an action's outcome allows certain actions which are predisposed to failure to be avoided, while more successful actions can be chosen in their place. However as mentioned above in order to continue to make more informed decisions, it is important the model does not become out of date. If an action fails several times the system will learn to avoid it, but that can also mean it stops gathering new data for it. If these failures were due to a temporary environmental factor or an external interference it should not remove the possibility of this action from all future executions.

In reinforcement learning literature this is often addressed through an attempted balance of exploration over exploitation [52]. This tries to ensure that rather than always opting for the best choice of action sometimes the system will make a sub optimal choice in order to gather more information. The executive controller however is designed primarily to support intelligent plan execution, and therefore seeks to minimise deviations from the actions within a plan, rather than encourage them. Instead a decay function, $\delta \in [0, 1]$, is in place which aims to regress the estimates of actions which have not been selected over a period of time back towards their initial prior estimates ϕ_a^o derived from the training data:

$$T(s, a, o^i) = \alpha(\phi_a^{o^i} + \delta^t T(s, a, o^i)) \qquad \begin{array}{r} \text{if } T(s, a, o^i) \leq \tau, \\ \text{and } \Omega_a \geq \psi, \\ \text{and } T(s, a, o^i) < \phi_a^{o^i} \end{array} \tag{4.8}$$

Where α is again a normalisation factor, as show in Equation 4.7, but has been recalculated to reflect the updated values of Equation 4.8.

This provides the opposite effect of the learning formula, and reduces the influence of the system's experience and moves the estimate back towards the initial optimistic priors. To qualify for decay an action has to have both a probability of success below a threshold value τ, and the time since it has last been chosen for execution Ω should be above a set value ψ. Each action has associated with it a timestamp of its last known execution, and if upon initialisation of the MDP the executive identifies any actions that meet the set criteria, it marks them for decay. This allows the executive to store learned knowledge of an action until a sufficient time has passed that the environment may have changed.

Once an action has been marked for decay it is an iterative process that occurs gradually

over future executions. The decay rate δ controls how quickly this transition occurs, and is designed to allow the system sufficient opportunity to select the action before it has regressed fully. As an action's estimates begin to improve, it is less likely to be avoided by the executive and can be incorporated back into the policy. Once an action is executed again, it stops decaying and starts integrating new experience into its current value, allowing actions to improve naturally or quickly be discarded again. If an action is not executed it will continue to decay until it reaches a value within range of the original prior.

4.4.4 Classifying Failure

When a failure occurs it is important that the executive can recognise it, determine the cause, and select an appropriate response. The introspective models can identify when an action is not performing within the boundaries of its nominal behaviour, however they do not provide an explanation as to why. Fault identification in complex systems such as robotics can be a difficult problem due to the multitude of issues that can be caused by both the hardware and the software. Identifying these faults from noisy sensor data can be a computationally expensive process, and is a separate research area that is outside the scope of this work.

Instead some simplifying assumptions have been made in order to allow the robot to rapidly identify and respond to failures. First the executive focuses on behavioural faults within the execution, rather than hardware faults. This concentrates on problems such as becoming delocalised, being obstructed, and dropping objects rather than a deflated left tire or stuck wheel shaft[1]. The introspective models themselves can detect any deviation that affects an execution regardless of the source, however within this work the classifier will only be trained on behavioural faults. Secondly, it is assumed that the failures associated with each action have been identified and represented in the control model. This reduces the possible failures to a relatively small set of states for each action.

This second assumption is quite limiting, but necessary within the confines of the control model. The MDP needs to have each state defined during initialisation, and an action associated with each for the policy. These failure states are identified during

[1] Although it is possible for some of these to be addressed through the use of the executive's learning and redundant systems, for example if the robot is equipped with a front and back facing camera, if one fails it can learn to use the other.

the training stage for the action models. As each model is learned from real data, if during the collection stage an action encounters a failure it is separated from the rest of the runs and manually labelled with the time and type of failure that occurred. This is typically used in the verification of the model, however the labels can provide the basis for the failure states and the corresponding execution data can be used to generate control rules to identify these states.

The data collected from these executions is a set of features derived from the robot's sensors that describe the progression of an action through an environment, as outlined in the previous chapter. The features relate to aspects of the execution such as distance traveled, angular momentum, velocity etc. and can be used to characterise a given failure. These characteristics can be learned by a classification algorithm, and used to identify future failures.

In this work classification is achieved through the use of decision trees [78]. Decision trees are a non-parametric supervised learning technique which can derive a set of rules from a collection of labelled input data. The trees are learned through analysing the training set and determining the features which most effectively splits the data. To determine the most effective split the information gain ratio is used. This estimates how much influence a single feature has on determining the class of an execution's feature set. The algorithm places the feature with the highest information gain at the top of the tree, and the different values of this feature which split the data are used to create branches to subsets of the data. If a subset represents a labelled class it becomes a leaf node, otherwise the algorithm recursively selects the next most effective feature to split the subset. This continues until each branch ends in a leaf representing a label.

A tree is learned for each of the possible action classes within the domain. Once these trees are learned a set of rules can be extracted which can be used to classify future data. When a problem arises during an execution the executive can use the current features to feed into the decision tree and identify the most likely cause of failure. As there are a relatively small predefined number of failure states per action this method is satisfactory for this work as long as it can differentiate between them. Once the failure is identified the controller's state is updated and passed to the executive to determine the appropriate response.

4.4.5 Execution Verification

One of the issues raised from the last chapter was the lack of accountability over resources with regards to the recovery actions. These models were designed to intervene and recover executions, however without concern for resource consumption it becomes possible to save an action while jeopardising the plan. The control model presented so far features similar limitations, the abstraction of resources from this model provides a more manageable representation with which to reason, but allows the creation of polices that are infeasible within the confines of the robot's capabilities. Being able to deviate from the plan allows for more dynamic execution, however this needs to be balanced against the real world constraints.

The policy itself is designed to be a more robust execution strategy than the corresponding plan when executed in a real environment. The rewards are set to encourage the policy to follow the plan unless the risk of failure becomes too great, at which point alternative actions are searched for. Even when the policy does deviate from the plan, the rewards are structured to encourage the derived policy to realign with the later plan steps as it attempts to achieve its goal. As the plan is created with resource constraints this helps to somewhat limit the policy's resource consumption.

In practice however this is only a guide, and changing one action to avoid failure can require the addition of several more into the execution in order to converge back with the plan. These additional actions will not have their resources accounted for, which could cause the goal to become unreachable. In order to assist with this a policy validator was created, which examines a policy whenever it has been created or altered to determine if it is feasible. This validator uses Monte Carlo simulation to verify the policy and estimate if it will complete within a set of resource boundaries. Monte Carlo simulation estimates a path through the policy by choosing the associated action for the current state and randomly sampling the probability distribution over its outcomes to determine the next state. This repeats for every state encountered until the simulation reaches the goal or hits a predefined threshold for the number actions. The pseudo code for the resource simulation can be shown in Figure 4.5.

While the executive does not reason with resources, it is aware of them. The planner's domain knowledge passed in to create the MDP contains both the system resources and associated cost of each action. The policy verifier can use these to maintain a belief over the current resource level of the system and estimate the cost of executing the

```
function resourceSim(MDP, policy, startState, goal, numSimulations)
    resourceTotal ← 0
    simActionCost ← zeros(size(numSimulations, MDP.actions))
    for i ← 0, ..., numSimulations do
        state ← startState
        actionCost ← 0
        runCost ← 0
        while state! = goal do
            act ← selectActon(policy, state)
            if isFailedState(state) == FALSE then
                coreAction ← act
            end if
            successorStates ← applyAction(MDP.trans, state, act)
            succState ← selectRandomSuccessor(successorState)
            actionCost ← actionCost + findCost(act)
            if isFailedState(succState) == FALSE then
                simActionCost[i, coreAction] ← actionCost
                actionCost ← 0
            end if
            runCost = runCost + findCost(act)
            state ← succState
        end while
        resourceTotal ← resourceTotal + runCost
    end for
    meanResourceTotal = resourceTotal/numSimulations
    meanActionCosts = rowMean(simActionCost, numSimulations)
    return meanResourceTotal, meanActionCosts
end function
```

Figure 4.5: Pseudo code for simulating a policy and estimating its resource consumption. This simulates the cost of both the policy and the cost of each 'plan' action in the policy.

current policy. As each action is chosen in the simulation, the corresponding cost can be accumulated into a total. Once the goal has been reached the cost of the simulated policy can be compared against the current resource level to determine if it is viable or not. These simulations can be run thousands of times, and an average cost of the policy reaching its goal can be calculated. If this average cost is below the available resources, then the policy can be permitted for execution.

If a policy is estimated to cost more than the available resources, it is prevented from executing and the executive can attempt to revise it. The verifier can enact a temporary cost function $C(s, a, s')$ to dissuade certain actions from being selected, and a new

policy can be generated that incorporates this. The new value function for each state can then be represented as:

$$V^*(s) = max_{a \in A} \left[R(s) + \gamma \sum_{s' \in S} \tau(s, a, s') V(s') - C(s, a, s') \right] \tag{4.9}$$

The cost function is initially determined by analysing the simulation data to discover any actions from the current state to the goal which consume higher resources than expected. Aside from monitoring the total resource consumption during the simulation, the policy verifier can derive the average cost of each action inclusive of its failure and recoveries by establishing if the simulated outcome is a failure state or not. This can be observed in the pseudo code presented in Figure 4.5, where a simulated resource consumption is only associated with an action that starts and ends in a normal execution state. Any interim actions executed as a result of ending up in a failure or recovery state are incorporated into the initial action's resource estimation to provide a result comparable to the initial plan allocations. The action's simulated resource consumption can then be compared to the domain supplied estimate of resource consumption, and actions which are above a specified threshold can be assigned a cost. Pseudo code for the cost allocation algorithm can be viewed in Figure 4.6. This allows a cost to be applied to both high resource actions, or all actions chosen in a resource hungry policy.

The verifier can go through several iterations of discounting actions and re-evaluating the policy in an attempt to find a suitable solution that completes within the resource restrictions. If no executable policy is found however it can pass the problem up to the planner and instigate a replan, allowing the deliberator to reason over the full domain with resources from the current state. The similar representations between the MDP and the planner allows the executive to augment the current state with resources and pass it up to the deliberator as a new initial state.

Separating resources from the model and using a simulation provides benefits to this system. It enables the control model to be significantly abstracted, which allows potentially cumbersome state spaces to be more tractably solved online. Additionally it allows the solutions to be verified over multiple resource types. If for example a robot has to execute with limited battery power and time, while operating within range of a communications beacon, the simulation can check for violations in any of these constraints and prompt the system to seek a better solution. Incorporating these into the control model itself would result in a significantly larger state space which can be

function applyCosts(MDP, $simActRes, actPenalty, excessPentalty, th$)
  $resourceRatios \leftarrow$ `calculateRatio`$(simActRes, MDP.actResources)$
  $selectedActions \leftarrow$ `find`$(resourceRatios > 0)$
  for $i \leftarrow 0, \ldots,$ `size`$(selectedActions)$ **do**
    $act = selectedAction[i]$
    $oState \leftarrow$ `getOriginatingState`$(MDP.trans, act)$
    $dState \leftarrow$ `getDestinationState`$(MDP.trans, act)$
    if $resourceRatios[act] > th$ **then**
      $MDP.cost[oState, act, dState] \leftarrow MDP.cost[oState, act, dState]$
        $+excessPentalty$
    else
      $MDP.cost[oState, act, dState] \leftarrow MDP.cost[oState, act, dState]$
        $+actPenalty$
    end if
  end for
  return MDP
end function

Figure 4.6: Pseudo code for applying an additional cost function to specific actions. This algorithm derives the ratio of each actions simulated resource consumption to the domain allocated consumption, and applies a cost to actions which consume excessive resources.

slow to solve. An alternative solution is to incorporate the resources into the reward function of the MDP, providing inherent resource reasoning within the policy itself, however this reward function can become difficult to define when trying to balance multiple resource types along side existing goals.

4.4.6 Opportunistic Events

The use of an online solver which can continually update its policy allows for external opportunities to be considered by the executive should they arise. An example of an opportunity could be during the robot's execution a user sends a request for an item to be delivered to them. The system should be able to reason about this and determine if this request is feasible within the context of its current goals, or is something that will need to be achieved at a later time. The request can be formalised and sent to the robot as a new goal state, which can be passed to the executive and incorporated into the reward function. Upon receiving these new goals the system can instigate a policy update, and ensure any changes are viable within the current resource constraints. In order to minimise the disruption to the execution, rewards can be set based on the new

goal's importance. This allows smaller tasks to be achieved only if they fit around the current execution, while urgent goals can be sent that may require an entirely new policy.

In this system most of these opportunities arise from the robot itself. As mentioned briefly in the previous chapter the work in this thesis is set within the MADbot (Motivated and goal directed robot) [15] architecture. The MADbot architecture was developed to investigate how motivations could be used to automatically generate new goals for a system, and how these goals could be incorporated into a plan. The motivations represent concepts such as conserve energy or acquire data, and change over time as the robot depletes its battery or operates without taking photographs. Once these motivations hit certain thresholds they generate a goal state, which can be passed to the executive.

4.5 Executive Overview

Having defined the major components, the system architecture outlined in Section 4.2 can now be revisited in more detail. Figure 4.7 steps inside the executive and highlights the interactions between each of these components, and presents a flow through the system. This section will discuss these interactions and how each part works together to provide a more intelligent executive.

Upon initialisation the deliberative component creates a plan to reach the system's goals and passes this to the executive along with a domain file and a MDP specific initial state. The initial state and the domain file are used to generate the state space of the MDP. The system checks whether the derived MDP is consistent with a previously learned control model, using the seed state and number of states and actions within the newly generated control model to verify it. If the verification is successful the relevant transition function is loaded from the domain knowledge, otherwise a new function is initialised.

With the control model fully initialised the executive controller converts the plan into a reward function. This reward function is then combined with the current domain knowledge to generate a policy of appropriate actions that will transition the system toward its goal through a dynamic environment. If it is the first time the robot has operated within an environment this policy will reflect the plan's actions, however if

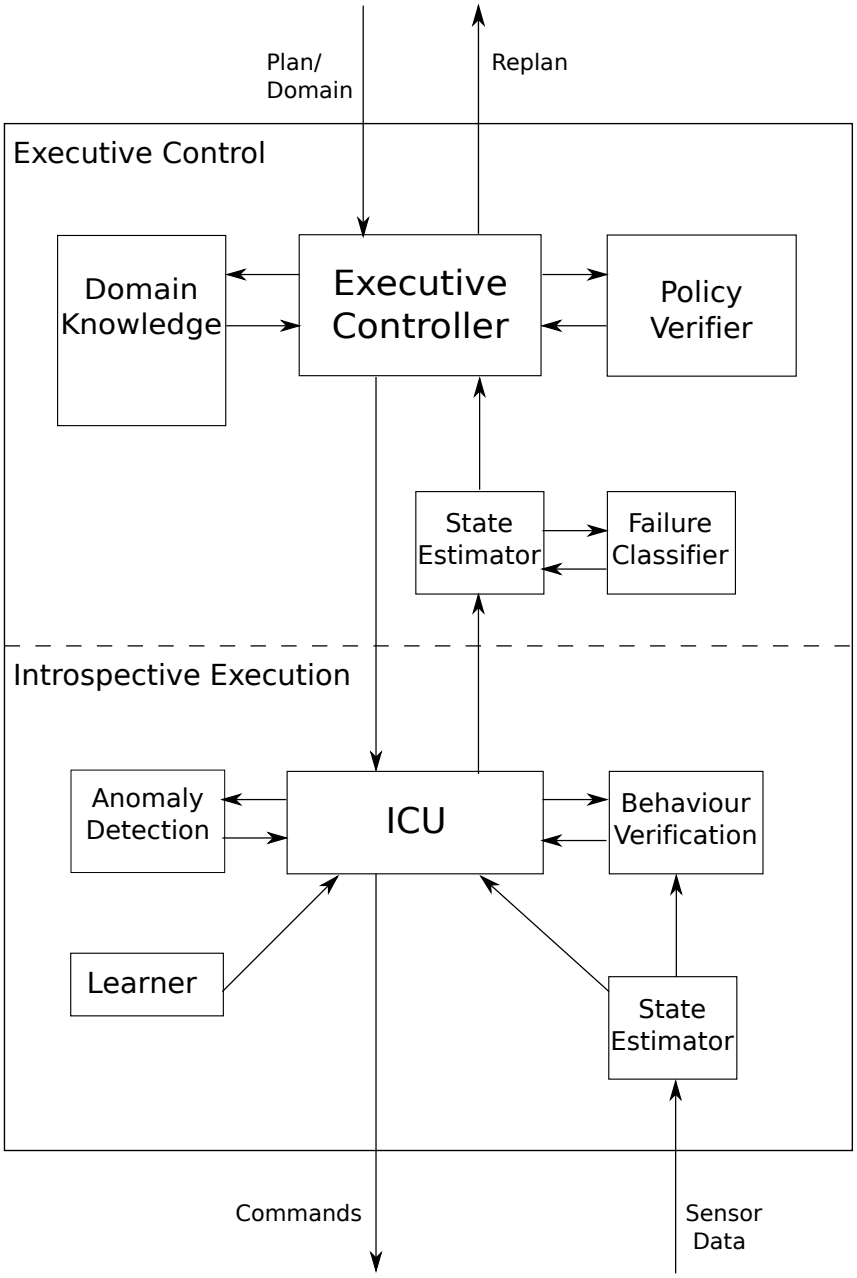


Figure 4.7: A look inside the executive representing the major components and their interactions with each other. The upper half is the control layer, supporting plan execution through additional reasoning based on knowledge from the environment. The lower half represents the introspective layer, dedicated to monitoring complex actions and their interaction with the environment.

the system already has experience in the domain it may make alternative choices that better support the execution. Once the policy has been created it is passed over to the execution verification component to ensure that it is viable under the current system conditions. The execution verification uses the systems resource levels, passed in with the initial state, and runs simulations of the policy to estimate its resource consumption. If the policy fails the executive controller is updated and searches for a new solution, otherwise it is cleared for execution.

Once the policy is verified the first action is selected and dispatched to the introspective execution control. The introspective control unit loads up the corresponding introspective model for the action, and initialises the execution. As the robot executes the action the data from the sensors and actuators flow into the introspective model's state estimator and it begins to monitor the execution of the action. This operates as described in Section 3.6, ensuring the behaviour of the action falls within the nominal range until it either completes or deviates enough to be classed as a failure. If an action completes successfully the introspective control unit informs the executive's state estimator, which in turn updates the state of the system and passes this to the controller. The controller updates the domain knowledge, then dispatches a new action based on the current state.

If an action's behaviour is found to be deviating from expectations, the flow of control changes slightly from the previous system. Before when an action failed the introspective control unit automatically transitioned into a recovery model in an attempt to save the execution. Now upon acknowledgement of a failure, the execution is still paused and the progression of the current action backed up, however the introspective control unit updates the executive's state estimator with a report of a problem. The state estimator activates the failure classifier to detect the most likely cause of the issue, which makes a diagnosis and passes this back to the estimator. This updates the controller with the new state, which selects the appropriate recovery response and dispatches it back to the introspective layer. Upon completion of the recovery action the executive's state estimator is notified again, and the controller can choose whether to resume the previous action or not. If it decides to continue it dispatches a resume command and the introspective execution operates as before, loading up the previous actions's progression, pruning the bad states from the trajectory, and resuming the action from a better position to complete.

Through the presentation of this system it may seem that there are now several steps

involved between the detection of failure and the recovery. However these additional steps afford the system a greater opportunity for reasoning, providing explicit states within the execution from which the executive can intelligently respond to problems. Further more much of this reasoning is captured within an efficient policy mapping, which when coupled with the close integration of the executive controller and the models, allows for a seamless response to failure without any breaks within the execution.

If a failure does occur, aside from dispatching a recovery action, the executive controller takes the opportunity to re-evaluate the execution as a whole. When an action is found to be failing the controller updates its domain knowledge as it would as if it completed any other way. However once the recovery action is dispatched, it uses this updated domain knowledge to ensure the current policy is still achieving the goals of the system. If an action is consistently failing, or has failed repeatedly in the past, the updated transition function for that action may make it no longer a desirable path on the route to the goal. If this is the case, and other actions are available that can achieve the same results, the policy can be updated to take advantage of this. Any change to the policy results in the execution verification being rerun, so if the actions change they are checked to ensure the policy is still feasible. Once the recovery action has completed executing, the new policy actions can be dispatched, and the system can continue moving towards its goal.

4.6 Discussion

This chapter presents a system for increasing the robustness of executing high level symbolic task plans in dynamic real world environments. The creation of a more intelligent executive through the introduction of a stochastic controller allows more appropriate decisions to be made about the execution at both the level of individual actions, and throughout the plan as a whole. With regards to actions the controller extends the approach described in the previous chapter, introducing probabilistic outcomes and intermediate states that allow further reasoning during the execution of an action. These states allow the executive to classify the type of failure an action has encountered, and select an appropriate response with which to intervene and recover the current execution. This improves the robustness of actions and further improves their probability of success in dynamic worlds.

