

ADAPTABLE
BELIEFS-DESIRES-INTENTIONS
REASONING

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2025

Peter Stringer

Department of Computer Science

Contents

Abstract	8
Declaration	10
Copyright	11
Acknowledgements	12
Glossary	13
1 Introduction	15
1.1 Motivation	15
1.1.1 Research Questions and Objectives	17
1.2 Thesis Structure	19
1.3 Contributions	23
2 Background	25
2.1 Agents and the Beliefs-Desires-Intentions paradigm	25
2.1.1 Agents	25
2.1.2 BDI Agents	26
2.1.3 Programming BDI Agents	28
2.1.4 Practical Applications	29
2.1.5 Limitations of BDI	30
2.1.6 Actions in BDI Programs	31
2.2 The GWENDOLEN Programming Language	33
2.2.1 Operational Semantics for GWENDOLEN	34
2.3 Automated Planning	41
2.3.1 STRIPS and PDDL	41
2.3.2 Action learning in AI planning	42

3	Related Work	43
3.1	BDI Action Theory	43
3.2	Actions with Durations	44
3.3	Failure Handling	47
3.4	Learning Action Descriptions	49
4	A Framework for Adaptive Cognitive Agents	51
4.1	The Reasoning Cycle for Cognitive Agents	51
4.2	Introducing a Framework for Adaptive Cognitive Agents	52
5	Durative Actions	57
5.1	Terminating Conditions	57
5.2	Extending Action Theory	59
5.2.1	Managing actions with terminating conditions	59
5.3	Example	64
5.4	Implementing Durative Actions	68
6	Detecting Failures and Learning New Action Descriptions	71
6.1	Failure Detection	71
6.1.1	Failure Detection Method	72
6.2	Stage D_1 : Learning New Action Descriptions	75
6.2.1	Algorithm Selection	76
6.3	Machine Learning with GWENDOLEN	81
6.3.1	Agent learning process	82
6.3.2	Plan Library Reconfiguration with Learned Action Descriptions	85
7	Evaluation	88
7.1	Methodology	88
7.1.1	Simple Testing	89
7.1.2	Experimental Evaluation	89
7.1.3	Evaluation Criteria	91
7.2	Experiment Domains	92
7.2.1	Simulated Environments	92
7.2.2	Generation of Experiment Files	98
7.3	Experiment Scenarios	99
7.3.1	Waypoints Environment	99
7.3.2	Hallway	103

7.3.3	Reconfiguration Timing and Memory Cost	105
7.4	Experiment selection	107
7.5	Results	108
7.5.1	Waypoints	108
7.5.2	Hallway	114
7.5.3	Reconfiguration Timing	116
7.5.4	Reconfiguration Cost	118
7.5.5	Criteria Satisfaction	120
8	Conclusions	123
8.1	Contributions, Research Questions, and Objectives	123
8.1.1	Chapter 5: Durative Actions	123
8.1.2	Chapter 6: Detecting Failures and Learning New Action Descriptions	124
8.1.3	Chapter 7: Evaluation	126
8.1.4	Challenges	127
8.2	Future Work	128
8.2.1	Limitations	132
8.2.2	Weaknesses	134
8.3	Conclusion	136
	Bibliography	137
A	GWENDOLEN Semantics	151
A.1	Intentions	151
A.2	Plans, Applicable Plans and Intentions	152
A.2.1	Applicable Plans	153
A.3	The Environment	154
A.4	Multi-Agent System Semantics, Scheduling, Reasoning Cycle	155
A.5	Stage Rules: The Agent Reasoning Cycle	157
A.5.1	Stage A	159
A.5.2	Stage B	160
A.5.3	Stage C	161
A.5.4	Stage D	162
A.5.5	Stage E	167
A.5.6	Stage F	168

List of Tables

2.1	GWENDOLEN Notation conventions	39
5.1	Notation conventions for Action Theory Extensions	60
6.1	Comparison of Learning Algorithms	77
7.1	Table of simple test scenarios for agile development.	90
7.2	Table of experiment relation to evaluation criteria	107
7.3	Summary of Action Failure Rates for 4 Node Waypoint Environments	109
7.4	Summary of Action Failure Rates for 9 Node Waypoint Environments	111
7.5	Summary of Action Failure Rates for Hallway Environments	114
7.6	Replanning Timing Costs for 4-node Waypoint Environments	117
7.7	Replanning Timing Costs for 9-node Waypoints Environments	118
7.8	Memory Cost for Replanning 4 Node Waypoint Plans	119
7.9	Memory Cost for Replanning 9 Node Waypoint Plans	119
A.1	Methods implemented by GWENDOLEN Environments	155
A.2	Notations for deed type checks	158
A.3	Operations on Intentions	159

List of Figures

2.1	Example GWENDOLEN program for a route navigation task	36
2.2	GWENDOLEN Reasoning Cycle	37
4.1	Generic Sense-Reason-Act cycle	51
4.2	Extended Sense-Reason-Act cycle to account for action deprecation, synthesis of new action descriptions, and the patching of plans.	52
4.3	GWENDOLEN reasoning cycle with additional stages showing the pro- posed extension.	53
5.1	Action log for a GWENDOLEN agent	58
5.2	Action log for an inspection robot in a hallway environment with suc- cessful actions	67
5.3	Action log for an inspection robot in a hallway environment with a failed action	67
5.4	Action log for an inspection robot in a hallway environment with an aborted action	68
5.5	Durative Action - Partial Class Diagram	69
5.6	Handle Action Operational Rule	70
6.1	Detecting Failures in an Action Log	74
6.2	Example of an action log with variable post-conditions for the same action (<i>move(0,1)</i>)	84
6.3	Post-conditions extracted from Figure 6.2, added with their respective weights which are calculated based on how recent they are.	85
6.4	Four GWENDOLEN plans for a patrolling robot	86
7.1	4-Node and 9-Node Waypoints Environments. Dotted lines represent the actions that can be performed to traverse between waypoints.	93
7.2	Available capabilities in the 4-node waypoints environment	93

7.3	Available plans in the 4-node waypoints environment	94
7.4	Hallway Environment	95
7.5	Available capabilities in the hallway environment	95
7.6	Available plans in the hallway environment	96
7.7	Time and Computational Costs Experimentation Process	97
7.8	4 Node: Route Patrol Example	99
7.9	4 Node: Route Patrol with Failure	100
7.10	9 Node: Route Patrol Example	102
7.11	9 Node: Route Patrol with Failure	103
7.12	Hallway Patrol Example	103
7.13	Hallway Patrol Action Failure	104
7.14	Action Failure Rate (%) against Success Rate in 4 Node Waypoint En- vironments	110
7.15	Action Failure Rate (%) against Average Action Descriptions Learned in 4 Node Waypoint Environments	110
7.16	Action Failure Rate (%) against Success Rate in 9 Node Waypoint En- vironments	113
7.17	Action Failure Rate (%) against Average Action Descriptions Learned in 9 Node Waypoint Environments	113
7.18	Action Failure Rate (%) against Success Rate in Hallway Environments	115
7.19	Action Failure Rate (%) against Average Action Descriptions Learned in Hallway Environments	116

Abstract

ADAPTABLE BELIEFS-DESIRES-INTENTIONS REASONING

Peter Stringer

A thesis submitted to The University of Manchester
for the degree of Doctor of Philosophy, 2025

Deploying an autonomous robot into a dynamic environment can lead to actions becoming unreliable over time, producing unexpected outcomes that were unforeseeable before runtime. Most agent programming languages, commonly used for high-level control of autonomous robots, do not support the adaptation of agent programs at runtime to deal with changes in an environment. In addition, modifying an agent program at runtime can be dangerous due to the unpredictable consequences.

These limitations present significant challenges for autonomous systems, as environmental changes can render pre-programmed behaviours ineffective or even counterproductive. As a result, an agent-controlled autonomous robot may be left without a viable plan and forced to abort the mission.

This thesis aims to provide a comprehensive framework for creating more adaptive and resilient autonomous agents, capable of operating effectively in dynamic and unpredictable environments. In pursuit of this aim, this thesis makes four main contributions. A formal semantics for actions with explicit durations, pre-conditions, post-conditions, and terminating conditions; a method for detecting persistent action failures using a recorded history of action execution outcomes; a method for learning new action descriptions; and a cohesive implementation that combines these three contributions with a framework for reconfiguring agent plan libraries.

Together, these extensions to agent programming languages enable agents to monitor actions during missions, detect when actions are no longer performing as expected,

learn updated action descriptions from data gathered during action executions, and then use these new descriptions to repair plans that contained failing actions.

An implementation of the extensions was written for the GWENDOLEN programming language and combined with an existing framework for reconfiguring agent plan libraries. The integration of these components creates a comprehensive framework that can adapt to unexpected action failures at runtime whilst prioritising safety. The implementation was then validated with an evaluation in simulated dynamic environments, demonstrating a 55% average increase in agents completing their assigned missions after experiencing failures on every available action.

The extension of agent programming languages to enable graceful degradation and long-term autonomy in dynamic environments is an important foundation for the future development of reliable and resilient robotic systems that can operate entirely without human intervention.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

- i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.
- ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.
- iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.
- iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see <http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420>), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see <http://www.library.manchester.ac.uk/about/regulations/>) and in The University’s policy on presentation of Theses

Acknowledgements

I would like to express my gratitude to my supervisory team, Dr. Louise Dennis, Prof. Clare Dixon, and Dr. Rafael Cardoso, for their guidance, expertise, and feedback. Their support and mentorship has been indispensable throughout my development as a researcher.

I am truly grateful to my wife, Amelia, for her endless emotional support, patience, and understanding during my research. Her encouragement and belief in my abilities kept me going when the results let me down.

Last but not least, I would like to thank my family for their unwavering faith in me and for providing the emotional support required to help me to achieve everything I wanted from my PhD journey.

Glossary

3APL An Abstract Agent Programming Language. 29

AI Artificial intelligence (AI) leverages computers and machines to mimic the problem-solving and decision-making capabilities of the human mind.. 26–28, 41, 42, 48, 75

AJPF Agent Java Pathfinder (AJPF) model checker (L. A. Dennis et al. 2012) extends the Java Pathfinder (JPF) model checker to prove Linear Temporal Logic properties of BDI programs. 33, 131

BDI A software model developed for programming intelligent agents based on human practical reasoning, firstly theorised and named by Michael Bratman. 15–19, 21–34, 38, 41–45, 48, 49, 51, 52, 65, 85, 123–128, 130, 132, 136

GOAL Goal-oriented Agent Language. 29

HTN The Hierarchical Task Network (HTN) paradigm is an approach to automated planning that takes advantage of domain knowledge to reduce the search space when developing a solution to a planning problem.. 27

MCAPL Model-Checking Agent Programming Languages. 68, 131

NLP Natural Language Processing (NLP) is an area of research and application that explores how computers can be used to understand and manipulate natural language text or speech to do useful things.. 27

PDDL Planning Domain Definition Language is a formalism for describing planning problems and domains, developed initially by Drew McDermott in 1998. 41, 42, 46, 127

PRS A procedural reasoning system (PRS) is a framework for constructing real-time reasoning systems that can perform complex tasks in dynamic environments. It is based on the notion of a rational agent or intelligent agent using the belief–desire–intention software model. 28, 33

QLAP Qualitative Learner of Action and Perception. 42

Seeing To It That STIT logic is a formalism for modelling the semantics of agent interactions and choices in the context of branching time.. 46

STRIPS Stanford Research Institute Problem Solver. 41, 42, 56

UAV Unmanned Aerial Vehicle. 30

Chapter 1

Introduction

This thesis aims to provide a comprehensive framework for creating more adaptive and resilient autonomous agents, capable of operating effectively in dynamic and unpredictable environments. In this chapter, the research motivation, questions and objectives are introduced, the structure of the thesis is outlined, and the contributions made as a result of this research are detailed.

1.1 Motivation

For robotic systems to pursue long-term autonomy in unpredictable environments, they must be able to adapt to unforeseen changes and continue functioning effectively. Unforeseen changes in the environment can lead to actions becoming unreliable over time, producing unexpected outcomes that can cause mission failure. This poses a significant challenge to the operation and deployment of autonomous robots.

Most agent programming languages, commonly used for the high-level control of autonomous robots, do not support the adaptation of agent programs at runtime, which can be problematic when deploying autonomous robots in dynamic environments. On top of this, modifying an agent program during operation carries significant risks, as the consequences of such modifications can be unpredictable, potentially leading to mission failure. The lack of adaptability in existing agent programming languages presents a significant obstacle to the realisation of long-term autonomy in robotic systems.

Beliefs-Desires-Intentions (BDI) agents offer a promising paradigm for achieving adaptability in robotic systems. BDI agents are built upon the principles of practical reasoning (Rao & Georgeff 1991), modelled after human cognitive processes (Bratman

1987). These agents possess beliefs about their environment, desires (goals) that they strive to achieve, and intentions (plans) that guide their actions to fulfil those goals. This framework makes BDI agents well-suited to navigate dynamic and unpredictable environments.

There are several key advantages of the BDI model that contribute to the development of adaptable agents. BDI agents maintain a clear separation between their knowledge about the world (beliefs) and the procedures for acting upon that knowledge (plans) (Rao & Georgeff 1991). This declarative representation allows for easier modification and updating of an agent's understanding of the environment as changes occur. An agent's beliefs encapsulate its understanding of the environment, which can be readily updated in response to perceived changes (Wooldridge 1999). This flexibility is important in dynamic environments where unforeseen events may invalidate prior knowledge. BDI agents exhibit goal-directed behaviour, as they constantly work towards achieving their desires. This behaviour is determined by a dynamic selection of plans based on the agent's current beliefs and goals (Wooldridge 2001). This flexibility allows BDI agents to readily adapt to changes in their environment or goals, making them suitable for pursuing long-term autonomy in unpredictable settings. BDI agents can also reason about their actions and plans. This reasoning ability is based on the agent's beliefs about the world and its understanding of the effects of its actions. The capacity for reasoning could enable agents to handle unexpected situations. For instance, if an action deviated from its anticipated outcome, the agent could reassess the situation, considering alternative plans to reach its goals.

While BDI agents are a good starting point for developing adaptable autonomous systems, there are several limitations of BDI languages which need to be addressed to enable the development of mechanisms for adaptation in dynamic environments.

Many BDI languages represent actions as atomic (Meyer et al. 2015), and assume they execute instantaneously. This simplification doesn't reflect the reality of real-world actions, which often have durations. Without explicitly accounting for action durations, agents risk becoming unresponsive to dynamic events while awaiting action completion.

While some BDI languages have more complex action representations that state the conditions required for execution, and the outcome that the action should achieve (Dastani et al. 2005, Hindriks 2009), they lack methods for detecting when actions consistently fail to achieve their intended effects. Agents that are capable of adapting to changes in their environment need to identify action failures and distinguish whether

those failures are transient or persistent.

Traditional BDI agents rely heavily on pre-programmed plans and action descriptions, limiting their flexibility in dynamic environments. The ability to adapt existing action descriptions or learn new ones as the environment changes is critical for extending an agent's operational lifetime.

There have been research efforts to address these limitations, each with strengths and shortcomings. Work by Harland et al. (2014) on goal life-cycles in BDI agents offers a theoretical basis for managing goals with changing states throughout their lifetimes. Their framework introduces the concept of suspending goals while actions are executing, however, this work lacks a specific focus on individual action monitoring, focusing primarily on the effects of action outcomes on goal status. Dennis & Fisher (2014) extend Harland et al. (2014)'s life-cycle concept to apply states and life-cycles to actions. Their extended operational semantics provide a theoretical foundation for improved action monitoring, though it does not provide mechanisms for acting on the action outcomes, such as detecting recurrently failing actions.

Research on plan failure in BDI agents predominantly centres on goal-level failures (Bordini & Hübner 2010, Sardiña & Padgham 2007). This research concentrates on mechanisms for dropping or re-planning goals when plans fail to achieve their objectives. While essential, this focus on goal-level failure overlooks action-level failures, where the specific post-conditions of an action might not be met, even if the overarching goal is achieved. This distinction is important for identifying the root cause of plan failures and enabling more targeted adaptation strategies.

The limitations in existing BDI languages highlight the need for more robust mechanisms for representing action with durations, detecting action failures, and learning new action descriptions that are accurate to their perceived effect on the environment. These limitations must be addressed to enable the development of adaptable BDI agents capable of pursuing long-term autonomy without human intervention in dynamic and unpredictable environments.

1.1.1 Research Questions and Objectives

The limitations of autonomous agents, particularly concerning their ability to adapt to dynamic environments where actions may become unreliable and produce unexpected outcomes, have motivated this research. These limitations informed four key research questions and eight corresponding research objectives. By addressing these research questions and objectives, the thesis aims to provide a comprehensive framework for

creating more adaptive and resilient BDI agents, capable of operating effectively in dynamic and unpredictable environments.

Many real-world robotic tasks involve actions that take time to complete. However, many traditional BDI languages often operate under the assumption that actions are near-instantaneous. *Research Question 1* asks how BDI languages could handle actions that have pre-conditions, post-conditions, durations, and terminating conditions. *Objective 1.a* focuses on the development of this theory, whilst *Objective 1.b* calls for an implementation to recognise the theory, to lay the foundation for further research and development into adaptable agents.

Many BDI agent programming languages do not support the adaptation of agent plans during execution. *Research Question 2* asks how extended action theory can help to enable plan reconfiguration for BDI agents. *Objective 2.a* aims to build on the extended action theory, developing the concept of an “Action Log”, to allow agents to track the performance of actions over time. *Objective 2.b* subsequently builds on the capabilities unlocked by the extended action theory, by using the Action Log to determine when actions are no longer consistently achieving their post-conditions.

Machine learning algorithms have demonstrated their ability to identify recurring patterns in datasets (Mitchell 1997). *Research Question 3* asks how machine learning could be used to update action descriptions to more accurately reflect an action’s effect on the environment. *Objective 3.a* addresses the research question directly by calling for a method to update action descriptions using the data collected in the Action Log. *Objective 3.b* aims to integrate the learned action descriptions into the agent’s planning process, using an existing framework for reconfiguring agent plan libraries, allowing agents to use the updated knowledge to continue operating after experiencing failure.

Research Question 4 seeks to evaluate the feasibility and effectiveness of the proposed extensions for BDI agent architectures. *Objective 4.a* involves designing and conducting experiments in simulated environments to assess how well the modified BDI agent performs in different domains, particularly when action failure rates are increased. *Objective 4.b* covers the analysis of the experiment results based on metrics relevant to pursuing long-term autonomy, such as success rate, robustness to action failures, and computational efficiency.

1.2 Thesis Structure

In this section, the structure of this thesis is outlined, with a description of the content of each chapter, and the first-author publications that each chapter's content informed.

Chapter 1 - Introduction

The research motivation is presented, describing the research problem that this thesis aims to solve, the current state of adaptable BDI reasoning research, and what is proposed to address the research problem. A summary of the thesis structure is then given before the novel contributions of the thesis are presented.

The high-level concepts and research outline from this chapter were published as part of the following papers:

Adaptable and Verifiable BDI Reasoning (Stringer et al. 2020)

Peter Stringer, Rafael C. Cardoso, Xiaowei Huang, Louise A. Dennis

AREA 2020, First Workshop on Agents and Robots for reliable Engineered Autonomy. Volume 319 of Electronic Proceedings in Theoretical Computer Science, pp. 117–125

Personal contributions to this publication include: high-level foundational concepts, a research outline, and an initial system architecture for BDI autonomous agents capable of adapting to changes in dynamic environments.

Adaptable and Verifiable BDI Reasoning - Extended Abstract (Stringer, Cardoso, Huang & Dennis 2021)

Peter Stringer, Rafael C. Cardoso, Xiaowei Huang, Louise A. Dennis

AAMAS 2021, 20th International Conference on Autonomous Agents and Multiagent Systems. ACM, pp. 1835–1836

This extended abstract is an adapted version of the previous publication (Stringer et al. 2020).

Chapter 2 - Background

The adaptability of autonomous agents, specifically those that follow the BDI model of agency, is discussed. The mechanisms behind actions in BDI agent programs are described, and the semantics of the GWENDOLEN agent programming language are explained. The importance and applications of BDI agents in various domains are highlighted, and the action theory of AI planning is explored, focusing on STRIPS

and PDDL languages and the integration of action learning from AI planning into BDI agents.

Chapter 3 - Related Work

This chapter examines related research in four key areas that form the foundations of the contributions in this thesis. Firstly, foundational action theories are analysed, highlighting semantics for goal and action management but identifying gaps in explicit failure handling. Next, approaches to durative actions in agent frameworks and planning systems are reviewed, though these lack mechanisms to monitor runtime failures. Then, failure recovery methods are critiqued for focusing on goal-level responses rather than action failure logging. Finally, learning techniques for action descriptions are assessed, emphasising the risks of trial-and-error exploration.

Chapter 4 - A Framework for Adaptive Cognitive Agents

The approach of the research proposed in Section 1.1 of this chapter is presented, supported by illustrations of the newly proposed extensions and their position in the existing architecture. A walk-through of the proposed system is then given, using a diagram of the new system architecture and a pseudo-code algorithm. This chapter aims to situate each of the proposed extensions together in a cohesive system architecture, to aid the explanations of the extensions in the subsequent chapters.

Chapter 5 - Durative Actions

This chapter sets the foundational action theory required by the mechanisms for failure detection and learning action descriptions. The necessity of incorporating durations into actions within BDI languages is discussed, with a focus on agents deployed in physical systems where actions may take considerable time to complete. The chapter begins by justifying the introduction of durative actions and outlining the required mechanisms for integrating durations into existing action semantics. Key concepts such as terminating conditions (success, failure, and abort) are defined and explained. The details of an extended action theory are provided alongside operational semantic rules for actions with explicit durations, pre-conditions, post-conditions and terminating conditions. An example is then used to demonstrate how the extended semantics operate in practice, including the use of an action log to track action outcomes. Finally, the chapter details the implementation of these concepts in the GWENDOLEN

programming language, extending the underlying Java code.

The extended operational semantics were published as part of the following paper:

**Implementing Durative Actions with Failure Detection in GWENDOLEN
(Stringer, Cardoso, Dixon & Dennis 2021)**

Peter Stringer, Rafael C. Cardoso, Clare Dixon, Louise A. Dennis

EMAS 2021, 9th International Workshop on Engineering Multi-Agent Systems. Volume 13190 of Lecture Notes in Computer Science, Springer, pp. 332–351

Personal contributions to this publication include: An extension of the action execution semantics in the GWENDOLEN BDI programming language. A theory for actions that have durations, with a method for detecting action failures. An implementation of the extension with a simple case study.

Chapter 6 - Detecting Failures and Learning New Action Descriptions

In this chapter, a method for detecting action failures using the *Action Log* is presented. The necessity of failure detection in dynamic environments is highlighted, emphasising its role in enabling autonomous agents to adapt to unforeseen circumstances. The mechanisms underlying failure detection are then discussed with an example scenario to illustrate its application.

The process of learning new action descriptions to replace failing ones is then explored. Several learning algorithms were evaluated for their suitability, leading to the selection of a hybrid algorithm combining supervised learning, rote learning, and ranking. The hybrid algorithm uses historical data from the *Action Log* to identify the most likely outcomes for actions and update their descriptions accordingly. Implementation details are then provided, showing how the learning algorithm is integrated into the GWENDOLEN programming language. The details of how the new action descriptions are used in conjunction with a framework for reconfiguring agent plan libraries are then presented.

The mechanisms for detecting persistent failures and learning new action descriptions were published as part of the following papers:

Updating Action Descriptions and Plans for Cognitive Agents - Extended Abstract (Stringer et al. 2023b)

Peter Stringer, Rafael C. Cardoso, Clare Dixon, Michael Fisher, Louise A. Dennis

AAMAS 2023, 22nd International Conference on Autonomous Agents and Multiagent Systems. ACM, pp. 2370–2372

Personal contributions to this publication include: an integration of the work from the previous paper, ‘Implementing Durative Actions with Failure Detection in GWEN-DOLEN, with a method for learning new action descriptions, encapsulated in a practical implementation.

Chapter 7 - Evaluation

In this chapter, a suite of simple tests is used to test the implementation of the learning algorithm proposed in Chapter 6 against a set of defined requirements, confirming core functionality like handling action failures and partial successes. Then, two simulated environments are introduced with visual representations, along with the experiment scenarios used to test the system proposed in Chapter 4 against a set of evaluation criteria. Additional experiments that compare time and memory costs were also performed using the outputs of the simulated environment experiments, to isolate the AI planner’s time and memory costs for an evaluation of the computational efficiency of single-plan patching against full re-planning of the plan library. Finally, the results of the experiments are presented and analysed against the evaluation criteria, highlighting the system’s adaptability in learning action descriptions and computational efficiency.

The waypoints environment experiments, and the implementation of the system used to perform the evaluation were published in the following paper as an extended version of the previous paper, ‘Updating Action Descriptions and Plans for Cognitive Agents’:

Adaptive Cognitive Agents: Updating Action Descriptions and Plans (Stringer et al. 2023a)

Peter Stringer, Rafael C. Cardoso, Clare Dixon, Michael Fisher, Louise A. Dennis
EUMAS 2023, 20th European Conference on Multi-Agent Systems. Volume 14282 of Lecture Notes in Computer Science, Springer, pp. 345–362

Personal contributions to this publication include: more detailed theoretical mechanisms, a practical implementation of the mechanisms, and an evaluation of the system.

Chapter 8 - Conclusions

This chapter revisits the research questions and objectives outlined in Section 1.1.1 of this chapter, and the contributions made to adaptable BDI reasoning research are detailed. The strengths, weaknesses, and limitations of the research are also discussed. Additionally, future work is suggested, focusing on improving the system’s scalability,

enhancing failure detection, and exploring real-world practical implementations. The chapter concludes by summarising the research's impact, highlighting the successful extension of agent programming languages to adapt to dynamic environments, thus increasing the operational lifetime of autonomous agents.

1.3 Contributions

Operational semantics, theoretical mechanisms and practical implementations were developed to address the research questions and objectives raised by the motivation of this thesis, forming four novel contributions to BDI agent research.

An extended formal semantics for actions with explicitly defined durations, pre-conditions, post-conditions, and terminating conditions for BDI agents. An extension of existing BDI action theory was developed to consider actions with explicit success, failure, and abort conditions, and durations, along with the addition of an *Action Log* that acts as a record of all action executions. The semantics are general to BDI languages that have actions with explicit pre-conditions and post-conditions. This extended formal semantics is presented in Chapter 5.

Failure detection for BDI agents. To effectively handle unreliable actions, this thesis introduces the Action Log, a data structure for tracking action performance. The Action Log records the outcomes of executed actions, noting whether they successfully achieve their intended post-conditions or fail to do so. The Action Log is then used to identify consistently failing actions, and can trigger mechanisms for failure recovery. The Action Log is introduced in Chapter 5 and its role in failure detection is detailed in Chapter 6.

Learning new action descriptions. A mechanism for learning updated action descriptions was implemented into an existing BDI programming language. Recognising the need for dynamic adaptation in unpredictable environments, this thesis explores the use of machine learning to update action descriptions based on the agent's action execution experiences. By leveraging the information gathered in the Action Log, the thesis proposes a machine-learning mechanism that can learn new action descriptions to replace action descriptions that are no longer accurate. The data collected in the

Action Log is used as input for a hybrid machine learning algorithm that can determine a set of post-conditions that better reflect the effect of the action on the agent's environment. The mechanism for learning updated action descriptions is presented in Chapter 6.

Integration of action failure detection and a mechanism for learning updated action descriptions with a plan repair framework. The novel contributions of this thesis, including the extended representation of actions with explicit durations and terminating conditions, the Action Log, and the machine learning mechanism for updating action descriptions, were integrated with Cardoso et al. (2019)'s plan library reconfiguration framework that can repair agent plans from existing plans. This integration provides a comprehensive framework for BDI agents that can adapt their actions and plans in response to changes in dynamic environments. The integration of action failure detection and a mechanism for learning updated action descriptions with a plan repair framework is detailed in Chapter 6, and followed by an experimental evaluation in Chapter 7.

Chapter 2

Background

In this chapter, the ability of autonomous agents to adapt to changes in their environment is discussed, specifically, agents that follow the Beliefs-Desires-Intentions (BDI) model of agency. Next, the mechanisms behind the actions in BDI agent programs are described, before detailed semantics of the GWENDOLEN agent programming language are explained.

2.1 Agents and the Beliefs-Desires-Intentions paradigm

This thesis studies the ability of agents, specifically those that follow the BDI model of agency, to adapt to changes in their environment.

2.1.1 Agents

An ‘agent’ is an abstraction developed to capture autonomous behaviour within complex, dynamic systems (Wooldridge 2002). It is defined by Russell & Norvig (2020) as something that “can be viewed as **perceiving** its environment through **sensors** and **acting** upon that environment through **effectors**”. Agents are frequently considered a suitable paradigm for decision-making in autonomous systems because of their encapsulation of the Sense-Reason-Act cycle. This cycle describes the process by which an agent observes its environment (sense), processes this information to make decisions (reason), and acts upon the environment to affect a desired change (act). The Sense-Reason-Act model is highly suited to dynamic decision-making systems where adaptive responses to real-time data are required.

The importance of agents in Computer Science and specifically Artificial Intelligence (AI) research is evident from their widespread applicability over several domains: autonomous vehicle navigation, where agents process vast amounts of sensory data to make real-time driving decisions (Kuutti et al. 2021); healthcare, particularly in robotic surgery, where precision and adaptability of agents lead to enhanced surgical outcomes (Yu et al. 2024); financial markets, where trading agents automatically execute trades based on dynamic market conditions (Bahoo et al. 2024); and in smart grids, where agents can manage the demand and supply of electricity in an efficient and sustainable manner (Omitaomu & Niu 2021).

2.1.2 BDI Agents

Cognitive agents (Bratman 1987, Rao & Georgeff 1992, Wooldridge & Rao 1999) have explicit reasons for the choices they make. These are often described in terms of the agent's *beliefs* and *goals*, which in turn determine the agent's *intentions*. This view of cognitive agents is encapsulated within the BDI model (Rao & Georgeff 1991, 1992). *Beliefs* represent the agent's (possibly incomplete, possibly incorrect) information about itself, other agents, and its environment, *desires* represent the agent's long-term goals while *intentions* represent the goals that the agent is actively pursuing (the representation of intentions often includes partially instantiated and/or executed plans and so combines the goal with its intended means).

BDI agents, with their high-level and declarative representation of concepts such as beliefs and goals, are an attractive paradigm where the control over agent behaviours and decision-making processes needs to be amenable to analysis (e.g., for assurance purposes (Dennis et al. 2012)). Accordingly, this research is focused on the use of BDI agent programming languages (Mascardi et al. 2005, Rao & Georgeff 1992) for the high-level control of autonomous robotic systems and particularly in providing support at the agent level for *long-term autonomy*. In this context, *long-term autonomy* refers to the capability of agents to operate reliably over extended periods without human intervention. Whilst 'extended periods' is an undefined length of time, one of the objectives of the research in this thesis is to maximise this undefined length of time.

The future of BDI agent architectures has been thoroughly discussed (Bordini et al. 2021, Cardoso & Ferrando 2021, De Silva et al. 2020, Logan 2018). In (Logan 2018), the limited impact of BDI research on AI and industrial applications is explored. Logan's manifesto states that a key factor behind the poor uptake is due to a lack of incentive to switch to agent programming languages from mainstream programming

languages. As a result, expanding the feature set of BDI architectures is heavily encouraged by Logan (2018), with a focus on encapsulating complex behaviours that are not easily replicated in mainstream languages.

The current state of BDI research has been assessed by De Silva et al. (2020), highlighting that the synergy of BDI languages and other areas such as Machine Learning, Hierarchical Task Network (HTN) planning, and Natural Language Processing (NLP) is an important direction for the future of BDI agent architectures. De Silva et al. (2020) specifically highlight a renewed interest in applying planning to BDI agents, driven by advances in logic-based planning algorithms and HTN planning. It is stated that the integration of AI planning algorithms with BDI languages can expand the set of plans that are available to agents to help them cope with environmental changes.

The current state-of-the-art BDI architectures are evaluated by Bordini et al. (2021). It is stated that AI capabilities, such as machine learning and automated planning, can enhance the decision-making and reasoning abilities of BDI agents. The integration of these capabilities could enable BDI agents to learn from experience, adapt to changing environments, and make more informed and intelligent decisions.

Cardoso & Ferrando (2021) conducted a survey of current and ‘veteran’ BDI languages. In the discussion, Cardoso and Ferrando also consider why BDI agents are not as broadly used as they could be. It is concluded that the lack of implementations and practical evaluations of BDI languages could be a factor, in addition to the factors presented in Logan’s agent manifesto (Logan 2018), namely the comparative ease of implementing the desired agent behaviour in a mainstream programming language instead.

There are alternative architectures for practical reasoning agents that differ from BDI agent architectures, each with their own assumptions, strengths, and limitations. Wooldridge (2001) discusses four architectures for implementing practical reasoning agents: Logic-based, Reactive, Layered, and BDI architectures.

Logic-based agents use symbolic representations and logical deduction to make decisions. This type of architecture is capable of modelling complex environments and behaviours using formal logic, though logical deduction can be computationally expensive, leading to slow decision-making which is not suited to dynamic environments. Oppositely, reactive agents directly map situations to actions rather than using an internal model of the world. This type of agent prioritises fast reaction times and simple implementation, but lacks long-term planning capabilities and has difficulty implementing goal-directed behaviours. Layered architectures combine reactive and

deliberative behaviours in multiple interacting hierarchical layers, allowing this type of agent to reason at different levels of abstraction. However, the interaction between layers needs to be managed, increasing design and implementation complexity. Also, if not managed correctly, this inter-layer interaction can introduce even more computational complexity.

While each of these architectures has its own strengths, the BDI model offers a structured approach that balances reactivity and deliberation, and has a high-level declarative representation of reasoning, which provides an ideal framework for integrating a wide range of AI capabilities, including machine learning, reasoning, and planning (Bordini et al. 2021).

2.1.3 Programming BDI Agents

There are *many* different agent programming languages and agent platforms based, at least in part, on the BDI approach (Bordini et al. 2021, Cardoso & Ferrando 2021, Logan 2018). Most BDI languages typically operate within a *reasoning cycle* that captures the agent Sense-Reason-Act cycle. Agents programmed in these languages commonly contain a set of *beliefs*, a set of *goals*, and a set of *plans*. Plans determine how an agent acts based on its beliefs and goals and form the basis for *practical reasoning* (i.e., reasoning about actions) in such agents. Many BDI plans are expressed in terms of some *guard*, which can be considered a pre-condition for the whole plan, and a *goal* which can be considered a goal state for the plan. As a result of executing a plan, the beliefs and goals of an agent may change, and actions may be executed.

The BDI model is commonly referred to in this thesis, and some BDI languages are specifically referred to when discussing individual mechanics and semantics.

- Procedural Reasoning System (PRS) (Georgeff & Lansky 1987) is the first implementation of a BDI architecture. It was presented as a system for reasoning about and performing complex tasks in dynamic environments. The first major application of PRS was for monitoring and fault detection in the NASA Space Shuttle’s reaction control system. This implementation ran on a LISP machine, using multiple PRS instances to manage over 1000 facts about the system (Georgeff & Ingrand 1989).
- AgentSpeak (Rao 1996) is a programming language for BDI agents, based on the PRS architecture. AgentSpeak(L) aimed to bridge the gap between the theoretical specification of BDI agents and practical implementations like PRS.

- An Abstract Agent Programming Language (3APL) (Hindriks et al. 1999) is also a programming language for BDI agents, and focuses on more complex reasoning and agent adaptability. 3APL combines imperative and logic programming paradigms, allowing agents to have belief or knowledge bases and use practical reasoning rules to monitor and revise their goals.
- Jason (Bordini et al. 2005) is an open-source Java-based extension of the AgentSpeak(L) language. It enhances AgentSpeak by adding new features and tools for agent development, including debugging and simulation environments.
- GWENDOLEN (Dennis & Farwer 2008) was originally developed from the Jason agent programming language, with the intention of addressing the absence of agent programming languages with the capability to formally verify agent programs.
- Goal-Oriented Agent Language (GOAL) (Hindriks 2009) is an agent programming language focused on goal-directed behaviour, where goals are declarative and only describe the state in which the agent desires to be in, rather than how to achieve them.

2.1.4 Practical Applications

The deployment of autonomous robots is increasingly becoming the best solution in scenarios where there is a possibility of danger to human life (Evertsz et al. 2014). However, autonomous robot control systems do not have the same cognitive capabilities as humans yet, despite the human-like reasoning achieved by the BDI model of agency (Adam & Gaudou 2016). Nevertheless, in some scenarios, autonomous robots are the only means for achieving a goal. For example, space exploration: where it is not yet feasible to send human operators to remote destinations due to the physical limitations of a human body.

BDI agents have been proposed to control an array of cyber-physical autonomous systems such as autonomous vehicles, spacecraft and robot arms (e.g., Mars Rover (Cardoso et al. 2020), earth-orbiting satellites (Dennis & Fisher 2023, Dennis et al. 2010) and robotic arms for nuclear waste-processing (Aitken et al. 2018)). In addition, despite the obvious need to avoid detailed explanation of operations, BDI agents have also been deployed into various military environments and have been responsible for high-stakes decision-making in world-leading commercial companies. Examples for

the applications of BDI agents include fault diagnosis in spacecraft (Ingrand et al. 1992); the OASIS aircraft management system (Rao & Georgeff 1995); simulating air mission dynamics and pilot reasoning (SWARMM) (Tidhar et al. 1995); decision support for human operators at the largest retail supplier of petroleum, Equinor (Ølmheim et al. 2010); managing UK Ministry of Defence Unmanned Aerial Vehicles (UAVs) to complete various missions (*Remote control: remotely piloted air systems - current and future UK use* 2014); and simulating enhanced behavioural realism for live fire training (Evertsz et al. 2014).

2.1.5 Limitations of BDI

Traditional BDI agents (Rao 1996) rely on the programmers to predict, at design time, all eventualities that an agent will encounter during their deployment. Agents that are designed using this approach have limited flexibility at runtime (Meneguzzi & Luck 2008). By allowing plans and actions to be adapted to better reflect the outcomes they achieve or do not achieve in their environment, agents can operate using plans and actions that are more accurate to the reality of their environment when compared to using static plans and actions.

In (Dennis & Fisher 2014), Dennis and Fisher highlight two problems with the way BDI languages typically handle actions. Firstly, it is claimed that in some languages “execution of the BDI program waits for the action to complete before processing other intentions, goals and plans” — in robotic systems, where an action such as moving between two waypoints in a map may take some time to complete can prevent the agent from monitoring the environment for other events (for instance, the need to avoid obstacles during movement). It should be noted that the assumption of BDI programs waiting for actions to complete before processing other intentions, goals and plans is only true for some BDI languages and there are notable exceptions to this assumption (Boissier et al. 2020, Howden et al. 2001). In JaCaMo (Boissier et al. 2020), which integrates Jason (Bordini et al. 2005) agents with CArTAgO (Ricci et al. 2011) environments, agents can handle asynchronous operations allowing them to remain responsive through the duration of action executions. While JACK (Howden et al. 2001) agents can handle parallel action execution and multi-threading to perform multiple tasks simultaneously.

Secondly, many languages lack principled mechanisms within their semantics for detecting whether an action has succeeded and reacting appropriately to failure if it has not. Dennis and Fisher addressed both of the highlighted problems by proposing a

generic BDI semantics inspired by the concept of *goal lifecycles* (Harland et al. 2014, 2017) to allow for actions with durations and failures, although they did not provide an implementation of this semantics (Dennis & Fisher 2014). However, Dennis & Fisher’s semantics did provide a most suitable foundation for the further extended semantics founded in this thesis, that not only support actions with durations and failures, but also integrates the concept of failure detection.

This research focuses primarily on action failures, with an emphasis on *failure detection*. Robotic autonomous systems operating in real-world environments may encounter intermittent failures of actions. Sometimes these failures will be transitory and can be ignored in terms of the long-term operation of the system, such as momentary spikes in temperature sensors or brief losses of GPS signal. In other situations failures may be persistent indicating some change in the robot or its environment which must be compensated for. For example, permanently damaged sensors, faulty actuators, or an environmental change; a Martian dust storm could completely change the terrain of a previously planned route. In Chapter 5 the semantics from (Dennis & Fisher 2014) are integrated with the concept of failure detection, and implemented in the GWENDOLEN agent programming language.

2.1.6 Actions in BDI Programs

Many BDI languages represent environmental interaction as an atomic *action* (Meyer et al. 2015). Atomic actions are assumed to execute immediately and complete in a single step from the perspective of the BDI reasoning cycle. When an action is invoked it executes some low-level code invisible at the BDI level or interacts directly with the external world. Such an action is often an atomic command, and its effect is judged via agent perception, though it may have an explicit return value (typically, ‘success’ or ‘failure’). Languages that treat interaction in this way include Jason (Bordini et al. 2021) and GWENDOLEN (Dennis 2017).

Interaction may also be modelled as *capabilities*. Languages that treat interaction in this way include GOAL (Hindriks 2009) and 3APL (Dastani et al. 2005). These have explicit *pre*-conditions and *post*-conditions such that the interaction is executed only if the pre-conditions are true and, after the interaction has concluded, the post-conditions are asserted explicitly by the language. Other effects may be subsequently observed via perception mechanisms. Non-deterministic actions, that have more than one possible outcome, can be modelled by defining more than one set of post-conditions for a single action (Yao et al. 2016), where the action outcome is

judged via agent perception. However, it should be noted that the most widely used BDI languages that model interaction as capabilities cannot handle non-determinism in action specifications.

It is possible that a capability executes no low-level code, particularly when an agent is executing in some simulated setting where it is considered sufficient to use just the post-conditions to represent the result of the interaction. In this thesis, atomic actions are represented in the format:

$$action(terms) \quad (2.1)$$

Capabilities are represented in the format:

$$\{pre - conditions\}action(terms)\{post - conditions\} \quad (2.2)$$

Durative actions are represented as:

$$\{pre - conditions\}action(terms)\{post - conditions\}[duration] \quad (2.3)$$

Action representations like *capabilities* and *durative actions* contain mechanisms like pre-conditions and post-conditions which can aid the detection of action failure, however, most BDI language semantics do not support these mechanisms yet. Vikhorev et al. (2011)'s AgentSpeak(RT) supports the use of *deadlines* which specify the time by which the agent should respond to an event, though the temporal constraint applies specifically to intentions rather than to individual actions.

BDI languages with consideration for failures also already exist, such as Sardina & Padgham (2011)'s CANPlan, and Bordini & Hübner (2010)'s Jason, though these languages do not distinguish goal failure from action/capability failure, whereby the postconditions stated in the action description are not believed to have been achieved after execution.

In (Dennis & Fisher 2014), Dennis and Fisher propose a generic framework for integrating durative actions (represented in Expression 2.3) with failure modes into BDI programming languages. This proposal builds on work on Goal life-cycles for BDI languages (Harland et al. 2014) that have *active*, *suspend*, and *abort* stages. Dennis & Fisher (2014) propose that actions be associated with *success*, *failure*, and *abort* conditions; the abort condition is intended to handle situations where the action execution

is deemed to have continued for “too long” without either success or failure being detected. When an action is executed as part of a plan, the corresponding goal for which the plan was formulated is temporarily suspended until the action is completed. This suspended state allows the processing of other goals to occur. If the action’s success condition is met then the goal is returned to the *active* state and the plan continues processing, otherwise the goal is returned to a *pending* state for re-planning.

2.2 The GWENDOLEN Programming Language

The GWENDOLEN programming language (Dennis & Farwer 2008) is used for the implementation of this research for two main reasons. Firstly, GWENDOLEN provides a high-level agent reasoner in a number of autonomous robotic applications. Secondly, GWENDOLEN’s link to the Agent JavaPathFinder (AJPF) model-checker (Dennis 2018) enables the possibility of incorporating formal verification processes in future research.

The GWENDOLEN programming language was originally developed from the Jason programming language. Jason is an open-source Java-based interpreter for an extended version of AgentSpeak, and as such reflects the semantics of AgentSpeak. As AgentSpeak is based on the PRS architecture (the first implementation of a BDI language), much of its semantics are also seen in other BDI languages. Consequently, the theories discussed in this thesis are deliberately kept general to as many BDI concepts as possible to maximise their applicability across various BDI language semantics. Furthermore, the specific implementation in GWENDOLEN is designed to be largely translatable to other BDI languages.

The development of GWENDOLEN addressed the absence of agent programming languages with the capability to formally verify agent programs, using a formal verification technique called model checking. Model checking (Clarke 1997) is the verification of properties of finite-state systems. Model-checking agent programs before deployment can offer guarantees about safety that otherwise would not be possible. The verified agent programs are mathematically proven to solely operate within a pre-defined specification, enabling agents to work in safety-critical environments without the risk of performing unpredictable and potentially dangerous actions.

The importance and relative absence of robust verification techniques integrated with agent programming languages has since been highlighted by Doan et al. (2014).

By developing the theoretical concepts from this thesis in GWENDOLEN, the requirements for verifying agent programs with similar semantics can be identified for future integration.

2.2.1 Operational Semantics for GWENDOLEN

This research primarily aims to expand upon the capabilities of existing BDI programming languages. Throughout this thesis, an operational semantics for the chosen representative BDI language, GWENDOLEN, is used as a means to communicate the existing capabilities for most BDI programming languages. With this formal specification for the underlying implementation, the research contributions can be highlighted without disseminating many lines of uncompiled Java code.

A version of the full operational semantics for the GWENDOLEN programming language can be found in Appendix A of this thesis, or alternatively, in (Dennis 2017). Its key components are, for each agent:

- B , a set of beliefs that are ground first-order formulae.
- I , a set of intentions that are stacks of *deeds* associated with some event.

Deeds can be the addition or deletion of beliefs, the adoption of new goals, and the execution of primitive actions. A GWENDOLEN agent may have several concurrent intentions and will, by default, execute the first deed on each intention stack in turn. GWENDOLEN is event-driven and events include the acquisition of new beliefs (typically via perception), messages and goals. A programmer supplies plans that describe how an agent should react to events by extending the deed stack of the relevant intention. These plans contain actions for execution.

Intentions in GWENDOLEN

Intentions are used in BDI languages to describe a sequence of deeds for achieving a goal. A sequence of deeds in GWENDOLEN is called a *deed stack*. For each deed, there is an associated event which is responsible for the deed's addition to the deed stack, these events are stored in the *event stack*. At the implementation level, an intention is a collection of tuples, with each tuple containing an event, a deed, and a unifier (unifiers are ignored in this thesis as they are an added complexity which has been left to further work).

Example The following table shows the structure for a single intention to perform a route navigation task:

event	deed
+!route	+!at(1)
+!route	+!at(0)

The intention is represented as a table containing a heading for the events and a heading for the deeds. Rows in the table associate a particular deed with the event that has caused the deed to be placed on the intention, for example, +!route in the ‘event’ column is the event responsible for the addition of +!at(1) to the ‘deed’ column. These columns form an event stack and a deed stack. Standard BDI syntax is followed: !g is used to indicate a goal, and +!g signifies the commitment to achieve that goal (i.e., a new goal that g becomes true is adopted).

The route navigation intention, represented by the table, has been triggered by the *route* goal, which aims to perform a maintenance task. The commitment to the *route* goal can be seen in the table as the trigger event for both rows in the intention. An intention is processed from top to bottom; it can be seen here that the agent first intends to commit to the goal $at(1)$. Once it has committed to that goal it then commits to the goal $at(0)$.

In GWENDOLEN the process of committing to a goal causes an expansion of the intention stack, first making the goal into an event which has yet to be planned (so is associated with the empty deed ϵ) and then selecting a plan to execute which replaces ϵ and pushes the deeds from the selected plan onto the intention stack to be processed.

Goals can be defined as “achieve” goals or “perform” goals in GWENDOLEN. The goal type handles the commitment assigned to the goal: an achieve goal’s plans are re-planned until the goal is believed to have been achieved, a perform goal’s plans are always triggered but are not re-planned if they fail to achieve the goal state.

If $at(1)$ is an achievement goal then the intention is expanded *before* the agent commits to $at(0)$. If the agent commits to the $at(1)$ achievement goal, that has a plan containing two actions, $move(0,1)$ and $printlogs$, the above intention becomes first:

event	deed
+!at(1)	ϵ
+!route()	+!at(1)
+!route()	+!at(0)

and then becomes:

event	deed
+!at(1)	move(0,1)
+!at(1)	printlogs
+!route()	+!at(1)
+!route()	+!at(0)

At any moment outside of the initial state, it is assumed that there is a *current intention* which is the intention being processed at that time. The stacks that form the intention are further paired with two booleans, which indicate the intention's status: *suspended*, meaning that by default it is *not* selected at the intention selection phase of the agent's reasoning, and *locked*, meaning the intention has been forced to remain current until it is unlocked, which could be used to allow a complete sequence of belief changes to be processed before any other reasoning takes place (Dennis 2017).

Because intentions are stacks, the top row of the intention is often of most interest: *the head*, written $hd_i(i)$, and the rest of the intention, *the tail*, written $\tau l_i(i)$. Sometimes it is only the top event on the intention, hd_e or the top deed on the intention, hd_d , that is of interest.

Figure 2.1 represents an example of a GWENDOLEN agent program.

```

GWENDOLEN

:name: Rover

:Initial Beliefs:

:Initial Goals:
  route [perform]

:Plans:
  +!route() : {True} <- +!at(1), +!at(0);
  +!at(1): {B at(0)} <- move(0, 1), printlogs;
  +!at(0): {B at(1)} <- move(1, 0), printlogs;

```

Figure 2.1: Example GWENDOLEN program for a route navigation task

In this agent program, the agent begins with no initial beliefs, and only one initial goal: to perform the 'route' goal. The agent's plan 'library' contains three plans that follow the syntax:

$$event : \{guard\} \leftarrow body$$

The event is the trigger for the the plan, which can be matched to the top event of an intention. In the example, the trigger event for $+\!at(1)$ and $+\!at(0)$ was the initial goal, $+\!route$. The guard is the condition that is checked against the agent's beliefs to determine whether the plan is applicable or not. The body is the deed stack that the plan proposes to execute.