The executive can also learn from and adapt its execution based on these failures. If

an action is constantly failing, either due to an environmental factor or just the robot's own limitations, it can recognise this and seek new actions to avoid the problem and continue towards the goal. This increases the executive's introspective capabilities from an action level to a plan level, allowing it to observe each action as it executes, and use the outcomes from those actions to determine if the execution as a whole is still serving the goals of the system. Furthermore knowledge learned from actual executions can be stored and used to influence future executions.

The desire for a better method of executing plans in dynamic environments is common throughout robotics. In Ghallabs's position paper [38], discussed in Chapter 2, he describes the need for better actors: systems of hierarchal deliberative layers which work together, continually planning, monitoring and adapting their execution to solve complex real world problems. While the proposed scope and solutions vary significantly the ideas behind this paper align with the core ideas presented in this thesis. With the addition of the executive controller a system for plan execution has been created that uses increased deliberation across the entire platform. Each component operates at a different level of abstraction, and can communicate to effectively achieve a common goal. Finally the learning component allows the integration of online reasoning, such that the execution strategy can be updated in real time based on feedback from the environment.

A limitation when going forward with this work can stem from the use of a MDP as the basis for the controller. Based on the size of the problems and complexity of the domains, using a MDP was an appropriate choice to introduce stochastic actions and a learning component to this system. However if the problems become more complex, and the size of the domains grow then the state space is likely to explode, which can make the resulting MDPs unwieldily to solve and store. Moving towards trial based solutions for solving these models, such as methods based on RTDP [4] or UCT [53], can continue to allow online creation of polices without needing to evaluate the complete state space.

Another solution is to change the representation of the model from a standard MDP into a factored version. A factored MDP represents states as a collection of variables, and an actions outcome updates these variables. It provides a compact representation for the problem, and prevents the need to enumerate the state space. The propositional nature of the model used in this work is a sensible fit for factoring, making this a logical extension if the system is applied to larger problems.

CHAPTER 5

EVALUATION

The previous chapters discussed methods for increasing the intelligence of a system's executive through the addition of introspective techniques. This chapter presents an evaluation of these techniques by implementing them onboard a mobile robot and conducting a series of experiments in dynamic environments. A physical robot is used to demonstrate the system's practicality in real world applications, and highlight its ability to improve execution as well as demonstrate any limitations. These are presented as a series of test cases, followed by a selection of simulated results to provide a more quantitive evaluation.

The chapter begins with an explanation of the experimental setup used for the rest of the chapter, and provides an overview of some of the additional components of the system. Section 5.2 examines the anomaly detection algorithms, and compares the performance of each on a series of failures. Section 5.3 presents the first set of experiments, which focus on testing the introspective execution recovery to discover how the system can handle failure at the level of action execution. Section 5.4 then begins to evaluate how creating a more intelligent executive as a whole can learn and adapt to failure. This is followed by 5.5, which uses a simulation to evaluate a wider selection of executions. The chapter is then finished off by a discussion of the results.

Figure 5.1: The Pioneer 3-DX robot used in the evaluation of this work.

5.1 Experimental Setup

The experiments take place onboard a Pioneer 3-DX robot, as seen in Figure 5.1, equipped with a laser ranger finder and sonar positioning system. This robot is placed within an active office environment within the university, and tasked with navigating between the various offices and taking pictures. This provides a series of corridors, open areas and doors for the robot to operate within, and due to a mix of both academic offices and student laboratories there is a constant source of people moving through and affecting the area. The robot is equipped with a pre-defined map of this environment, detailing the static structures it can encounter and a series of waypoints it can navigate between.

The work presented so far is integrated into the MADbot architecture [15]. MADbot

represents a plan execution architecture which contains a series of motivations which relate to the current domain. These motivations represent desires the system can have such as the need to recharge, relocalise or transmit data. The value of these motivations can change as the plan progresses, which in turn can trigger the generation of new goals for the system to achieve. For the purpose of this work MADbot primarily provides a pre-existing platform for connecting a high level planner to low level control, however its motivation and goal generation features can work within the presented system.

MADbot implements a deliberative layer, which is supported by the LPG [35] planning system. LPG is a local search based planner and is particularly suitable for this work as problems are expressed using propositional and numerical constraints. As the executive itself cannot reason directly about resource consumption, a planner that can incorporate numerics is essential. The input to the planner is a PDDL 2.1 [32] domain file and problem file, representing the planner's abilities and knowledge of the world. The output is a deterministic sequence of actions which can achieve specific goals. At the opposite end the low level software utilised by this architecture is provided by ActivMedia's ARIA and ARNL control libraries, which produce the basic behaviours that compose the actions such as path planning, obstacle avoidance and localisation.

The limitations of this robotic platform restricts the experiments in this chapter to focus primarily on navigation tasks, however it is important to emphasise again that all of the work described so far generalises to other types of compound task.

5.2 Anomaly Detection Routines

In order for an executive to be able to overcome the failures it encounters it has to be able to reliably detect them. Chapter 3 introduced three different anomaly detection algorithms for this purpose, Cumulative Log Probability Difference (CLPD), Temporal State Counting (TSC), and Gradient Log Probability Difference (GLPD). Both the CLPD and TSC algorithms have been presented previously in [39] while the GLPD variant has been developed specifically for this work. As both the platform the original algorithms were deployed on has been changed, and the models they relied on have been retrained for different tasks, all three algorithms will be re-evaluated to determine their ability to detect errors.

For this evaluation the navigate corridor model has been chosen. This represents a

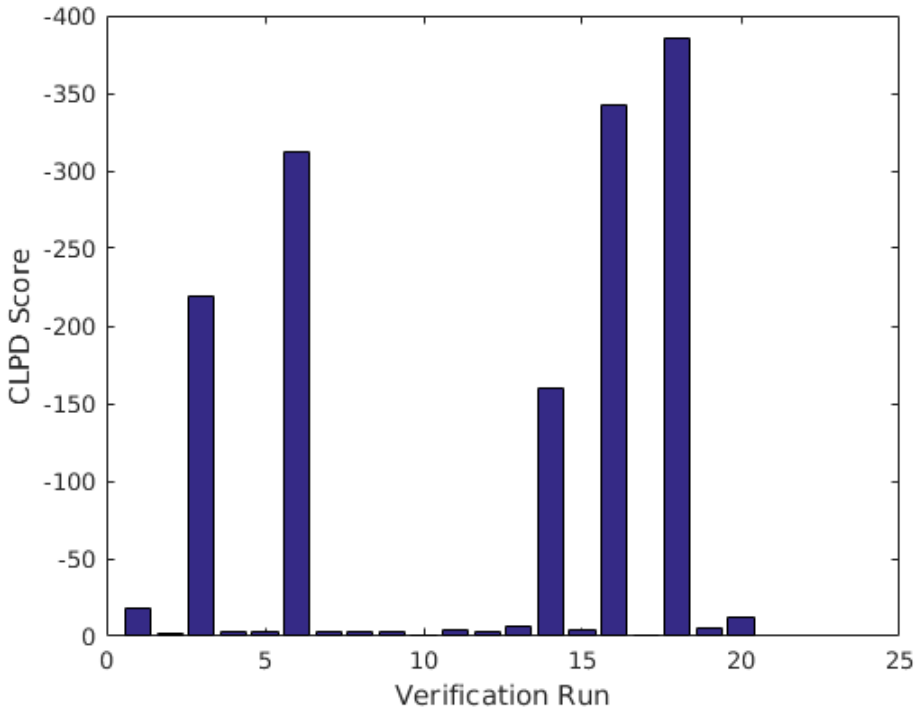


Figure 5.2: Chart showing the CLPD scores from each of the verification runs.

compound action with multiple interacting sub-behaviours that maintains a relatively straightforward progression through the task. A training set of 60 corridor executions has been used to learn this model, while 20 additional runs were taken and held separately as a verification set. Finally 10 runs were captured with specific errors induced into them. For the corridor navigation two types of errors were introduced, a blockage which prevented the robot from progressing with its navigation, or a group of people surrounding the robot which slowed down its progression and caused it to become delocalised.

To compare these algorithms thresholds need to be established that can define when a task has veered too far from the normal execution that it can be labelled anomalous. This can be achieved by using the verification data, which is a set of standard task executions not used within the training of the model. As these executions are deemed normal, the trained model should be able to explain their behaviour, and if passed through the error detection algorithms any accumulated error score can be assumed to be within the boundaries of acceptability. This method for determining the threshold for each of these algorithms is based on what was initially presented by Gough in [39]. As an update to this, rather than selecting the thresholds manually from a visual

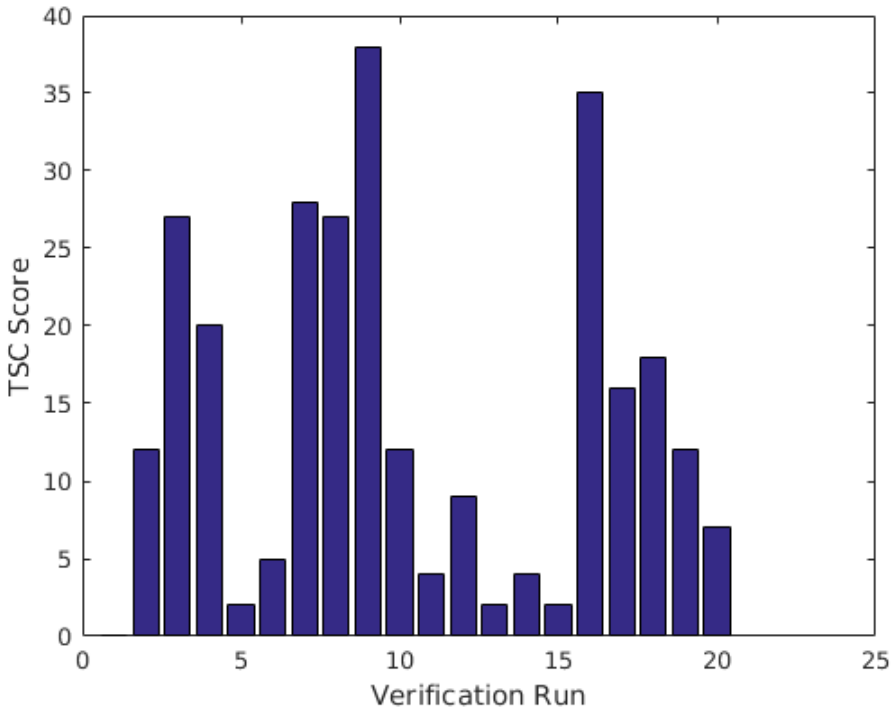


Figure 5.3: Chart showing the TSC scores of the verification runs.

inspection of the verification data, for this work the maximum error score plus an additional 5% margin is used in order to create more comparable results between the methods.

For each of these algorithms an error score close to zero implies that the execution is well explained by the model, while the further away from zero a score gets is indicative that there is discrepancies within the execution. The CLPD scores for each of the verification executions can be seen in Figure 5.2. The maximum CLPD score from the verification data is 381, and after adding an additional 5% error margin the threshold for a normal execution can be set at 403. If a new run breaches this threshold, it can be regarded as erroneous. Figure 5.3 shows the TSC scores for the same verification data. From this taking the maximum score and adding the error margin, a threshold of 40 can be applied. Finally the GLPD scores for the verification data is shown in Figure 5.4. The maximum value here is 87, which results in a threshold of 92 being set.

With the thresholds for each of the three algorithms determined, the executions containing induced failures can be examined. Each of these executions were interrupted

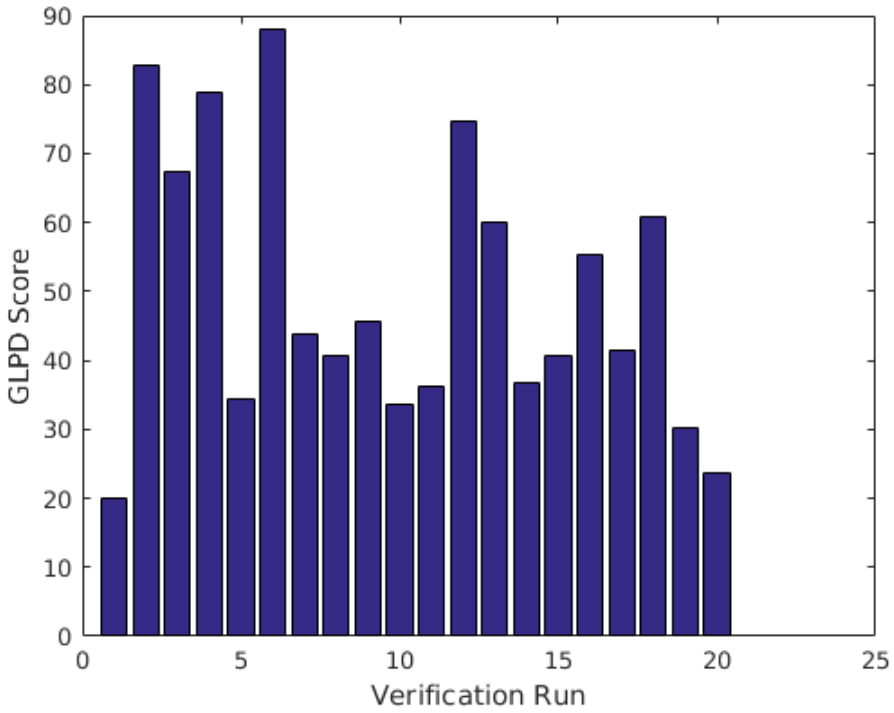


Figure 5.4: Chart showing the GLPD scores of the verification runs.

part way through the completion of the task with one of the failure types discussed above. Table 5.1 displays the results of when the failures were detected by each of the respective algorithms.

Focusing on the first 5 executions which contain the blocked error all three algorithms perform reasonably well, with the GLPD identifying the potential problems the quickest in each instance. This is due to a blocked error manifesting itself as the robot being repeatedly stuck within the same state, which levels out the Viterbi trace due to the model's high self transitioning probability. This change in trajectory is picked up by the GLPD algorithm, and the error is quickly recognised. The same state repetition also tends to be picked up quickly by the TSC algorithm, however depending on the time point and the state repeating it may take longer to reach the threshold. Finally the CLPD algorithm has a more mixed performance, and fails to identify one of the errors. This is due to the trajectory of the Viterbi trace for run 4 levelling out at an angle that caused it to stay within the probability window set by the training data. Given enough time this would eventually trigger an error, however after 10 seconds of being stuck the execution was ended.

Comparison Of Failure Detection Times				
Run	Error Type	CLPD Time (s)	TSC Time (s)	GLPD Time (s)
1	blocked	28.5	25.5	24.3
2	blocked	19.9	22.0	19.3
3	blocked	17.1	16.1	15.2
4	blocked	-	28.6	28.2
5	blocked	23.6	24.1	23.0
6	de-localised	33.7	25.7	22.9
7	de-localised	39.5	-	31.7
8	de-localised	35.4	23.4	22.6
9	de-localised	33.4	24.3	32.8
10	de-localised	31.5	35.9	-
	Mean	29.18	25.06	24.44

Table 5.1: A comparison of the different anomaly detection algorithms focusing on the times they detect the failures induced into each run.

The results for executions featuring the delocalised error was more mixed. The GLPD algorithm performs well, detecting the failure the fastest in 3 out of the 5 executions, however it does fail to detect a problem within run 10. The delocalised failure tends to cause the robot to enter a cycle of slowed progress, hesitating, and increased turning. This typically causes the Viterbi trace to either have periods of levelling out, or sharp decreases in the log probability. For run 10 there was enough room around the robot to allow slowed progression despite the hesitation, preventing the trace's trajectory from featuring a significant change. At the point the execution was ceased the GLPD was approaching the threshold value, however it failed to breach it before the execution was terminated. The TSC algorithm similarly fails to identify one of the failures, this time run 7. The cycle of states observed during the failure in this run had high state counts from the training data, causing the sequence to be masked.

Unlike the previous two, the CLPD algorithm did manage to detect all the failures of this type, however typically slower than the others. The small changes within the trajectory of some of these failures, coupled with the requirement for the trajectory to reach the edge of the training data's probability window, caused some failures to be detected over 10 seconds slower with the CLPD than with other methods. While the majority of time using the CLPD is slower, there are executions where it detects some errors faster than one or both of the other algorithms. This might indicate it would be sensible to run all three algorithms in parallel to ensure all errors are identified in

Failure Rate From 20 Normal Executions		
Algorithm	Failures Detected	Maximum Error Score
CLPD	3	-788
TSC	0	35
GLPD	0	74.5

Table 5.2: A comparison of the three anomaly detection algorithms applied to normal executions of the same task in a different environment.

shorted possible time. However continued testing of these algorithms using successful executions of tasks of the same class but in different environments highlights the CLPD algorithm having the highest incidence of false positives. Table 5.2 shows the results of failures detected in a set of 20 normal executions set within a different corridor from what was used in the model training. This new environment can cause traces that are interpreted as less probable by the model, and therefore can dip outside the probability window set by the training data. This has little affect of the GLPD and TSC algorithms, but can cause large error scores to be accumulated in the CLPD, triggering failures in normal executions.

5.3 Introspective Action Execution

The introspective action execution framework, discussed in Chapter 3, outlined a method for using introspective models to monitor an action's progress, identify possible failures, and attempt to overcome these failures from within the current execution. This provides a method to recover from various execution problems without having to abandon the current action or disrupt the overall plan. This section describes a set of test cases based on this framework, where the robot is placed within a dynamic office environment and tasked with executing simple plans. The robot then has to ensure the completion of these plans using the introspective models. The introspective execution framework on its own lacks any means of formal control, therefore the recovery action is associated directly with the plan action. The following experiments are designed with this in mind, such that the injected failures will be correctable by the known recovery action.

Model Selection Based on Execution Data

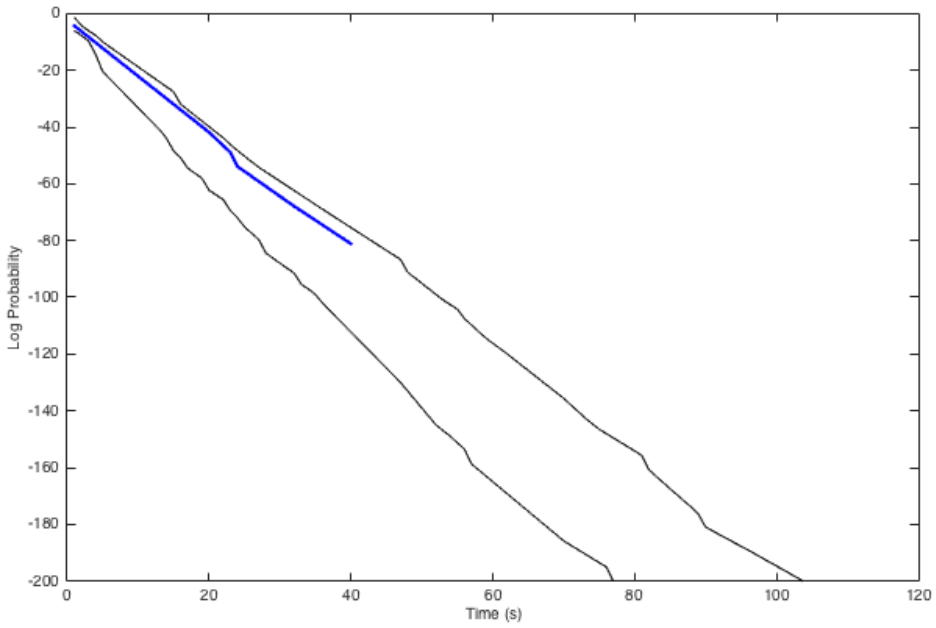


Figure 5.5: A Viterbi sequence showing the initial execution through the corridor navigation model. The black outer lines represent a probability envelope for this model, derived from the training data.

The first test case is designed to demonstrate the executive's ability to differentiate between the possible models based on execution data, and select the appropriate one for the current action as described in Chapter 3.5.4. If two environment types differ significantly then separate models may be required for the same action to preserve their predictive capability. In order to not add redundant actions into the planner, further increasing the complexity of the domain, these models are grouped together under one action type and the executive can select the appropriate model at run time.

The robot begins in a corridor and has the goal of reaching the opposite end. The planner dispatches the action `NAVIGATE(BOT1 WP6 WP8)`, which has no knowledge of the class of environment the robot is in, only that it needs to execute a navigate action. The executive begins the execution of this action, and transforms the sensor readings into observations for both the navigate corridor model, and the navigate open area model. These observations are used in conjunction with the Viterbi algorithm to predict state sequences through these models, and the associated probability of these sequences.