The GWENDOLEN Reasoning Cycle

Reasoning in the context of cognitive agents refers to the process of drawing inferences, making decisions, and formulating plans based on the agent's knowledge, beliefs, and perceptions of the environment. Reasoning in cognitive agents is generally governed by a reasoning cycle. GWENDOLEN reasoning follows a six-stage cycle, shown in Figure 2.2.

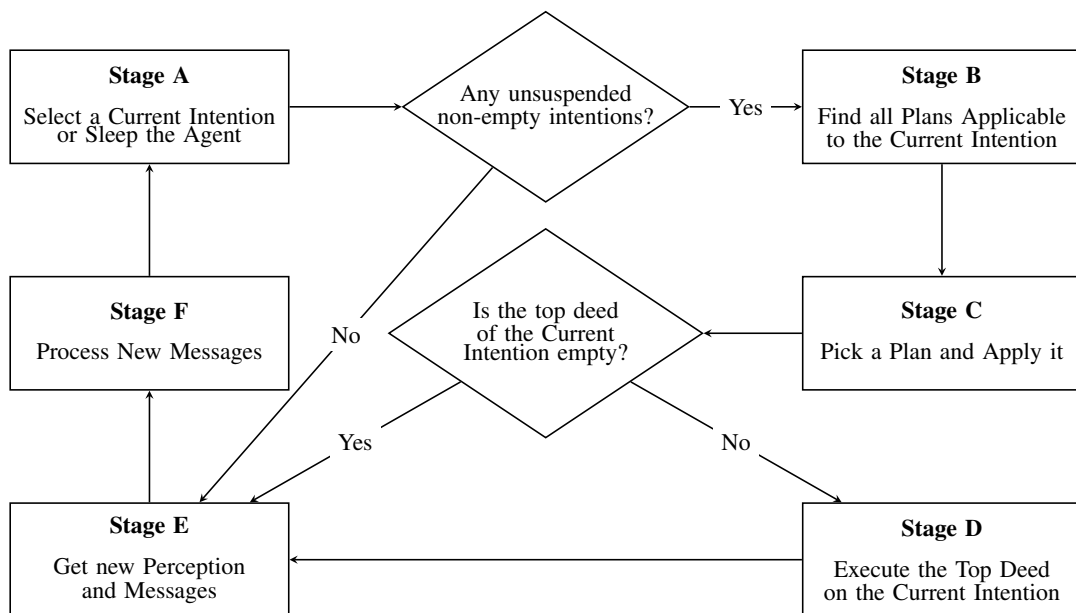


Figure 2.2: GWENDOLEN Reasoning Cycle

The reasoning cycle begins with **Stage A**, where the agent either selects a current intention or goes into a sleep state. This decision depends on the status of the agent's intentions; if all are empty or suspended, the agent will sleep. The sleep state can be overturned by new perceptions or messages that wake the agent and continue the cycle. After **Stage A**, the agent evaluates whether there are any unsuspended, non-empty intentions. If unsuspended intentions are present, the agent moves to **Stage B**. Here, the agent identifies all plans that apply to the current intention, storing them as applicable plans. Once applicable plans are identified, **Stage C** involves selecting and

applying one of these plans to the current intention. After executing a plan, the agent checks if the current intention is empty. If the current intention is not empty, the agent proceeds to **Stage D**, where the top deed of the current intention is executed. If either the current intention is empty or after **Stage D** is completed, the agent moves to **Stage E**. Here, new perceptions and messages are gathered from the environment. Finally, in **Stage F**, any new messages received during **Stage E** are processed. After this, the cycle loops back to **Stage A**.

Action Theory in GWENDOLEN

In GWENDOLEN actions are treated as atomic actions. When an action appears in a plan, it is placed on an intention as a deed, then when the processing of the intention reaches the action, it is executed and all other processing ceases until execution has completed. As a result of the issues with this style of action execution (particularly that no further perception takes place and so new events can not be reacted to), it has become typical in GWENDOLEN programs to treat the execution of actions as the *initiation* of the action. A specialised “wait for” construction is then used to *suspend* the intention containing the action until some success criterion is perceived. This is analogous to the “suspend” state in the Goal-lifecycle literature (Dennis & Fisher 2014, Harland et al. 2014) since intentions are often created by the acquisition of a goal. Thus, GWENDOLEN loosely supports the concept of actions with durations by making use of this “wait for” command.

The way in which GWENDOLEN handles and executes actions is described by a set of operational semantics (Dennis 2017). These operational semantics are subdivided into transition rules that govern each individual operation. Specific rules are selected at each stage of the reasoning cycle, which cause transitions on the agent. Table 2.1 shows the commonly used notational conventions in these transition rules from the GWENDOLEN semantics.

In the representation of the GWENDOLEN operational semantics presented here, references to unifiers, edge cases, and some specialised action types have been removed for simplicity. It should be noted that unifiers are a commonly used concept in BDI language semantics and are used to match abstract plans to specific situations, reducing the need for context-specific plans (Rao 1996). Agents can ‘apply’ their context to applicable abstract plans by using a unifier to bind a known value to a variable defined in a plan. However, the extended semantics proposed in this thesis do not directly affect the operation of unifiers. Therefore the transition rules in this thesis have

Symbol	Explanation
a	An action performed by the agent.
ξ, ξ'	The agent's state before and after an action, respectively.
i	The current intention of the agent.
$\text{hd}_d(i)$	The top deed of the current intention stack.
$\text{tl}_i(i)$	The tail of the intention stack, excluding the top element.
$\mathbf{do}(a)$	Represents the execution of an action.
$\rightarrow_{\text{action}}$	Denotes the transition of the agent's state as a result of performing an action.
$* \dots b$	Represents the instruction to wait for a belief to become true.
$B \models b$	Expresses that the formula b is a logical consequence of the agent's belief base.
$+b$	Represents the addition of a belief to the belief base.
$\text{new}(+b, \epsilon)$	Creates a new intention from an event and a deed.
$B \cup \{b\}$	Adds a new belief to the belief base.

Table 2.1: GWENDOLEN Notation conventions

been simplified by removing all reference to unifiers.

The operation of action execution is represented by Rule (2.4).

$$\frac{\text{hd}_d(i) = a \quad \xi \xrightarrow{\mathbf{do}(a)} \xi'}{\langle \xi, \langle \dots i \dots \rangle \rangle \rightarrow_{\text{action}} \langle \xi', \langle \dots \text{tl}_i(i) \dots \rangle \rangle} \quad (2.4)$$

The state of the system is represented by an environment, ξ , coupled with a large tuple containing the components required by the agent to function, of which the current intention, i , is most important. As noted above intentions represent a stack of deeds to be executed in order to handle some event (e.g., to achieve a goal). $\text{hd}_d(i)$ is used to represent the top (head) deed (hd_d) on this intention stack (i). a is used to represent an action, and $\text{tl}_i(i)$ to represent the tail (tl_i) of the intention stack (i) after the top deed is removed¹.

Thus Rule (2.4) states that when the top deed on the intention is flagged as an action, $\text{hd}_d(i) = a$, and the outcome of that action is to change the environment, ξ , to become ξ' , represented by $\xi \xrightarrow{\mathbf{do}(a)} \xi'$, then performing the action transforms the pair of the environment and the tuple of agent components by changing the environment and removing the top of the current intention, i , (i becomes $\text{tl}_i(i)$). The other items in the agent's tuple remain unchanged (represented by ' \dots ').

¹For simplicity of presentation, the intention is treated as a stack of deeds but it should be noted that more information, such as the goal to be achieved, is also included in the full semantics.

The “wait for” command is governed by Rules (2.5) and (2.6) (again these have been simplified to remove unifiers, edge cases and references to Prolog-style *reasoning rules*). Here $*\dots b$ represents the instruction “wait for b to become true”, $B \models b$, expresses that the formula b is a logical consequence of the agent’s belief base, B . While the instruction: $\text{suspend}(i)$, suspends the intention, i .

$$\frac{\text{hd}_d(i) = *\dots b \quad B \models b}{\langle \xi, \langle \dots i \dots B \dots \rangle \rangle \rightarrow_{\text{wait_for}} \langle \xi, \langle \dots \text{tl}_i(i) \dots B \dots \rangle \rangle} \quad (2.5)$$

$$\frac{\text{hd}_d(i) = *\dots b \quad B \not\models b}{\langle \xi, \langle \dots i \dots B \dots \rangle \rangle \rightarrow_{\text{wait_for}} \langle \xi, \langle \dots \text{suspend}(i) \dots B \dots \rangle \rangle} \quad (2.6)$$

Rules (2.5) and 2.6 handle the two cases for the “wait for” instruction ($*\dots b$) when it appears as the top deed of an intention. In Rule (2.5), when the condition b is true in the agent’s belief base ($B \models b$), the instruction to wait for the belief to become true is removed and the intention continues processing. In Rule (2.6), when the condition is not true ($B \not\models b$), the intention is suspended ($\text{suspend}(i)$) until the condition becomes true.

Lastly, existing intentions are unsuspending when new beliefs are added, as shown in Rule 2.7. This can happen either as a result of perception or as deeds that appear in plans, in both cases the new beliefs appear on the top of some intention:

$$\frac{\text{hd}_d(i) = +b}{\langle \xi, \langle \dots i, I, B \dots \rangle \rangle \rightarrow_{\text{add_belief}} \langle \xi, \langle \dots \text{tl}_i(i), \mathbf{unsuspend}(I, b) \cup \text{new}(+b, \epsilon), B \cup \{b\}, \dots \rangle \rangle} \quad (2.7)$$

Here $\mathbf{unsuspend}(I, b)$ unsuspending all suspended intentions in I (the agent’s intention set) that are waiting for the belief, b , to be perceived by the agent. Whilst $\text{new}(+b, \epsilon)$ creates a new intention from an event, $+b$, and a deed, ϵ (in this case, an ‘empty’ deed). Therefore this rule adds a new belief, $+b$, to the belief base, B , and a new intention noting the appearance of the new belief. At the same time, all intentions that are waiting for b to be achieved as part of their suspend condition are unsuspending. The work in this thesis attempts to merge the processes described in these semantic rules into more principled extended semantics that consider actions with durations and terminating conditions.

2.3 Automated Planning

Automated planning (also called AI planning) is an area of AI research that focuses on developing and optimising algorithms that can find sequences of actions that will take an agent from an initial state to a goal state. AI planning can be categorised into several different types based on the representation of the environment, the available actions and information, and the planning methodology.

The work in this thesis is integrated with a framework for reconfiguring agent plan libraries (Cardoso et al. 2019) that can use the updated action descriptions produced by the algorithm in Chapter 6 to repair plans that contain failing actions, prolonging the effective lifetime of the agent. The framework proposed by Cardoso et al. (2019) uses existing plans in the plan library to find alternative solutions to achieve the post-conditions of capabilities that no longer achieve their post-conditions. The reconfiguration framework works in conjunction with the method for learning new action descriptions presented in this thesis. If a new action description is learned, it is available to the reconfiguration framework, meaning that the AI planner will have more plans available to attempt a reconfiguration with.

To find the alternative solutions, Cardoso et al. (2019) used a classical planning algorithm. In a classical planning problem, the goal is to find a sequence of actions that leads from an initial state to a desired goal state. Classical planning algorithms operate in a deterministic fully-observable environment, meaning that the solution that is produced will only be correct if actions achieve their stated post-conditions. In classical planning, actions also have defined pre-conditions and ‘effects’, similar to the pre-conditions and post-conditions used in the BDI action theories. Although in the case of classical planning, the action theories are used to construct plans directly from actions.

2.3.1 STRIPS and PDDL

The Stanford Research Institute Problem Solver (STRIPS) (Fikes & Nilsson 1971) is a foundational representation language that influenced many modern planning languages in the AI Planning domain. The language’s name was inherited from the popular planner that uses the language. Planners that can solve problems defined in this language or a subsidiary of the language are referred to as STRIPS-like planners.

The planning language used in this research is the Planning Domain Definition

Language (PDDL) 2.1 (Fox & Long 2003). A planning language is required by Cardoso et al.'s (2019) plan library reconfigurability framework to define GWENDOLEN plans and the BDI agent's environment in a language that can be interpreted by an AI planner. PDDL 2.1 is part of the PDDL (Mcdermott et al. 1998) family, originally developed to improve upon the STRIPS language, by allowing greater expressiveness in domain descriptions and problem definitions. There are six generally recognised evolutions of the PDDL family, but PDDL 2.1 is favoured by Cardoso et al. (2019) as this iteration specifically introduced time, pre- and post-conditions, and numeric fluents (to express resource levels). This version of PDDL is also particularly suited to the research in this thesis as it supports the concept of actions with durations, and can also manage the expression of actions with pre-conditions and post-conditions, which are relied upon in this thesis to allow action descriptions to be updated.

2.3.2 Action learning in AI planning

Features of the learning algorithms used in the AI Planning domain have influenced the method for learning updated action specifications in this research. There is a considerable body of work on learning action specifications from an environment in the AI planning domain, however, the solutions often depend upon exploration of the environment to allow the discovery of new solutions, which may have negative implications for safety critical scenarios.

Mugan & Kuipers (2011) introduced the Qualitative Learner of Action and Perception (QLAP). When deployed in an unknown, continuous, and dynamic environment, QLAP constructs a hierarchical structure of possible actions in an environment based upon the consequences of actions that have happened before.

Pasula et al. (2007) explored the use of Machine Learning and probabilistic planning in complex environments to cope with unexpected outcomes. A learning algorithm is used to determine an action model with the greatest likelihood of attaining the perceived action effects of another different set of actions. The method is discussed in full in Chapter 6.

Chapter 3

Related Work

In this chapter, related research is examined and evaluated against the research in this thesis. Similarities found in other works are highlighted, and the advantages and disadvantages of each are outlined. This chapter is divided into four sections covering action theory, actions with durations, failure handling, and learning action descriptions. These sections were identified in the research questions and objectives in Chapter 1, and help to categorise the related works.

3.1 BDI Action Theory

Harland et al.'s operational semantics for goal life-cycles in BDI agents (Harland et al. 2014) formed the theoretical foundation for Dennis and Fisher's (Dennis & Fisher 2014) actions with durations and failures. Harland et al.'s operational semantics allow agents to activate, abort, suspend, and resume goals based on their current state and the agent's perceived knowledge. For example, if an agent's goal is not consistent with its knowledge, then it can be suspended or aborted. If the agent's knowledge changes, and makes a goal valid again, the goal can be resumed.

Dennis and Fisher adapted Harland et al.'s semantics to treat actions as capabilities. These capabilities contain a reference to the original action but also pre-conditions and post-conditions, and three terminating conditions that describe the beliefs that are expected when that action has: succeeded, failed, or requires an abort. The research on durative actions and failure detection in this thesis is based on the semantics introduced in these two works.

3.2 Actions with Durations

In general BDI languages do not explicitly treat actions as having durations. A notable exception is the *Brahms* language (Sierhuis 2001, Sierhuis et al. 2007) in which actions, called *activities*, explicitly involve durations. *Brahms* was originally developed as a simulation language and its focus was upon answering questions about whether human-agent teams could complete tasks within particular times. The simulation language has been used to model the work practices of air traffic controllers in air traffic management, and the activities of the Apollo astronauts on the surface of the moon. In its original presentation, *Brahms* had no formal semantics. However, one was later provided by Stocker et al. (2011), although these semantics are primarily concerned with the effect of activity duration on simulations with mechanisms for monitoring the behaviour of an activity during its execution. Nonetheless, the formal semantics provided a definition of the *Brahms* language containing 41 rules that capture its operational behaviour, enabling formal reasoning about *Brahms* models.

There has been more recent works on actions with durations in the BDI domain (Traldi et al. 2022) that highlight the importance of an explicit representation of time. Traldi et al.’s work aims to improve upon the scheduling process of BDI agents, allowing agents to also reason about the temporal implications of each action, in an effort to prioritise actions that may be otherwise delayed. This is particularly useful in scenarios where there is inherent risk if important actions are delayed and are blocked by a queue of less critical actions. Traldi et al.’s (2022) introduction of durative actions here is concerned with avoiding failure, rather than considering how failure has occurred and how a subsequent recovery could be managed.

The modelling of actions with durations has also been considered in recent versions of the GOAL programming language (Hindriks 2021), with ‘timers’ allowing programmers to generate percepts at predefined time intervals. Whilst this concept does not directly support actions with explicitly defined durations, the implementation of ‘timers’ introduces a notion of time to GOAL, allowing agents to perform time-sensitive reasoning about ‘action-selection’. For example, if a timer was used to generate a percept that states that it has been 24 hours since the plants in a greenhouse have been watered, and this percept is a precondition for an action to water the plants in the greenhouse, the agent’s rule-based action-selection will now see that the preconditions have been met for this action and can be selected for execution. Whilst the consideration for actions with durations allows GOAL agents to handle time-sensitive tasks, there is no support for failures and dealing with failures.

Vikhorev et al. (2011) developed a ‘real-time’ extension of the AgentSpeak programming language, AgentSpeak(RT), which supports the use of *deadlines* and *priorities*. With AgentSpeak(RT), goals can specify a *deadline* by which the goal should be achieved, and a *priority* to indicate their relative importance with higher-priority goals taking precedence over lower-priority goals. The temporal constraint and importance only applies specifically to intentions rather than to individual actions. These additions enable the concurrent execution of multiple intentions, using a more intelligent scheduler that only commits to a set of intentions that is maximally feasible.

Ferber & Müller (1996) highlighted poor domain specification as an issue in multi-agent systems and subsequently considered an extension of action theory to account for a more dynamic environment. Their approach recognises that actions may not always affect their environment in the predefined and intended way. To address this, they defined a formalism that separated actions into two phases: the ‘influences’, the intended consequence of an action, and the ‘reaction’, the actual outcome of the action’s influences on the agent’s environment given its previous state.

Ferber and Müller stated that there was a lack of a formal specification for agent architectures based on mental states, such as the BDI agent architecture, which was correct at the time of writing, although now there are many formal specifications (Bordini & Hübner 2010, Dennis et al. 2007, d’Inverno et al. 2000, 1997, Hindriks 2021). It was also stated that agent architectures that are based on mental states have an inability to take into account “reactive agents”, which are agents that are not characterised by their mental states but rather by their perception and action capabilities with respect to the environment. This statement has also since been rectified as agents following the BDI model are now widely capable of reacting to their environments through perception (Mascardi et al. 2005).

The task of defining and deploying durative actions has also been well explored in the multi-agent systems research area (Ferber & Müller 1996, Helleboogh et al. 2007, Ricci et al. 2010). A formalism for modelling dynamic environments is proposed by Helleboogh et al. (2007) with specific consideration for actions that might not be achieving expected outcomes, however, the formalism focuses on defining the effects of environmental interference on actions rather than detecting failures. Whilst the research considers many of the issues encountered whilst deploying an agent into a dynamic environment, it concentrates specifically on defining a formalism for multi-agent simulation (as intended by the authors) rather than implementation.

Work by Ricci et al. (2010) critiques the modelling of actions in BDI languages,

outlining not only the inadequacy of current action theory for use in a dynamic environment but also the additional facets of feedback for actions that can be exploited whilst using a dynamic environment. Specifically, the ability to consider an action's success or failure from environment perceptions. The focus of Ricci et al. (2010) was to provide an improved action and perception model for complex endogenous environments and takes a different research direction to that which is considered by the research in this thesis. The added complexity introduced by Ricci et al. (2010)'s action and perception model could lead to scalability issues when deployed in large-scale environments due to the increasing number of artefacts required to define and manage actions and perceptions in the environment.

The approach taken in this thesis for defining and deploying durative actions is presented in Chapter 5.

The field of AI Planning has invested considerable effort in the modelling of actions and capabilities with durations and stochastic outcomes, both theoretically as variants on Markov Decision Procedures (Mausam & Weld 2008, Younes & Simmons 2004) and practically capturing the concepts of actions and capabilities with durations and stochastic outcomes in planners (e.g., (Cirillo et al. 2010)) and domain description languages such as the PDDL 2.1 extension of PDDL (Fox & Long 2003). In this domain, the effect of the action duration is of most importance during the generation of the plan, rather than its execution. In automated planning, executable plans are represented as sequences of actions and lack the manipulation of mental states which is the defining feature of BDI approaches.

Epistemic planning enriches the automated planning process with the epistemic notions of knowledge and beliefs (Bolander 2017). The introduction of knowledge and beliefs allows the planner to reason with more context about the agent's environment. However, the greater level of expression needed for state representation in epistemic planning significantly increases the complexity of the planning problem as the state space increases, requiring greater computational power and time (Hu et al. 2022).

The modelling of actions with durations has been considered in logics for agency. Troquard & Vieu (2006) represent these using continuations within Seeing To It That (STIT) logic. The logic does not explicitly link the issue of durations with aborts, nor does it adequately account for the need to suspend working on a goal while waiting for an action to complete. As such this work was a less attractive influence for a

GWENDOLEN implementation than Dennis & Fisher (2014)'s abstract formal semantics, where there is already consideration for actions that do not complete instantaneously, and semantics for goal suspension whilst actions are executing.

Whilst the concept of durative actions has been adequately explored in the mentioned works, there has not been an implementation that focuses on monitoring individual actions for failure.

3.3 Failure Handling

There has been a great deal of work on plan failure in BDI programming languages (e.g., (Bordini & Hübner 2010, Sardiña & Padgham 2007, Sardiña & Padgham 2011)). This has not distinguished goal failure from action/capability failure, whereby the post-conditions stated in the action description are not believed to have been achieved after execution. This is understandable considering that when an action fails its most important effect is on the goal which may need to be dropped or re-planned. As a result, work has focused on goal-dropping and re-planning mechanisms, which are captured in the work on BDI goal life-cycles by Harland et al. (2014) upon which much of the work in this thesis is also based. In Chapter 6 the integration of an action failure logging mechanism with mechanisms for goal dropping and re-planning is discussed, which addresses the need for a distinction between goal and action or capability failures.

Yao, Logan and Thangarajah (Yao et al. 2016) developed a method for recovering from plan execution failures that identifies and exploits positive interactions between an agent's intentions. They leveraged the effects of performing action(s) from other intentions that bring about the conditions required to progress intentions with previously failing plans. The method focused on the development of a Monte-Carlo Tree Search scheduler with an extended selection phase, whereupon plan failure, the scheduler attempts to find an execution order that can re-establish the context or pre-condition of the next step in the intention. This work provides a more flexible and robust procedure for plan execution, however, if an intention cannot be progressed by the scheduler, the plan is dropped and regular backtracking occurs to find an applicable plan. The failing plan is left in the library for future selection, where failing actions may derail plan execution again.

Cardoso et al. (Cardoso et al. 2019) have developed a method for reasoning about replacing malfunctioning actions with alternate existing actions to achieve the same desired goal, reusing the domain entities and predicates that are already available.

The formal framework is implemented into the GWENDOLEN programming language, using an agile AI planner to identify viable replacements for failing actions, before merging the replacement plan into the plan library. The work in this paper provided a significant base for the research in this thesis.

While the integration of AI planning into BDI architectures is an active area of research, with various frameworks focusing on plan adaptation, evolution, and execution monitoring (Meneguzzi & de Silva 2015), only Cardoso et al. (2019) have developed a method for finding plan replacements for plans containing failing capabilities for BDI agents at runtime using an AI planner. Without this framework, there would be a requirement for three additional mechanisms to enable the work proposed in this thesis. A mechanism for translating BDI agent capabilities and environments to an AI planning domain and planning problem, a method for specifying plans containing a failing capability, and a method for running an AI planner on demand when capability failures are detected.

Whilst Cardoso et al. (2019)'s framework addresses these requirements, there were two assumptions stated for its implementation: firstly, the availability of a method for identifying capability failures, and secondly, a method for parsing plans from the AI planner back to the GWENDOLEN plan library. A method for identifying capability failure is required as this acts as the trigger for a plan library reconfiguration. Furthermore, if a valid plan is found to replace plans containing the failing capability, then a method for parsing STRIPS-like plans, generated by the AI planner, is required to translate the new plan back to a format that is comprehensible to the agent. This thesis addresses both of these assumptions, by introducing a method for identifying capability failures in Chapter 6, and by implementing a STRIPS-like plan parser that translates the output of Cardoso et al. (2019)'s framework back into a GWENDOLEN plan library, as part of the cohesive GWENDOLEN implementation evaluated in Chapter 7.

Recent work by Ferrando & Cardoso (2022) has introduced an automated failure handling mechanism for BDI agents. The mechanism is enforced at runtime to ensure that agents perform within a formal safety specification. Every event in a 'shielded' plan is checked before its execution, and if an event leads to violating a safety specification, then the shield can intervene. This mechanism would be highly effective in safety-critical environments, although it could also limit the flexibility and longevity of agents in dynamic environments if deployed without an efficient method for recovery.

Research in runtime verification has taken a different approach to implementing failure detection. Runtime monitors (Ferrando & Cardoso 2022, Ferrando et al. 2020,

Salfner et al. 2010) are generated from formal properties to ensure system traces do not violate the given formal specification. If a property is in danger of being violated by an action, the action can be stopped or flagged as a violation. Using this mechanism, runtime verification aims to completely avoid failure. However, when applied practically, there can be limitations that are difficult or even impossible to overcome: poor scalability in large distributed systems, limited observability of full system states, and performance overhead. Runtime verification can also struggle with rapidly changing and complex environments, making comprehensive monitoring difficult to achieve (Sánchez et al. 2019).

3.4 Learning Action Descriptions

There have been efforts towards introducing explicit mechanisms to allow BDI agents to be able to learn (Guerra-Hernández et al. 2004, Phung et al. 2005), however, other research areas such as AI planning have considerably more experience in applying learning algorithms to existing systems (Arora et al. 2018). In this domain, reinforcement learning has been used to learn the ‘rules’ of an unknown environment. The general method centres on allowing agents to explore the environment by executing actions available to them, collecting the resulting sensor data from the outcomes, and subsequently using the data from action executions to infer the pre-conditions and post-conditions for each.

Reinforcement learning is evidently the leading method for learning action descriptions in the AI planning domain (Arora et al. 2018, Celorrio et al. 2012), which is likely due to their complementary use cases (Lee et al. 2022). AI planning focuses on finding high-level plans to solve problems based on defined domain knowledge, whereas reinforcement learning is able to learn lower-level actions through interaction with the environment (Sutton & Barto 2018). However, the safety implications of trial and error exploration used by reinforcement learning algorithms meant that a different, more safety-focused approach was taken for the research in this thesis. The exploration requirement for these algorithms is driven by their narrow use case: learning rules from a completely unknown and previously unvisited environment (Grounds & Kudenko 2007). In the applications considered by the research in this thesis, the agent has access to historic action outcomes, and this allows for a safer approach to be taken, without the need to perform actions in an unknown environment with no prior knowledge of the consequences.

Learning algorithms have been extensively implemented and explored for their ability to adapt to unknown environments, including inductive learning (Mitchell 1997), genetic algorithms (Goldberg 1989), transfer learning (Pan & Yang 2010), and supervised learning (Cunningham et al. 2008). However, these methods each face individual challenges surrounding their requirements for high-quality data, potentially unsafe exploratory behaviour, noise sensitivity, and computational complexity. In Chapter 6, these challenges are discussed and a hybrid learning algorithm is introduced, tailored to meet the specific requirements of the proposed system. This algorithm integrates features from supervised learning, rote learning, and ranking techniques to ensure safety, efficiency, and effective use of the available historical data from an action log.

Chapter 4

A Framework for Adaptive Cognitive Agents

This chapter presents an overview of the approach of the research in this thesis, supported by illustrations of the newly proposed extensions and their integration into the existing architecture. A walk-through of the proposed system is then given, using a diagram of the updated system architecture and a pseudo-code algorithm. The novel contributions outlined in this chapter are discussed in detail in subsequent chapters: Chapter 5 presents the integration of durative actions, while Chapter 6 addresses action logging, failure detection, and the learning of action descriptions.

4.1 The Reasoning Cycle for Cognitive Agents

The Sense-Reason-Act cycle, shown in Figure 4.1, is the fundamental process used by cognitive agents (Millican & Wooldridge 2014). Agents sense their environment, reason about it, and then act accordingly by performing an action based on the reasoning.

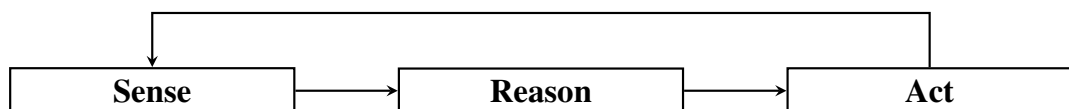


Figure 4.1: Generic Sense-Reason-Act cycle

The BDI model operates by using this process. BDI agents are programmed to behave rationally based on their beliefs about the environment, their desires (goals), and their intentions (plans) to achieve those goals (Rao & Georgeff 1995). However,

the standard BDI model, cannot account for changes experienced in dynamic environments if the available actions to ‘act’ with are static for the duration of execution (Archibald et al. 2024). In this context, ‘static’ means that the agent’s understanding of the effect of the action can not change even if the effect of the action is changing. Agents therefore need a mechanism for detecting failure, and learning from it, to adapt their behaviour accordingly.

In a BDI architecture, mechanisms for detecting failures, learning from failures, and adapting behaviours, belong in the *reason* stage of the Sense-Reason-Act cycle. For BDI agents, reasoning is generally governed by a reasoning cycle.

4.2 Introducing a Framework for Adaptive Cognitive Agents

This thesis proposes a framework to extend BDI theory. The proposal is illustrated in Figures 4.2 and 4.3. Each extension is discussed in this thesis.

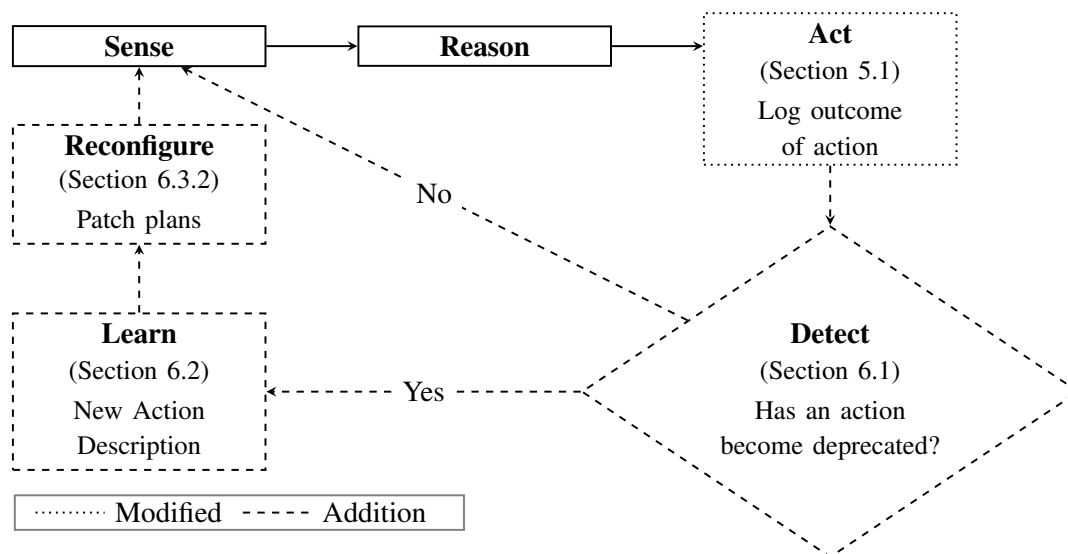


Figure 4.2: Extended Sense-Reason-Act cycle to account for action deprecation, synthesis of new action descriptions, and the patching of plans.

In Figure 4.2, a Sense-Reason-Act cycle is depicted with the additional processes required by the proposed extension. There are four notable additions:

- The act phase now requires the agent to log the outcome of each action execution;

4.2. INTRODUCING A FRAMEWORK FOR ADAPTIVE COGNITIVE AGENTS 53

- The agent now also checks to see if the executed action is deprecated by comparing the expected outcome with the actual outcome;
- Two new phases were also added:
 - In the synthesis phase, new action descriptions are learned from the data collected from action executions;
 - In the patch phase, if a new action description is learned, the affected plans in the plan library are reconfigured using the updated descriptions of the existing actions.

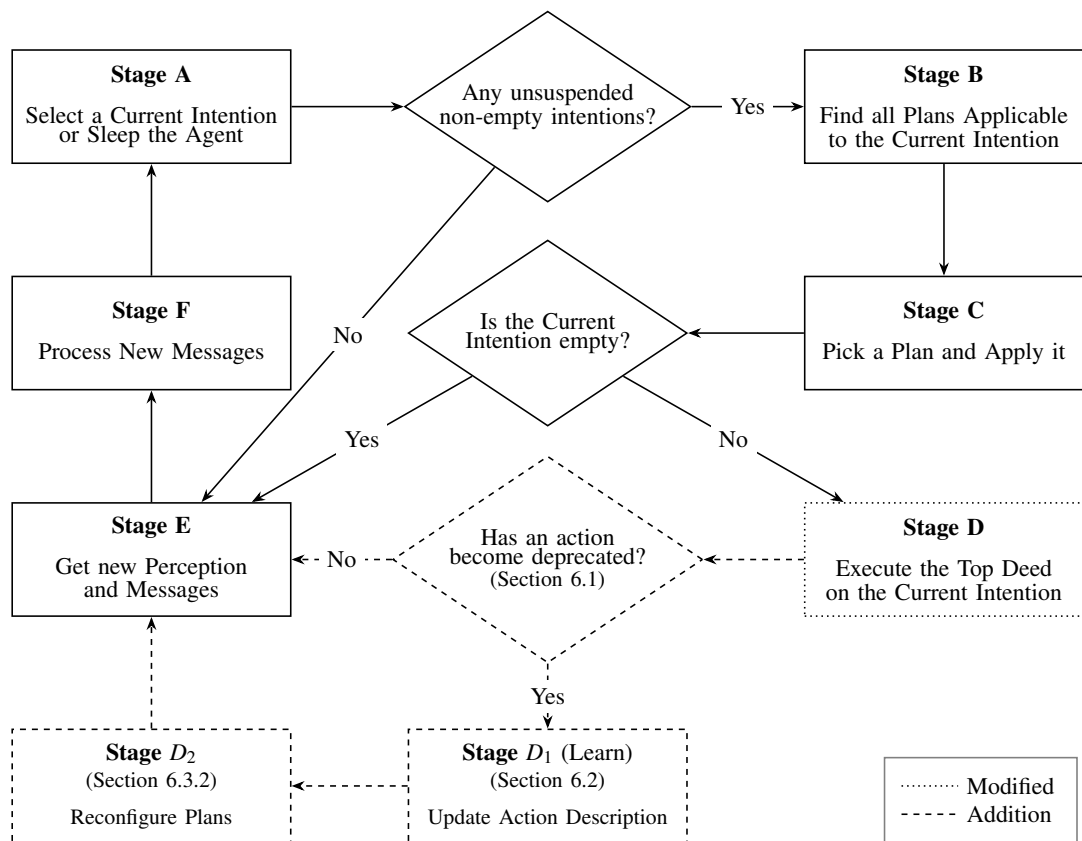


Figure 4.3: GWENDOLEN reasoning cycle with additional stages showing the proposed extension.

In Figure 4.3, the proposed extension has been mapped onto the existing reasoning cycle, that was mentioned in Section 2.2.1. In particular, ‘Stage D’, the stage where the top deed on the current intention is executed, has been expanded. In the original reasoning cycle, ‘Stage D’ is solely responsible for executing the top deed on the current intention.

After executing the top deed on the current intention, a check for deprecated actions is made. The last entry of the action log is checked: if it is an entry for anything other than an action failure then nothing further happens. No action becomes deprecated, and the cycle continues to Stage E. However, if it is an entry for an action that has failed, the number of entries containing a failure for this specific action is checked against a *failure threshold*. The use of a failure threshold enables the failure detection method to distinguish between transient failures and persistent failures. The threshold ensures that actions will not have their descriptions immediately replaced with a new set of post-conditions based on a single failure, unless the threshold is set to 1, as the post-conditions logged from a single failure may be anomalous to the true action post-conditions. Different actions may be executed in varying frequencies depending on their usage in the plan library, and as a result, the threshold value of an action is context-specific, and must be defined prior to execution in this implementation. In any case, if the threshold has been reached, the reasoning cycle moves to the new Stage D_1 in which a new action description will be learned, and then moves onto Stage D_2 where plans will be patched using Cardoso et al. (2019)’s reconfiguration mechanism.

This extracts all the action descriptions from the agent and translates them into STRIPS operators (Fikes & Nilsson 1971). It should also be noted that the pre-conditions and post-conditions used in the agent program examples in this thesis are stated in absolute terms, i.e., $at(0)$, rather than in relative terms such as $at(X + 1)$, for compatibility with Cardoso et al. (2019)’s reconfigurability framework. Let the old action description for the failed action a be:

$$\{Pre\}a\{Post\}[n]$$

The reconfiguration mechanism computes initial and goal states for a planning problem from $\{Pre\}$ and $\{Post\}$. This planning problem is then given to a STRIPS planner together with the STRIPS operators of the agent’s plan descriptions. If the planner computes a new plan this is translated into a sequence of GWENDOLEN actions, a_l , and this sequence replaces a everywhere it appears in the agent’s plans. Algorithm 1 shows this process. The extended ‘Stage D’ is shown on lines 10-18, with lines 13-16 representing the implementation of Cardoso et al. (2019)’s framework as ‘Stage D_2 ’.

Starting on line 1 of Algorithm 1, the loop is initiated and continues as long as the agent is active. On line 2 **Stage A** is executed, where a *currentIntention* is selected, or if all intentions are empty or suspended then the agent’s sleep flag will be set to true. New perceptions and messages can change this flag to false to ‘wake’ the agent and

Algorithm 1: Gwendolen Reasoning Cycle with Extended Stage D

```

1 while Agent is active do
2   Stage A: Select a currentIntention or Sleep the Agent;
3   if unsuspended non-empty intentions exist then
4     Stage B: Find all applicablePlans to the currentIntention;
5     Stage C: Pick a plan and Apply it;
6     if currentIntention empty then
7       Stage E: Get new perceptions and messages;
8     else
9       Stage D: Execute the topDeed on the currentIntention;
10      if lastActionLogEntry == failure then
11        if actionFailureCount > actionFailureThreshold then
12          Stage D1: Learn(action);
13          Stage D2: Reconfigure(action);
14          i: Translate action descriptions to STRIPS operators;
15          ii: Compute initial and goal states for planning;
16          iii: Execute planning algorithm;
17          if newPlan exists then
18            Replace action a with sequence al in agent's plans;
19        Stage E: Get new perceptions and messages;
20      else
21        Stage E: Get new perceptions and messages;
22    Stage F: Process new messages;

```

continue the cycle. If no unsuspended non-empty intentions existed when executing line 3 of the algorithm, **Stage B**, **Stage C**, and **Stage D** are skipped, and instead **Stage E** gets all new *messages* and *perceptions* from the environment (line 20 and 21) before continuing to **Stage F** (line 22), where these are processed.

If any unsuspended intentions do exist and have deeds to perform then **Stage B** is executed. In **Stage B** (line 4) all of the plans that apply to the *currentIntention* are found and stored in the agent's *applicablePlans*. Once this has been done, on line 5 **Stage C** is started, where a plan from the *applicablePlans* is chosen and applied. From here if the *currentIntention* is empty, then **Stage E** gets all new messages and perceptions from the environment.

If the *currentIntention* is not empty, then **Stage D** (line 9) executes the *topDeed* on the *currentIntention*. After this stage, the *lastActionLogEntry* is checked. If the

log entry contains a failure then the *actionFailureCount* is checked against the *actionFailureThreshold* for that action (lines 10 and 11). If the count is above the threshold then **Stage D₁** is started (line 12), where the *action* is passed to the ‘Learn’ rule for a new action description to be learned. Next, the action log entries for the deprecated action are filtered into a separate log in reverse order in preparation for the ranking policy. Each variation of post-conditions in the separate log is assigned a weight, calculated by their index position (representing recency with linear decay) and the frequency of that variation. The variation with the highest weight is assigned as the new success post-conditions for this action. This process is discussed in more detail in Section 6.3.1.

Once the new set of success post-conditions has been learned, **Stage D₂** (line 13) is started, where the ‘Reconfigure’ rule is executed on the *action*. The ‘Reconfigure’ rule is broken into four parts *i*, *ii*, *iii*, and a final stage is performed if a new plan is found. In part *i* on line 14, the action descriptions of all of the actions are converted to STRIPS operators. In part *ii* (line 15), the initial state and the goal state of the planning domain are created from the failing action’s old pre-conditions and post-conditions (from before Stage D₁). In part *iii* (line 16), these states and the actions are passed to the planning algorithm to find a sequence of actions that solves the planning problem.

If a sequence is found, a *newPlan* is produced (lines 17), then any plans containing the failing action description are patched using the replacement sequence of actions found by the planning algorithm (line 18). **Stage E** (line 19) now gets any new *perceptions* and *messages* before **Stage F** (line 22) processes any new messages received during **Stage E**.

Chapter 5

Durative Actions

In this chapter, the introduction of durations for actions is justified, the required mechanisms for integrating durations into existing action semantics are provided alongside an example use case. Lastly, details of the implementation of the extended action theory are given.

5.1 Terminating Conditions

Dennis & Fisher (2014) advocated associating actions not only with pre- and post-conditions containing durations but also with explicit success, failure and abort conditions (an abort is used if the action is ongoing but needs to be stopped) and suggest goals be suspended while an action is executing and then the action's behaviour be monitored for the occurrence of its success, failure or abort conditions. When one of these occurs the goal then moves to the *Active* or *Pending* (where re-planning may be required) part of its life-cycle as appropriate.

In the proposed mechanism, the terminating conditions of actions can be defined. There are three sets of conditions that can be defined: *Success*, *Failure*, and *Abort*. Each of these sets of conditions is associated with a set of operational rules, defined in Section 5.2.1, that manage the processes that are invoked when an action is executed and one of the conditions is subsequently believed to be true.

Conditions for *Success*, *Failure*, and *Abort* are especially useful in detecting persistent failure and driving algorithmic learning. In the proposed mechanism, an *action log* is kept, containing a historical log of action outcomes for each action and a record of the change in beliefs from before execution compared to after execution.

Action Logging

To assess an action's status (i.e. Active, Pending, or Deprecated), a historical record of all action executions must be kept, alongside the condition under which they were terminated and the beliefs held by the agent both before and after the execution.

Action Log		
Capability	Change in beliefs	Action Outcome
{at(0)}{move(0, 1)}{at(1)}	-at(0), +at(1)	Success
{at(0)}{move(0, 1)}{at(1)}	-at(0), +at(3)	Failure

Figure 5.1: Action log for a GWENDOLEN agent

Figure 5.1 is a visual representation of a Java ArrayList containing two Action Log Entry objects. The Action Log Entry object holds a Capability, a list of the difference in beliefs before and after the action executed, and finally the outcome for that action once it terminated. The action shown in the figure was programmed to move the agent from a waypoint 0 to another waypoint 1. In the first entry, the action is believed to have succeeded as the change in beliefs results in the agent believing that it is at waypoint 1, matching the expected post-conditions for that action. In the second entry, the change in beliefs results in the agent believing that it is not at the expected waypoint, producing a failure as the action outcome.

The action log is represented in the implementation as an Array List, with a fixed size defined before execution. There is no default value as it should be calculated for each deployment. The fixed size should consider the number of actions available to the agent and the classification of these actions (i.e., navigation, activating effectors, recording from sensors). The fixed size should also be a balance between keeping enough entries to make informed learning decisions, but also to discard ageing entries. More available actions requires a larger size as more actions will be sharing space in the same log. Special consideration should be taken for actions that may be executed more frequently.

When a new entry is created for the log, if the size limit has been reached the oldest entry in the log is removed before adding the new entry. With the action log acting as a queue, entries can be treated with various weights dependent on their index position: with more recent entries representative of more up-to-date data. Data obtained from

repeated patterns in the action log is also given more value in the learning process by individually weighting each entry at first, then summing the weights of entries with the same post-conditions. Additional data points are important when constructing a dataset that is suitable for machine learning and have consequently been considered in the design of the Action Log data structure.

5.2 Extending Action Theory

The action theory presented in this chapter extends beyond existing approaches highlighted in Chapter 3, by allowing for the definition of actions with explicit notions of durations, pre-conditions, post-conditions, and terminating conditions, while incorporating an action log for action monitoring and failure detection. Specifically, an implementation in the GWENDOLEN programming language leverages existing semantics for goal suspension whilst actions are executing, meaning agents can continue operating throughout the duration of action executions.

Following Dennis & Fisher (2014), actions are associated with three terminating conditions: *Success*, *Failure*, and *Abort*. These conditions are represented in an extended action notation as:

$$a : (\phi_s, \phi_f, \phi_a)$$

In this extended action notation, a is an action, ϕ_s is the success condition, ϕ_f is the failure condition and ϕ_a is an *abort* condition after which the action should be aborted. The agent has also been extended to maintain an *action log* that tracks action successes, failures and aborts. It is used to determine when an action is *Suspect* or *Deprecated* (Stringer et al. 2020).

When an action is executed, either it is an instantaneous action (in which case one of its termination conditions is either instantly or trivially true), or it has a duration and the current intention is suspended. This follows Dennis & Fisher’s semantics where the goal is suspended since GWENDOLEN’s intention life-cycles map onto the goal-lifecycles of Harland et al. (2014).

5.2.1 Managing actions with terminating conditions

In the following sections, operational semantic rules for actions with durations and terminating conditions are presented. These semantics extend the abstract semantics presented by Dennis & Fisher (2014) and have been applied to the operational semantic

rules of the GWENDOLEN programming language with the intention of implementation. In the extended semantics, actions with explicitly defined durations that also have terminating conditions are considered, and the concept of an *action log* is introduced.

The notational conventions used in the rules are presented in Table 5.1 for reference.

Symbol	Explanation
$a : (\phi_s, \phi_f, \phi_a)$	An action a with success condition ϕ_s , failure condition ϕ_f , and an abort condition ϕ_a .
a	An action performed by the agent.
ϕ_s	The success condition for an action.
ϕ_f	The failure condition for an action.
ϕ_a	The abort condition for an action.
i	The current intention of the agent.
ξ, ξ'	The agent's state before and after an action, respectively.
B	The belief base of the agent.
$\text{suspend}(\text{executing}(a) : \text{tl}_i(i))$	The suspension of an intention with an action marked as executing .
$\text{executing}(a : (\phi_s, \phi_f, \phi_a))$	An action that is currently being executed.
$\text{do}(\text{abort}(a))$	The aborting of an action a .
L	The action log.
\oplus	Used to denote appending a new element (such as a tuple) to a list.
$\Delta(B, B')$	The difference in beliefs in the belief base before and after an action.
$\text{tl}_i(i)$	The tail of the intention stack, excluding the top element.
$\text{hd}_d(i)$	The top deed of the current intention.
$\rightarrow_{\text{action}}$	Denotes the transition of the agent's state as a result of performing an action.
$B \models \phi_s$	The belief base contains the beliefs stated in the success conditions of an action

Table 5.1: Notation conventions for Action Theory Extensions

Executing actions with terminating conditions

When an action with an expected duration is executed, it is likely that none of the termination conditions are believed instantaneously. In this case, the intention with the action marked as **executing** is suspended, as shown in Rule (5.1).

$$\frac{\text{hd}_d(i) = \mathbf{executing}(a : (\phi_s, \phi_f, \phi_a)) \quad \xi \xrightarrow{\mathbf{do}(a)} \xi' \quad B \not\models \phi_s \quad B \not\models \phi_f}{\langle \xi, \langle \dots i, B \dots \rangle \rangle \rightarrow_{\text{action}} \langle \xi', \langle \dots \mathbf{suspend}(\mathbf{executing}(a) : \text{tl}_i(i)), B \dots \rangle \rangle} \quad (5.1)$$

The top deed on the intention stack, $\text{hd}_d(i)$, is an action with a success, failure and abort condition, $a : (\phi_s, \phi_f, \phi_a)$, which has been executed in the current state, ξ . In this case, the agent does not believe the success, $B \not\models \phi_s$, and failure conditions for the action, $B \not\models \phi_f$. It should be noted that in this set of rules, only the intention, i , and belief base, B , are shown, with unused agent features represented by ‘...’. A tuple containing the current state of the environment and a tuple that represents the agent’s state,

$$\langle \xi, \langle \dots i, B \dots \rangle \rangle$$

describes the current state of the system. After the agent’s state has been changed by performing the action, $\rightarrow_{\text{action}}$, the intention containing the executing action is suspended by performing the `suspend` operation on the intention that contains the executing action. The notation, $:$, used in the intention, indicates the concatenation of a deed to the top of an intention stack:

$$\mathbf{suspend}(\mathbf{executing}(a) : \text{tl}_i(i)).$$

A suspended intention is, by default, not selected at the intention selection phase of the agent’s reasoning (Dennis 2017). Following the original GWENDOLEN semantics, an intention will remain suspended until some belief condition occurs, normally that a belief is acquired via perception or from the receipt of a message. In the case of actions with terminating conditions, their intention is unsuspended if any of the termination conditions are perceived.

When success, failure, and abort conditions are encountered, for any action execution, an entry containing the action, a , the difference in beliefs from before and after the action execution, $\Delta(B, B')$, and the type of action outcome (‘success’, ‘failure’, or ‘abort’) is added to the *action log*. This is shown in the semantics by:

$$L \oplus \langle a, \Delta(B, B'), \text{‘outcome’} \rangle$$

Where the action log is represented by L , and an action log entry is represented as:

$$\langle a, \Delta(B, B'), \text{'outcome'} \rangle$$

The notation, \oplus , signifies a new entry being appended to the action log.

In the case of failure conditions, once the outcome has been logged, the action is retried. The transition rule (2.7) for unsuspending intentions from the original GWENDOLEN semantics (see Appendix A) is left unchanged, and is shown in Rule (5.2), for reference.

$$\frac{\text{hd}_d(i) = +b}{\langle \xi, \langle \dots i, I, B \dots \rangle \rangle \xrightarrow{\text{add_belief}} \langle \xi, \langle \dots \text{tl}_i(i), \mathbf{unsuspend}(I, b) \cup_{\text{new}}(+b, \epsilon), B \cup \{b\}, \dots \rangle \rangle} \quad (5.2)$$

After this, Rules (5.3), (5.5) and (5.7), described below, are used to process the rest of the intention and update the action log. In the case of failure terminations, the action is attempted again, and in the case of abort terminations, the action execution is aborted in the environment.