An action's behaviour tends to be the most clearly defined and consistent between

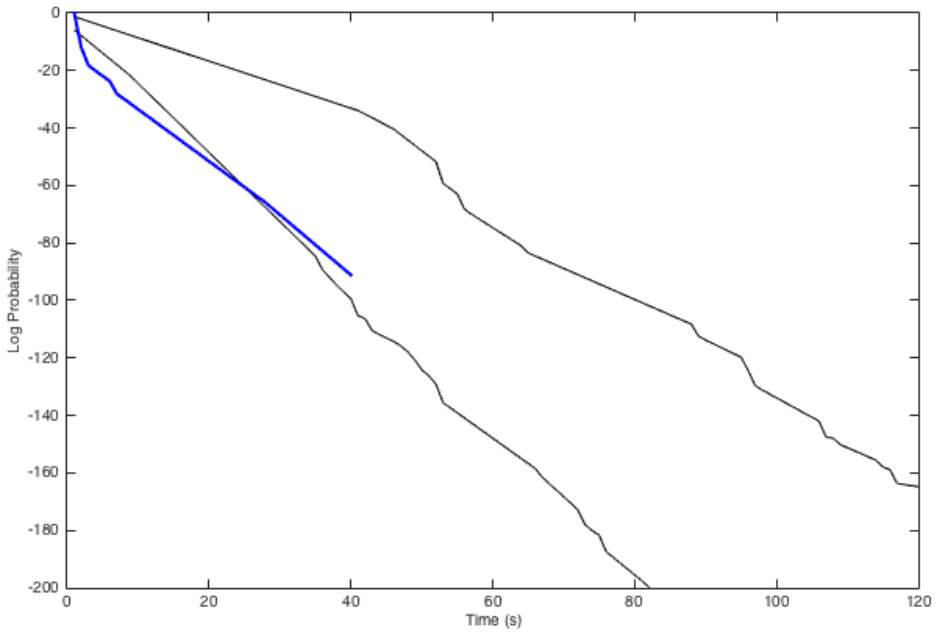


Figure 5.6: The Viterbi sequence of the initial execution through the open area navigation. The trace can be seen here stepping outside the probability envelope of the model, demonstrating a poor fit.

executions during the initial few seconds of a task. This lack of variation is reflected within the probability envelope of an action's model, providing a tighter boundary for acceptable behaviour. Small changes in an action's behaviour at this time can result in significant deviations within the estimated probability of an execution. In the navigation example this is primarily driven by the laser readings, which change considerably between the narrow corridor and a wide open area. In Figure 5.5, the first few seconds of the navigate action's execution can be seen represented in the navigate corridor model, while in Figure 5.6, the same execution is represented in a navigate open area model. The executive can then differentiate between these two models by examining the associated probability of each trace, alongside the number of error states generated by each. In this experiment after 4 seconds the corridor navigation had a log probability of -79.9 and no identified error states, while the same trace through the open navigation model had a probability of -91.1 and 17 error states. This caused the executive to dismiss the open navigation model and continue to monitor the execution using the navigate corridor model, which is the correct choice.

This test case provides an example of the executive's ability to differentiate between

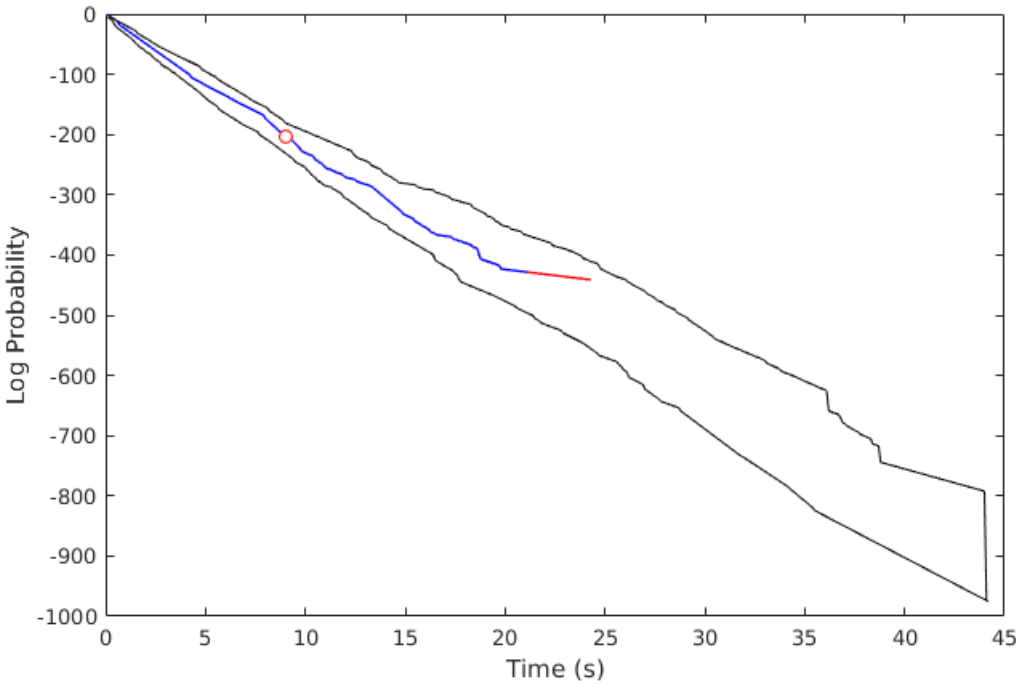


Figure 5.7: A Viterbi sequence representing a corridor navigation with an error induced that delocalises the robot. The error is induced at around 9 seconds, as represented by the red circle, and at the 20 second mark the trace can be observed to level out, indicating abnormal behaviour.

different models of the same action class. By using the generated probability and feedback from the anomaly detection algorithms the executive can select the correct model for the current environmental condition.

Corridor Navigation With Induced Failure

In this test case the robot was initialised at the start of a corridor and tasked with navigating to the other end to capture a picture. The planner again generates the plan, `NAVIGATE(BOT1 WP6 WP8)`, and dispatches it to the executive for execution. The executive loads up the corresponding navigation model, and begins executing the action and monitoring its progress.

The execution begins as expected, however around 9 seconds into the navigation a group of people enter the corridor and stop to observe the robot, a common occurrence when operating within the University. These people stopped a short distance away from the robot, and were spread out in the middle of the corridor. As the robot approached, the crowd was detected in its sensor readings as solid objects, introduc-

ing anomalies into the system's localisation routines. This causes the robot to slow down and become hesitant. In the middle of a navigation action where the behaviour is typically one of steady progression, this is seen as unusual.

To help contextualise this execution the output from the Viterbi Algorithm is shown in Figure 5.7, with an outline of the navigate corridor's training data surrounding it as a representation of the introspective model. The red circle represents where the problem was first injected, which begins to manifest itself in the execution trace around 20 seconds in where the trace can be observed to level out. This is detected by the anomaly detection routines, and at 24.3 seconds into the execution the GLPD algorithm reaches its threshold and raises an abnormal behaviour alert. This causes the executive to launch the associated recovery action, which in these experiments is scan action.

This action surveys the environment by rotating the robot 360 degrees, identifying landmarks in the environment to reinitialise the localisation routine and uncover any possible paths to the goal. Upon completion it passes control back to the navigation action, now with the robot having a firmer grasp of where it is and where it needs to go. Before resuming the navigation the executive initiates the state pruning algorithm, as described in Section 3.5.3, to remove the component of the execution corresponding to the failure and ensure the navigation resumes from a suitable internal state to proceed. This removes 23 states from the end of the execution, which are highlighted in Figure 5.7 with the red line.

Upon completion of this, the executive transitions back to the navigation model and execution and monitoring of that action resumes. This finishes executing, and the deliberative component is updated that the action has completed successfully and the goal has been achieved. The final execution trace for this task is shown in Figure 5.8, which with the failure overcome and the error states removed, appears as a normal Viterbi sequence.

An interesting comparison is if the same scenario is performed again using the planner coupled with introspective monitoring, or with just the default control software. If the planner is set up to use the models to detect anomalies, but not recover from them, it executes as above until the point of failure. Upon discovering the failure it attempts a replan, however as the planner cannot reason about the execution itself, it needs to generate a state to replan from. As there are only two waypoints in this problem, it has to choose one of them. If it chooses the destination waypoint, upon formulating the state the planner recognises it is actually at the goal and assumes the plan has completed. If

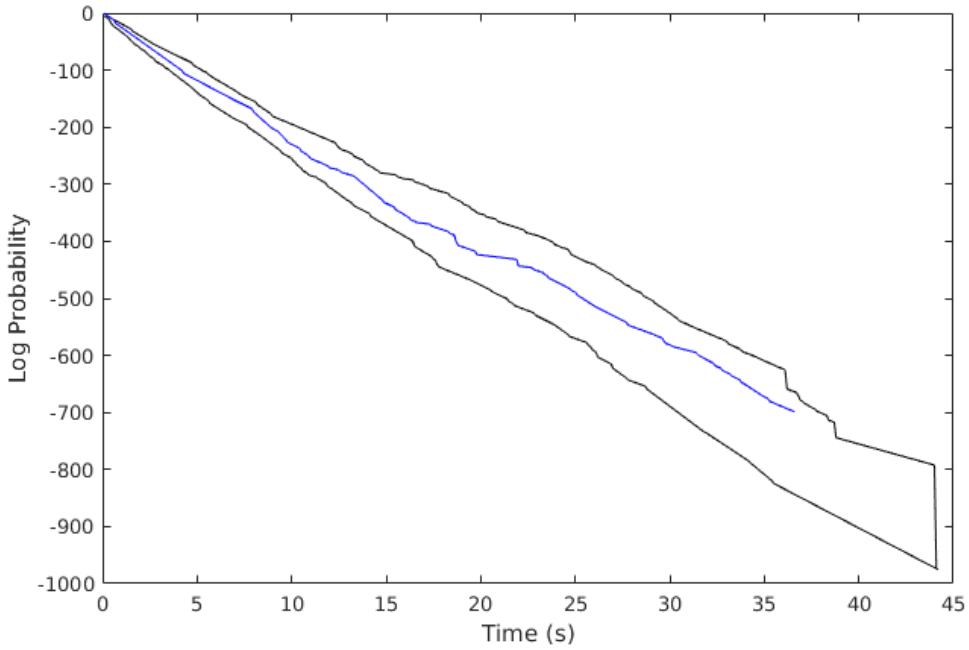


Figure 5.8: The final Viterbi sequence after the problem shown in Figure 5.7 has been over come and the failure states pruned. The levelled out section has been removed, and the trace looks like a normal successful execution.

it chooses the initial waypoint then the planner reissues the navigation action, however the supplied waypoint's co-ordinates do not match the robot's current position, further compounding the robot's localisation problem rather than fixing it. Greater coverage of waypoints could fix this problem, however it increases the complexity of the domain model, and puts the responsibility onto the engineer to determine how far apart waypoints should be for optimal navigation.

If the planner relies purely on the control software for execution, then it attempts the action without any means to determine if a problem arises. This can lead to one of two outcomes in the navigation action. If when the robot reaches the crowd it cannot find a safe path to its goal, the low level software generates a path planning error and puts the robot into a safe state. This state persists even if the crowd disperses, and the robot needs to be manually recovered. If the robot can find a path through the crowd it continues its navigation despite the localisation module no longer being certain of its exact position. This results in the robot reaching its destination at the end of the corridor, but usually over or undershooting the actual waypoint. For a simple navigation task this may not be an issue, however if the robot needed to reach a specific location for its next action, such as within reach of a table, it can impact future actions.

This test case demonstrates the use of the introspective action execution system to help increase the robustness of a robot's operation in a dynamic environment, and recover from an unusual situation without having to abandon its execution. It demonstrates how an anomaly can be detected before it becomes a critical problem, and the steps the executive can take to overcome it at the execution level.

Doorway Navigation With Controller Failure

The aim of this test case is to demonstrate how the introspective actions can overcome problems with the low level software. As has been mentioned the robot can struggle to enter through doorways, especially if the door is not fully opened or the robot approaches from an awkward angle. In this case the robot is placed in front of a doorway, and tasked with capturing an image of the corridor on the other side. The planner generates and dispatches the action `NAVIGATE_DOORWAY(BOT1 WP3 WP4)`.

The executive takes this action and initialises the doorway navigation model. The execution begins and the robot moves towards the doorway and immediately enters a hesitation and search cycle as it tries to identify a possible path through the door. The observations generated from this behaviour get mapped to an unusual sequence of states within the introspective model. The output from this execution's Viterbi sequence is shown in Figure 5.9 against the doorway model's training envelope, where from 6 seconds onwards the trace can be observed to descend at a much shallower progression. This is due to the cycle of the same few states occurring is regarded as more probable within the model.

The anomaly detection routines recognise this as abnormal behaviour and at 11 seconds the GLPD's threshold is breached. This triggers the recovery action for the doorway navigation model to be loaded. This recovery action attempts to identify the centre of the doorway, and position the robot directly in front of the largest opening. Once complete, the executive resumes the initial action and prunes the hesitating behaviour. For this execution 36 states are pruned from the state trace to remove the fault, which is shown on Figure 5.9 as the red line. This allows the navigate door action to resume and continue to be monitored until the robot reaches its goal on the other side. Figure 5.10 shows the output from the completed trace with the error states removed.

This is an interesting example as it demonstrates a problem that cannot be resolved in a satisfactory manner by either replanning or the control software. If the robot uses the planner with only the execution monitoring capabilities enabled, it recognises the

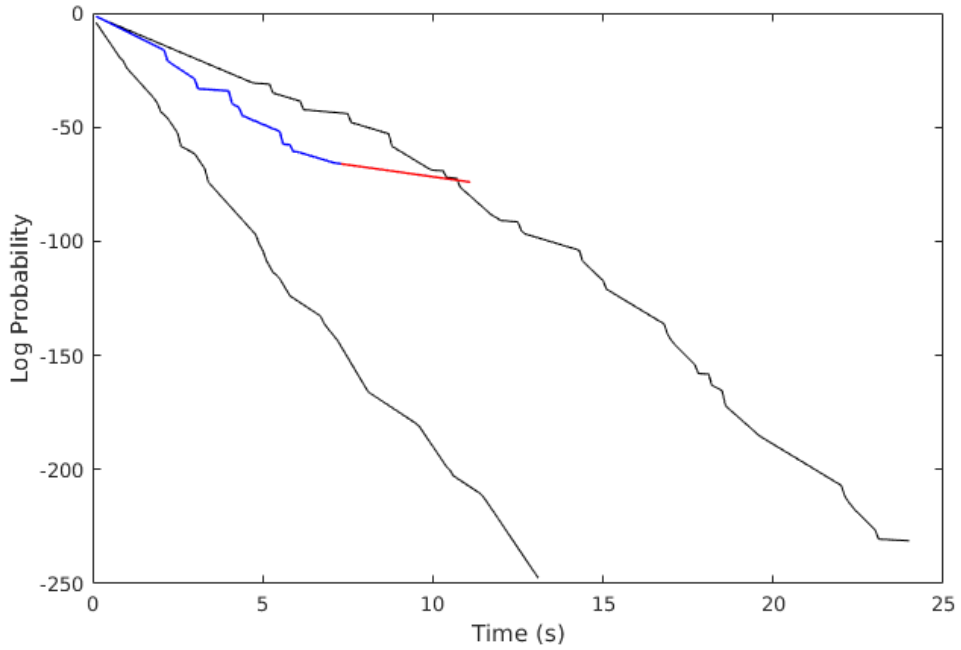


Figure 5.9: A doorway navigation trace, showing an abnormality towards the start of the execution represented by an erratic trace ending in a levelled out sequence.

problem of the robot failing to pass through the doorway. However upon replanning, it can once again only choose the previous state of the robot to replan from. In this scenario the original plan state is still representative of the robot's current state due to its lack of progress, however if the planner has no additional actions which can change the current state of the robot or the door, it will continue to issue `NAVIGATE_DOORWAY` commands which will continue to encounter the same problem and fail. If the planner uses the default control software it will issue a path planning error and halt the execution, stopping the robot from progressing at all.

In this test case the introspective execution control overcomes a low level software problem which otherwise would need to be handled by an alteration to the code. The default control software is overly cautious with path planning through narrow spaces, resulting in problems when the robot attempts to enter doorways. By using introspective models and recovery actions the executive can recognise and overcome this problem, providing a more robust execution.

Corridor Navigation With Multiple Induced Failures

This test case examines the effect of encountering multiple failures within a single

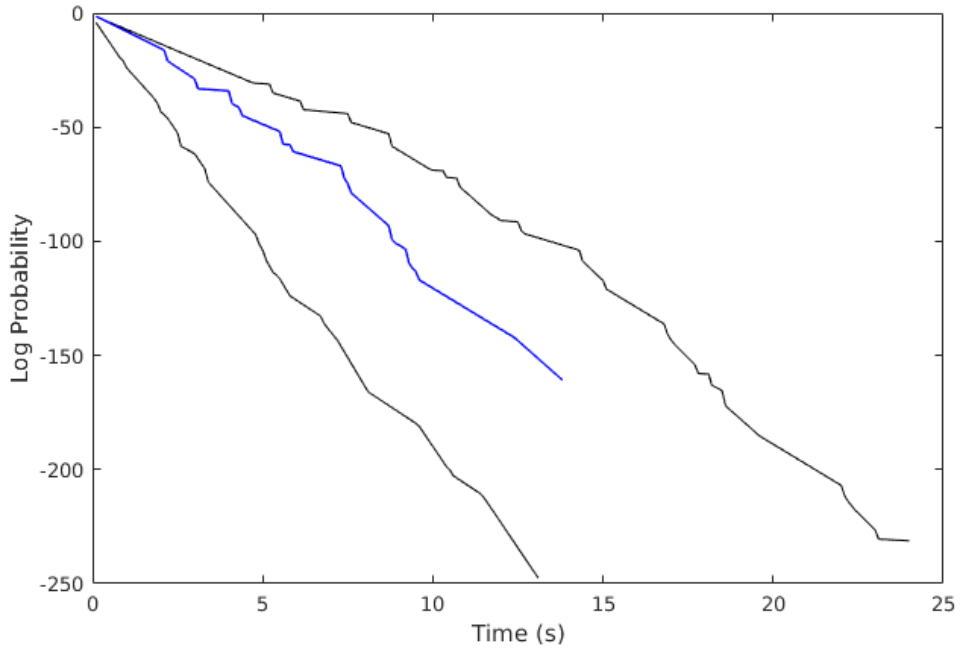


Figure 5.10: The corrected Viterbi sequence for the doorway navigation, with the failure overcome and the errors removed.

execution. Similar to the first navigation test case, the robot is situated at the start of a corridor, and given the goal of reaching the opposite end. The planner dispatches the action `NAVIGATE(BOT1 WP6 WP8)`, and the robot begins execution. At roughly 6 seconds into the execution, a group of people enter the corridor and group slightly before the halfway point, forming what appears to be a set of obstructions to the robot's sensors. This causes the robot's localisation score to drop, and the hesitant progressing behaviour to emerge.

This causes a significant change to the execution's trajectory, which can be visualised using the output from the Viterbi trace as shown in Figure 5.11. The GLPD detects this shift in the execution at 13.4 seconds into the run, and raises an anomaly. This causes the scan recovery action to execute to overcome the issue, and upon its completion the state pruning algorithm to be applied. This removes 40 states in this trajectory, as highlighted in red in Figure 5.11.

The result of a successful recovery action is that the system continues to execute the original action as if the previous problems had not occurred. The robot resumes navigating towards its goal as before; the erroneous state sequence is pruned leaving the action looking normal from the model's monitoring perspective, and as the failure was

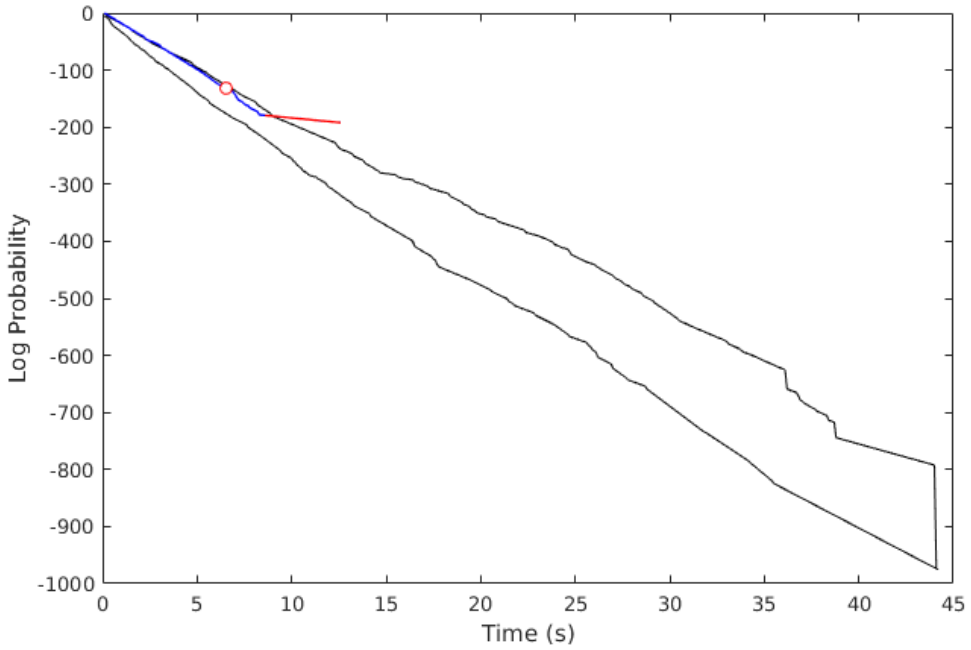


Figure 5.11: A Viterbi sequence for a corridor navigation with an error induced. The trace flattens out at the end, showing anomalous behaviour.

handled by the executive the higher level deliberative systems were never alerted to the occurrence of a problem at all. As such when the crowd of people, rather than disperse or move behind the robot to watch it pass, move further down the corridor towards the goal the robot encounters them as a problem as if for the first time. As the robot approaches it once again registers the people as solid objects which cannot be located on its internal map, and localisation inconsistencies are generated.