Abort conditions are intended to stop the agent from retrying actions when an action fails significantly enough to be deemed unsafe to reattempt. Failure conditions can then be used to define conditions that represent a tolerance for what is considered acceptable performance, allowing the agent to continue if the action still performs within an acceptable range. Success conditions are then used to explicitly define when an action has achieved its intended outcome, ensuring the agent can recognise and confirm successful execution. A richer set of tools for reacting to failures and aborts, involving the analysis of the *action log*, is discussed in Chapter 6.

Section 5.3 works through an example that follows the operational semantics presented in this section, including example action logs for successes, failures and abort outcomes in Figures 5.2, 5.3, and 5.4.

Terminating with success

In Rule (5.3), a durative action is executing, and the agent believes the success condition.

$$\frac{\text{hd}_d(i) = \mathbf{executing}(a : (\phi_s, \phi_f, \phi_a)) \quad B \models \phi_s}{\langle \xi, \langle \dots i, B, \dots, L \rangle \rangle \xrightarrow{\text{action}} \langle \xi', \langle \dots \text{tl}_i(i), B', \dots, L \oplus \langle a, \Delta(B, B'), \text{'success'} \rangle \rangle \rangle} \quad (5.3)$$

Rule (5.4) shows the case where the success condition is believed to be true at the

point of action execution, such as when an instantaneous action is executed e.g. taking a picture. Rule (5.4) is very similar to Rule (5.3), as both are concerned with the success condition of the action being believed to be true, the difference between the two rules is the time frame in which the conditions are believed: in Rule (5.3), the agent believes the success conditions during the execution of the durative action, whereas in Rule (5.4), the agent believes the success conditions immediately, at the point of action execution. Both transitions result in the same outcome. Rule (5.4) also behaves similarly to Rule (2.4), since the situation where an action achieves its expected outcome is analogous to action execution in the existing GWENDOLEN semantics, but with the addition of an action log. In Rule (5.4), the top deed of the intention stack is an action, and performing the action transforms the pair of the environment and the tuple of agent components by changing the environment and removing the top deed of the current intention.

$$\frac{\text{hd}_d(i) = a : (\phi_s, \phi_f, \phi_a) \quad \xi \xrightarrow{\text{do}(a)} \xi' \quad B \models \phi_s}{\langle \xi, \langle \dots i, B, \dots, L \rangle \rangle \rightarrow_{\text{action}} \langle \xi', \langle \dots \tau 1_i(i), B', \dots, L \oplus \langle a, \Delta(B, B'), \text{'success'} \rangle \rangle \rangle} \quad (5.4)$$

Terminating with failure

In Rule (5.5), a durative action is executing, and now the agent believes the failure conditions. The action failure is recorded in an entry in the action log, then the action is concatenated to the tail of the intention stack to be re-attempted. Rule (5.5) is similar to Rule (5.4) in that the agent believes in a terminating condition during the execution of a durative action, and both record the outcome and the change in conditions they experienced during execution in the action log. However, as the agent believes the failure conditions have been met in Rule (5.5), instead of removing the top deed of the intention like when the agent believes in the success conditions of an action, the failed action is concatenated to the tail of the intention stack to be re-attempted.

$$\frac{\text{hd}_d(i) = \mathbf{executing}(a : (\phi_s, \phi_f, \phi_a)) \quad B \models \phi_f}{\langle \xi, \langle \dots i, B, \dots, L \rangle \rangle \rightarrow_{\text{action}} \langle \xi, \langle \dots a : (\phi_s, \phi_f, \phi_a) : \tau 1_i(i), B', \dots, L \oplus \langle a, \Delta(B, B'), \text{'failure'} \rangle \rangle \rangle} \quad (5.5)$$

In Rule (5.6), the failure condition is believed to be true at the point of action execution. The same operation as in Rule (5.5) is performed. The failure is logged in the *action log* and pushed onto the tail of the intention stack to be re-attempted.

$$\begin{array}{c}
\text{hd}_d(i) = a : (\phi_s, \phi_f, \phi_a) \quad \xi \xrightarrow{\text{do}(a)} \xi' \quad B \models \phi_f \\
\hline
\langle \xi, \langle \dots i, B, \dots, L \rangle \rangle \rightarrow_{\text{action}} \\
\langle \xi, \langle \dots a : (\phi_s, \phi_f, \phi_a) : \text{tl}_i(i), B', \dots, L \oplus \langle a, \Delta(B, B'), \text{'failure'} \rangle \rangle \rangle
\end{array} \quad (5.6)$$

Terminating with the abort condition

Rule (5.7) shows the case where an action is executing, and now the agent believes that the abort condition is true. In this case, the action is immediately aborted in the environment, $\text{do}(\text{abort}(a))$, the abort outcome is logged in the *action log*, and the action is not attempted again.

$$\begin{array}{c}
\text{hd}_d(i) = \mathbf{executing}(a : (\phi_s, \phi_f, \phi_a)) \quad B \models \phi_a \\
\quad \xi \xrightarrow{\text{do}(\text{abort}(a))} \xi' \\
\hline
\langle \xi, \langle \dots i, B, \dots, L \rangle \rangle \rightarrow_{\text{action}} \\
\langle \xi', \langle \dots \text{tl}_i(i), B', \dots, L \oplus \langle a, \Delta(B, B'), \text{'abort'} \rangle \rangle \rangle
\end{array} \quad (5.7)$$

In Rule (5.8), the abort condition is believed to be true at the point of action execution.

$$\begin{array}{c}
\text{hd}_d(i) = a : (\phi_s, \phi_f, \phi_a) \quad \xi \xrightarrow{\text{do}(a)} \xi' \quad B \models \phi_a \\
\hline
\langle \xi, \langle \dots i, B, \dots, L \rangle \rangle \rightarrow_{\text{action}} \\
\langle \xi', \langle \dots \text{tl}_i(i), B', \dots, L \oplus \langle a, \Delta(B, B'), \text{'abort'} \rangle \rangle \rangle
\end{array} \quad (5.8)$$

In this case, the action cannot be aborted as it completed instantaneously and is not still executing in the environment, though the action's deed is removed from the current intention to prevent the action from being re-attempted. The abort outcome is then logged in the *action log*. Rule (5.7) handles all other cases where an agent believes an action's abort conditions and the action has not completed instantaneously.

5.3 Example

The operation of the semantics presented in this chapter can be illustrated in a simple example. In this example, a wheeled inspection robot is tasked to navigate around a space represented as a topological map and take images at specific locations in that map. Specifically, this example considers a robot moving through a hallway towards a fire exit, where it is supposed to log an image of the fire exit (for later human inspection to verify that the fire exit is clear). However, the hallway is constructed in such a

way that, particularly at times when many people are moving through it, the obstacle avoidance behaviour of the robot means it sometimes ends up at the wrong location — the entrance door, at the opposite end of the hallway, and sometimes the movement takes far longer than the expected duration (5 minutes) of the move.

Therefore two actions are considered, one with a duration (*move_to_fire_exit*) and one without (*take_image*). Their success, failure, and abort conditions and durations are:

$$\textit{move_to_fire_exit} : (\{at(\textit{fire_exit})\}, \{\neg(at(\textit{fire_exit}))\}, \{time(a,t) \geq 300\})[300]$$

$$\textit{take_image} : (\{\top\}, \{\perp\}, \{\perp\})[0]$$

The definition of their terminating conditions follows the format:

$$\textit{action} : (\{\textit{success_condition}\}, \{\textit{failure_condition}\}, \{\textit{abort_condition}\})[\textit{duration}]$$

The success condition for the *move_to_fire_exit* action is met if *at(fire_exit)* is believed, the failure condition is met if $\neg(at(fire_exit))$ is believed, the abort condition is met if the time taken *t*, to execute the action *a*, is over 5 minutes (300 seconds), $time(a,t) > 300$, and the expected duration is set to 300 seconds. The terminating conditions for the *take_image* action are trivial placeholder conditions as it is an instantaneous action. In the description of the conditions, the success condition is set to True, \top , the failure condition is set to False, \perp , the abort condition is set to False, \perp , and the action duration is set to 0.

The agent's intention stack is represented as a stack of deeds, ignoring some of the other information GWENDOLEN stores in intentions which is irrelevant here. At the point where it starts traversing the hallway, it has two deeds on the stack – the action to move to the fire exit, followed by the action to take an image. These deeds were triggered by the commitment to a *maintenance* goal, which is shown in the corresponding event stack, and is indicated by the standard BDI syntax for committing to achieve a goal: $+!g$.

event	deed
$+!maintenance()$	<i>move_to_fire_exit</i>
$+!maintenance()$	<i>take_image</i>

The agent attempts to execute *move_to_fire_exit*. At this point, the agent is neither at the fire exit nor at the entrance meaning neither the success or failure conditions

are believed for the `move_to_fire_exit` action. As neither the success or failure conditions are believed, the process described by Rule (5.1) is followed: the intention is suspended and the `move_to_fire_exit` action is marked as executing). The intention becomes:

event	deed
Suspended: <code>!maintenance()</code>	executing (<code>move_to_fire_exit</code>)
Suspended: <code>!maintenance()</code>	<code>take_image</code>

Three things may now happen.

1. Firstly, the robot may reach the fire exit. The agent perceives *at*(`fire_exit`) and the intention is unsuspended. At this point the agent makes a transition in accordance with (5.3) and the intention becomes:

event	deed
<code>!maintenance()</code>	<code>take_image</code>

The agent may then execute `take_image`. This is an instantaneous action with a trivial success condition. A transition occurs according to (5.4) and the intention becomes:

event	deed

This intention is now complete and will be cleared away as part of the rest of the reasoning process. The resulting action log is shown in Figure 5.2, with two entries, where a green background represents a success entry. Each action that was performed during maintenance has been logged, with the change in beliefs from before the action executed compared to after execution and the outcome of the action.

2. Secondly, the robot may reach the entrance. The agent perceives *at*(`entrance`) and the intention is unsuspended following the process of Rule (5.2). At this point the agent makes a transition in accordance with (5.5) and the intention becomes:

event	deed
<code>!maintenance()</code>	<code>move_to_fire_exit</code>
<code>!maintenance()</code>	<code>take_image</code>

Action Log		
Capability	Change in beliefs	Action Outcome
move_to_fire_exit	+at(fire_exit)	Success
take_image	⊥	Success

Figure 5.2: Action log for an inspection robot in a hallway environment with successful actions

At the same time, the failure log is updated to note the failure of `move_to_fire_exit`. The original state has been reached again and the action will be re-attempted. Figure 5.3 shows an example of the log from this action outcome, with a blue background representing a failure entry.

Action Log		
Capability	Change in beliefs	Action Outcome
move_to_fire_exit	+at(entrance)	Failure

Figure 5.3: Action log for an inspection robot in a hallway environment with a failed action

- Thirdly, after five minutes the robot may have reached neither the fire exit nor the entrance and may still be attempting to move through the hallway. The agent receives the percept $time(\text{move_to_fire_exit}, n)$ where $n \geq 300$, meaning that the time that the `move_to_fire_exit` action has been executing has surpassed the expected duration of the action, and the intention is unsuspending. At this point, the agent makes a transition in accordance with (5.7). The agent performs the action, `abort(move_to_fire_exit)` and the intention becomes:

event	deed
+!maintenance()	move_to_fire_exit
+!maintenance()	take_image

At the same time, the abort log is updated to note the abort of

`move_to_fire_exit`. Figure 5.4 shows an example of the log from this action outcome, with a red background to represent an abort entry.

Action Log		
Capability	Change in beliefs	Action Outcome
<code>move_to_fire_exit</code>	<code>time(move_to_fire_exit, n)</code>	Abort

Figure 5.4: Action log for an inspection robot in a hallway environment with an aborted action

5.4 Implementing Durative Actions

The semantics presented in this chapter show the operations of actions with terminating conditions and durations as applied to the existing GWENDOLEN semantics. To integrate the extended action theory with the Java implementation of GWENDOLEN, extensions to the underlying Java code were required. It should be noted that GWENDOLEN is distributed as part of the Model-Checking Agent Programming Languages (MCAPL) framework, and the most recently available MCAPL distribution of GWENDOLEN supports *Capabilities* (actions with defined pre-conditions and post-conditions).

The *Capabilities* class was extended to support durations, and extended further to handle defining success, failure, and abort conditions. The standard environment class was extended to handle actions with defined durations, and finally, the reasoning rule for handling standard instantaneous actions was extended to handle actions with defined durations.

Durative Actions Class This class is an extension of an existing *Capabilities* class in the MCAPL distribution of GWENDOLEN, which is an extension of the *Action* class. Figure 5.5 shows a partial class diagram indicating the relationship between these classes. Three member variables were added to the standard *Capabilities* class; the three variables can be used to define the belief conditions for successes (*success*), failures (*failure*), and aborts (*abort*) for each action.

Durative Actions hold three more member variables, in addition to the *Capability* class: *duration*, which is filled with an integer value for the expected duration of the action; *threshold*, which holds an integer value representing the number of failures for

that action in the *Action Log* which will trigger the next stage of failure detection; and *actionstate*, a byte value which is updated automatically to represent when the action is executing or not executing in an environment.

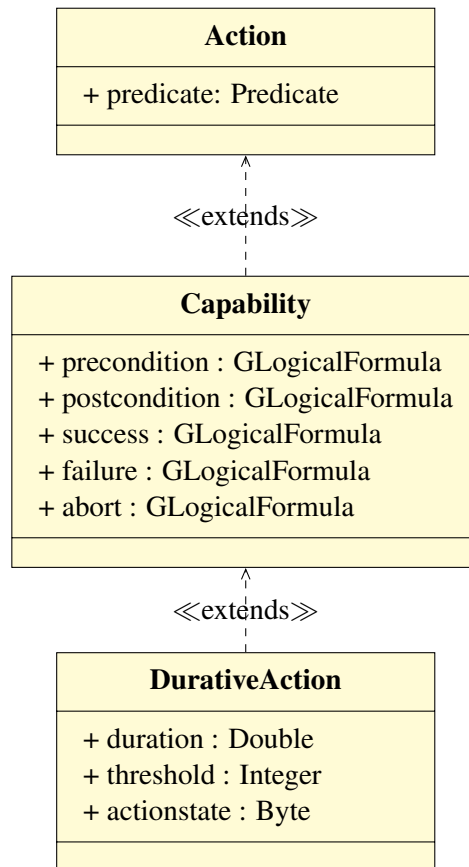


Figure 5.5: Durative Action - Partial Class Diagram

Durative Action Environment For action durations to be honoured by agents, considerations for time must be added to the default environment. A percept for time is kept in this environment and is updated automatically.

Handle Action Operational Rule In Stage D of the GWENDOLEN reasoning cycle (see Figure 4.3), the operational rule for handling actions can be selected and executed. Within the body of this rule, the method for executing actions in the environment is called. Before introducing the concept of Durative Actions with terminating conditions, actions were considered to complete instantaneously and could not fail. To manage various action outcomes, each action is now executed and followed until termination.

In Figure 5.6, the process for resolving each of the three action outcomes is shown. If the action execution terminates with failure, and the threshold number of failures has been reached for that action, then the learn rule is applied in the following stage in the reasoning cycle. See Chapter 6 for more details about the learn rule. Once a new action description is learned, the agent plans must be reconfigured before continuing normal action execution.

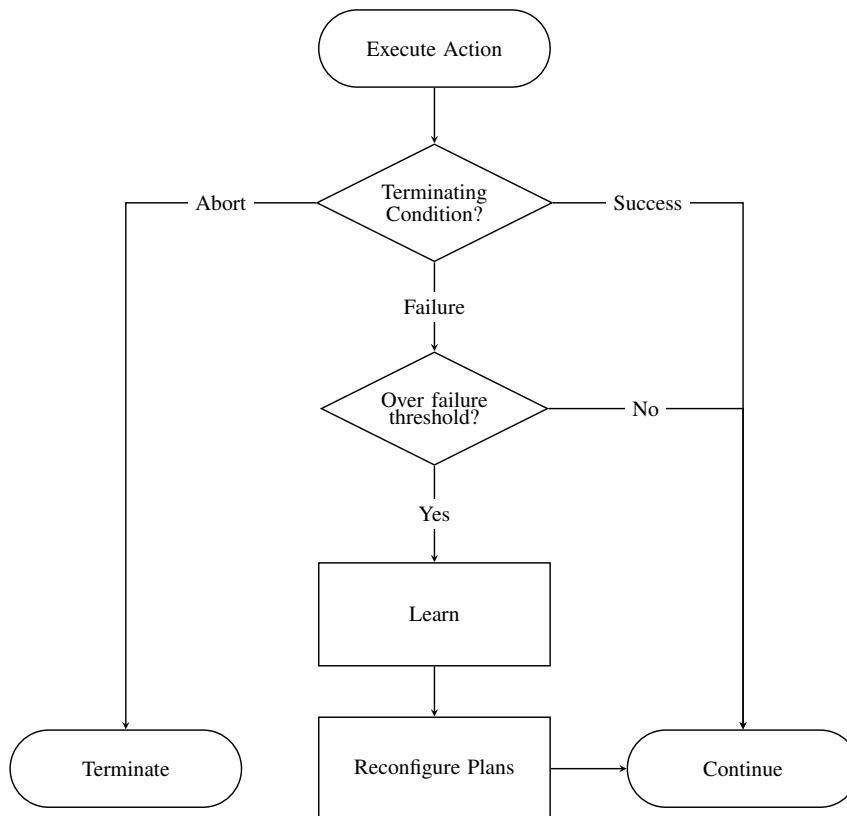


Figure 5.6: Handle Action Operational Rule

Chapter 6

Detecting Failures and Learning New Action Descriptions

Dynamic environments introduce uncertainties that can disrupt the operation of autonomous agents: obstacles may appear, environmental conditions may change, or the agent's own hardware may experience degradation. These unforeseen circumstances necessitate a mechanism for adapting to change. Recognising when an action is no longer consistently achieving its intended outcome is the first step towards enabling agents to adapt to dynamic environments. A system that is capable of detecting failure can flag deprecated actions and trigger corrective measures. New action descriptions that more accurately reflect the action's affect on the environment can be learned by leveraging the information recorded in the *Action Log*, and agent plan libraries can be reconfigured to ensure that agents can remain operational without intervention.

In this chapter, a method for detecting action failures is introduced, in Section 6.1. The underlying mechanisms are then detailed in Section 6.1.1, and the method is demonstrated using the running example in the waypoints environment. A method used for learning new action descriptions to replace failing action descriptions is then detailed in Section 6.2. The influences and origin of the chosen algorithm are also given. Lastly, specific details of the algorithm and its related mechanisms are justified and explained.

6.1 Failure Detection

Failure detection within the context of this research can be described as the monitoring of outcomes for individual action executions to keep an updated record of all recent

results with the ability to autonomously flag characteristic patterns of action deterioration. It follows that the ability to detect failure is a prerequisite for adapting to failure itself (Cardoso et al. 2019), as the failure must be identified in order to resolve it. In this research, the detection of failure relies upon comparing the actual outcomes with expected outcomes.

Identifying failures in robotic systems is important for maximising longevity. Frequently identifying and resolving issues through failure detection can prevent them from escalating into major problems that could terminate a robot's operational lifetime earlier than expected. Above this, detecting and diagnosing failures in any system has clear advantages over a system that does not. Failure detection can be most helpful when a system is subject to conditions that can not be predicted before deployment.

In this research, failure detection has been designed to run 'quietly' alongside normal agent execution so that it can be applied to existing systems without causing disruption to the agent's execution. Logs must be gathered through monitoring as it is not possible to collect detailed information on action execution in any other way. The value is seen when the collected data is used, not gathered. The collected data is used when enough entries that match certain criteria exist in the data set. The criteria are discussed in Section 6.1.1.

Whilst reasoning systems can replicate human decision-making, the resultant decisions are subject to two major assumptions: perceptions of the agent environment are always assumed to be accurate, and actions are assumed to perform as described before execution (Wooldridge 1999). If a human was tasked to detect failures in the operation of an autonomous agent, they would monitor the interaction between the agent's actions and the environment, keeping a note of past interactions and identifying emerging patterns of change. This behaviour can be reproduced programmatically by adding entries to a data structure containing the changes from before and after action executions. These entries can then be monitored after each action execution for patterns of change.

In this chapter, the concept of action logging and monitoring for failure detection is expanded and discussed. The explanation builds upon two previously discussed requirements from Chapter 5: actions with durations, and actions with post-conditions.

6.1.1 Failure Detection Method

Two prerequisites for detecting failing actions were considered in Chapter 5. Specifically, actions can have durations that 'time out', and actions can have conditions that

trigger their termination. Action durations and post-conditions are used when comparing the expected outcome of an action compared to the actual outcome after execution. The baseline comparison between the expected outcome and the actual outcome of an action execution is used to determine whether actions have succeeded or failed, and consequently failures cannot be detected without it. In Section 4.2, all requirements for the proposed extension and their positions in the current system architecture are explained. To make use of the action outcomes, a log of the action outcomes is collected. The ‘Action Log’ is the final prerequisite for failure detection.

Detecting Failures in the Action Log

At Stage D in the reasoning cycle, once the top deed on the current intention has been executed, the last entry of the action log is evaluated, to see if an action has become deprecated. If the last entry of the action log is an entry for anything other than an action failure, then the cycle continues to the next stage. However, if the last entry on the action log is an entry for an action that has failed, then the number of entries in the action log containing a failure for this specific action is compared with the failure threshold that has been defined for that action (in the action’s description). If it is determined that the threshold has been reached, the action is considered deprecated, and the cycle continues to the Learn rule in Stage D_1 of the reasoning cycle.

In Figure 6.1, five entries in an action log can be seen. An example that could produce this action log is given below. When a new entry is made by an agent following an action execution, the size of the log is recorded, and measured against the predefined size limit. In this case a single action success has been experienced for the *move(0, 1)* capability, followed by four failures. For this capability, in this case, the failure threshold is four. The learning algorithm will now consider the frequency, similarity and age weights to calculate a new action description to represent the consistent result seen in the recent executions.

Example

A scenario using the Mars rover agent presented by Cardoso et al. (2019) is used in this example to illustrate how Action Log entries are recorded. In the scenario, there is an autonomous rover navigating a topological map of Mars that contains waypoints. The goal in this scenario is for the rover to reach a specific waypoint on the topological map, and when it reaches the goal waypoint, the agent navigates back to waypoint 0, before another waypoint goal location is provided. If an action failure occurs, the

Action Log		
Capability	Change in beliefs	Action Outcome
{at(0)}{move(0, 1)}{at(1)}	-at(0), +at(1)	Success
{at(0)}{move(0, 1)}{at(1)}	-at(0), +at(3)	Failure
{at(0)}{move(0, 1)}{at(1)}	-at(0), +at(3)	Failure
{at(0)}{move(0, 1)}{at(1)}	-at(0), +at(3)	Failure
{at(0)}{move(0, 1)}{at(1)}	-at(0), +at(3)	Failure

Figure 6.1: Detecting Failures in an Action Log

agent has been instructed to return to its original state from before the failing action was executed.

During execution, the rover produced an action log. In Figure 6.1, a filtered version of the action log containing only the $move(0, 1)$ action is displayed, showing the change in the action's outcomes over time. The action log shows the logs recorded by the rover at the point where the $move(0, 1)$ action has reached the action failure threshold of four.

In the first row of the action log, the rover tried to navigate to waypoint 1 by executing the action $move(0, 1)$. It successfully moved and updated its belief to indicate that it was at waypoint 1 ($-at(0), +at(1)$). The action was marked as a "Success", and the rover continued on with its journey to the goal waypoint on the map.

In the second row of the action log, the rover had completed the previous mission to reach the goal waypoint and was back at waypoint 0. The new goal waypoint was assigned and the rover executed the $move(0, 1)$ action to start the route. This time, it encountered an obstacle. Due to its obstacle avoidance override behaviour, it moved to waypoint 3 instead ($-at(0), +at(3)$). This action execution was marked as a "Failure" in the action log as the action did not achieve its postconditions. The rover then returned to grid space 0, its original state from before the action failure.

In the third row of the action log, the rover, in its original state from before action failure, made another attempt to get to waypoint 1, by executing the $move(0, 1)$ action. It still encountered the obstacle and ended up at waypoint 3 again ($-at(0), +at(3)$). The action was marked as a "Failure" and the rover returned to waypoint 0 again.

In the fourth row of the action log, the rover made yet another attempt to move

to waypoint 1 ($move(0, 1)$), resulting in a failure due to the same obstacle. The rover found itself at waypoint 3 ($-at(0), +at(3)$) again, and the action was marked as a “Failure”, before the rover returned to waypoint 0 once again.

In the last row of the action log, the rover made a final attempt to move to waypoint 1 with the $move(0, 1)$ action. It still could not move past the obstacle and ended up at waypoint 3 ($-at(0), +at(3)$). The action was marked as a “Failure” in the action log.

Whilst the obstacle avoidance behaviour in this example is intentional, the action executions are marked as failures due to the actions not achieving their post-conditions. This approach ensures that any deviation from the expected outcome, even if intended and safe, is flagged and analysed. In this example, if the obstacle does not move and the obstacle avoidance behaviour is consistently taking the agent to grid space 3 when executing the $move(0, 1)$ action, the action log will reflect that this action now consistently takes the agent to grid space 3 instead.

The action log can then be used to inform the learning algorithm, in Stage D₁ of the reasoning cycle, so that the action description can be updated to accurately reflect its effect on the environment.

6.2 Stage D₁: Learning New Action Descriptions

Whilst considering the issue of learning new action descriptions, six existing types of learning algorithms were evaluated, in addition to learning methods developed for the AI Planning domain (Mugan & Kuipers 2011, Pasula et al. 2007). Algorithms were selected for consideration based on their suitability for learning action descriptions. Each of the selected algorithms is listed in Table 6.1 and then further analysed on their suitability for learning action descriptions in Section 6.2.1. The table headings were influenced by the comparison table from Arora et al.’s survey of learning algorithms in the AI planning domain (Arora et al. 2018).

In Table 6.1, the *Input* and *Output* columns state the required inputs for the algorithm, and the product that is generated by each. The *Technique* column states a short description of the method that is used by each algorithm. A list of benefits and limitations are then summarised in the *Benefits* and *Limitations* columns. Finally, the *Robust to Noise* column refers to the algorithm’s ability to maintain performance despite the presence of noise in the data. In this context, noise can be defined as any random error or variance in the data that can potentially mislead the learning process (Arora et al. 2018). For instance, random errors can be caused by readings from sensors that are

temporarily disturbed by dust or electromagnetic interference.

6.2.1 Algorithm Selection

The algorithm selection process was guided by three general requirements: the limited available data collected from failure detection, the requirement to prioritise safety, and keeping the computational complexity as low as possible.

The input data that is available for the learning algorithms to use is constructed from the entries in the Action Log. The Action Log contains an action description with pre-conditions and post-conditions, the outcome of the action execution determined by the action's stated terminating conditions, and the difference in beliefs from before and after the action's execution.

Some learning algorithms may employ exploratory behaviours to collect additional data although this could pose safety risks to the agent through the execution of actions where the outcomes are not yet known. There is a large range of inputs required by different learning algorithms to operate effectively. Unfortunately, the limited data collected by the system in this thesis reduces the number of algorithms that are suitable for the required application.

As the primary goal of this research is to achieve long-term autonomy, safety is considered a priority, and therefore features as a requirement for algorithm selection. Performing unsafe operations to learn new action descriptions could put agents in an unrecoverable state, consequently ending their autonomy. On the requirement of safety, it should be noted that actions always have the possibility of failing in unsafe ways, even when their expected outcome is known. In the system proposed in this thesis, it is assumed that the abort conditions stated in each action's description are defined to cover any condition that is deemed unsafe to continue from.

The third requirement of computation complexity refers to the amount of computational resources (time and memory) required to execute the algorithm. There is a balance between the output accuracy and the algorithm's resource requirements that is considered in this algorithm selection. For this reason, the resulting system should not be bound to the requirement of high-performance hardware to perform as intended, and should keep computational complexity as low as possible.

Inductive Learning Inductive learning uses pattern recognition to create generalisable rules from specific examples so that the rules can be applied to new and unseen

Learning Algorithm	Input	Output	Technique	Benefits	Limitations	Robust to Noise
Inductive Learning	Historical action logs	General rules describing action pre/post conditions	Pattern recognition in data	Good at finding underlying patterns; Requires less computational power	May oversimplify; Depends heavily on quality and diversity of data	Moderate
Reinforcement Learning	Feedback from the environment (reward function)	Policies maximising reward	Learning through trial and error	Adaptive to dynamic environments; Continuously improves with more data	Can require a lot of data; Potentially dangerous in safety-critical scenarios	Low to Moderate
Genetic Algorithms	Set of potential solutions	Optimised set of solutions (action rules)	Evolutionary algorithms	Good at exploring a wide state space; Effective in complex domains; Capable of optimal or near-optimal solutions	Computationally expensive, Potentially dangerous in safety-critical scenarios	High
Transfer Learning	Data from related tasks/domains	Knowledge applicable to new actions	Applying knowledge from one domain to another	Speeds up learning on new tasks; Efficient use of existing data	Performance dependent on similarity between domains; Requires access to relevant data from source domain	Varies (depends on source domain)
Supervised Learning	Labelled historical action logs	Predictive models for action outcomes	Learning from labelled examples	High accuracy with sufficient labelled data; Clear evaluation metrics	Requires large amounts of labelled data; May not generalise well to unseen data	Moderate to High
Rote Learning	Historical action outcomes	Replication of actions for known scenarios	Memorisation without 'understanding' underlying patterns	Fast recall for known scenarios; Simple to implement	No generalisation to new scenarios; Requires large memory for complex tasks	High

Table 6.1: Comparison of Learning Algorithms

situations. Inductive algorithms are good at finding underlying patterns in data (Russell & Norvig 2020), and they require less computational power compared to more complex algorithms (Mitchell 1997) such as genetic algorithms.

Inductive learning algorithms are also susceptible to over-generalisation, which could lead to the algorithm generating vague action descriptions if the input data is not diverse enough (Mitchell 1997). The results produced by this algorithm also heavily depend on the quality of the historical data (Russell & Norvig 2020). However, most of the algorithms discussed in this chapter are subject to the same limitation of being highly dependent on the quality of the historical data. Inductive learning algorithms are able to handle some level of noise by identifying consistent patterns, but excessive noise can lead to incorrect generalisations (Mitchell 1997).

An inductive learning algorithm may not be appropriate due to its dependence on high-quality and diverse data, which may not be available from the historical action log data in the case of the system proposed in this thesis.

Reinforcement Learning Reinforcement learning agents typically learn through trial and error, making decisions and receiving rewards and penalties for the outcomes of those decisions. Reinforcement learning is adaptive to dynamic environments and continuously improves with more data (Sutton & Barto 2018). However, learning through trial and error, and executing actions when the post-conditions are not already defined, could be dangerous for an agent in a safety-critical scenario, or lead to unpredictable results in a remote deployment. The reliance on a reward system severely limits the algorithm. Reinforcement learning algorithms struggle to learn when rewards are infrequent (Ng et al. 1999), such as in maintenance missions where an agent may only operate for short periods at a time to perform a maintenance route, then lying dormant for the rest of the time. The design of the reward function is also very important and must be designed and used to train the agent prior to execution, meaning the agent is susceptible to over-fitting to the specific reward structure they were trained with (Nguyen et al. 2020). Reinforcement learning can tolerate some noise in the input data by averaging over multiple trials, although significant noise can mislead the learning process when rewards are sparse.

If the environment could be simulated or a safe level of exploration could be guaranteed, then reinforcement learning could be applicable in this work. However, this would introduce even more computational overhead making it unsuitable for the proposed system requirements.

Genetic Algorithms Genetic learning algorithms mimic natural evolution, selecting and combining the best of a collection of solutions to produce a new generation of solutions until all of the solutions are considered to represent the same solution. Genetic algorithms are good at exploring a wide state space, making them effective in complex domains (Goldberg 1989). They are capable of optimal or near-optimal solutions by combining the best characteristics of existing solutions (Holland 1992). However, this type of algorithm works most effectively on large data sets, and are therefore computationally expensive if the optimal results are to be found (Goldberg 1989). Genetic algorithms handle noise effectively because they rely on population-based search and multiple generations to filter out the noise (Holland 1992). As state space exploration is the core mechanic in this algorithm, the agent may perform undefined experimental actions in order to find a solution. Executing actions where the consequences are not known can be dangerous. For example, if a genetic algorithm was used to discover the post-conditions of an action that is used to control a robotic arm in a nuclear power plant's waste categorisation room, the agent's exploration behaviour could lead to the robotic arm colliding with radioactive containers or causing a spill of hazardous materials, resulting in severe contamination and safety risks.

If the agent was in a position where it held little information about a very large and complex domain, and safety is not a priority, a genetic algorithm would be suitable. Though outside of these conditions, the safety implications and increased computation costs would outweigh the potential benefits.

Transfer Learning Transfer learning applies knowledge gained in one domain to a different, but related domain. It uses data from related tasks to inform knowledge applicable to new actions. Transfer learning speeds up learning on new tasks by leveraging existing knowledge (Pan & Yang 2010). It also makes efficient use of existing data from related domains (Weiss et al. 2016). This could mean that transfer learning would operate well in a safety-critical domain, as action descriptions would only be learned from existing and presumable safe knowledge. However, this method is only appropriate for scenarios where the algorithm has access to relevant data from the source domain (Pan & Yang 2010). The algorithm is not robust to noise, as the output of this algorithm is directly tied to the source domain's knowledge. If the source data is noisy, it will be unintentionally transferred to the target domain (Weiss et al. 2016).

Considering the primary aim of the extension presented in this thesis is to allow agents to be able to adapt to dynamic environments, it is assumed that the agent will

encounter and be able to deal with unknown action post-conditions. This is clearly not the intended use of this learning algorithm and is therefore not appropriate for the proposed system.

Supervised Learning Supervised learning involves creating models from complex datasets to predict the output of future inputs. This type of algorithm achieves high accuracy with sufficient labelled data, although they struggle to generalise learned policies to new data, which can limit their effectiveness in dynamic environments (Cunningham et al. 2008). Input noise can also degrade the performance of supervised learning algorithms, although noise filtering and regularisation techniques can be applied to reduce the effect (Zhang & Zhu 2019).

Supervised learning requires large labelled datasets to train a model effectively (Alloghani et al. 2020). The action log automates the data collection and labelling process, as the action terminating conditions and agent perception are used to categorise and label the action execution data. However, despite this automation, the agent would need to operate for a period long enough to collect enough data to train models with before they can be used. Considering this, there would likely not be enough data for each action, at least initially, to adequately inform a supervised learning algorithm without augmenting the existing data, which can lead to over-fitting to the augmented data (Li et al. 2019, Wong et al. 2016).

Rote Learning Rote learning could be considered the most simple implementation of machine learning when compared to the algorithms considered in this section. The learning technique simply involves keeping a record of historic data for a required application, and recalling the most appropriate record based on a set of selection criteria. For instance, for a customer service chatbot, rote learning could be used to store and recall standard responses to frequently asked questions, quickly providing users with relevant information without the need for repeating the complex language processing. Rote learning is highly robust to noise because it simply recalls exact matches based on selection criteria, unaffected by variations unless the noise level is extreme (Cohen & Feigenbaum 1982).

The technique is aimed at minimising processing time and cost for previously encountered scenarios, which is not the desired output for an action description learning algorithm. However, the action log provides a considerable record of historical data for a required application, and with the addition of a ranking policy, this learning method

could be applied to the proposed system.

Learning Algorithm Categorisation

The learning algorithm used by the system proposed in this thesis is a hybrid algorithm, combining features from supervised learning, rote learning, and ranking, which is a commonly used technique used by many machine learning algorithms (Chen et al. 2009). The action log, in this context, acts as the labelled data required by a supervised learning algorithm (Cunningham et al. 2008), and the memorised historical action outcomes required by rote learning (Cohen & Feigenbaum 1982). To determine the most suitable action description from the historical log, a weighting policy is used to rank the candidate action descriptions.

The hybrid algorithm described here minimises computational overhead and prioritises safety unlike the learning algorithms presented in the related works in Chapter 3. The hybrid algorithm does so without compromising on the quality of the outcome. The developed algorithm is described in the next section.

6.3 Machine Learning with GWENDOLEN

The GWENDOLEN programming language was used as a foundation for implementing the selected learning algorithm. In GWENDOLEN the operational semantics are defined by ‘rules’. These rules are separated into different stages of the reasoning cycle, depending on their operation. When a stage has been reached, the preconditions of each rule that has been assigned to that stage are examined, to check that the conditions for the rule to be executed have been met. If the preconditions for a rule are met, the logic in the rule is executed. If the preconditions for are not met, the cycle proceeds to the next stage.

To allow the agent to reason about learning new action descriptions for failing actions, the implementation of the learning algorithm is contained within an operational rule class that is assigned to stage D_1 of the proposed extended reasoning cycle. Similarly to the existing rules, the ‘Learn’ rule class implements two main methods that are required by the reasoning cycle. Most importantly, a method that is used to verify that the prerequisites to the rule have been met, and secondly, a method that applies the logic stated in the rule. The implementation of these two methods are described in the following section.

6.3.1 Agent learning process

Check Preconditions As previously stated, there are prerequisite processes that need to be performed before the learning algorithm can be executed. Firstly, the current state of the agent is recorded, and compared with the preconditions required to start the learning process. The action log is retrieved and the latest entry in the log is noted. If the latest entry in the action log meets the following criteria, then the learning algorithm is executed: the action recorded in the entry is a durative action, the action outcome is not a success, and the number of entries in the action log for that action is over the action's threshold value. If the latest entry in the action log does not satisfy the criteria, then the reasoning cycle continues onto the next stage.

Applying the rule Once the prerequisites have been met, three variables are initialised. A clone of the agent's environment is taken, and two sets of literals are established: one set holding the success conditions for the failing action, which are used as the goal state for the plan library reconfiguration described in Section 6.3.1; the other set of literals is populated later with the newly generated post-conditions.

The action log entries for the deprecated action are filtered into a separate log in reverse order in preparation for the ranking policy, and the failing action is recorded.

If all of the failure entries in the filtered action log are exactly equal (e.g., the action failed with exactly the same postconditions for every failure in the action log), then the next stage is skipped, as the post-conditions from one of the failure entries can be assigned as the new success post-conditions for this action and do not need to be ranked.

If the failure entries in the filtered action log are not exactly equal, each variation of action post-conditions is extracted into a new array list and assigned a weight, calculated from the index position in the single action log (representing recency with linear decay) and the frequency of that variation. The variation with the highest weight is assigned as the new success post-conditions for this action.

In either case, the learning outcome is stored, and the action's failure count is reset. If the action starts failing again, and the number of action failures in the action log is over the action's failure threshold, then the learning process is triggered again. If the learning process has been successful, the environment's capability library requires updating with the latest action description. The old description is removed and the new version is added.

Algorithm for Learning New Action Descriptions

Algorithm 2 presents the pseudo-code for applying the ‘Learn’ rule described above, whereby a new action description is synthesised by taking all the failed instances of the deprecated action from the action log, and determining which of their post-conditions most accurately reflect the actual action outcome.

A list (probably containing duplicates, as can be observed from Figure 6.1) of new candidate post-conditions for the deprecated action is produced by extracting all of the post-conditions from its action failure entries in the action log. The new candidate post-conditions consist of the changes in beliefs from before the action executed compared to after the action executed. Each item in this list is assigned a weight score based on how recent the item is. The weights for identical items are then summed and that with the highest score selected as the new post-condition for the action. Pseudo-code for this process is shown in Algorithm 2.

Algorithm 2: Algorithm for synthesising post-conditions when an action is detected to be deprecated.

```

1 if action is deprecated then
2    $n \leftarrow 1$ ;
3    $post\_scores \leftarrow \{\}$  // map of post-conditions to scores
4    $action\_log \leftarrow reverse(action\_log(action))$ 
5   for  $entry \in action\_log$  do
6     // NB. the action log consists of tuples (action,
7     // change in beliefs, outcome)
8     if ( $entry[0] = action$ ) & ( $entry[2] = Failure$ ) then
9        $post\_scores[entry[1]] \leftarrow post\_scores[entry[1]] + n$ ;
10       $n \leftarrow n + 1$ 
11    $best \leftarrow 0$ ;
12   for  $post \in keys(post\_scores)$  do
13     if  $post\_scores[post] > best$  then
14        $best \leftarrow post$ 

```

Line 2 instantiates the initial weight score (n) to 1, and in Line 3 it sets $post_scores$ to an empty map. In Line 4, the action log is reversed to ensure the entries are processed from oldest to newest. Lines 5–8 will loop through every entry in the action log to find entries that match with the deprecated action (same action) and where the outcome of the entry was reported as a failure. When this happens, the post-conditions of the action are added to the $post_scores$ map along with the weight score, which is then

incremented by one for future iterations of the action log. In line 9, *best* is initialised as 0. Lines 10–12 iterates over the keys in the *post_scores* map to select the candidate post-condition with the highest weight score.

If the action log in Figure 6.1 is considered and the failure threshold is four, then the agent’s ‘act’ phase should now attempt to synthesise a new action description from the log. It extracts the list of failures which, in the action log in Figure 6.1, contains four items all of which have identical new post-conditions — namely $\{-at(0), +at(3)\}$. This therefore becomes the new post-condition for the action *move(0, 1)*.

However, consider an example where the action log is more variable, such as in Figure 6.2.

Action Log		
Action	Change in Beliefs	Action Outcome
$\{at(0)\}\{move(0, 1)\}\{at(1)\}$	$-at(0), +at(1)$	Success
$\{at(0)\}\{move(0, 1)\}\{at(1)\}$	$-at(0), +at(3)$	Failure
$\{at(0)\}\{move(0, 1)\}\{at(1)\}$	$-at(0), +at(3)$	Failure
$\{at(0)\}\{move(0, 1)\}\{at(1)\}$		Failure
$\{at(0)\}\{move(0, 1)\}\{at(1)\}$		Failure

Figure 6.2: Example of an action log with variable post-conditions for the same action (*move(0, 1)*)

For example, the agent may have employed an obstacle avoidance algorithm to circumnavigate the obstacle between waypoint 0 and 1 but this has resulted in the agent arriving at waypoint 3 instead, but suppose the obstacle has become more serious — perhaps sand and debris is piling up as the result of storms — and now the low-level movement behaviour causes a failure that returns the agent to waypoint 0. This would result in the action log shown in Figure 6.2, where in the last two entries there is no perceived change in beliefs, as the agent has stayed at its original state, or returned to its original state from before the action had been executed. This could be the result of encountering an obstacle that cannot be circumnavigated by obstacle avoidance.

Table 6.3 shows a list of candidate post-conditions, extracted from the action log,

Candidate Post-Condition	Weight
$\{-at(0), +at(3)\}$	1
$\{-at(0), +at(3)\}$	2
$\{\}$	3
$\{\}$	4

Figure 6.3: Post-conditions extracted from Figure 6.2, added with their respective weights which are calculated based on how recent they are.

weighted by how recent they are. Identical post-conditions in the table are individually weighted first and then summed to make a total weight, to consider the entry’s frequency in the action log in the ranking policy. Entries in the action log are extracted from the oldest entry first, into the candidate post-condition table, and assigned a weight value that is incremented by 1 for each entry that is extracted.

Of the two candidate post-conditions $\{-at(0), +at(3)\}$ has a total weight of 3, while $\{\}$ (no change) has a total weight of 7. Therefore, for the action log presented in Figure 6.2 the empty post-condition, $\{\}$, is selected for the new action description.

6.3.2 Plan Library Reconfiguration with Learned Action Descriptions

Once a new action description is stored, a plan reconfiguration mechanism can be used to patch any plans containing the action. Cardoso et al. (2019) describe how an AI planning problem can be extracted from two inputs: the failing action description, and a list of action descriptions for all actions in the domain (including learned action descriptions and excluding the failing action description). The AI planning process is constructed by a process of:

1. using the failed action’s pre- and post-conditions as initial and goal states respectively for the planning problem; and
2. using the set of all other action descriptions as an action model for the planner.

This planning problem can then be solved to create a “patch” for any BDI plan containing the failed action. The proposed framework uses this mechanism by supplying the action description of the action that has been flagged as deprecated after some pre-defined number of failures. The set of action descriptions sent to the planner is then

```

+!at(1) : {at(0)} ← move(0, 1), +!at(2);
+!at(2) : {at(1)} ← move(1, 2), +!at(3);
+!at(3) : {at(2)} ← move(2, 3), +!at(0);
+!at(0) : {at(3)} ← move(3, 0), +!at(1);

```

Figure 6.4: Four GWENDOLEN plans for a patrolling robot

created from the agent's current set of action descriptions, including the newly learned description of the deprecated action.

The action log from Figure 6.1 is used to show the base process of reconfiguring the plan library when an action description is learned that can be used in a replacement patch. In this example, the $move(0, 1)$ action has become deprecated. Attempts to move from waypoint 0 to waypoint 1 now result in the agent arriving at waypoint 3 (based on the action log from Figure 6.1). A STRIPS-type planner (Fikes & Nilsson 1971) is called with the updated action descriptions and an initial planning state — $at(0)$ (the agent is at waypoint 0) — and goal state — $at(1) \ \& \ \neg at(0)$ (the agent should end up at waypoint 1) — created from the pre-conditions and post-conditions of $move(0, 1)$. Along with the action descriptions for the unchanged $move$ actions, the planner has the new description for $move(0, 1)$ available:

$$\{at(0)\}move(0, 1)\{-at(0), +at(3)\}.$$

An action describing a move from waypoint 3 to waypoint 1 is also available:

$$\{at(3)\}move(3, 1)\{-at(3), +at(1)\}.$$

In this case, it is straightforward for the planner to create the plan $move(0, 1), move(3, 1)$ to solve this problem (note that $move(0, 1)$ now takes us to waypoint 3, not waypoint 1). If plans similar to the GWENDOLEN plans in Figure 6.4 were used, this means that the plan

$$+!at(1) : \{at(0)\} \leftarrow move(0, 1), +!at(2)$$

contains the deprecated action and will not succeed in moving the agent to waypoint 1. This patch produced by the planner, replaces the appearance of $move(0, 1)$ in the original plan, now that $move(0, 1)$ results in the robot ending up at waypoint 3. This produces the new plan:

$$+!at(1) : \{at(0)\} \leftarrow move(0, 1), move(3, 1), +!at(2)$$

which is stored for reuse.

If the same plans from Figure 6.4 were available to the planner, but instead a new action description with no post-conditions has been learned using the action log from Figure 6.2, the planner cannot find a viable patch for the action as there is no sequence of other actions in the plan library that can navigate through any path to waypoint 1 from waypoint 0. However, if additional actions were available, such as:

$$\{at(0)\}move(0,2)\{-at(0),+at(2)\}$$

$$\{at(2)\}move(2,1)\{-at(2),+at(1)\}$$

The failing plan could be patched to produce the following reconfigured plan:

$$+!at(1) : \{at(0)\} \leftarrow move(0,2),move(2,1),+!at(2)$$

This patched plan is then added back to the plan library, allowing the agent to navigate to waypoint 1 from waypoint 0.

Chapter 7

Evaluation

In this chapter, a suite of simple tests is used to validate the implementation of the learning algorithm proposed in Chapter 6 against a set of requirements, listed in Table 7.1, confirming core functionality like handling action failures and partial action successes. Then, in Sections 7.2 and 7.3, two simulated environments are introduced with visual representations, along with the experiment scenarios used to test the system proposed in Chapter 4 against the evaluation criteria outlined in Section 7.1.3.

Additional experiments that compare time and memory costs were also performed using the outputs of the simulated environment experiments, to isolate the AI planner's time and memory costs for an evaluation of the computational efficiency of single-plan patching against a full re-planning of the plan library.

Finally, in Section 7.5, the findings of the simulated experiment scenarios are presented, before the results of the reconfiguration cost experiments are discussed. Both sets of experiment results are then analysed against the evaluation criteria, highlighting both the system's adaptability in learning action descriptions and streamlined computational efficiency.

7.1 Methodology

The methodology for evaluation was framed around assessing a fundamental set of five criteria: success rate, robustness, scalability, time cost, and memory cost. The formal evaluation was preceded by simple testing, executed throughout the implementation process to ensure the implementation operated as described in the theory before it was evaluated with simulated environments.

7.1.1 Simple Testing

Throughout the implementation discussed in Chapters 5 and 6, a suite of trivial tests was used to ensure the correct operation of the underlying code when compared to the theory. Each test was designed to check the outputs of different components in the agent source code. The simplest model of each scenario was constructed and the testing data was kept trivial for each; the pass or fail outcome of the tests quickly diagnosed any failing components. The combined suite of tests only covered the core use cases of the system during implementation, as intended. Whereas the simulation experiments covered a wider range of scenarios, in more realistic environments.

Each test is shown in Table 7.1 with a description of the scenario that is being tested; the original action description before any change occurs; the expected action description replacement from the learning algorithm; and finally, an explanation of the expected result. Action descriptions in Table 7.1 are represented in the format described in Section 2.1.6:

$$\{pre - conditions\}predicate(terms)\{post - conditions\}[duration]$$

In Table 7.1, scenario 1 describes a situation where an action no longer produces the expected result, specifically where the post-condition $p2$ is not detected by the agent after the action's duration has lapsed. Scenario 2 addresses the case of a partially successful action after the action duration has completed, where the agent detects some but not all of the expected post-conditions. In scenario 3, after action execution, the agent believes that a different outcome has been achieved compared to the outcome that was expected. Scenario 4 describes the case for actions that fail under certain variable conditions. Here the action $a(N)$ now affects $p1$ by $N/2$ instead of N . Scenario 5 highlights the issue of timing out, where an action takes longer than expected. Lastly, Scenario 6 considers actions resulting in values outside the expected range.