In this instance the people are spread out in the corridor, allowing the robot more space to move, but still obstructing it and interfering with its internal localisation. This causes a change to the trajectory that is more subtle, as can be seen in the Viterbi output in Figure 5.12 from the 20 second mark. In the state trajectory itself this is represented by the repetition of the same states. This is caught by the TSC anomaly detection algorithm, which detects the high occurrence of a particular state for the current time period, and raises an exception 25.1 seconds into the execution. This is passed to the executive and the corresponding recovery action is selected. Upon completion 37 states are pruned from the trace, and the execution resumed. The action finally competes, and the resulting traces can be viewed in 5.13.

The test case demonstrates an example of the system's ability to handle multiple fail-

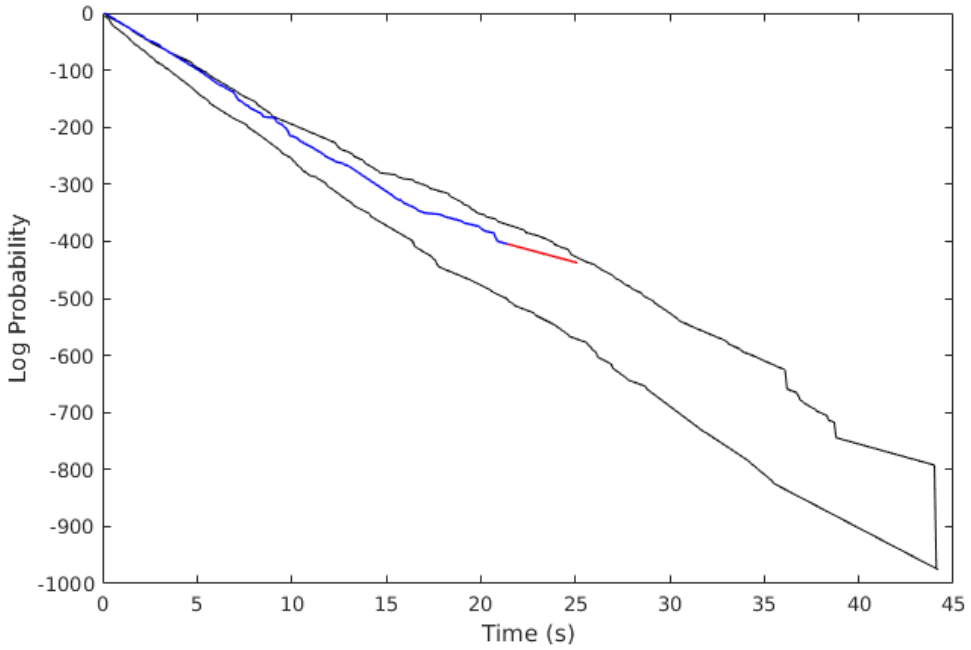


Figure 5.12: The same navigation task, with the first failure recovered from. The execution looks normal, and continues on until it once again starts to encounter anomalous behaviour, and the executive detects another potential failure.

ures within a single action, without having a detrimental effect on the success of the overall execution. As each failure is handled individually, and the errors are removed upon recovery, from the executive's perspective when it resumes an action it is executing as if nothing had gone wrong. This allows subsequent failures to be recovered in the same manner as the original problem, without any propagating effects. It does however highlight the lack of communication with the planner, and the issue this could pose in real tasks. In this particular case each scan recovery action takes roughly 18 to 20 seconds to complete, which if executed multiple times in a single action, could lead to execution problems in later stages of a plan due to violating resource of time constraints.

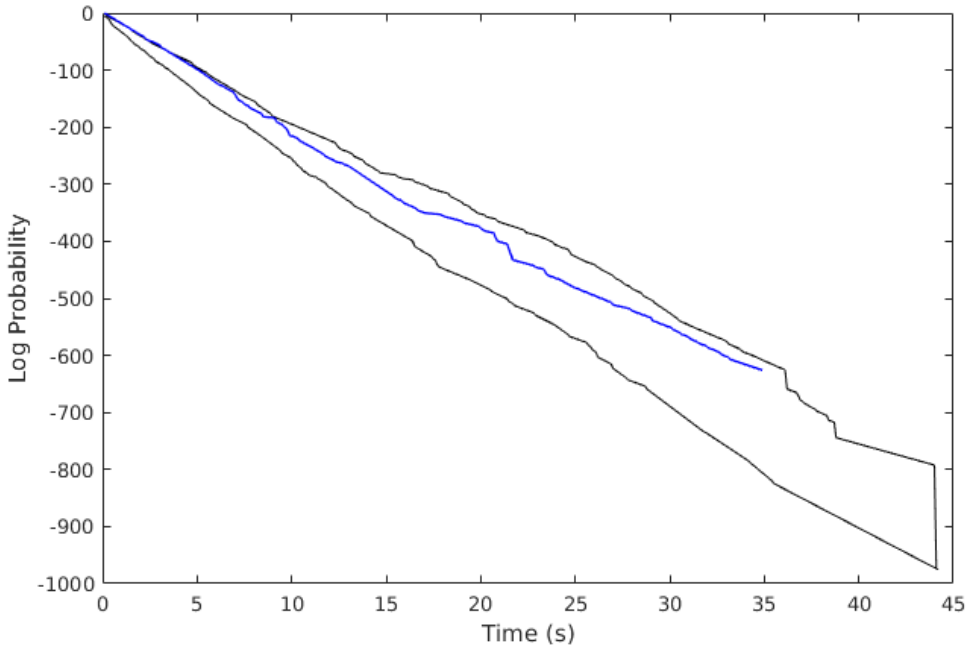


Figure 5.13: The final Viterbi sequence, with both failures recovered.

Corridor Navigation With Unrecoverable Failure

This test case is set up as before with the robot navigating between two points within a corridor environment. As normal the planner dispatches the navigation action, and the robot begins moving towards the goal with the executive monitoring its progress as it goes. However unlike in the previous test cases, this time instead of people crowding round the robot, a barrier has been erected in the middle of the corridor. This barrier affects the full width of the corridor, preventing the robot from reaching its goal.

As the robot approaches the barrier it starts to slow down, before coming to halt shortly before it. This is noticed by the anomaly detection routines, and the executive transitions into a recovery scan action. The scan action executes and monitors as normal, before passing back control to the navigation action. However unlike the previous examples, the physical state of the robot has not been improved. As such the executive prunes the error states and resumes the navigation, however the robot cannot move or progress any further resulting in another failure. The executive is notified and as before initiates the scan action in an attempt to recover from the problem. As these recovery actions do not improve the state of the robot in this instance, the system can get stuck in a fail-recover-fail cycle. To combat this a a simple function has been implemented

into the framework to detect if the execution has failed multiple times in a row, and if so halts the current action.

This case test highlights two problems of this system. Firstly if the robot cannot select an appropriate action for the type of failure it encounters, the recovery action can stop being an effective tool and instead simply prolong the problem. It also demonstrates if an action is no longer feasible, as in the case of the blockage where no recovery action would be applicable, then another solution is needed in order for the robot to reach its goal. In Chapter 4 the system was extended to address these issues, and this is evaluated in the next section.

5.4 Introspective Executive

The Introspective Executive, presented in Chapter 4, was designed to address some of the limitations highlighted in the above experiments. By implementing a controller within the executive that sits on top of the introspective models, it allows an element of reasoning to be introduced to the execution which can overcome a greater variety of obstacles. Under the introspective action execution framework, failure was an implicit state detected by one model before the system transitioned automatically into another. With the addition of a controller to the executive this failure state within an execution is made explicit, providing an additional point of reasoning from which decisions can be made.

Furthermore, by monitoring these actions the controller can learn an estimation of how they perform in a realistic environment, and use this information to improve the system's overall execution. Actions that regularly fail can be substituted for more reliable ones, and problems that arise during an execution can be addressed through constantly refining the execution strategy based on the model's feedback. The following experiments are designed to demonstrate how a more intelligent executive, which can deliberate alongside the planner, allows for more robust execution in dynamic environments.

5.4.1 Acting Appropriately

The first set of experiments examine the executive's ability to determine the cause of a failure and select an appropriate response. When acting in a dynamic environment where a wide range of issues can arise, it is important that the system can distinguish between them in order to select the action that has the greatest chance of recovering the execution. These experiments continue the setup described in the previous section, with the robot navigating between waypoints in a corridor. This time however there is no predetermined recovery procedure, and a successful execution relies on the executive's ability to determine the appropriate action. Discussion of the executive's learning and policy derivation are purposely omitted from these test cases, and will be visited in detail in Section 5.4.2.

Corridor Navigation With Induced Localisation Failure

This test case is designed to demonstrate the executive's ability to recognise a specific type of failure, and react appropriately. Similar to the previous experiments, the robot is set within a corridor, and given the goal of traversing to the opposite end. The deliberative component produces a single action plan to achieve this, `NAVIGATE(BOT1 WP6 WP8)`, and passes it to the executive. An important difference to before, which will be further illustrated in future test cases, is that it is the full plan which is now passed to the executive instead of a single action. The executive uses this plan to derive an execution strategy which will reach the goal, and dispatches a navigation action. Simultaneously it loads up the corresponding introspective model, and begins monitoring the action's behaviour.

The navigation initially progresses as normal, however 14 seconds in a crowd appears, obstructing the robot's view of environmental features. This results in the same abnormal behaviour as before, and is detected by the robot's anomaly detection routines. This time however, rather than automatically transitioning into a recovery action the Introspective Control Unit (ICU) halts the current action, and notifies the executive controller's state estimator that a problem has arisen. Knowing only that a problem has occurred, the state estimator enlists the failure classification module to help diagnose the current state of the system. This examines the sequence of features derived from the robot's sensors, and compares them to the learned decision tree for the navigation action. Based on the low velocity and localisation score the classifier determines a delocalisation has occurred.

This classification is passed back to the state estimator, which updates the executive controller to the current state of the system. This new state reflects an abstract localisation failure having occurred at some point within the current execution, the identification of which allows the executive a point to reason and make a choice of recovery action from within the execution. For the current failure state the executive dispatches a scan action, and initialises the associated introspective model. This executes as before, and upon completion notifies the controller's state estimator, which identifies the system is now in a recovered state. This state reflects the system still being within the navigation action's overall execution, but now having overcome a problem. It provides an additional decision point where the executive can choose to resume executing the initial action, or abandon it and attempt a new action to return to the previous state.

In this case the executive chooses to resume the navigation, and the corresponding model alongside the current progress is reloaded. The error states are identified and pruned, and the execution resumes its progress towards the goal.

This test case highlights the executive's ability to recognise and react appropriately to failures within the environment. It can now attempt to classify the type of failure it encounters, and select a recovery action which is best suited to the type of problem identified. From an observer's perspective this test case and the previous cases dealing with localisation failure will progress identically, however an additional layer of reasoning now supports the overarching execution. The precomputed policy coupled with the quick failure classification routines minimises disruptions during the execution, and that the models and actions transition smoothly into one another as before.

Corridor Navigation With Induced Blockage

In this test case the robot was again placed within a corridor environment and given the goal of reaching the other end. This generates the simple plan `NAVIGATE(BOT1 WP6 WP8)`, which gets passed to the executive and incorporated into a control strategy. The executive begins a navigation action, and initialises the corresponding introspective model.

This navigation begins as normal, however this time instead of a crowd appearing, two people step out from an adjacent office directly into the robot's path. This causes the robot to stop abruptly before the obstruction. If the people disperse the robot can continue its navigation, however if they linger their proximity to the robot's sensors can hinder its ability to determine a suitable alternative path around them, which can

result in the low level path planning routine failing and the robot becoming stationary.

This behaviour is reflected in the introspective model through the Viterbi sequence constantly self transitioning into the same state, a problem that is quickly recognised by the Temporal State Counting anomaly detection algorithm. As above, upon detecting the error the introspective execution alerts the controller via its state estimator, which attempts to diagnose the failure. The diagnostics compares the current features against the respective decision tree. Based primarily on the "clutteredness" rating, a summation of the laser readings used to indicate how close the robot is to other objects, and the velocity, the rules determine the robot has likely become blocked. This is passed back to the state estimator, which updates the controller.

The controller acknowledges this new state and dispatches the appropriate recovery action. For this situation it chooses a clearance action, which moves the robot backwards away from the object blocking it, allowing the path planning routine to determine any possible routes to the goal that may have been obstructed. The action executes alongside the respective introspective model, and upon completion the executive is updated. A resume action is dispatched, and the navigation action is reloaded and its error states pruned, allowing the robot to continue, now with a clearer path to the goal.

This test case is designed to further show the executive's ability to detect different types of failure, and choose an appropriate action. This can be expanded to multiple different types of failure, each being accompanied by its own recovery action, as long as there is a means for the executive to differentiate between the problems.

5.4.2 Learning From the Environment

Being able to select a response based on the context of a failure is an important part of operating in a dynamic environment, however sometimes even this is not enough. If the robot encounters a situation that it cannot overcome, such as a completely blocked path or a closed door, it may have to abandon the action completely and seek an alternative. Furthermore, sometimes certain actions in the environment may experience consistently high rates of failure, which may be recoverable but can degrade the robot's overall execution performance.

This section outlines a set of experiments that demonstrates that by incorporating an element of learning into the executive's controller, the robot can avoid problematic

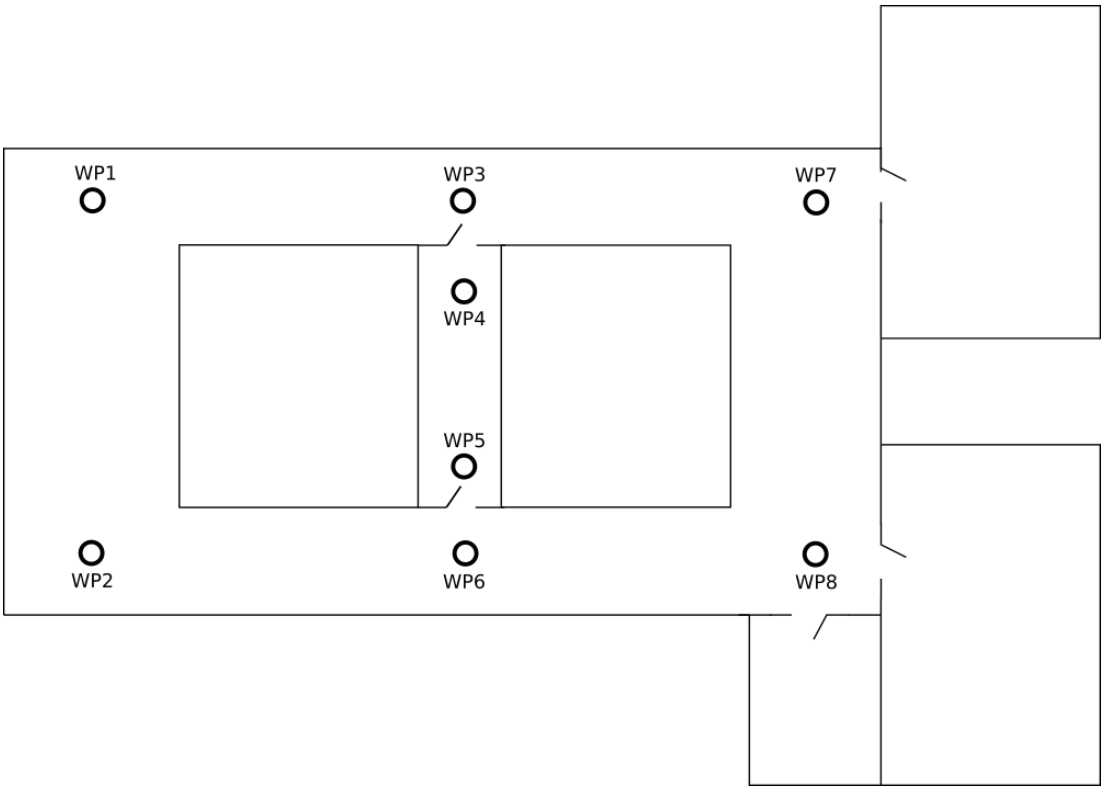


Figure 5.14: An example of the office environment the robot has to navigate through

actions and make better decisions with respect to its goals. The environment for these experiments is now opened up to the full office floor as shown in Figure 5.14, featuring multiple doors, corridors and open areas. For these experiments it is assumed resource consumption is not an issue, and that any policy change can be verified and executed.

Learning From Successful Executions

This test case begins with the robot sitting at `WP1` in the office environment, with the goal to take a picture to confirm if anyone is in the office at `WP8` . The deliberative component examines this problem and produces a plan that will transition the robot towards its goal, as follows:

```
(NAVIGATE BOT1 WP1 WP3)
(NAVIGATE BOT1 WP3 WP7)
(NAVIGATE BOT1 WP7 WP8)
(TAKE_IMAGE C1 BOT1 WP8)
```

As the plan is being constructed, the executive initialises its own controller and evaluates the domain files to identify if there is an existing control model available that

matches the current specifications. No such model is provided, therefore a new controller is generated and initialised with the default probabilities for each action derived from the verification data. The plan is then passed into the executive to be used as the basis for the reward function.

Starting with the initial state, each plan action is applied in sequence and the resulting state is assigned a reward within the MDP. For the following experiments each state from the plan sequence is given a reward of +3, except the goal state which is set to +15. Failure states are given a reward of -1, and all other states are valued at 0. These values are set through empirical evaluation and can be altered to produce different behaviours within the controller. Higher values assigned to plan steps encourage the robot to continue to follow the plan in spite of greater levels of failure, while lower values will allow the controller to deviate quicker with regards to environmental interference.

With the rewards set the executive begins to evaluate the constructed MDP and produces a policy. As this is the first time this control model has been used, each action is initialised to its default optimistic probability, resulting in a policy that aligns strongly with the provided plan. The executive begins execution by dispatching the first action, `(NAVIGATE BOT1 WP1 WP3)`. The appropriate behavioural model is loaded into the ICU, and the system begins to monitor its progress as before. The environment is cleared for this test, resulting in the first action completing without incident and the executive controller being informed of its success. This prompts the controller to update its domain knowledge to register the action's outcome, incorporating the successful observation into its estimates for future predictions. For these experiments the discount factor λ, applied to the prior estimates as discussed in Section 4.4.2, is set at 0.8. This allows the priors to reduce their influence over a few observed results, which is beneficial for these experiments where only small quantities of experience is used. This alters the system's belief of this action successfully executing from 72% to 86.7%.

The successive actions dispatched undergo a similar process. The executive selects the appropriate behavioural model, monitors the progress of the execution, and updates its control knowledge based on the outcome. In this experiment all three actions complete without incident, raising the executive's confidence in them for future executions. This test case demonstrates how the executive can incorporate the plan into its execution, and begin to learn the underlying probabilities associated with executing actions within the current environment.

Learning From Failed Executions

This experiment begins with a similar set up to the previous test case. The robot begins in `WP1` and has the goal of reaching the office at `WP8`. However this time the environment has been altered to be more difficult for the robot to traverse. Along the corridors connecting `WP1` to `WP3` and `WP3` to `WP7`, boxes have been scattered to represent groups of people idling or obstructions along the path. This information is not known to the robot prior to execution, and as such the task begins as before. The planner assesses the problem and domain provided, and produces the same plan as the previous experiment to reach `WP8`:

```
(NAVIGATE BOT1 WP1 WP3)
(NAVIGATE BOT1 WP3 WP7)
(NAVIGATE BOT1 WP7 WP8)
(TAKE_IMAGE C1 BOT1 WP8)
```

During the initialisation of the executive the domain is recognised as one which has been used before, so instead of creating a new control model it rebuilds the previous one. This includes the updated transition function learned from the prior execution. The plan is taken and transformed into the reward function and the executive assesses the model and constructs a policy to direct the system towards its goal. As the results from the previous experiment only reinforced the previously selected actions, the policy once again follows the plan closely.

The executive dispatches the first action, `(NAVIGATE BOT1 WP1 WP3)`, and begins to monitor its execution with the appropriate behavioural model. The execution commences, and progresses as normal until the first set of obstructions are encountered. At this point the robot starts to slow down and hesitate, as its tries to circumvent the new objects. As it gets deeper into the middle of the cluster of objects the navigation behaviour deteriorates further and the introspective model recognises this and alerts the system to possible impending failure. The executive diagnoses this as a localisation problem, and updates the state of the system and instigates a scan recovery action.

As the recovery action is dispatched, the executive updates its knowledge base to inform it of the current failure. This increases the number of observed localisation failures for this particular action and recalculates the probability distribution over its outcomes. This is then passed back to the executive controller, which recalculates the

policy in light of this new update to ensure the current execution strategy is still the recommended course of action. This update is applied during the recovery process, which attempts to ensure that if any major changes are needed to the policy they can be calculated without disrupting the execution. In this example no change occurs, and the executive maintains the same policy as before.

Upon the recovery action completing the executive controller registers the change into a recovered state, updates the knowledge base again, and resumes the navigation action towards `WP3`. Before reaching this waypoint another failure occurs due to continued obstruction. The system handles it as before, issuing a recovery action to overcome the problem, and updating its knowledge base to capture the experience from the execution. The next action, `(NAVIGATE BOT1 WP3 WP7)`, fares slightly better however it still incurres a single failure during its execution, and the last action, `(NAVIGATE BOT1 WP7 WP8)`, completes without incident.

This test case is designed to demonstrate how the executive can react to and learn from failures when acting within a dynamic environment. Aside from being able to acknowledge and respond to an execution failure, the system can capture this experience and use it to refine its estimations of an action's performance. This helps the system learn possible sources of failure within its domain, that it can use to enhance future executions.

Refining Plans Based on Experience

Once again this experiment begins with the robot at `WP1` with the goal of reaching the office adjacent to `WP8`. Prior to this experiment beginning however the previous experiment has been repeated two more times, executing the robot through an environment with obstructions designed to hinder its progress. Each of these executions incurred additional failures, totalling 4 localisation problems between `WP1` and `WP3`, and 3 between `WP3` and `WP7`. This reduces both actions estimated probability of success below 40%.

The planner uses a static model for its domain, and therefore has no knowledge of these failures or any of the previous executions. As such it constructs the plan as before, and sets out the same plan steps:

```
(NAVIGATE BOT1 WP1 WP3)
(NAVIGATE BOT1 WP3 WP7)
(NAVIGATE BOT1 WP7 WP8)
```

```
(TAKE_IMAGE C1 BOT1 WP8)
```

During the executive's initialisation it recognises the domain and constructs the appropriate control model. It then receives the plan, and transforms it into the reward function. This starts to differ from previous executions when the executive begins to create its policy. Previously the policies have been constructed around the plan steps, assuming the robot starts in the same initial state and executes each action successfully in turn. However due to the obstructed corridors the first two plan actions have failed several times, and the executive is aware of their low probability of success. It is also aware from the control model that there is an alternative route through the bottom half of the environment which leads to the goal in a similar number of steps which still has its initial optimistic action values.

Upon deriving a policy this time the executive recognises this alternative set of actions to the plan may provide a more successful means of achieving its goal. The rewards set by the plan steps are weighted together with the learned likelihood of success and found to be a worse choice than taking the alternative route via the bottom corridors. This results in a policy that still achieves the system's goals, but differs from the initially supplied plan, as shown in Figure 5.15.

As before, with the policy created the executive dispatches its first action `(NAVIGATE BOT1 WP1 WP2)`. The introspective control unit receives the action and begins executing it, while monitoring it with the appropriate behavioural model. As there are no obstructions along the bottom path of the map, this action completes as normal. This updates the executive controller, and in turn the domain knowledge, and dispatches the next action. `(NAVIGATE BOT1 WP2 WP6)` and `(NAVIGATE BOT1 WP6 WP8)` complete without incident, reinforcing the policy selection of the executive.

For a comparison the above experiment is repeated several times using the policy derived from the robot's experience, and then again enforcing the original plan with introspective recovery active. The results produced are shown in Table 5.3. The route devised by the planner can be seen to incur multiple execution failures, due to attempting to navigate corridors littered with unmapped obstructions. The policy on the other hand uses the robot's experience to learn which actions are likely to produce these failures and manages to reduce both encountered failures and execution time.

A similar example of this beneficial learning can be demonstrated with regards to the robot's ability to navigate through doors. Starting a new execution, the robot was

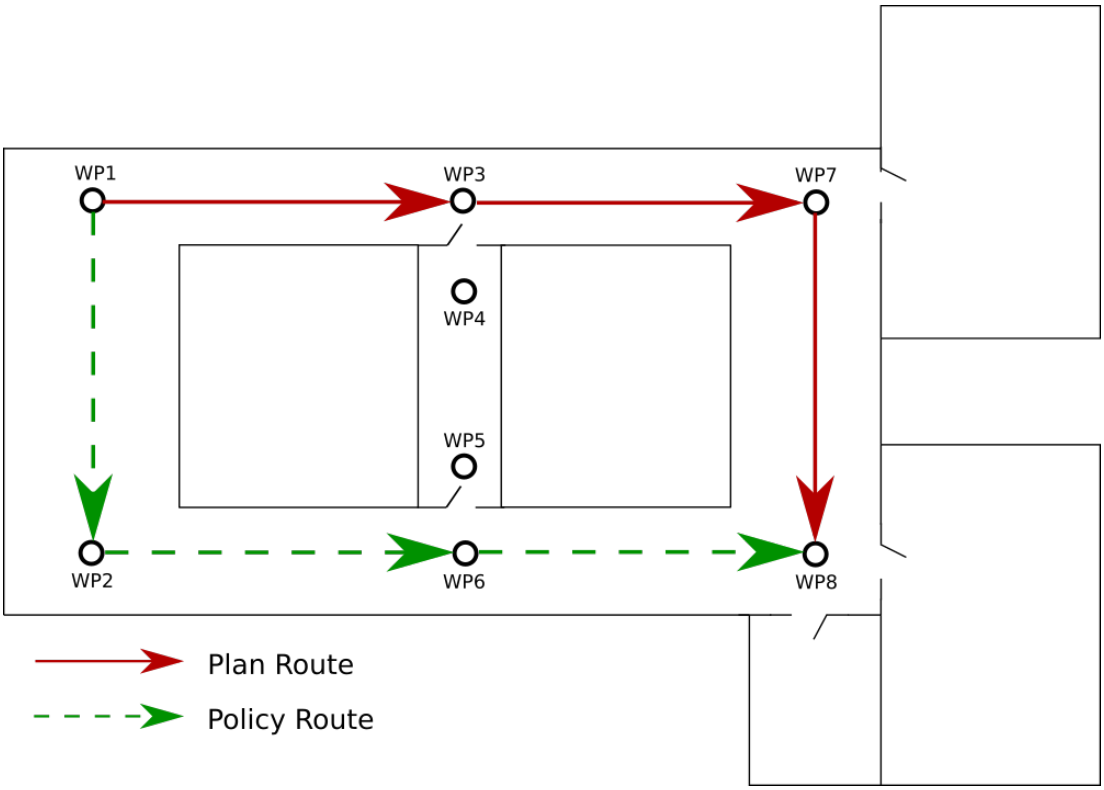


Figure 5.15: A representation of the two different sequences of action the robot can choose. The red solid arrow shows the actions set out by the plan, which take no knowledge from the environment. The green dashed arrows show the path following the policy based on the robot's experience.

positioned at `WP3` and given the task of reaching `WP6`, requiring it to pass through two separate doors to reach its goal. One instance of this execution results in two failures upon attempting the first door, and two additional failures attempting the second. When this experiment was repeated a second time, the number of failures coupled with the previous successes in the environment lead to a policy being created which ignores the middle corridor and its two doors, and directs the robot back to `WP1` and around the outer corridors to the goal. This is shown in Figure 5.16.

Once again as a comparison this experiment has been repeated multiple times following both the learned policy and initial plan, the results of which can be seen in Table 5.4. Following the plan in this instance produces several failures, with the robot consistently encountering problems attempting either doorway. The policy on the other hand completes successfully without failure. However in spite of this, the direct route proves to be quicker in most instances as the doorway navigations and subsequent recovery actions are all relatively quick to execute in comparison to having to navigate

Plan Versus Policy with Corridor Clutter			
Controller	Route	Number of Failures	Time (s)
Plan	1-3-7-8	3	189.2
Plan	1-3-7-8	2	166.3
Plan	1-3-7-8	2	160.1
Policy	1-2-6-8	0	124.3
Policy	1-2-6-8	0	119.4
Policy	1-2-6-8	1	143.7

Table 5.3: A comparison showing the different times and number of failures associated with each control strategy. The plan route take the robot across the top corridors, which are littered with objects, the executive route goes via the bottom path which is clear.

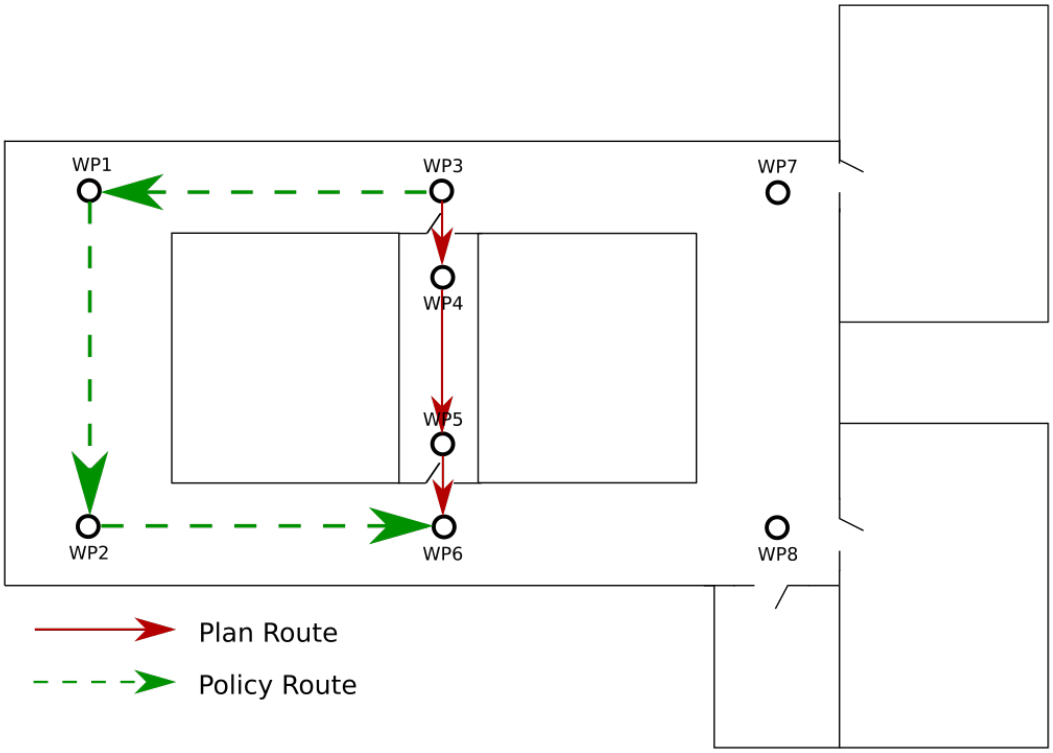


Figure 5.16: A representation of the two approaches the robot can take to reach `WP6`. The green path is the plan which follows the most direct route to the goal. However due to numerous failures attempting the doors, the executive learns to use an alternative route, via the outside corridors represented by the green dashed arrows.

Plan Versus Policy with Doorways			
Controller	Route	Number of Failures	Time (s)
Plan	3-4-5-6	4	113.6
Plan	3-4-5-6	3	105.2
Plan	3-4-5-6	5	128.3
Policy	3-1-2-6	0	120.9
Policy	3-1-2-6	0	125.2
Policy	3-1-2-6	0	126.8

Table 5.4: A comparison showing the different times and number of failures associated with each control strategy. The plan takes the direct route via the middle corridor and two doorways, the policy redirects the path via the corridors.

across three large areas.

Balancing execution time versus number of failures is a important consideration. If failures are simple and easy to fix, then encountering them is inconsequential and the set of actions which achieve the goal in the minimal time may be recommended. If failure can be catastrophic, then it becomes much more important to avoid them at all costs to ensure the success of the execution. In this situation the robot's inability to easily traverse doorways is not critical, however there are times where even the recovery actions prove ineffective and it cannot navigate through to its destination. In these instances being able to learn and attempt a separate path can be highly beneficial.

This test case demonstrates how the system manages to learn from its previous experience within a domain, and use this knowledge to refine its execution. Using the experience gained from executing actions, the system can create more robust polices that avoid potential sources of failure while still achieving its high level goals.

Online Adaption to Failure

For this test case the robot is positioned at `WP6`, and given the goal of reaching `WP3`. A clear path exists between these two points via a corridor, however this requires the robot traversing through two doors to reach the other end. As shown in the previous test case this can be problematic for the robot, however this time the door between `WP6` and `WP5` has been shut, completely blocking access. This is unknown to the planner, which has had the status of doors abstracted from its domain due to the regular alteration of their state by other actors within the environment. Upon initialisation, the following plan is produced:

```
(NAVIGATE_DOORWAY BOT1 WP6 WP5)
(NAVIGATE BOT1 WP5 WP4)
(NAVIGATE_DOORWAY BOT1 WP4 WP3)
```

This plan is passed to the executive and a policy is derived. As the executive has no previous experience executing any of these plan actions they maintain their original estimations, and the policy aligns with the plan. The initial policy is shown in Figure 5.17 as the solid red arrows following the projected plan. The first action is dispatched, `(NAVIGATE_DOORWAY BOT1 WP6 WP5)`, and the introspective execution loads the appropriate behavioural model and begins to execute it. With the door shut however the robot's movement is severely impeded, and it quickly becomes hesitant about moving forward. This is acknowledged by the execution monitoring routines, which recognise a failure and pass a notification to the executive controller. This updates the domain with the new observation and dispatches a recovery action to align the robot with the doorway. The controller re-evaluates the policy in light of this failure, and finds it is still appropriate for the current execution.

The recovery action completes and notifies the executive, and the navigate door action resumes. As the environment has not changed the robot begins to instantly exhibit the previous hesitating behaviour, and another failure is generated. This is passed to the controller, and the knowledge base is again updated and the policy re-evaluated. This evaluation occurs online and is designed to explore if a new, more suitable policy is available based on the knowledge the robot has gained from executing in the world. In this case due to multiple failures with the current action, coupled with successful previous executions between `WP6` and `WP8`, when the policy is re-evaluated it is found to be no longer the best sequence of actions to reach the goal.

As the executive executes the recovery action, a new policy is derived based on the latest information. This policy encourages the robot to abandon its current action and traverse along the bottom path to `WP8`, up to `WP7` and finally along to the goal at `WP3`. The updated policy is shown in Figure 5.17 as the dashed green arrows. This is calculated such that by the time the recovery action has completed the new policy is in place, and the abandon action is dispatched. This causes the robot to attempt to navigate back to `WP6`, which in this case it has not moved far from. Once there it begins dispatching and executing the actions to reach the goal via the alternative path.

A traditional plan execution approach to this problem would be to update the state of the system upon failing to execute the navigate door action, then replanning. The relies

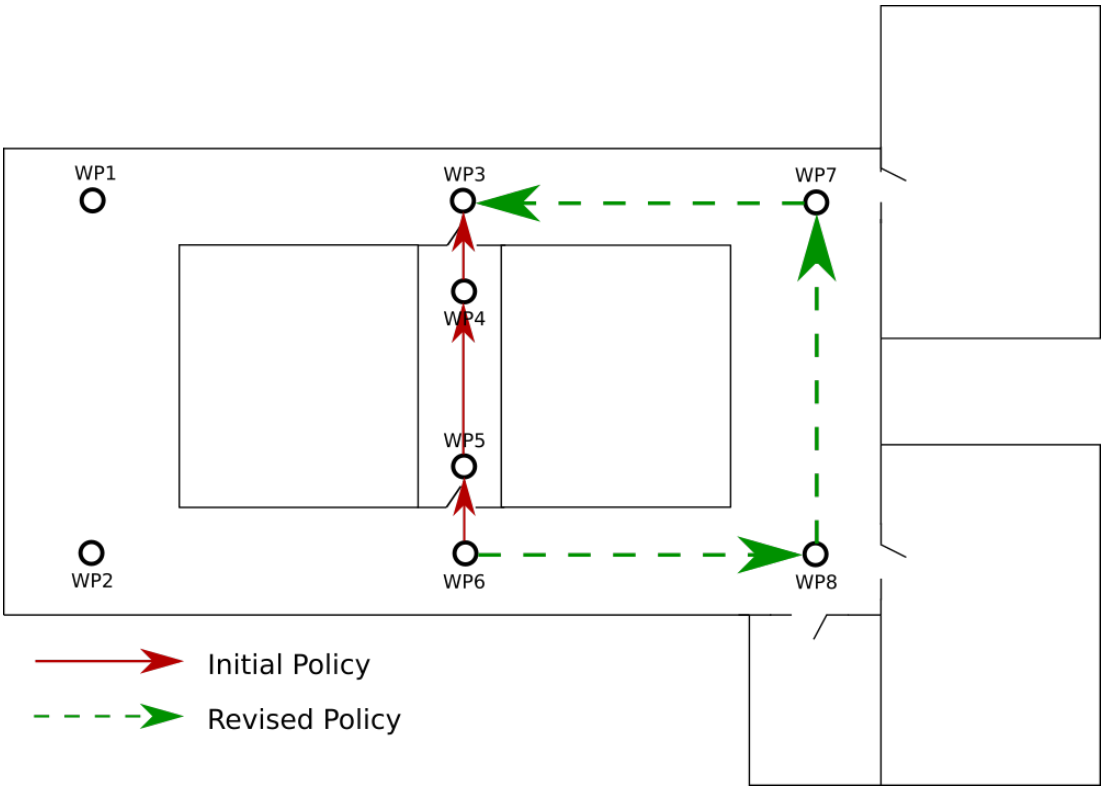


Figure 5.17: A representation of the robot having to navigate a closed door. The initial policy follows the plan actions directly to the goal, and is shown by the red arrows. The updated policy after experiencing failures updates the actions to go around the doors to the goal.

on the ability of a state estimator to accurately detect the problem, in this case a door being shut, and a world model that can express this problem. It can also require the robot to idle while a plan or plan repair takes place. In contrast the approach presented can learn an action is failing without having to know the specific reason. If a failure occurs that the robot cannot recover from, either through it being impossible or simply not having a predefined recovery action in place, the system can still learn from the failure and adapt the execution.

This test case shows an example of how the robot can monitor its actions online and update its reasoning based on feedback from the environment. If a certain action is frequently failing and alternative actions are available, the executive can recognise and react to this. As updating the execution strategy happens while the system is attempting to recover from whatever caused the problem, it allows the robot to fluidly change policies mid execution without having to pause or break control. Furthermore future executions can then avoid these actions, lessening the number of future failures.

Forgetting Failures

At the end of the last test case navigating between WP6 and WP5 was found not viable for the system due to a closed door, and an alternative path was chosen. It is useful for the robot to avoid this route to prevent future failures, however to avoid this path completely can become detrimental. In this case specifically the main cause of the failure was an environmental issue, and the door can become open again. While it is important to learn from the previous actions and use this knowledge to improve execution, it is also important not to allow negative experiences to completely prevent action selection.

In order to avoid this issue a decay function δ has been implemented, described in Chapter 4.4.3, such that over time if an action has not been chosen due to a sufficiently low probability, it can start to slowly increase back to its original estimated value. For these experiments δ is set at 0.8, to allow a reasonably rapid progression back to the prior probability. An action is selected for decay based on comparing a timestamp representing when an action has been last selected for execution and threshold value for its estimation of success. The threshold value τ has been set at 35%, and the time requirement ψ has been adjusted such that an action will qualify for decay after an hour.

The robot begins once again at WP6 with the goal to reach WP3, and the door between WP6 and WP5 has been reopened. The system initialises and a plan is created that directs the robot through the corridor directly to the goal, as above. This is passed to the executive, which upon creating and analysing the control model, recognises the plan actions have a high likelihood of failure and directs the robot to the goal via WP8 and WP7. Each of these actions dispatches and is monitored to completion, with a single failure recognised and recovered in the corridor between WP7 and WP3.

This experiment is then repeated after a period of time, with the robot again placed at WP6 with the goal of reaching WP3. The plan is produced is unchanged from before, and is passed down to the executive to form the reward function. This time however when the executive initialises, as it is constructing the control model it recognises that the transition between WP6 and WP5 via the navigate door action is not been executed in over an hour, and is below the 35% threshold of success. This causes the executive to tag it to be decayed, raising its probability of success to 46.2%, and normalising the distributions over the failure states.

With the adjustments made to any actions which match the decay criteria, the executive solves the MDP and produces a control policy. As the path around the corridors has been chosen and successful several times now, and the direct path still has a poor probability of success, the first decay does not impact the policy, and the executive continues as before taking the longer but more reliable route to the goal.

Once again this experiment is repeated. As the navigate door action between `WP6` and `WP5` has been tagged for decay and still not been selected, it is again selected for adjustment, and its probably raised to 62.8%. This time when the plan is passed to the executive and the policy is created the change in probability of the `(NAVIGATE_DOORWAY BOT1 WP6 WP5)` action is enough to encourage the policy to resume following the plan and take the direct route to the goal.