7.1.2 Experimental Evaluation

The majority of the evaluation for the system created in this thesis was based on the results of experiments performed in two simulated domains: the waypoints environments, where an agent must navigate through a series of waypoints on a topological map; and the hallway environment, where an agent must navigate a route through a number of 'rooms' to reach a final area to take a picture of an emergency exit, representing the actions of a maintenance robot. The two experiment domains were chosen

No.	Name	Original Description	Failure Description	Change
1	Action no longer has result	$\{p1\}a\{p2\}[[1s]$	$\{p1\}a\{\}\{[1s]$	p2 no longer detected by the agent after duration
2	Action is partially successful	$\{p1\}a\{p2,p3\}[[1s]$	$\{p1\}a\{p2\}[[1s]$	p3 no longer detected by the agent after duration
3	Action has a different outcome	$\{p1\}a\{p2\}[[1s]$	$\{p1\}a\{p3\}[[1s]$	p2 no longer detected by the agent, p3 is believed by the agent after duration
4	Action fails for some variables	$\{p1(X,Y)\}a(N)\{p1(X, (Y + N))\}[[10s]$	$\{p1(X,Y)\}a(N)\{p1(X, (Y + N/2))\}[[10s]$	a(N) now affects p1 by N/2
5	Action times out	$\{p1\}a\{p2\}[[1s]$	$\{p1\}a\{p2\}[[2s]$	a now takes 2s to achieve p2
6	Action outside expected range	$\{p1\}a\{p2(N)\}[[1s]$	$\{p1\}a\{p2(N + X)\}()$	a results in a value outside of the accepted range for p2

Table 7.1: Table of simple test scenarios for agile development.

to test against rising failure rates. A simulated environment with a navigation task provides a basic benchmark for measurements, whilst allowing random and varied failures to be introduced in a controlled and reproducible manner.

The waypoints environment was inspired by a ‘gridworld’ domain with an agent-controlled Mars rover, proposed by Cardoso et al. (2019), and designed to assess an agent’s ability to overcome action failures.

The hallway environment builds on the waypoints environment by decreasing the connectivity between the available states, allowing for a comparison of how the proposed system handles reduced domain connectivity.

Both simulated environments were also designed with the intention of developing physical counterparts using real sensor data to validate system performance in real-world conditions.

Each experiment covered different evaluation criteria and is discussed in the corresponding experiment descriptions.

7.1.3 Evaluation Criteria

The full list of evaluation criteria were:

Success Rate: The average completion percentage, measuring successful executions over the total number of executions.

Robustness Robustness is assessed as the ability to manage reduced domain connectivity, and high failure rates.

Scalability Scalability is assessed as the system’s ability to remain functional and maintain performance when handling larger domains. In this case, scalability is measured by comparing the success rate, time cost and memory cost between differently sized domains.

Time Cost Time cost is measured as the amount of time, measured in milliseconds, taken by the AI planning algorithm to generate a solution.

Memory Cost Memory cost is measured as the overall amount of computer memory, in Kilobytes, (KB), used by the AI planning algorithm.

7.2 Experiment Domains

A series of environments modelling plausible real-world examples were created to survey interactions with the existing GWENDOLEN mechanics in more realistic scenarios. Software-breaking edge cases that passed the simple tests, discussed in Section 7.1.1, were identified and fixed in this stage. Whilst breaking down testing and evaluation with simple and context-less unit tests is useful for development, the scenario-based experiments were important for validating and analysing the theory under increased complexity, as well as ensuring the implementation remained robust for contextual inputs.

7.2.1 Simulated Environments

The simulation environments used for this research are listed below, the rules for each environment are stated alongside a diagram representing the Java environment code.

Waypoints Environment

This environment was originally based on the ‘Mars Curiosity rover’ example used in Cardoso et al.’s work on plan library reconfigurability (Cardoso et al. 2019). In the paper, Cardoso et al. evaluate their implementation by deploying an agent with the goal of navigating through a predetermined route of waypoints on a topological map of Mars. Given that the framework created in this thesis incorporated Cardoso et al.’s work on plan library reconfiguration, it was expected that this example was also used in the overall evaluation.

In this environment, there is a single agent and a number of ‘waypoints’, which are posed as locations on a topological map. A four-waypoint variant of this environment, with the agent starting at waypoint ‘A’, is illustrated in Figure 7.1.

Another iteration of this environment contains nine waypoints, with the agent also starting at waypoint ‘A’. The nine-waypoint environment is also represented in Figure 7.1. There are 12 available actions in the 4-node environment, and 40 available actions in the 9-node environment. The available actions in the 4-node environment are represented by the capabilities shown in Figure 7.2.

As the original example used in this research, the waypoints environment was subjected to the most rigorous experimentation. A Python script was written to produce uniquely random scenario files based on a set of two criteria: the number of generated examples, and the number of failing actions. For example, assuming the number of

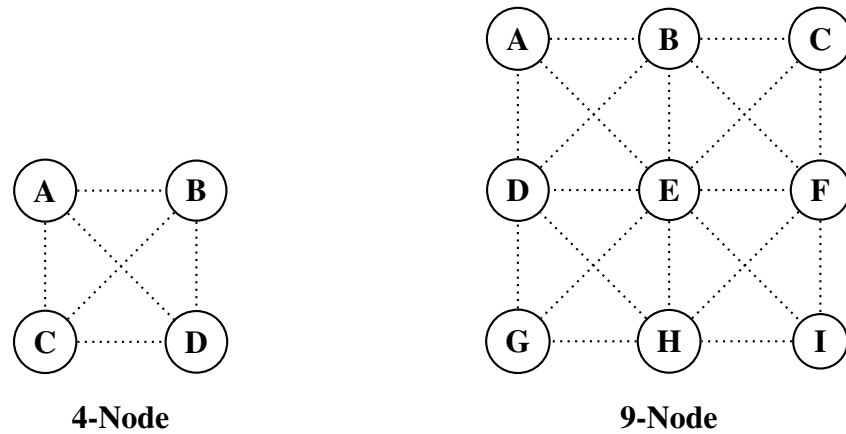


Figure 7.1: 4-Node and 9-Node Waypoints Environments. Dotted lines represent the actions that can be performed to traverse between waypoints.

```

{at (A) } move (A, B) {at (B) }
{at (B) } move (B, A) {at (A) }
{at (A) } move (A, C) {at (C) }
{at (C) } move (C, A) {at (A) }
{at (B) } move (B, D) {at (D) }
{at (D) } move (D, B) {at (B) }
{at (C) } move (C, D) {at (D) }
{at (D) } move (D, C) {at (C) }
{at (A) } move (A, D) {at (D) }
{at (D) } move (D, A) {at (A) }
{at (B) } move (B, C) {at (C) }
{at (C) } move (C, B) {at (B) }

```

Figure 7.2: Available capabilities in the 4-node waypoints environment

failing actions has been set to 8, the script would randomly select 8 different actions from the available actions in the environment and change the result of their execution in the environment. If the action was intended to take the agent from waypoint 1 to waypoint 2, then the script would have changed the destination of that action to be any waypoint other than waypoint 2, to ensure an unexpected result. In total, 1200 sets of synthetic experiment files were deployed into the waypoints environment, ensuring a thorough spread of scenarios over this domain.

An example plan library for this experiment domain is shown in Figure 7.3, where each action is used in a plan to take the agent to each waypoint, and a plan called ‘route’ has been generated that takes the shortest path through a set of four randomly selected waypoints (B, D, C, B), starting and ending at waypoint A.

The ‘connectivity’ of the two different-sized domains was calculated as the average

```

+!at (A) : {B at (B)} <- move (B, A) ;
+!at (A) : {B at (C)} <- move (C, A) ;
+!at (A) : {B at (D)} <- move (D, A) ;
+!at (B) : {B at (A)} <- move (A, B) ;
+!at (B) : {B at (C)} <- move (C, B) ;
+!at (B) : {B at (D)} <- move (D, B) ;
+!at (C) : {B at (A)} <- move (A, C) ;
+!at (C) : {B at (B)} <- move (B, C) ;
+!at (C) : {B at (D)} <- move (D, C) ;
+!at (D) : {B at (A)} <- move (A, D) ;
+!at (D) : {B at (B)} <- move (B, D) ;
+!at (D) : {B at (C)} <- move (C, D) ;
+!route : {B at (A)} <- +!at (B) , +!at (D) , +!at (C) , +!at (B) , +!at (A) ;

```

Figure 7.3: Available plans in the 4-node waypoints environment

number of waypoints that can be reached in one direct action from another waypoint, and is used to quantify the scale difference between the domains. For the 4-node environment, the average connectivity was 3 nodes, and for the 9-node environment, the average connectivity was 4.4 nodes. The ‘average journey length’ is also used for this purpose, and was calculated by finding the average length of the shortest path journey from each waypoint to every other waypoint in the domain. The 4-node environment has an average journey length of 1, as all of the waypoints are directly connected, whereas the 9-node environment has an average journey length of 1.47 as some waypoints need more than one action to reach other waypoints.

By comparing the results of a 4-node variant with a 9-node variant of the waypoints environment, the effects of increased domain connectivity and longer average journey lengths can be considered (see Section 7.5.1).

Hallway Environment

The hallway environment is similar to the waypoints environment, in that they are both grid environments and only movement actions can fail, although the hallway environment has more complex connections between the available actions, due to the reduced connectivity between the available rooms. The similarity of the environments allows for a direct comparison of how reduced connectivity affects the performance of the proposed system.

In this environment, the agent is tasked to perform a routine inspection, where it must take a picture of the emergency exit in a hallway (highlighted in red in Figure 7.4).

However, it can be seen that not all of the rooms are directly connected, and the agent

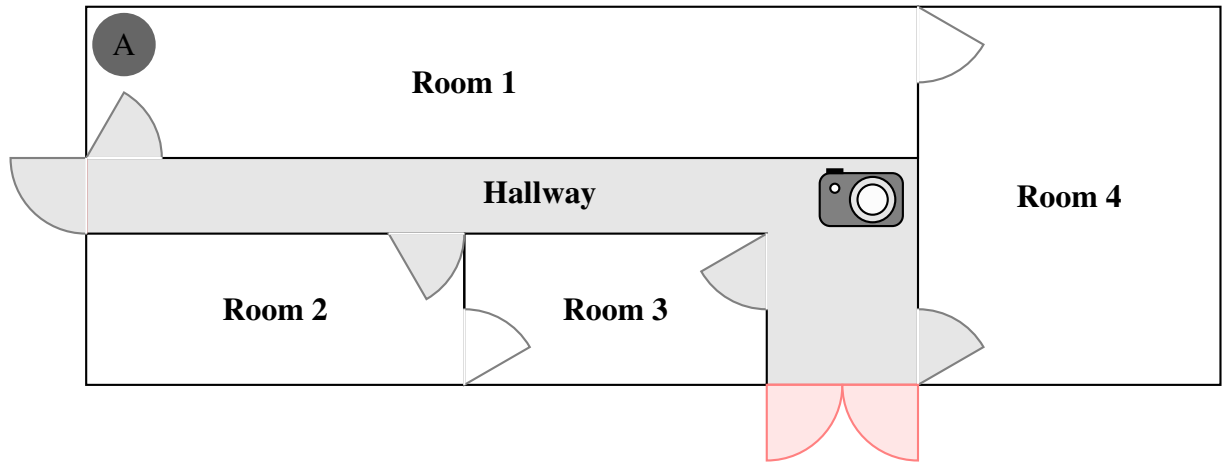


Figure 7.4: Hallway Environment

must be located in a predefined area to take the picture. This creates a chain of actions that is more sensitive to failure as there is only one viable action that will move the agent to the location to take the picture from the hallway and only one action to leave this location to get back to the hallway. There are two possible actions to leave each room, meaning that the agent may need to find an alternative route to reach the hallway if one of these actions is not performing as expected. There are 15 available actions in the hallway environment which are represented by the capabilities shown in Figure 7.5.

```
{at(room1)} move(room1, room4) {at(room4)}
{at(room2)} move(room2, room3) {at(room3)}
{at(room3)} move(room3, room2) {at(room2)}
{at(room4)} move(room4, room1) {at(room1)}
{at(room1)} move(room1, hallway) {at(hallway)}
{at(room2)} move(room2, hallway) {at(hallway)}
{at(room3)} move(room3, hallway) {at(hallway)}
{at(room4)} move(room4, hallway) {at(hallway)}
{at(hallway)} move(hallway, room1) {at(room1)}
{at(hallway)} move(hallway, room2) {at(room2)}
{at(hallway)} move(hallway, room3) {at(room3)}
{at(hallway)} move(hallway, room4) {at(room4)}
{at(picture_location)} move(picture_location, hallway) {at(hallway)}
{at(hallway)} move(hallway, picture_location) {at(picture_location)}
{at(picture_location)} take_picture {at(picture_location)}
```

Figure 7.5: Available capabilities in the hallway environment

An example plan library for this experiment domain is shown in Figure 7.6, where

each action is used in a plan to take the agent to the desired location, and a plan called ‘route’ has been generated that takes the shortest path to the ‘picture_location’ to take a picture, starting and ending at a randomly selected room.

```

+!at(room4) : {B at(room1)} <- move(room1,room4);
+!at(room3) : {B at(room2)} <- move(room2,room3);
+!at(room2) : {B at(room3)} <- move(room3,room2);
+!at(room1) : {B at(room4)} <- move(room4,room1);
+!at(room1) : {B at(hallway)} <- move(hallway,room1);
+!at(room2) : {B at(hallway)} <- move(hallway,room2);
+!at(room3) : {B at(hallway)} <- move(hallway,room3);
+!at(room4) : {B at(hallway)} <- move(hallway,room4);
+!at(hallway) : {B at(room1)} <- move(room1,hallway);
+!at(hallway) : {B at(room2)} <- move(room2,hallway);
+!at(hallway) : {B at(room3)} <- move(room3,hallway);
+!at(hallway) : {B at(room4)} <- move(room4,hallway);
+!at(picture_location) : {B at(hallway)} <- move(hallway,picture_location);
+!at(hallway) : {B at(picture_location)} <- move(picture_location,hallway);
+!take_picture : {B at(picture_location)} <- take_picture;

+!route : {B at(room4)} <- +!at(hallway), +!at(picture_location),
                               +!take_picture , +!at(hallway), +!at(room4);

```

Figure 7.6: Available plans in the hallway environment

The hallway environment can also be used to evaluate the overall success rate of the system. The success rate can be measured by the percentage of routes that are completed by each agent as the percentage of action failure is increased. The robustness of the system can be evaluated by assessing the difference in results compared to the waypoints environment. The hallway environment contains an action to traverse the hallway to the site marked for taking a picture. This action is critical for completing the mission and is therefore more sensitive to higher action failure rates, due to the increased likelihood of the critical action failing, and this is expected to negatively affect the performance of the agent in this environment.

Time and Memory Cost Experiments

To discover an accurate time and memory cost for running the AI planning algorithm to re-plan a GWENDOLEN plan library, the re-planning mechanism of the system was evaluated in isolation. The experimentation process is illustrated in Figure 7.7.

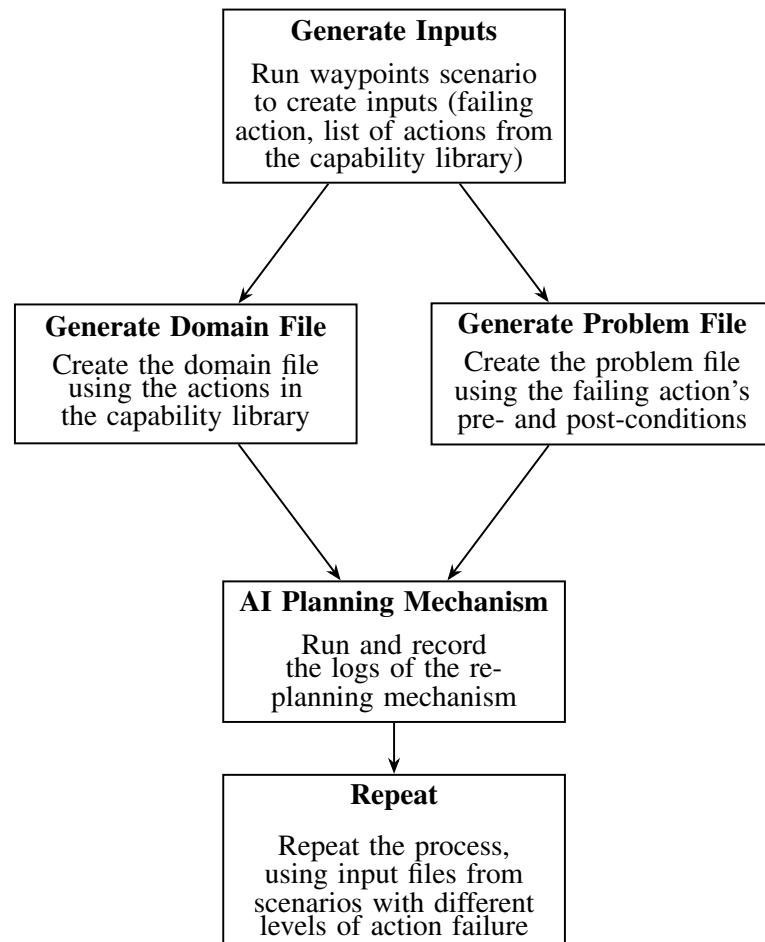


Figure 7.7: Time and Computational Costs Experimentation Process

The mechanism requires two inputs to operate: the failing action description, and a list of action descriptions for all actions in the domain (including learned action descriptions and excluding the failing action description). The required inputs for the re-planning mechanism experiments were generated by running a waypoints environment scenario, up to the point at which re-planning would be performed. This means that the list of action descriptions given to the planner would contain a learned action description. A set of input files was generated for every action in the capability library so that the planner could be evaluated against a range of learned action descriptions. This process was repeated three times for three different levels of failure in the domain, to generate input files for a domain with one failure, a domain where half of the actions are failing, and a domain where all of the actions are failing. This experiment was performed using the 4-node waypoints environment and then repeated using the 9-node environment, to assess the mechanism's performance when handling more complex

plans.

The re-planning mechanism was manually executed via terminal commands using the input files. This isolated execution ensured that only the planner's time and memory costs were measured, independent of external system interactions. By decoupling the planner from the rest of the system, the experiment produced results that could be used to evaluate how reducing the planning problem size (i.e., replacing a single failed action versus re-planning the entire library) impacts computational costs. For each set, the planner's output was recorded; the results included the action failure identifier, overall outcome, peak memory usage and the duration of processing.

For comparison, this experiment was also run as an 'optimal' agent where the whole plan library is re-planned, producing the optimal plan library using the available capabilities. The same domain file is used, but the planner was asked to re-plan each of the plans from scratch. The time and memory costs to find the solutions were recorded and used to compare the cost of completely re-planning the plan library against the cost of finding just one replacement plan for a single action, as required by the re-configuration framework proposed by Cardoso et al. (2019).

7.2.2 Generation of Experiment Files

Each execution of a GWENDOLEN program requires an agent program, an environment, and an agent infrastructure layer file.

To create a large number of random inputs, Python scripts were written to generate and run the required files. For each domain, 50 trials were conducted to cover action failure levels ranging from 0% to 100% failure rates. The number of available actions influenced the action failure percentages in the experiments. In the 4-node waypoints environment with 12 actions and the hallway environment with 14 actions, each failure level was created by setting one additional action to fail per category. For instance, in the 4-node environment, the failure rates increased from 0% to 8.3% by setting one more action to fail out of the available 12 ($\frac{1}{12}$). The 9-node waypoints environment had 40 available actions, meaning the failure rates could be set at intervals that allowed close comparison to the other environments.

Once the files had been generated, 650 scenarios were executed in the 4-node waypoints environment, 550 scenarios in the 9-node waypoints, and 750 scenarios were executed in the hallway environment. Each execution of the agent programs produced output based on the agent's experience, this was saved alongside the scenario files for each execution in aid of repeatability and to allow further interpretation of results.

Another Python script was written to extract the success rates and learning outcomes of the recorded outputs. This script verified if the agent completed the assigned route and recorded the number of new action descriptions learned for each.

The use of synthetic experiments significantly increased the total number of experiments performed: 1950 scenarios were executed in total, with each agent given a randomised route through the environment, and experiencing varying action availability.

7.3 Experiment Scenarios

The experiment scenarios used in this evaluation are described in this section. A diagram representing the plan that the agent will attempt to perform is given for each experiment, along with an explanation of the scenario. Then the method for introducing failures into the scenario is described, with an accompanying diagram showing an example of a failure as well as the action that could be produced after learning a new action description for the failing action. The expected outcome for the experiment, detailing how the proposed system should respond to the failure in the scenario is then justified, including an example of a re-planned route.

7.3.1 Waypoints Environment

4 Nodes

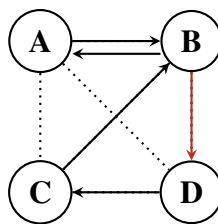


Figure 7.8: 4 Node: Route Patrol Example

Scenario In this example the agent, named rover, performs a patrol starting at waypoint A and visits four other waypoints (these can be waypoints that have been visited before, just not waypoint A) before posting the journey’s action log. The route is performed until the agent is stopped. In the experiments for 4 node and 9 node maps, the agent is given a list of four waypoints to visit to standardise the experiments between

all maps. An example route is shown in Figure 7.8, where solid lines represent the actions used to traverse the waypoints in the agent's planned route, and dotted lines represent the available actions for traversing between waypoints, some of which are covered by the agent's planned route.

The GWENDOLEN plan body for the route that starts and ends at waypoint A and visits four other waypoints: B, D, C, and B again, is shown in Equation (7.1).

$$\text{move}(A,B), \text{move}(B,D), \text{move}(D,C), \text{move}(C,B), \text{move}(B,A); \quad (7.1)$$

Failure A predefined percentage of randomly selected actions in the environment are modified. This is done without informing the agent via the capability library or changing the agent's plan itself. To do this, the post-condition for the action is changed from $at(X)$ to $at(Y)$ in the environment, to represent an external issue with that action, originating in the environment. The actions that are modified to simulate failure will take the agent to the location of the modified post-conditions every time it is used by the agent for that trial. The same action may fail in a different way in other trials, but each failure remains static once it has been selected to simulate failure for the trial. Action failures are treated in this way for all of the experiments in the evaluation.

For the scenario given in Figure 7.8, where the agent follows a route to visit four predefined waypoints, the $\text{move}(B,D)$ capability described in Equation (7.2), is now failing, and navigates the agent to waypoint C, as defined in Equation (7.3).

$$\{at(B)\}\text{move}(B,D)\{at(D)\} \quad (7.2)$$

$$\{at(B)\}\text{move}(B,D)\{at(C)\} \quad (7.3)$$

The result is illustrated in Figure 7.9, with a red arrow indicating that the $\text{move}(B,D)$ action now takes the rover to waypoint C.

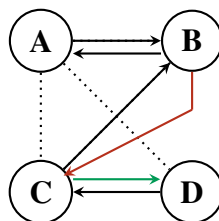


Figure 7.9: 4 Node: Route Patrol with Failure

The green arrow indicates an action in an example replacement route (shown in

Equation (7.4)) that would achieve the original action's postconditions, created by learning a new action description and reconfiguring the rover's plans.

Expected Outcome The agent will execute the failing action and believe the new post-condition once the action duration has elapsed. If the failure threshold has been set to 1 failure, the learning and reconfiguration process are triggered after one failure is detected for that action, and the agent will replace the action's post-condition with the newly believed post-condition. As there is only one failure in the agent's action log, the replacement is not weighted or scored against other entries in the log. The plans containing the replacement action description are then reconfigured using the new set of available actions.

An example replacement route written as a GWENDOLEN plan body is:

$$\text{move}(A,B), \text{move}(B,D), \text{move}(C,D), \text{move}(D,C), \text{move}(C,B), \text{move}(B,A); \quad (7.4)$$

The action in red has a new learned action description to reflect the new post-condition in Equation (7.3), and the plan has been reconfigured using this action in addition to the action in green to complete the route requirement. For this specific scenario, the agent performed the action $\text{move}(B,D)$ but ended up at waypoint C . This resulted in the action being flagged as deprecated, which required the agent to learn a new action description to reflect the change in destination for that action (removing the deprecated flag in the process). However, to achieve the desired result from the original action, the plan was reconfigured to include the action $\text{move}(C,D)$ to take the agent from waypoint C back to the desired waypoint from the original route, waypoint D . The action that failed (with its new action description) is retained in the new route plan in this case as it was used by the planner to find a replacement plan with the fewest actions. In other cases, different actions could have been used to patch the route.

9 Nodes

Scenario In this example, the rover is given four waypoints to visit, that are not necessarily the same as in the 4-node grid variant, before posting the journey's action log. As before, once the four defined waypoints have been visited, the rover navigates back to the start of the patrol and repeats the same route again until instructed to stop.