The executive dispatches the first action, and the robot attempts to navigate through the doorway to `WP5`. The door is now open, meaning the robot can reach the waypoint this way, however due to its normal trouble navigating doorways it fails the first attempt trying to pass through. This failure notifies the executive controller, and it updates its knowledge base to account for it. This reduces the action's probability enough that when the policy is re-evaluated it is once again found the most effective route to the goal is via the corridors to `WP8` and `WP7`, and is updated accordingly.

This experiment demonstrates how the executive can adapt its learning so that actions which are found to be poor choices are not permanently discarded. By decaying actions which no longer get selected it allows the executive to revisit them, and determine if they have become more feasible. If an action has failed due to an environmental factor, such as a door being shut or a corridor blocked, over time it may become accessible again, and it is important to learn this.

5.4.3 Executing With Resource Constraints

Up until this point, the previous experiments have assumed that there exists enough resources available to the system to accommodate any changes required for the execution. This is rarely the case under normal circumstances, and instead it becomes important to balance the number and type of actions selected against some form of constraint such as available power. If the system spends too much of its resources attempting to recover an action, it can result in later parts of the execution having insufficient means to complete. For these experiments each action has an associated

power cost taken from the plan domain, representing units of power consumed.

Offline resource monitoring

For this experiment the robot begins in WP3 and is tasked with the goal of reaching WP6. This continues to use the domain from the above experiments, and as such the robot has experienced failure when attempting to navigate the doorway between WP3and WP4. This time however the robot is given a fairly restrictive power resource limit of 50 units. The robot begins as always by producing a plan to the goal:

```
(NAVIGATE_DOORWAY BOT1 WP3 WP4)
(NAVIGATE BOT1 WP4 WP5)
(NAVIGATE_DOORWAY BOT1 WP5 WP6)
```

This plan is passed down to the executive alongside the resource limit, and it begins to form a policy. Based on its experience with the environment it initially disregards the plan, and a policy is generated which recommends that the robot moves to the goal via WP7 and WP8 based on its current state. Upon deriving this policy, the executive controller passes it to the policy verification module to ensure it is fit for execution. Using Monte Carlo techniques this simulates the policy several hundred times, and produces the average cost of its execution alongside the estimated cost of each chosen action. In this instance the simulated cost is 65.5 units, which exceeds the available resource limit.

The executive analyses the simulation data to identify actions which disproportionately consume resources, utilising over 50% of the expected action allocation. None are found, therefore every action on the simulated trajectory receives a cost of -1. This creates a new cost function which is now applied alongside the reward function as the executive attempts to derive a policy. These additional costs discourage a policy from simply following the previous actions, and a new policy is generated suggesting an alternative route to the goal via WP1 through WP2. This new policy is handed to the verifier and similarly is predicted to exceed the available resource constraints. Once again no single action is responsible for over consuming resources, and all simulated actions receive a cost.

When the executive tries to create a policy for the third time, due to both of the outer paths receiving costs and the plan rewarding the route through the middle corridor, the executive produces a policy that once again aligns with the planner. As demonstrated above with regards to the time of execution, the path with the doorways may incur more

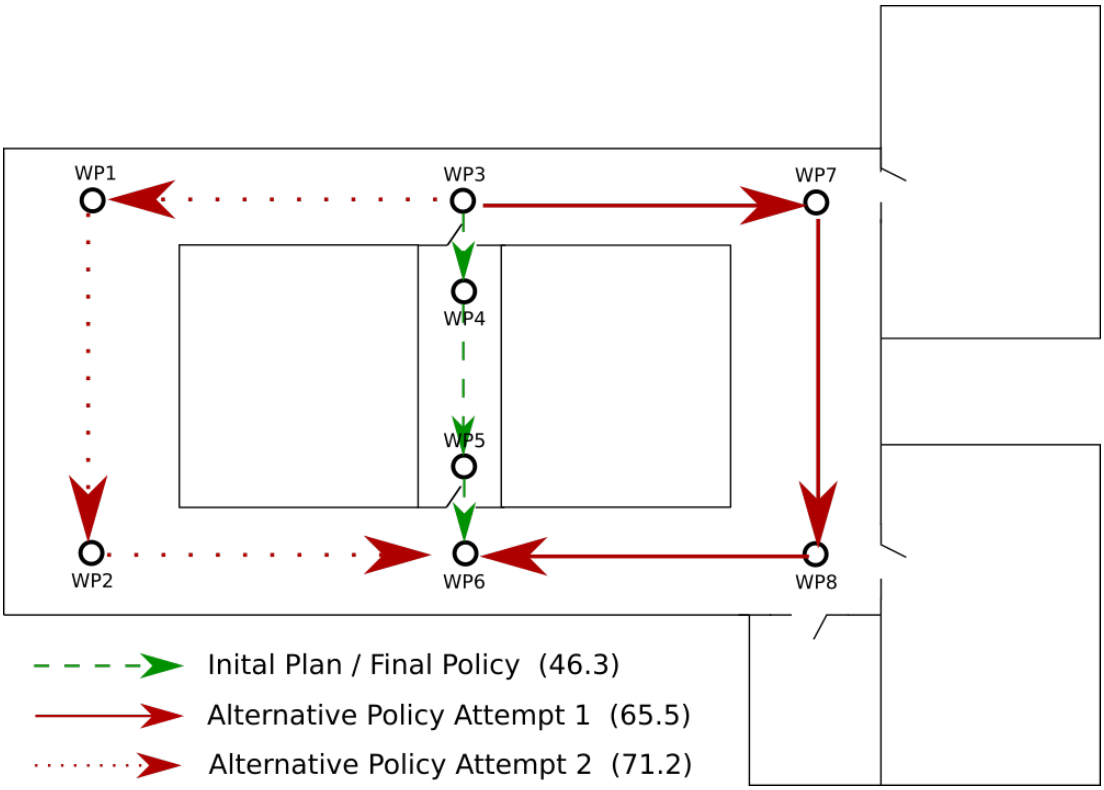


Figure 5.18: This diagram shows the polices the robot simulated before choosing an execution strategy. The system attempted both the right hand side path and then the left hand side path, however both were out with the resource limit provided. The simulated costs are displayed beside the policy attempt.

failures, but it is still the most direct route to the goal. Upon simulating this policy it is estimated to cost 46.3 units to execute, which falls under the set resource limit and is approved for execution. The executive can then begin to execute the actions as normal. The polices assessed by the executive can be visualised in Figure 5.18.

This case study demonstrate the executive's ability to reason with resources. The policies themselves are created without any knowledge of the cost of actions, in order to limit the complexity of the executive. However it can check that selected polices estimated costs fall within the resource limits of the systems, and if they do not it can take steps to reduce resource usage and find a more suitable execution strategy.

Online Resource Monitoring

In this test case the robot starts at `WP2` and has to reach the office at `WP7`. Prior to this experiment there has been a blockage failure in the open area between `WP8` and `WP7`, but otherwise the probabilities have been reset to their initial optimistic values.

The resource allocation for this task is set at 90. The planner examines the domain and produces a plan to reach the goal within the constraints.

```
(NAVIGATE BOT1 WP2 WP6)
(NAVIGATE BOT1 WP6 WP8)
(NAVIGATE BOT1 WP8 WP7)
(TAKE_IMAGE C1 BOT1 WP7)
```

Upon being passed to the executive the plan is set as the reward function, and even with the failures registered between `WP8` and `WP7` the policy produced adheres to the plan actions. This is passed to the verifier to ensure it executes under the resource level, and is found to have an average cost of 84.8 which is deemed executable. An overview of the initial policy can be seen in Figure 5.19. The first action is dispatched and the introspective execution begins to execute and monitor the navigation between `WP2` and `WP6`. The executive's resource level is lowered by the action cost, which for this corridor navigation is 25 power units.

During this navigation the execution is disrupted by people moving throughout the environment, causing a failure. This is recognised by the executive and a recovery action is dispatched. During the recovery the resource level is adjusted to account for the action, and the policy is rechecked to ensure it is still appropriate and executable. The executive finds no change or issue with the current policy, and upon recovering it continues towards the goal. As it continues however it is disrupted for a second time.

As before the executive recognises the failure and dispatches a recovery action. This depletes the resource count again, and now when the system comes to recheck the policy it discovers that while the error does not alter the policy directly, it has become unexecutable. Each recovery action costs the system 5 units of power, coupled with the estimated 25 units to finish executing the current action. The policy verifier has estimated the average cost of reaching the goal from the current state to be higher than the available resources. In this instance, when the verifier checks the individual action costs, the navigation between `WP8` and `WP7` are flagged as overly resource intensive. The previous failures coupled with the cost of executing back up actions, set at 10 units of power, make the resource estimates for this action high in comparison to its expected cost. This receives a negative cost of -5, and the other actions receive a -1.

As the robot recovers from the last failure, the executive attempts to derive a new policy. Even with the new negative costs included, due to the proximity to the goal

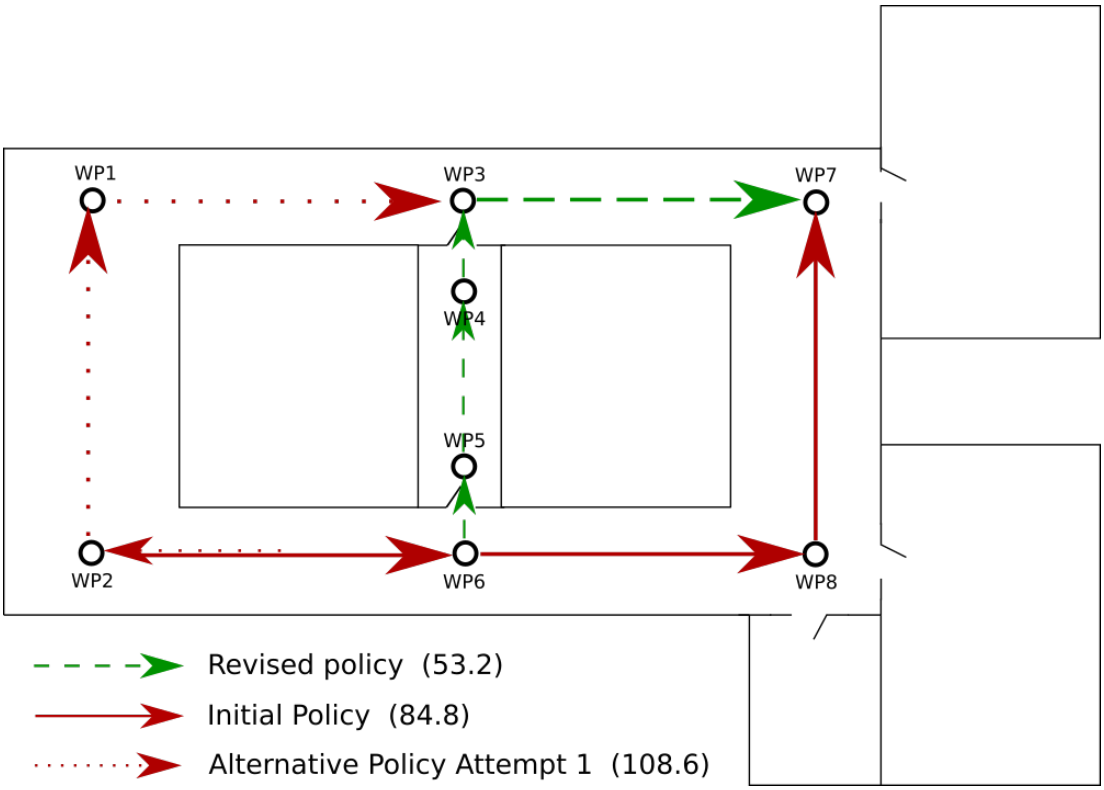


Figure 5.19: A representation of the robot updating its policy through resource constraints. The initial policy is shown is solid red, however during execution a string of failures makes the policy no longer feasible. The first attempt to revise this policy tries to send the robot back around the outer path, before finally settling on using the middle corridor. Resource estimates are shown beside each policy.

the initial policy is returned. This incurs a second round of additional costs. The next solution attempts to abandon the current action and reroute the robot back to `WP2` and around the outer corridor to `WP7`. This is dismissed by the verifier, which negatively rewards each action along this path. The number of negative actions forces this policy to be abandoned, and the executive chooses a policy which continues to `WP6` and then takes the corridor up the middle corridor to `WP3`.

For the purpose of this experiment each navigate doorway action is given a value of 5 power units, with a recovery value of 4. The middle corridor is similarly reduced, with a resource value of 10 units. This make the corridor an attractive proposition in spite of the lower doorway navigation probabilities. This new policy is passed to the verifier and is found to have a cost of 53.2, lower than the original path from the same state and within the current resource boundary. This is deemed executable, and when the robot has finished recovering from the previous failure it gets dispatched a resume

action and continues towards WP6, to now take the new route to the goal. This new policy is illustrated in Figure 5.19.

This experiment is designed to show the executive's ability to monitor its resource consumption online, and identify when the execution has veered too far over its resource limit and is no longer feasible. By taking into consideration the cost of executed actions and recovery actions, the system can quickly simulate the resources needed to complete the current goal, and if it has over stretched its limits, it can look for alternative actions which may complete in a shorter time period.

Plan Assisted Execution

This test case repeats the previous experiment with one difference, the navigate door values have been raised from 5 units of power to 10. The rest of the probabilities and resource costs are the same as before. This leads to the same plan being created, and in turn the executive generating the same policy. The first action is (NAVIGATE BOT1 WP2 WP6), and this gets dispatched by the executive to the introspective execution.

The execution of this action also progresses similar to above, with a disruption occurring shortly after the navigation begins causing the robot to delocalise. This problem is identified by the anomaly detection routines, and a recovery action is executed. The policy is still verified as acceptable, and the navigation action resumes executing. A second failure is induced towards the end of the navigation, once again causing the system to have to issue a recovery action.

As before when the executive attempts to verify the policy after the second failure it discovers it now no longer executes within the boundaries of the resource constraints. This time however, after discounting the initial path and the alternative route back via WP2, the middle path is also found to be infeasible. Due to the increase in the resource allocation for entering doors it is now no longer possible to take the middle corridor. The executive continues to search for a new policy, however no executable solution is found. In order to prevent the system from becoming stuck continually searching the executive has a preset number of attempts to derive a solution before giving up. For this experiment this is set to 10 – enough attempts to allow multiple polices to be attempted but not so many that the system becomes stalled trying every combination of actions.

Once the system reaches its allocated number of attempts the executive passes the problem back to the planner to try and solve. This involves generating a plan state

from the current executive state. Due to the two components sharing similar models of the domain this is straightforward, and involves adding the numeric elements stripped out for the MDP, and setting the resource allocation to the current levels based on the execution. It also requires the system to be moved into a state the planner can recognise, by either abandoning the current action and returning to the previous state, or completing the action and replanning from there. In this experiment the system resumes the navigation action and uses `WP6` as the base for replanning, putting itself into a safe state once it arrives there.

In this example no unrecoverable failure has occurred, the system has merely ran out of resources, and the provided plan state reflects that. It still allows all options to be selected, just with a new and tighter resource constraint. This leads to the planner producing a plan which continues the same path as the initial plan, as without knowledge of the action's probabilities of failure, this is still the quickest and most efficient route to the goal. The executive takes the newly provided plan, resets all reward and cost functions, and attempts to derive a new policy. As the new plan does not present a new solution, the executive fails to find a policy which can achieve the goal within the constraints. As the system has already attempted a replan and failed, the system ensures it is in a safe state and shuts down.

This experiment shows that communication is possible between the planner and the executive in both directions. If the executive cannot find a solution that it can execute, it has the option to initiate the planner and allow it another attempt to reach the goal. In the described experiment the planner was not provided with enough information to discover a workable solution, however in a more complex scenario the planner may be able to reason about the different goals and produce a plan that enables partial goals to be achieved. A more developed communication between the planner and the executive would also allow the planner to provide more appropriate solutions where the policy cannot.

5.5 Simulated Results

The previous sections have presented a series of case studies, each focusing on how different aspects of the system perform when deployed onboard a physical robot operating in an active office environment. This is intended to demonstrate the utility of this system when interacting within a real dynamic space. However in order to provide

Simulation Action Parameters			
Action	Resource Cost	Initial Probability (%)	Underlying Probability (%)
Corridor Navigation	30	72	50 - 80
Open Area Navigation	40	72	50 - 80
Doorway Navigation	5	64	30-60
Scan	6	100	100
Back Up	10	100	100
Door Alignment	4	100	100

Table 5.5: A summary of the different actions used in the simulation and their associated parameters. In this model the recovery actions have a 100% chance of moving the system to a recovered state, however from this state the action can either resume or fail again.

a greater level of quantitive analysis, a simulation of the introspective executive has been created. This allows not only a greater number of executions to be run, but for the system to be tested in a larger environment than was previously possible.

An image of the simulated environment can be seen in Figure 5.20. This maintains the office structure of previous experiments, consisting of corridors, doors and open spaces, while expanding it from 8 waypoints to 20. This allows the parameters used in the simulation to be similar to those used on the robot itself. Goal rewards are set at +15, plan steps carry a reward of +3, and recovery actions a negative reward of -1. Table 5.5 contains the resource costs of each action type, alongside their associated probabilities.

In Table 5.5 each action is associated with an initial probability and an underlying probability range. The initial probability represents the executives optimistic estimate of each action successfully completing prior to any learning occurring. The underlying probability has been included to represent the uncertainty of executing in a dynamic environment. One of the goals of this system was for it to learn how individual actions perform in the real world, and use this knowledge to improve future executions. To incorporate this in to the simulation each action has an associated underlying probability of successfully executing, which is drawn from the specified range at the start of a set of experiments. This probability is then used to determine the outcome of an action's execution for both plans and policies.

To test a wide range of executions a selection of initial states and goal states were cre-

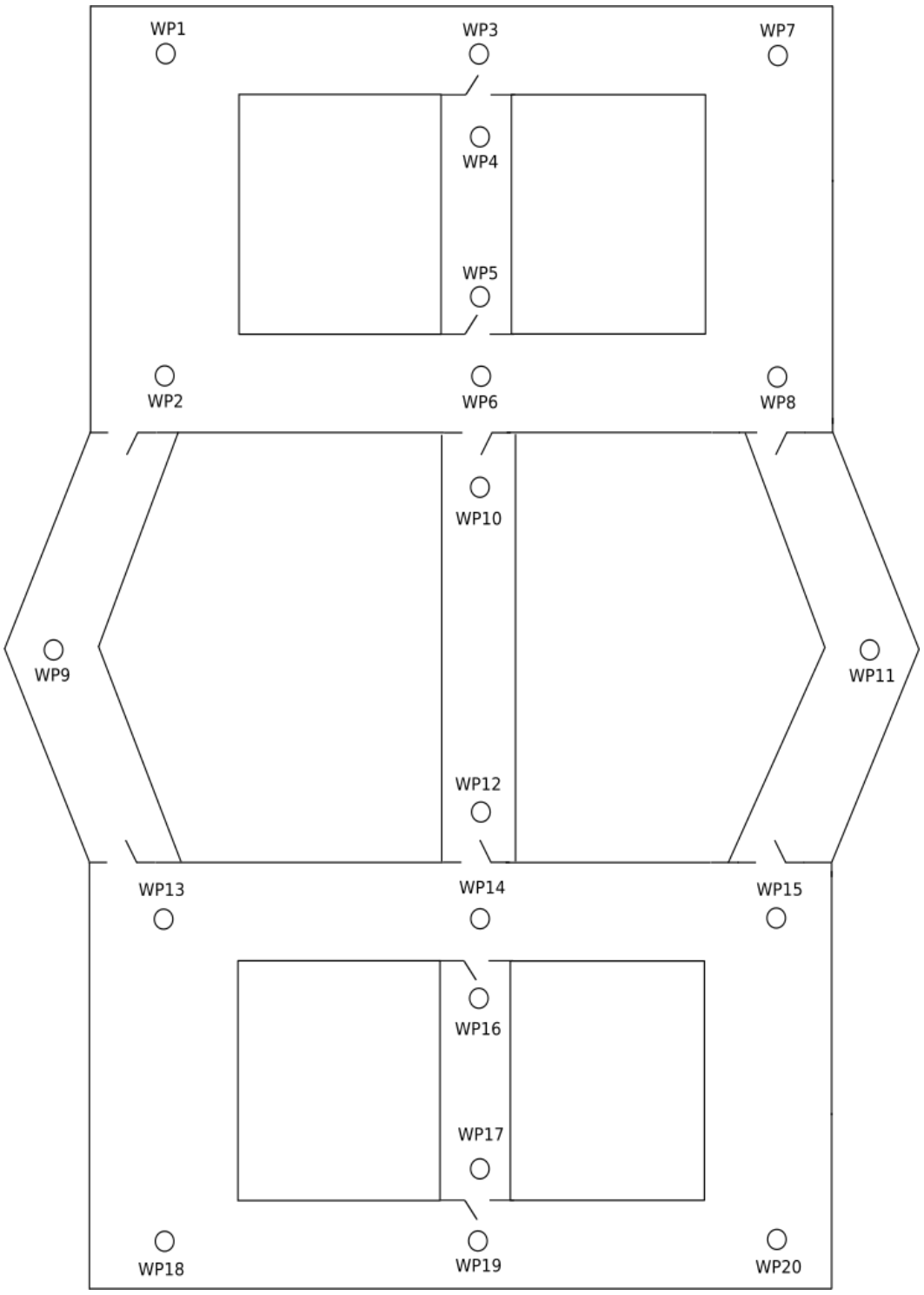


Figure 5.20: A representation of the environment used in the simulated experiments.

ated representing the robot starting and ending at each of the waypoints in the domain. These are randomly selected, and used to generate a sequence of actions representing a plan. This plan is passed to the executive and used to inform the policy as before. Both executing the plan, with recovery actions enabled, and the policy can be simulated allowing direct comparison of how they perform with the same start state, end state and underlying probabilities.

5.5.1 Simulated Learning

The first set of simulated experiments focus on the learning aspect of the executive. Through its interaction with the environment the executive should be able to learn which actions are most likely to succeed, and exploit this knowledge in future executions. In Section 5.4.2 a selection of test cases were shown to demonstrate this learning, and how when used in conjunction with a plan the executive can improve its execution. These test cases were performed onboard a physical robot in a noisy office environment, however they were limited in number. This section aims to expand this analysis of the executive's learning with the use of the simulation.

Setup

Using the simulator outlined above 1000 executions were performed. These were set in the larger environment specified in Figure 5.20, with randomised start states and end states to provide a variety of different plans. For these experiments the discount factor applied to the prior estimates to balance the learning rate remains the same as the physical experiments at 0.8. For each goal generated the simulation will attempt it with the executive using a planner with recovery actions enabled, and with the introspective executive with learning enabled, but forgetting failure and resource monitoring turned off.

Expectation

As the the underlying probabilities for the domain are static in this set of experiments it is expected that the introspective executive should be able to learn these and improve its execution after each iteration, resulting in a lower failure rate than the plan.

Results

The results from these experiments allow the learning capability of the executive to

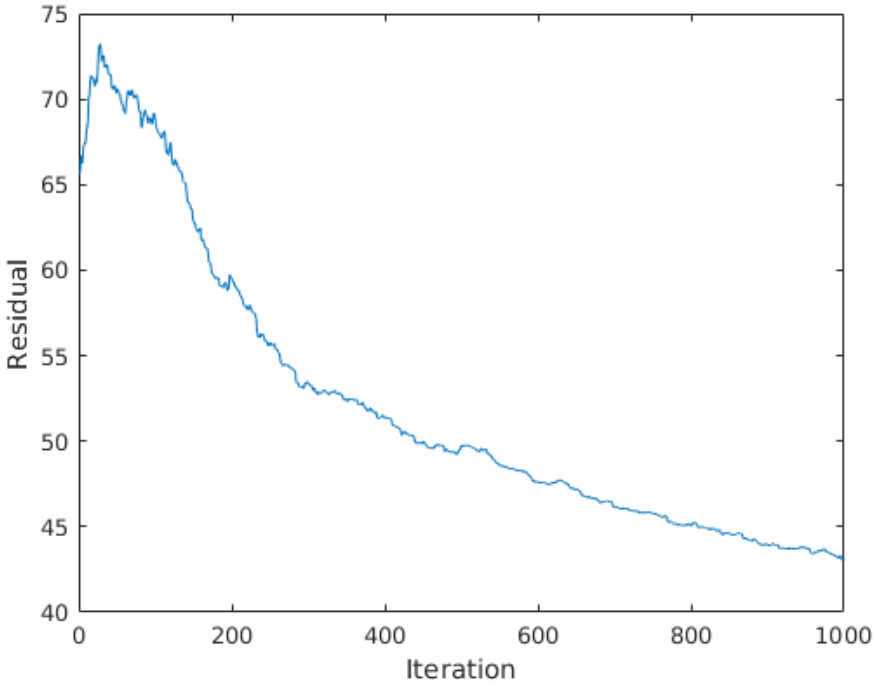


Figure 5.21: A representation of the executive's learning of its environment over time. This represents the summed absolute difference between the executive's transition probabilities and each action's underlying probabilities after each iteration.

be examined. At the start of the experiments each grounded action in the MDP has its underlying probability set, drawn from a range defined in Table 5.5. This range shown specifies the probability of an action's success, however in practice it provides a distribution over all of an action's potential outcomes. During each execution this distribution is used to determine which state the system will result in after executing an action. As the executive is designed to learn from the environment, over time it can be hypothesised that the transition probabilities associated with each action should start to reflect the underlying probabilities associated with the actions. To verify this after each iteration the absolute difference between each transition probability and underlying probability is calculated, and summed into a single figure.

The results of this calculation are shown in Figure 5.21, which plots the summed residual from the two probabilities against each iteration of the simulation. This shows that over time the difference between the transition probabilities and the underlying probabilities steadily decreases, indicating that the executive is learning how to better act in its environment. There is an initial spike at the start of the graph indicating the first set of iterations can actually reduce the executive's knowledge of its environment, how-

Simulated Plan Versus Policy		
Controller	Mean Failures	Mean Resource Consumption
Plan	3.75	129.87
Policy	2.05	119.65

Table 5.6: Results from 1000 simulated executions. This compares the executive using a plan with action recovery enabled versus the introspective executive equipped with learning.

ever that is to be expected as in the beginning any result will have a bigger impact on the residuals, which will only be magnified if is an unlikely success or failure.

The additional learning shows a noticeable affect on the overall results of the simulation, as shown in Table 5.6. This shows that the average number of failures encountered by the executive utilising the policy with learning is lower than trying to achieve the same goals using a planner with recovery actions enabled. This is due to the executive's ability to learn which are high risk actions, allowing it to adapt its policy to avoid these actions with low underlying probabilities of success. This can also be shown to reduce the average resource consumption. Examining individual executions it can be observed that as before the executive can be conservative with respect to failure when resources are unlimited, and choose routes that end up with higher corresponding resource counts to that of the planner. However overall the policy attempts to follow the plan enough that the occasional longer route can be mitigated by the reduction in failures.

A final piece of analysis which can be gained from these results is a closer look at the value iteration (VI) algorithm. For this experiment the convergence threshold ϵ is set at 0.05 and the discount factor γ is 0.9. This algorithm executes at two key points in the system, prior to the start of an execution once the plan has been provided, and

Value Iteration Analysis		
Mean Total Iterations	Mean Prior Iterations	Mean Execution Iterations
20.9	98.37	11.91

Table 5.7: Iteration analysis from 1000 executions. Mean total is the overall mean iterations from the simulations, mean prior is the average number of iterations taken before the execution begins, and mean execution is the average number of iterations during execution with the persisted value function.

during execution after each action has completed in either success or failure. Table 5.7 shows the different average iterations at these points. Prior to the execution starting the values are initialised at 0, and the VI algorithm has to calculate the value of every state, resulting in a higher iteration count. However by preserving these values and using them to initialise subsequent calls to the VI algorithm within the same execution, the iteration count is much lower. By running the VI algorithm after every action when only small changes to the transition function has occurred, coupled with initialising with informative values, allows the algorithm to converge and quickly update the policy as the robot executes.

5.5.2 Simulated Forgetting

The next set of simulated experiments are based around the executive's ability to slowly erode the learned probabilities of actions which have experienced high rates of failure and are no longer chosen for execution. This allows actions which may have been affected by environmental factors or other transient issues to be selected again by the executive, and prevent a few failures from discouraging an action's use in all future executions.

Setup

These experiments are setup similar to the previous batch, except to simulate operating within a dynamic environment every 100 iterations the underlying probabilities associated with each action are resampled from the ranges specified in Table 5.5. This means an action which may previously have had a high probability of successfully executing can suddenly begin to encounter much higher rates of failure, while conversely actions which had been associated with probable failure become much more likely to succeed.

To facilitate forgetting within the simulation the algorithm had to have its criteria for selecting which actions to decay changed from relying on the time since its last selected execution, to the number of iterations since its last execution. For this simulation an action had to have not been selected for 10 executions or more to be eligible for decay. Finally the decay rate at which the action moves back towards its initialised probability is set as above at 0.8. For these experiments another 1000 executions were simulated, and each goal generated will be attempted by the executive with a planner and recovery enabled, the introspective executive with only learning enabled, and a variant of the introspective executive with learning and forgetting of failure enabled.

Simulated Plan Versus Policy in a Dynamic Domain		
Controller	Mean Failures	Mean Resource Consumption
Plan	3.88	135.10
Policy with Learning	2.84	157.13
Policy with Forgetting	2.43	137.53

Table 5.8: Results from 1000 simulated executions with action's underlying probabilities changing every 100 executions. This compares the results from executing the plan, the executive using a policy with learning, and the executive using a policy with forgetting enabled.

Expectation

As the underlying probabilities can now change between executions, it is expected the introspective executive with forgetting should have the lowest failure rate. It's ability to learn from the environment while not enshrining past failures should allow it to more quickly adapt to changing conditions.

Results

The results from the simulation are displayed in Table 5.8. The planner's results in the dynamic environment are comparable to its results in the static variant. Intuitively this makes sense, as the planner attempts to find the shortest path to the goal using the least resources, with no concept of the underlying probabilities. As the start and end states are randomised for each execution the planner is just as likely to select actions that have increased or decreased in their probability of success.

The executive using the policy with only learning enabled performs noticeably worse in the dynamic simulation than in its static counterpart. This is due to the executive relying too heavily on its previous learned outcomes. When the underlying probabilities change this system can be slower to adapt, needing to accumulate a higher level of failure to override its existing experience, especially if alternative actions have previously experienced failures. This can cause the system to make poor choices with regards to action selection, and can result in it deviating from the plan unnecessarily. An interesting observation from these results is this technique has a lower fail count than the planner, but a higher average resource cost. This indicates that the system's reliance on out of date knowledge may not necessarily end up in failure, but result in the executive selecting actions that are suboptimal to its goal.

Finally the executive with forgetting enabled has the lowest fail count. By decaying the experience of failed actions over time, it helps to prevent the system from avoiding actions based on out of date information. This encourages the executive to try previously poor actions and to continue to learn from the environment. Which consequently results in the executive following the plan closer, as previously it would only deviate to avoid actions it has learned to associate with high risks of failure. As these failures decline the policy will rely less on its past learning and more on the planner to guide it, until it acquires new experience to inform it. This is reflected in the difference in resource count between the executive using a controller with forgetting enabled, versus the one without. This approach does however carry the negative effect of trying previously failed actions which are still problematic, which has resulted in a resource consumption which is similar to that of the planner. These attempts will not raise the failure rate dramatically, as typically it may only take one failure to verify the action is still a poor choice, but it can cause the executive to start following a sequence of actions only to have to change its strategy mid execution.

5.5.3 Simulation with Resource Monitoring

The final set of experiments incorporate resources into the simulation. In the previous experiments these have been recorded, but only to be used as an additional metric for each approach's performance. In these experiments resources act as a constraint, a level the executive can not go over if it intends to reach its goal. This will be used to evaluate on a larger scale how the executive can use the resource monitoring described in Section 4.4.5 to ensure any policies generated can operate within the allocated resource window.

Setup

The setup for this set of experiments continues to build off of the previous batch. 1000 executions were performed, with underlying probabilities changing every 100 iterations, however this time a resource limit of 250 has been set. This limit was selected as it allows the executive enough resources to reach any state in the environment from any other state, while still being restrictive enough that failures can prohibit an execution from completing. These experiments examine the performance of the planner, the executive with forgetting enabled but no resource monitoring, and the executive with forgetting and resource monitoring enabled.

Simulated Plan Versus Policy with Resource Constraints			
Controller	Mean Failures	Mean Resource Consumption	Failed Executions
Plan	3.42	131.7	94
Policy with Forgetting	2.39	130.34	100
Policy with Resource Monitor	2.45	122.53	83

Table 5.9: Results from 1000 simulated executions with a resource limit set. Failed executions represent any executions which exceed this limit.

When the executive has resource monitoring enabled, after each failure or alteration to the policy a resource simulation is executed to ensure it can still complete. If the policy's estimated consumption is above the available resource limit, a cost function is applied and the policy recalculated. This applies a cost of -1 for each action taken from the current state to the goal, and -3 for actions which consume in excess of double their allocated resource amount. The resource monitoring is provided 10 attempts to recalculate a policy that can operate within the resource bounds, before it is deemed no longer suitable for execution.

Expectation

Given a resource limit to operate within, it is expected that the executive with resource monitoring enabled should be able to significantly improve upon the number of failed executions when compared to the policy without monitoring, while still maintaining a low rate of failure.

Results

Table 5.9 shows the results from the resource monitoring simulation. Failed executions refers to any execution which exceeds the resource limit set for these experiments. During a simulation if this resource limit is reached the current execution is stopped, to allow a comparable set of results across the three approaches. Having this cut off enabled does produce a slightly lower mean failure and mean resource consumption when compared to previous experiments.

Out of the three approaches, the executive utilising a policy with learning and forgetting enabled demonstrates the weakest performance with regards to the number of failed executions. As this technique has no knowledge of resources this is understand-

able, and it will continue to prioritise selecting actions that can exploit its experience of the domain to minimise failure with little regard for overall cost. This can lead to it abandoning and recalculating new policies mid execution that involve backtracking, or selecting more expensive actions that are more likely to succeed. All of which can result in it unintentionally reaching the resource threshold, though this does however result in it having the lowest failure rate.

The approach utilising the plan fares better, but still features a high number of failed executions. In this simulation the plan will always attempt to take the least resource intensive set of actions that leads to the goal, which helps to minimise its failed executions. However its lack of knowledge about the underlying probabilities in the domain results in a higher failure rate, and it has no knowledge of the resources consumed by its recovery actions to overcome these failures. With a resource limit in place, these additional failures and recoveries can consume the available resources and result in a higher number of failed executions.

Finally, the approach which utilises the full features of the introspective executive, including resource monitoring, provides the lowest level of failed executions. By altering polices that are estimated to go above the resource threshold through the application of a cost function, it manages to mitigate some of the behaviour present in other iterations of the introspective based executive. As this cost function is only applied when the execution is estimated to breach the resource limit, this allows the executive to leverage its experience and minimise failures when it has plentiful resources, while attempting to find a better balance of failure versus action cost when it is constrained. While this approach does have the lowest number of failed executions, they are not significantly lower when compared to the planner. This could be due to the resource limit itself coupled with the underlying probabilities making some executions actually infeasible, or it could indicate that the cut off metric used in the resource simulation is too conservative.

The effect of the resource monitoring on the executive can be shown in Table 5.10, which examines the extent of its use across this experiment. These results shows that the number of executions predicted to exceed the resource threshold if unattended was similar to that of the normal policy in Table 5.9. The actual number is slightly above the corresponding non-resource monitored policy, which is likely due to the resource monitoring halting the execution before the full policy has been attempted. If left to execute it is probable some of these policies would successfully complete. However

Resource Monitoring Analysis of Executions		
Estimated Over Threshold	Adapted to Allow Continuation	Completed Successfully
111	54	28

Table 5.10: Analysis of the effectiveness of resource monitoring across 1000 simulated executions

of those estimated to fail, the application of the resource monitoring and cost function manages to allow nearly half to derive a new policy that allows them to continue executing within the resource constraints. Of these executions that continued, over half again then managed to complete their goal successfully.

Overall these results demonstrate how the resource monitoring within the executive can be beneficial to the system. In these experiments it allows it to complete the highest number of executions within a simulated dynamic environment, while still maintaining a low failure rate.

5.6 Discussion

The main focus of this chapter was to demonstrate that by expanding the executive's intelligence and reasoning through introspective capabilities, the system can more robustly execute symbolic task plans in dynamic environments. To achieve this the introspective action execution and the executive controller were implemented on a physical robot, and tasked with executing a variety of deterministic plans in a busy, indoor office environment.

The results presented in the Section 5.3 of this chapter demonstrate the introspective action execution and how it can be used to support task plans. By associating behavioural models of compound tasks with plan actions, then permitting the executive to select different variants of the action based on its environment, it can preserve the level of abstraction the planner is required to operate at and reduces the need for additional domain complexity.

The results also show how using introspective execution control can allow a greater level of success when executing plan actions. The abstract nature of the planner means that failures can occur during an execution that it cannot reason about without significant change to its domain model. Even if the executive has the ability to move the

system back to its previous state to facilitate a replan, it still requires abandoning the current action, and possibly the entire plan. The introspective execution can detect when an action starts to veer off the normal trajectory and attempts to recover from the problem before it becomes serious. The system can handle multiple failures within a single action, and tackle the problem at the execution level such that the rest of the system need not know anything has gone wrong.

This last part is both a strength and weakness of the system. While being able to recover from failure without passing it up through the system is a benefit, it does also mean that an action can execute for longer or consume more resources than is intended. A possible solution to this is to use a planner which can allocate flexible windows of time or resources to an action, and monitor if they execute within these. This approach once again pushes the reasoning back to the planner. Similarly the actions lack a formal means of control, requiring direct association with a recovery model. This means if an issues arises the selected model cannot solve, the system has to revert back to initiating the replanner.

To combat both these issues the second part of this chapter evaluates the executive controller, which enhances the executive by formalising a control structure over the introspective models that can reason about their execution and learn from their outcomes. The initial results here show the system's ability to select the appropriate action to recover an execution based on the context of the failure. This extends the utility of the introspective execution control by allowing a greater selection of failures to be recovered, and therefore a greater number of plan actions to be successfully completed.

The system's learning capability and how it can be used to improve the robots execution is evaluated both onboard a physical robot in Section 5.4.2, and through the use of a simulation in Sections 5.5.1 and 5.5.2. By observing the outcome of each action as it executes, the executive can learn a realistic estimate of its performance in the real world. These estimates can be used when a plan is passed to the executive for execution to provide an additional layer of reasoning which can identify any actions likely to fail and provide an alternative solution. This is shown to reduce the number of failures encountered during an execution in comparison to executing the same task with the plan alone. Furthermore these estimates can be learned during the execution, and used to provide a means of continual online reasoning. This allows the executive additional deliberative capabilities to determine if the current policy is still executable in light of recent failures, and if not adapt the execution strategy online to accommo-

date this. This extends the introspective capabilities of the system from being able to reason about actions, to being able to reason about plans.

One downside of this system is that given a sufficient supply of resources the executive will take the conservative path to the goal to minimise the number of encountered failures at the expense of execution time. As the executive lacks information about the numeric effect of actions, if the resources are available the policy will direct the robot to take several actions which can be much longer in duration than the initial plan steps to avoid a few recoverable failures. On the domain presented here this may not be an issue, however on a more complex domain with large numbers of actions, the difference in execution time may become a problem.

The argument of execution time versus number of failures is dependant on the specific robot and the domain it is operating in, as well as how much impact a failure has on the system's goals. In this work it is possible to alter the how conservative the system is based on its reward function. High plan step rewards will increase the policy's desire to align with the plan regardless of possible failures, while higher failure costs will result in the system becoming more conservative. The use of the resource validator over integrating resources into the MDP as action costs was also in part due to this consideration. By providing the executive with additional metrics such as action durations and a maximum execution time, the validator can include them in its simulations to alter the execution if a policy is predicted to take too long.

The system's ability to operate within resource limits is examined in Section 5.4.3 onboard a robot, and in Section 5.5.3 through a simulation. Here it is shown that after the creation of a policy, the executive can use simulation techniques to determine if its execution feasible, and attempt to update it if it is not. This is shown to work offline before an execution begins, and online to ensure any changes to the policy or execution in general do not invalidate the current path to the goal. This means that even though the executive cannot reason about resources directly, it can still prevent the system from making irresponsible choices and account for failed actions.

If the executive cannot find a solution that operates within the resource allocation, it can generate a plan state and pass this back to the planner to deliberate about. This allows reasoning to occur that includes the resource levels directly, and may involve the planner having to commence goal arbitration to determine what is achievable within the current constraints.

In the final test case invoking the planner proved little use, however in more complicated domains the communication between the planner and the executive could be essential. Furthermore the current level of communication between the planner and the executive is quite basic, with the executive state being converted back to a plan state through the addition of the original numeric plan propositions. As such the planner does not take into consideration any of the executive's additional knowledge, which can lead to it producing faulty plans based on out of date information. An interesting addition here would be to update the action costs with the executive's simulated variants when generating a plan state, allowing the planner to exploit the executive's experience.

Although both sections of this work have been confined to navigation actions for the evaluation, both the introspective actions and the executive controller can extend beyond this. Any compound action can be modelled using the techniques described in this thesis, and any failure they encounter can be associated with a recovery action. The recovery actions themselves are also not limited in anyway to the variants discussed, and additional actions can be added to the controller to support failures with a linear scaling of the state space. Similarly the controller can learn not just to avoid failure but to exploit redundancy within a domain. If the robot has a goal of taking a picture and is equipped with a front and back camera, the executive can learn if one of these stops working to choose the other. Likewise if there was different options for the same action, such as laser based navigation versus sonar navigation, the most successful variant for a specific environment can be learned, and applied in future executions.

CHAPTER 6

CONCLUSION

This thesis has presented a novel solution that supports the execution of high level task plans in dynamic unstructured environments. The primary focus of this work has been on increasing the intelligence of the system's executive through the application of introspective and experience based techniques. The executive leverages these introspective techniques to identify when an action from a plan, or even the entire plan itself, is no longer performing within its learned expectations, and can attempt steps to alter and correct the execution. The result is a more intelligent system which combines elements of continual online reasoning and deliberation across multiple levels of abstraction in order to achieve its goals.

6.1 Contributions

The main objective of this thesis was to investigate the application of introspective techniques at an executive level to increase a system's deliberative capabilities and provide a more intelligent platform for plan execution. To this end, the following contributions have been made.

- **Development of an introspective framework which extends the use of behavioural models from monitoring to control.**

  Prior to this work the introspective behavioural models [30] have only been used to monitor individual tasks within controlled environments. The introspective

framework presented in Chapter 3 both expands upon the monitoring capabilities of this work, and implements an element of control. From a monitoring perspective this represents the first time these models have been used to observe entire plans, seamlessly transitioning from one model to another without having to disrupt the execution. It allows multiple models to be assigned to a single action, and can distinguish between these models at run time based on the current context of the execution. This allows more accurate modelling based on environmental conditions while preserving the high level nature of plan actions. Lastly, several limitations are identified with regard to one of the anomaly detection algorithms presented in [39], and a new algorithm (GLPD) is developed to compensate for these.

The introspective execution framework also constitutes the first time these models have been used in controlling an execution. The introspective models provide the executive with an expectation of how each action should progress, and allows the system to identify anomalous behaviour before it results in a failure state. The use of recovery actions then allows the executive to intervene at this point, execute a new action to overcome or circumvent the possible failure, and allow the initial action to resume from a better state. This allows failure to be tackled within the executive, without needing to invoke higher level systems or even abandon the current action.

- **An executive level stochastic controller that can provide additional reasoning over the introspective models, and learn directly from the robot's experience.**

  The control aspect of this work is then expanded in Chapter 4, with the creation of the executive controller. This allows more effective use of the introspective models in adapting the execution through the inclusion of intermediate states within the actions. These intermediate states represent concepts of failure and recovery, and provide additional points of reasoning within the executive from which it can alter an action to better overcome an unexpected situation, or abandon it for the sake of the rest of the execution.

  The controller is based on a MDP, and a method is presented to automatically derive the states and actions of this model based on a planning domain. The transition function is initialised using the training data gathered from the introspective models, and then supported through learning. This uses the outcome

of each action the system executes to continually train the model, providing it with the ability to learn realistic estimations of each action's performance. A produced plan is then used as the basis of the reward function. This creates a system which can extend the deterministic reasoning used by the planner with the stochastic estimations learned by the executive to create a control policy that is more effective in a dynamic environment. It also provides the executive with a level of introspection over the plan, using the estimates to provide an expectation of how a plan should perform.

- **The implementation and evaluation of the above methods on a physical robot acting within dynamic environments.**

  Both the introspective execution framework and introspective controller where implemented on a physical robot and evaluated in Chapter 5. This represents the first time these introspective models have been used as part of a plan execution architecture and tested in an unstructured domain. This has shown examples of how the use of introspective techniques at an executive level can prove beneficial for more robust execution of task plans. From an action perspective the introspective execution framework has demonstrated the ability to detect failures within the environment and respond appropriately. This has allowed problems such as people crowding round the robot or obstructing its path to be overcome without needing to abandon the action or initiate a replan.

  The use of the introspective controller has also been shown to improve the execution of plans by reducing the number of failures encountered. By learning a stochastic controller and using this to provide additional reasoning over a deterministic plan, actions which are identified as having high probabilities of failure can be avoided and suitable alternatives chosen. This allows problems in the environment, such as the robot's inability to navigate through doors, to be taken into consideration and alternative actions suggested if available.

6.2 Future Work

The work presented in this thesis can be seen as initial steps towards a more intelligent, introspective plan execution architecture. With the major components defined there are many avenues of continued research and expansion that can be explored. This section

will provide a brief overview of a few of these.

- **Expanded Domain with Additional Actions**

  Through out this thesis it has been argued that the introspective action execution and introspective executive are entirely task independent, and can be applied across a varied spectrum of actions. However while there is no aspect of the models themselves or their use in recovering and learning from executions that is task dependant, they have been restricted in their application due to the limitations of the robotic platform they have been deployed on. It would be interesting to prove the utility of this system across a more diverse set of actions. Modelling a robotic arm picking up an object would allow the system to detect if it is approaching from a poor angle, and recovery actions could be used to realign the robot. Similarly a robot with a gripper could provide a "navigate with object" action, which could be monitored to detect if the object is slipping and help the robot reaffirm its grip on the package. Being able to learn within the executive which objects the robot can transport and which ones it struggles with would be useful to help refine the execution.

  Expanding the actions within a domain can also increase the functionality of the system. The addition of redundant actions, which can achieve the same goal through a different behaviour, can allow the executive to learn the most suitable action for a particular environmental context. If for example sonar navigation is used as the default action when traversing corridors, possibly to minimise resource consumption, and a particular corridor experiences regular failures then the executive can attempt a laser based navigation and observe if this improves the execution.

- **Improved Executive Control Model for Large Domains**

  An issue raised in Chapter 4, is that while the choice of a MDP is suitable for this work and the domains it has been tested on, it will scale poorly to larger state spaces. If the domain becomes more complex with the addition of several more state variables, this can cause an exponential increase in the number of states required to represent it. The current techniques require the enumeration of this state space, and the Value Iteration algorithm attempts to derive a policy for every possible state.

  One possible solution to this is to change the control model to an Stochastic

Shortest Path (SSP) MDP [8]. This is a similar control model to a standard MDP, allowing for a relatively straightforward transition, however it operates with a defined start state and end state, and associates varying costs with each action as opposed to a reward. This still requires the enumeration of the entire state space, however the addition of an initial state allows for the creation of partial policies, which only focus on the areas of the state space reachable between the initial state and the goal. Heuristic and sampling based techniques also allow for improved scaling of solutions over traditional dynamic programming techniques [56], allowing larger domains to be tackled. Algorithms such as RTDP [4] and LRTDP [9] are based on random walks from the initial state to the goal, and can provide anytime polices that would allow the executive to always select an action even if it is still attempting to derive a better solution.

A factored MDP [11] could also be an interesting possible approach. In many systems the components that compose the states are weakly connected, and it may be more appropriate to represent the environment as a set of state variables. This allows for a compact representation that avoids the need to enumerate the state space. Furthermore by representing the transition function using a Dynamic Bayesian Network (DBN), it allows actions to be described in terms of the individual state variables they effect. This can greatly reduce the learning required by a system, as a navigation between two waypoints can now be learned independently of other aspects of a state. However while a factored representation allows for larger domains to be compactly modelled it can be slow to solve, hindering its use in real time reasoning systems.

- **Further Integration Between the Planner and Executive**

  A strength of this work has been the multi faceted approach to deliberation, spanning different layers of the system. However the integration between these layers could be more comprehensive. The planner and executive share commonalities amongst their reasoning models that provides a basis for communication between the two components. This allows states that are estimated from a plan to be incorporated into the MDP's reward function, while also providing a means for the executive to generate a plan state to initiate replanning.

  This is a first step towards a tightly integrated planning and execution architecture, nevertheless it does not currently exploit this commonality to its full potential. Temporal reasoning is a core component of control architectures, and

would currently be confined to the planner. Having mechanisms in place to allow temporal information to be passed to and reasoned with at the executive could be an interesting extension of this work. A simple solution would be to use temporal estimations within the simulation of policies to ensure they can complete within an allotted boundary, however a more sophisticated solution may be appropriate. Similarly the executive itself can generate plan states to enable replanning, however these are generic states based on the original plan domain. Creating an updated state which incorporates some of the knowledge the executive has learned, such as estimated resource costs, has potential to allow more appropriate plans to be generated. Facilitating information sharing between the executive and the planner could allow a method of automatically updating the planners domain files with key information, keeping it updated with the current environments and ensuring more relevant plans are produced.

BIBLIOGRAPHY

[1] Rachid Alami, Raja Chatila, Sara Fleury, Malik Ghallab, and Félix Ingrand. An architecture for autonomy. *The International Journal of Robotics Research*, 17(4):315–337, 1998.

[2] Michael L Anderson and Donald R Perlis. Logic, self-awareness and self-improvement: The metacognitive loop and the problem of brittleness. *Journal of Logic and Computation*, 15(1):21–40, 2005.

[3] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[4] Andrew G Barto, Steven J Bradtke, and Satinder P Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

[5] Michael Beetz. Structured reactive controllers: controlling robots that perform everyday activity. In *Proceedings of the third annual conference on Autonomous Agents*, pages 228–235. ACM, 1999.

[6] Michael Beetz and Thorsten Belker. Learning robot action plans for controlling continuous, percept-driven behavior. In *Sixth European Conference on Planning*, 2014.

[7] R. Bellman. Dynamic programming. *Princeton University Press, Princeton, NJ*, 1957.

[8] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE, 1995.

[9] Blai Bonet and Hector Geffner. Labeled rtdp: Improving the convergence of real-time dynamic programming. In *ICAPS*, volume 3, pages 12–21, 2003.

[10] Abdelbaki Bouguerra, Lars Karlsson, and Alessandro Saffiotti. Semantic knowledge-based execution monitoring for mobile robots. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 3693–3698. IEEE, 2007.

[11] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1):49–107, 2000.

[12] AJ Brown, VM Catterson, M Fox, D Long, and SDJ McArthur. Learning models of plant behavior for anomaly detection and condition monitoring. In *Intelligent Systems Applications to Power Systems, 2007. ISAP 2007. International Conference on*, pages 1–6. IEEE, 2007.

[13] Jennifer Carlson, Robin R Murphy, and Andrew Nelson. Follow-up analysis of mobile robot failures. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 5, pages 4987–4994. IEEE, 2004.

[14] Steve A Chien, Russell Knight, Andre Stechert, Rob Sherwood, and Gregg Rabideau. Using iterative repair to improve the responsiveness of planning and scheduling. In *AIPS*, pages 300–307, 2000.

[15] Alex Coddington, Maria Fox, Jonathan Gough, Derek Long, and Ivan Serina. Madbot: A motivated and goal directed robot. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 20, page 1680. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

[16] Amanda Jane Coles, Andrew I Coles, Maria Fox, and Derek Long. Colin: Planning with continuous linear numeric change. *Journal of Artificial Intelligence Research*, pages 1–96, 2012.

[17] Michael T Cox, Tim Oates, and Donald Perlis. Toward an integrated metacognitive architecture. In *AAAI Fall Symposium: Advances in Cognitive Systems*, 2011.

[18] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational intelligence*, 5(2):142–150, 1989.

[19] Richard Dearden, Nicolas Meuleau, Sailesh Ramakrishnan, David E Smith, and Rich Washington. Incremental contingency planning. 2003.

[20] A.P. Dempster, N.M. Laird, D.B. Rubin, et al. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[21] Mark d'Inverno, Michael Luck, Michael Georgeff, David Kinny, and Michael Wooldridge. The dmars architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):5–53, 2004.

[22] Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and Jonas Kvarnström. Tal: Temporal action logics language specification and tutorial. 1998.

[23] Patrick Doherty, Jonas Kvarnström, and Fredrik Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, 2009.

[24] Richard E Fikes. Monitored execution of robot plans produced by strips. Technical report, DTIC Document, 1971.

[25] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[26] Shai Fine, Yoram Singer, and Naftali Tishby. The hierarchical hidden markov model: Analysis and applications. *Machine learning*, 32(1):41–62, 1998.

[27] R James Firby. An investigation into reactive planning in complex domains. In *AAAI*, volume 87, pages 202–206, 1987.

[28] Robert James Firby. *Adaptive execution in complex dynamic worlds*. PhD thesis, Citeseer, 1990.

[29] GD Forney Jr. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[30] M. Fox, M. Ghallab, G. Infantes, and D. Long. Robot introspection through learned hidden markov models. *Artificial Intelligence*, 170(2):59–113, 2006.

[31] Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan stability: Replanning versus plan repair. In *ICAPS*, volume 6, pages 212–221, 2006.

[32] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res.(JAIR)*, 20:61–124, 2003.

[33] Gordon Fraser, Gerald Steinbauer, and Franz Wotawa. Plan execution in dynamic environments. In *Innovations in Applied Artificial Intelligence*, pages 208–217. Springer, 2005.

[34] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *AAAi*, volume 1992, pages 809–815, 1992.

[35] Alfonso Gerevini and Ivan Serina. Lpg: A planner based on local search for planning graphs with action costs. In *AIPS*, volume 2, pages 281–290, 2002.

[36] Malik Ghallab and Hervé Laruelle. Representation and control in ixtet, a temporal planner. In *AIPS*, volume 1994, pages 61–67, 1994.

[37] Malik Ghallab, Dana Nau, and Paolo Traverso. The actor's view of automated planning and acting: A position paper. *Artificial Intelligence*, 208:1–17, 2014.

[38] Malik Ghallab, Dana Nau, and Paolo Traverso. The actors view of automated planning and acting: A position paper. *Artificial Intelligence*, 208, 2014.

[39] J. Gough. *Opportunistic plan execution monitoring and control*. Thesis, University of Strathclyde, 2007.

[40] Karen Zita Haigh and Manuela M Veloso. Planning, execution and learning in a robotic agent. In *AIPS*, pages 120–127, 1998.

[41] Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of markov decision processes using macro-actions. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 220–229. Morgan Kaufmann Publishers Inc., 1998.

[42] Nick Hawes, Christopher Burbridge, Ferdian Jovan, Lars Kunze, Bruno Lacerda, Lenka Mudrová, Jay Young, Jeremy Wyatt, Denise Hebesberger, Tobias Kortner, et al. The strands project: Long-term autonomy in everyday environments. *IEEE Robotics & Automation Magazine*, 24(3):146–156, 2017.

[43] Till Hofmann, Tim Niemueller, Jens Claßen, and Gerhard Lakemeyer. Continual planning in golog. In *AAAI*, pages 3346–3353, 2016.

[44] R. A. Howard. Dynamic programming and markov processes. *MIT press.*, 1960.

[45] Guillaume Infantes, Malik Ghallab, and Félix Ingrand. Learning the behavior model of a robot. *Autonomous Robots*, 30(2):157–177, 2011.

[46] Félix Ingrand and Malik Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 2014.

[47] Francois F Ingrand, Michael P Georgeff, and Anand S Rao. An architecture for real-time reasoning and system control. *IEEE expert*, 7(6):34–44, 1992.

[48] Francois Fe'lix Ingrand, Raja Chatila, Rachid Alami, and Frédéric Robert. Prs: A high level supervision and control language for autonomous mobile robots. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 43–49. IEEE, 1996.

[49] Luca Iocchi, Laurent Jeanpierre, Maria Teresa Lazaro, and Abdel-Illah Mouaddib. A practical framework for robust decision-theoretic planning and execution for service robots. In *ICAPS*, pages 486–494, 2016.

[50] Sergio Jiménez, Fernando Fernández, and Daniel Borrajo. Integrating planning, execution, and learning to improve plan execution. *Computational Intelligence*, 29(1):1–36, 2013.

[51] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1):99–134, 1998.

[52] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, pages 237–285, 1996.

[53] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[54] Jana Koehler. Planning under resource constraints. In *ECAI*, volume 98, 1998.

[55] T. Kohonen. Self-organisation and associative memory. In *Springer Verlag*, 1984.

[56] Andrey Kolobov. Planning with markov decision processes: An ai perspective. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–210, 2012.

[57] Evan A Krause, Paul W Schermerhorn, and Matthias Scheutz. Crossing boundaries: Multi-level introspection in a complex robotic architecture for automatic performance improvements. In *AAAI*, 2012.

[58] Bruno Lacerda, David Parker, and Nick Hawes. Optimal and dynamic planning for markov decision processes with co-safe ltl specifications. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1511–1516. IEEE, 2014.

[59] Bruno Lacerda, David Parker, and Nick Hawes. Optimal policy generation for partially satisfiable co-safe ltl specifications. In *IJCAI*, pages 1587–1593, 2015.

[60] Solange Lemai and Félix Ingrand. Interleaving temporal planning and execution in robotics domains. In *AAAI*, volume 4, pages 617–622, 2004.

[61] Hector J Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B Scherl. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1-3):59–83, 1997.

[62] Maxim Likhachev, Sebastian Thrun, and Geoffrey J Gordon. Planning for markov decision processes with sparse stochasticity. In *Advances in neural information processing systems*, pages 785–792, 2004.

[63] John McCarthy. Making robots conscious of their mental states. In *Machine Intelligence 15*, pages 3–17, 1995.

[64] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen. T-rex: A model-based architecture for auv control. In *Proceedings of ICAPS Workshop*, 2007.

[65] Michael Montemerlo, Joelle Pineau, Nicholas Roy, Sebastian Thrun, and Vandi Verma. Experiences with a mobile robotic guide for the elderly. In *AAAI/IAAI*, pages 587–592, 2002.

[66] Benoit Morisset and Malik Ghallab. Learning how to combine sensory-motor functions into a robust behavior. *Artificial Intelligence*, 172(4):392–412, 2008.

[67] Aaron Christopher Morris. *Robotic Introspection for Exploration and Mapping of Subterranean Environments*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, December 2007.

[68] Abdel-Illah Mouaddib, Shlomo Zilberstein, and Victor Danilchenko. New directions in modeling and control of progressive processing. In *ECAI*, volume 98, 1998.

[69] Nicola Muscettola, P Pandurang Nayak, Barney Pell, and Brian C Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1):5–47, 1998.

[70] Karen L Myers. A procedural knowledge approach to task-level control. In *AIPS*, pages 158–165, 1996.

[71] Bernhard Nebel and Jana Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, 76(1):427–454, 1995.

[72] R Peter Bonasso, R James Firby, Erann Gat, David Kortenkamp, David P Miller, and Mark G Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3):237–256, 1997.

[73] Gilbert L Peterson and Diane J Cook. Incorporating decision-theoretic planning in a robot architecture. *Robotics and Autonomous Systems*, 42(2):89–106, 2003.

[74] Ronald PA Petrick and Andre Gaschler. Extending knowledge-level contingent planning for robot task planning. In *ICAPS 2014 Workshop on Planning and Robotics (PlanRob)*, 2014.

[75] Ola Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.

[76] Ola Pettersson, Lars Karlsson, and Alessandro Saffiotti. Model-free execution monitoring in behavior-based robotics. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 37(4):890–901, 2007.

[77] M. Puterman. Markov decision processes—discrete stochastic dynamic programming. *Wiley, New York*, 1994.

[78] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[79] L.R. Rabiner et al. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[80] Kanna Rajan, Frederic Py, Conor McGann, John Ryan, Tom O'Reilly, Thom Maughan, and Brent Roman. Onboard adaptive control of auvs using automated planning and execution. In *15th International Symposium on Unmanned Untethered Submersible Technology*, 2009.

[81] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 3, pages 1931–1937. IEEE, 1998.

[82] Gerald Steinbauer. A survey about faults of robots used in robocup. In *RoboCup 2012: Robot Soccer World Cup XVI*, pages 344–355. Springer, 2013.

[83] Florent Teichteil-Koenigsbuch, Guillaume Infantes, and Ugur Kuter. Rff: A robust, ff-based mdp planning algorithm for generating policies with low probability of failure. *Sixth International Planning Competition at ICAPS*, 8, 2008.

[84] Florent Teichteil-Königsbuch, Charles Lesire, and Guillaume Infantes. A generic framework for anytime execution-driven planning in robotics. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 299–304. IEEE, 2011.

[85] Sebastian Thrun, Maren Bennewitz, Wolfram Burgard, Armin B Cremers, Frank Dellaert, Dieter Fox, Dirk Hähnel, Charles Rosenberg, Nicholas Roy, Jamieson Schulte, et al. Minerva: A second-generation museum tour-guide robot. In *Robotics and automation, 1999. Proceedings. 1999 IEEE international conference on*, volume 3. IEEE, 1999.

[86] Andrey Kolobov Mausam Daniel S Weld. Classical planning in mdp heuristics: With a little help from generalization. 2010.

[87] Brian C. Williams, P. Pandurang Nayak, and Urang Nayak. A model-based approach to reactive self-configuring systems. In *In Proceedings of AAAI-96*, pages 971–978, 1996.

[88] Håkan LS Younes and Michael L Littman. Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. In *In Proceedings of the 14th International Conference on Automated Planning and Scheduling*, 2004.

[89] Nevin Lianwen Zhang, Runping Qi, and David Poole. A computational theory of decision networks. *International Journal of Approximate Reasoning*, 11(2):83–158, 1994.

[90] Shlomo Zilberstein, Richard Washington, Daniel S Bernstein, and Abdel-Illah Mouaddib. Decision-theoretic control of planetary rovers. In *Advances in Plan-Based Control of Robotic Agents*, pages 270–289. Springer, 2002.

[91] Vittorio A Ziparo, Luca Iocchi, Pedro U Lima, Daniele Nardi, and Pier Francesco Palamara. Petri net plans. *Autonomous Agents and Multi-Agent Systems*, 23(3):344–383, 2011.