In Figure 7.10, the agent starts at waypoint A, before following a route to visit the waypoints B, D, H, and D, and then returning to waypoint A. The GWENDOLEN plan

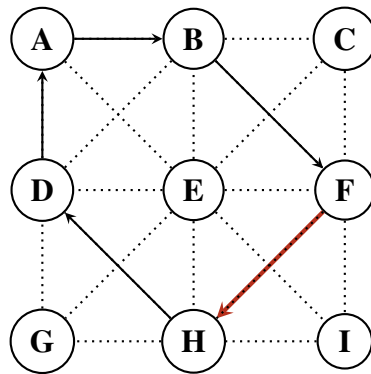


Figure 7.10: 9 Node: Route Patrol Example

body for the route shown in Figure 7.10 is represented as:

$$\text{move}(A,B), \text{move}(B,F), \text{move}(F,H), \text{move}(H,D), \text{move}(D,A); \quad (7.5)$$

Failure Action failures are introduced using the same method as in the 4-node environment: a predefined percentage of randomly selected actions in the environment are modified. This is done without informing the agent via the capability library or changing the agent’s plan itself. To do this the post-condition for the action is changed from $at(X)$ to $at(Y)$, to represent an external issue with that action, originating in the environment.

For the scenario given in Figure 7.10, the capability shown in Equation 7.6 could return the results expected from the capability defined in Equation 7.7, where an empty post-condition shows that the action is no longer achieving any post-conditions after execution.

$$\{at(F)\}\text{move}(F,H)\{at(H)\} \quad (7.6)$$

$$\{at(F)\}\text{move}(F,H)\{\} \quad (7.7)$$

Figure 7.11 shows the replacement actions with green arrows, and a no-entry symbol represents the blockage that caused the failure. The replacement actions $\text{move}(F,I)$ and $\text{move}(I,H)$, are shown in the patched plan in Equation 7.8.

Expected Outcome The expected outcome should be similar to the 4-node variant’s expected outcome, as the learning and reconfiguration mechanisms should perform the same in both environments. Some improvement is expected due to the greater number of available actions, as this will increase the number of possible routes available to the

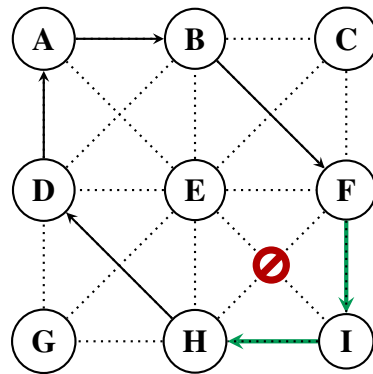


Figure 7.11: 9 Node: Route Patrol with Failure

planning algorithm to reconfigure plans. However, as the 9-node environment is larger, the agent often requires two actions to traverse between two waypoints instead of just one action. This increases the number of actions required to complete the route and subsequently increases the likelihood of encountering a failing action. An example replacement route for the route in Figure 7.10, written as a GWENDOLEN plan body is:

$$\text{move}(A,B), \text{move}(B,F), \text{move}(F,I), \text{move}(I,H), \text{move}(H,D), \text{move}(D,A); \quad (7.8)$$

7.3.2 Hallway

Patrol

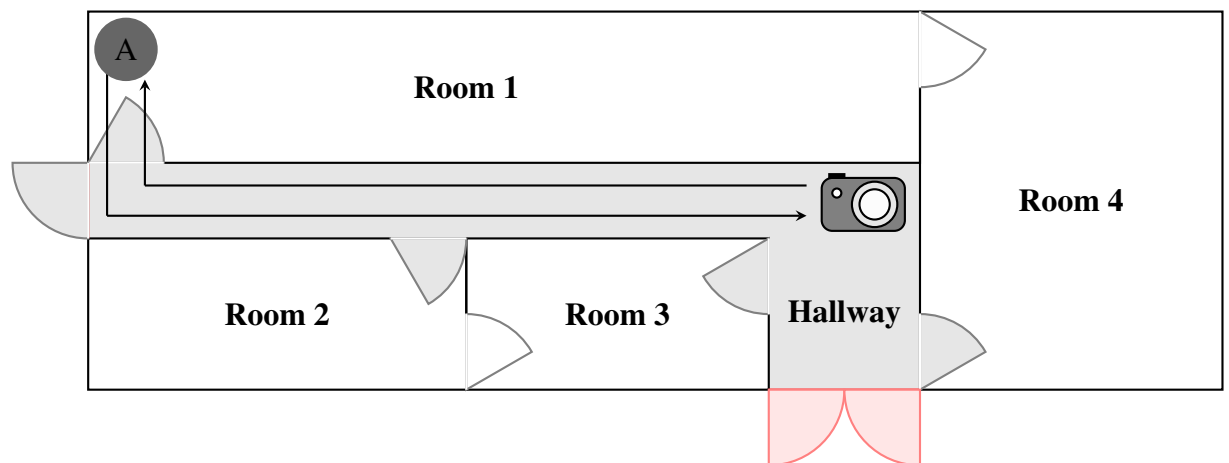


Figure 7.12: Hallway Patrol Example

Scenario In this example, illustrated in Figure 7.12, the agent navigates an environment to reach an endpoint where a picture is to be taken. The agent starts in any of the rooms, but never the hallway. Any route may be taken by the agent to reach the hallway to take a picture, after which the agent should return to its original room. An example GWENDOLEN plan body for a possible route when the agent starts in ‘Room 1’ is shown in Equation 7.9.

$$\text{move}(\text{Room1}, \text{Hallway}), \text{take_picture}, \text{move}(\text{Hallway}, \text{Room1}); \quad (7.9)$$

Failure A randomly selected action that appears in the route goal’s plan is modified, without informing the agent or changing the agent’s plan itself, similar to the way-point environment experiments. In the example plan shown in Equation 7.9, the action $\text{move}(\text{Room1}, \text{Hallway})$ is changed to no longer have any post conditions, meaning the agent remains in Room1 after execution. The post-condition for this action was changed in the environment which presents as an unknown change in that action to the agent.

$$\{at(\text{Room1})\}\text{move}(\text{Room1}, \text{Hallway})\{at(\text{Hallway})\} \quad (7.10)$$

$$\{at(\text{Room1})\}\text{move}(\text{Room1}, \text{Hallway})\{\} \quad (7.11)$$

Expected Outcome The expected outcome of this experiment is illustrated in Figure 7.13.

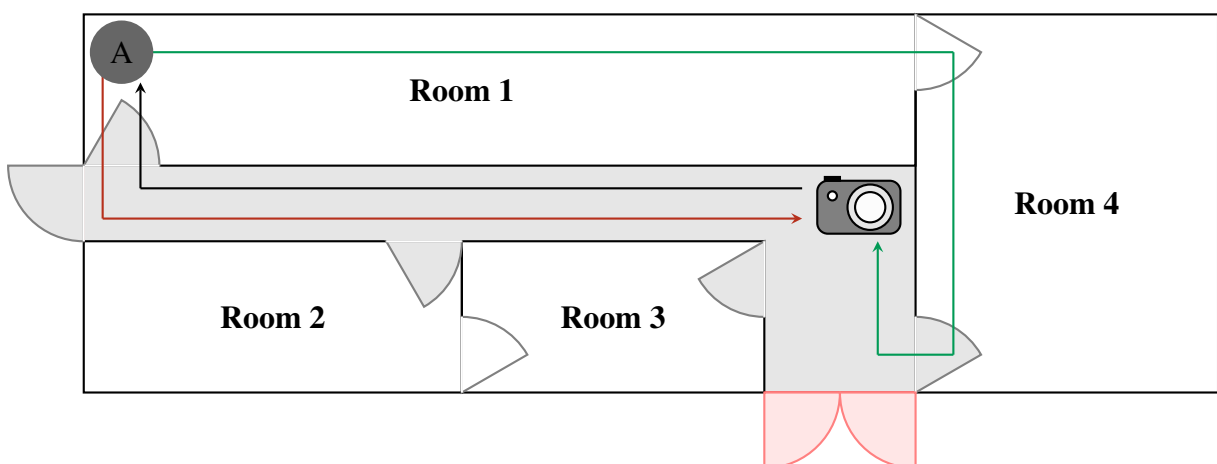


Figure 7.13: Hallway Patrol Action Failure

The agent will attempt to execute the first action in the plan body, but will remain

in ‘Room1’. The agent will log this as a failure, shown by the red arrow, and initiate the learning and reconfiguration process as the failure threshold has been set to 1 for this example. The new post-condition found in the failure log then replaces the action’s deprecated post-conditions, shown by the green arrow in the Figure 7.13. A new plan body is reconfigured using the available actions, and the failing action has been replaced by a different route to the location required for the *take_picture* action, described in Equation 7.12.

move(Room1,Room4), move(Room4,Hallway), take_picture, move(Hallway,Room1); (7.12)

7.3.3 Reconfiguration Timing and Memory Cost

The reconfiguration timing and memory cost experiments use the waypoint environment’s capability library. Two PDDL files are supplied to the AI planner at the point where reconfiguration commences, one describes the domain which contains all of the available actions, and the other describes the problem. In this case, the problem is a definition of the action that requires re-planning or reconfiguring. Every action in the capability library was given as a problem to the planner, totalling 12 actions in the case of the 4-node map, and 40 actions in the case of the 9-node map. The reconfiguration timing and memory cost experiments were based on the experiment data from the waypoints environment experiments, and not on the hallway environment. The waypoints environment, with its two differently sized maps, allowed for a close comparison of the effects of map size and scale on the costs of the planning algorithm, without introducing different metrics from another domain into the comparison.

The cost of generating a patch for the failing action was recorded along with the cost of re-planning the full plan library, to represent the costs induced by an ‘optimal’ agent. The integration of failure detection with Cardoso et al. (2019)’s reconfigurability framework allows the planner to selectively re-plan only the affected plans in the plan library. The planner only needs to find a single plan to take the agent from the pre-condition state, to the post-condition state of the failing action, in order to patch every plan that the failing action featured in, as this patch is then applied to all of the affected plans. As a result, the time and memory cost of the plan reconfiguration system used in this thesis is measured by recording the time and memory cost for reconfiguring this single plan. The time and memory costs induced by an optimal agent were also recorded to generate a comparison. This agent does not isolate the plans containing the failing action, and instead re-plans the entire plan library to produce the optimal plan

library rather than patching the failing action.

To illustrate the comparison, if the original plan for a route that starts and ends at waypoint A and visits the four waypoints B, F, H and D was:

$$\text{move}(A,B), \text{move}(B,F), \text{move}(F,H), \text{move}(H,D), \text{move}(D,A);$$

and the action $\text{move}(F,H)$ was seen to be failing consistently and the new action description was found to be:

$$\{at(F)\}\text{move}(F,H)\{at(I)\}$$

then the planner in the system proposed in this thesis would generate a single plan, such as:

$$\text{move}(F,H), \text{move}(I,H)$$

which can then be used to replace the failing action in the route plan, to produce:

$$\text{move}(A,B), \text{move}(B,F), \text{move}(F,H), \text{move}(I,H), \text{move}(H,D), \text{move}(D,A);$$

In this example, the optimal agent's planner would have produced a route plan with the same cost - as most optimal route that visits all of the required waypoints, where the action $\text{move}(F,H)$ is failing, will always be 6 actions in length.

4 Node Map

Scenario The 12 actions in the agent's capability library for the 4-node map are used to generate problem files for the AI planner to ingest. The problem files are individually processed by the planner, and the planner attempts to find viable plans that will navigate from the initial state (action pre-conditions) to the goal state (action post-conditions) stated in the problem file using the remaining 11 actions in the capability library.

Failure The percentage of actions that have failed (and have different post-conditions to the actions in the original plans) in each capability library was varied to represent three levels of degradation. The levels are: one failure, 50% failure and complete failure.

Expected Outcome The planner uses the actions provided in the capability library to find a plan that achieves the failing action’s post-conditions, starting from the initial state described by the failing action’s pre-conditions.

9 Node Map

Scenario This scenario is identical to the four node map timing scenario, although the capability library has been scaled up to account for the additional actions required for movement between nine nodes. There are 40 actions in the capability library for this scenario.

Failure The percentage of actions that have failed in each capability library was also varied over three levels of degradation for this map size. As before, the levels are: one failure, 50% failure and complete failure.

Expected Outcome The planner uses the actions provided in the capability library to find a plan that achieves the failing action’s post-conditions, starting from the initial state described by the failing action’s pre-conditions

7.4 Experiment selection

Table 7.2 shows which criteria each experiment is designed to evaluate. A tick (✓) indicates that the criterion was evaluated in that experiment, while a cross (✗) indicates that it was not.

		Criteria				
Environment	Experiment	Success Rate	Robustness	Scalability	Time Cost	Comp. Cost
Waypoints	4-Node	✓	✗	✓	✗	✗
	9-Node	✓	✗	✓	✗	✗
Hallway	4 Rooms	✓	✓	✗	✗	✗
Reconfiguration Timing	Patch	✗	✗	✓	✓	✗
	Full Plan Library	✗	✗	✓	✓	✗
Reconfiguration Cost	Patch	✗	✗	✓	✗	✓
	Full Plan Library	✗	✗	✓	✗	✓

Table 7.2: Table of experiment relation to evaluation criteria

In the waypoints environment, both the 4-node and 9-node experiments aim to test the success rate of the system - measuring successful plan completions as the percentage of randomly selected failing actions is increased. The systems ability to scale can also be evaluated as the success rate of the agent can be compared to the size of the environment.

In the hallway environment, the success rate is evaluated in the same way as the waypoints experiments. Additionally, the hallway environment's rooms are considerably less connected, meaning that the number of connections between different nodes or rooms is significantly less than in the waypoints environment - which helps to evaluate how the system performs when the pool of available actions for repairing plans is decreased. This is used to assess the robustness of the system in this evaluation.

The reconfiguration timing and memory cost experiments measure the costs of replanning a patch for a failing action compared to the cost of replanning the full plan library from scratch, from the plan libraries taken from the waypoints environment experiments.

Measuring the differences in cost between replanning patches and full plan libraries enables a comparison of the system presented in this thesis, against a system that requires the full plan library to be replanned. As the timing and memory cost experiments use the plans from the 4-node and 9-node experiments, an evaluation of how well the system scales was made by comparing the costs between these different environment sizes.

7.5 Results

In this section, the results of the experiments described in Section 7.3 are discussed. There is a tabulated summary of the results for each experiment, with units and measurements standardised across the experiments where possible.

7.5.1 Waypoints

4 Node Map

Summary A summary of action failure rates for 4 node waypoint environments is shown in Table 7.3.

A total of 50 tests were performed for each level of action failure (from 0% to

Action Failure (%)	Average Completion (%)	Average Actions Learned
0	100	0.00
8	100	0.32
17	100	0.60
25	100	0.98
33	100	1.16
42	100	1.50
50	100	1.54
58	100	1.90
67	100	1.96
75	100	2.16
83	99	2.35
92	97	2.51
100	99	2.61

Table 7.3: Summary of Action Failure Rates for 4 Node Waypoint Environments

100% action failure), with 650 tests overall¹. From the values shown in the second column (Average Completion(%)) in Table 7.3 it is clear that the average completion rate for a map of this size is very high. In the line graph presented in Figure 7.14, the average success rate shown in the Average Completion (%) column of each agent over 50 tests is plotted against the percentage of failing actions in the environment. It can be seen that the success rate does not go below 97% for any level of action failure in this domain, which shows the proposed system’s resilience to all levels of action failure.

The high success rate in this domain is likely due to the greater connectivity of a map with 4 nodes arranged in a square arrangement: each node has three connections and can therefore continue to operate until all paths are no longer traversable. The average number of actions learned, shown in the Average Actions Learned column in Table 7.3, ranges from 0 to 2.61, with a steady increase from 0% action failure to 100% action failure. This is shown in the line graph presented in Figure 7.15, where the average number of action descriptions learned (from the data in the Average Actions Learned column in Table 7.3) is plotted against the percentage of failing actions in the environment (from the data in the Action Failure(%) column in Table 7.3).

¹The files and results for all experiments are available at: <https://github.com/peterstringer/experiment-files>

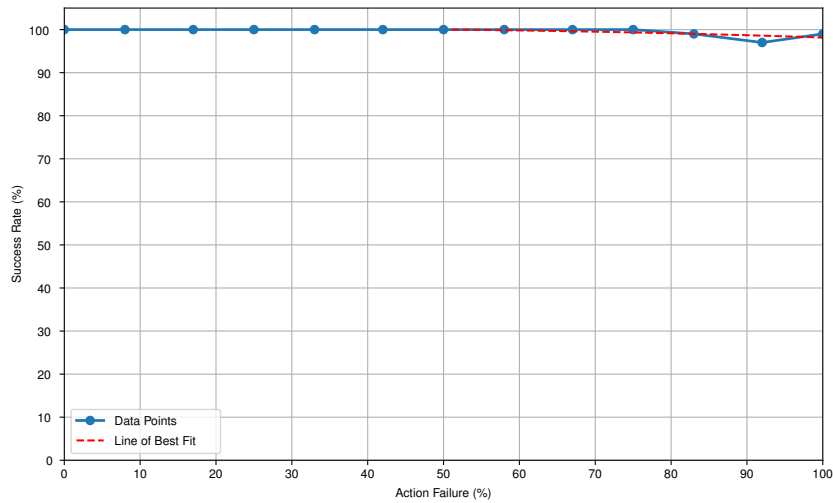


Figure 7.14: Action Failure Rate (%) against Success Rate in 4 Node Waypoint Environments

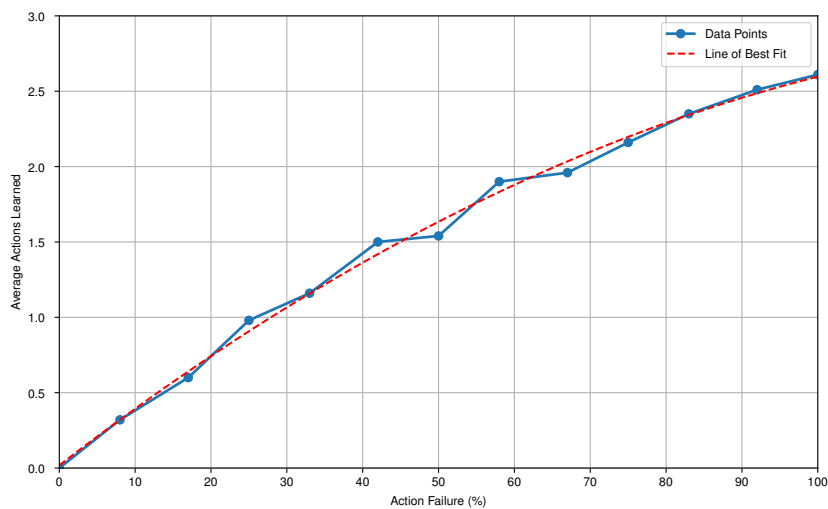


Figure 7.15: Action Failure Rate (%) against Average Action Descriptions Learned in 4 Node Waypoint Environments

The average number of actions learned increases from 0.00 to 2.61 as action failure increases from 0% to 100%. When considered alongside the plot shown in Figure 7.14, it can be determined that the agent can still complete 99% of the assigned routes despite needing to learn an average of 2.61 new action descriptions and reconfigure the plan library with patches for those actions.

Action failure ranged from 0% to 100% in this experiment. For each level of action failure, the corresponding percentage of the available actions will not achieve their expected post-condition, and will instead navigate a randomised waypoint that is not the same as the expected waypoint. For example, at 50% action failure, 50% of the actions available to the agent in the environment will navigate to a different waypoint than the waypoint in the action description in the agent’s capability library.

The relation is linear in this experiment, although this relationship would not hold for longer routes as more failures are discovered on the map. However, the relation would not differ considerably: when over two actions are learned on a route, there is only a 3% decrease in the average completion rate, as seen in Table 7.3.

9 Node Map

Summary A summary of action failure rates for 9 node waypoint environments is shown in Table 7.4. A total of 50 tests were performed for each action failure category, with 550 tests overall. As there were 12 actions available in the 4-node waypoint environment, the percentages do not directly translate to the 9-node waypoint environment where there are 40 actions available to the agent.

Action Failure (%)	Average Completion (%)	Average Actions Learned
0	100	0.00
10	96	0.50
20	85	0.86
30	84	1.74
40	78	1.68
50	82	2.20
60	62	1.92
70	70	2.58
80	62	2.48
90	69	2.86
100	58	2.84

Table 7.4: Summary of Action Failure Rates for 9 Node Waypoint Environments

There is significantly reduced connectivity in this domain when compared to the 4-node map. Whilst all nodes still have at least three connections, the average journey

length between nodes is 1.47 actions compared to 1 action on a 4-node map. The effect of the longer average journey can be seen in both the average completion rate shown in the Average Completion (%) column in Tables 7.3 and 7.4, and the average actions learned (shown in the Average Actions Learned column).

The average completion rate drops by a maximum of 3% in the 4-node environment, whereas in the 9-node environment, the completion rate drops by a maximum of 42%. The average number of actions learned also remains lower on the 4-node environment at every failure rate when compared to the 9-node environment. However, as the agent is travelling further on average, requiring more actions, a greater number of failing actions are encountered, suggesting that the average number of actions needing to be learned would increase as more failing actions need to be replaced. The effect is rather noticeable on the average completion rate, with only 58% of routes still being completed at 100% action failure in the 9-node environment, compared to 99% of journey still being completed at 100% action failure in the 4-node environment.

However, despite a 47% increase in average journey length in the 9-node environment, the average number of actions learned increased less than expected. An additional average of 0.47 more action failures are encountered per waypoint in the 9-node environment due to the increased average journey length. The increased average journey length results in an average of 5.88 actions being required to visit the four randomly selected waypoints specified in the scenario description, compared to a 4 action journey length on average in the 4-node environment.

Accordingly, the average actions learned was expected to increase by the additional average journey length multiplied by the chance of encountering an action failure (Action Failure %). For 9-nodes at 100% failure, this would be 0.47 ($0.47 \times \frac{100}{100}$) additional actions learned on average. Despite this, the relation between the average number of actions learned and action failure percentage remains similar to the 4 node map, peaking only 0.23 average actions learned higher at 100% action failure, as shown in the Average Actions Learned columns in Tables 7.3 and 7.4.

In Figure 7.16, the average success rate shown in the Average Completion (%) column of each agent over 50 tests is plotted against the percentage of failing actions in the environment. As the percentage of failing actions increases, the average success rate decreases. However, from the last row in Table 7.4, it can be seen that the agents still managed to maintain an average success rate of 58% even when all of the available actions directed the agent to a different waypoint than expected or didn't move the agent at all, at 100% action failure.

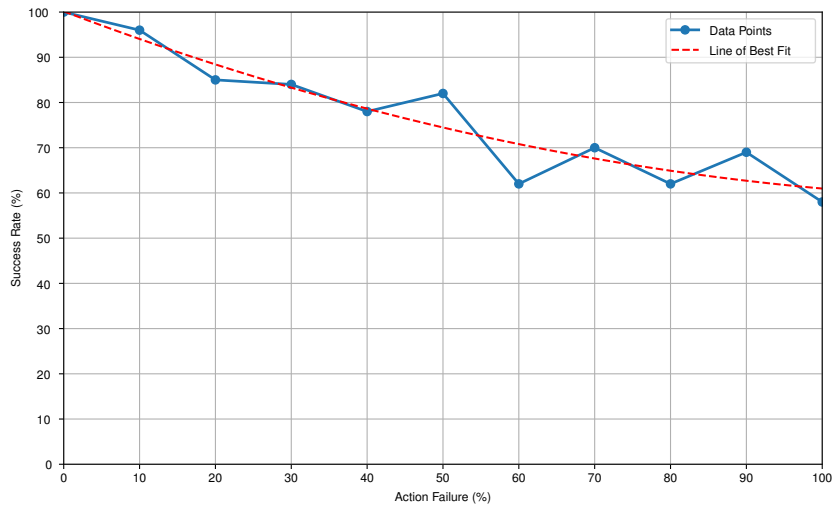


Figure 7.16: Action Failure Rate (%) against Success Rate in 9 Node Waypoint Environments

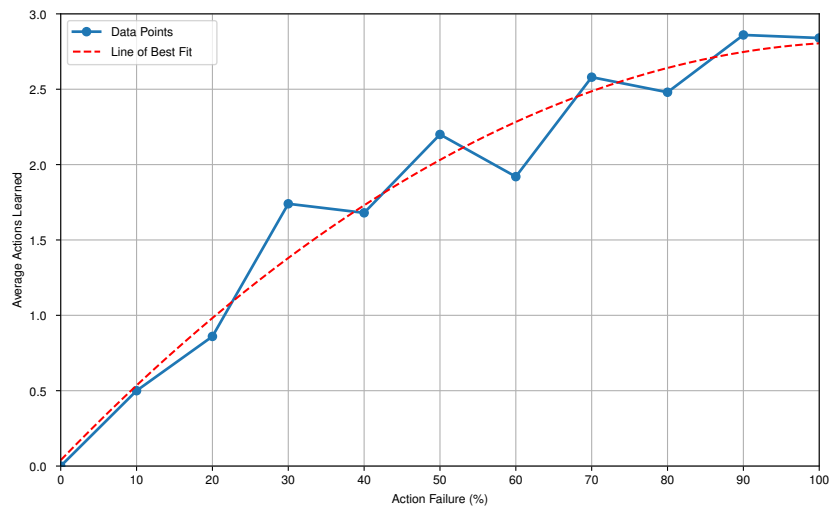


Figure 7.17: Action Failure Rate (%) against Average Action Descriptions Learned in 9 Node Waypoint Environments

In Figure 7.17, the average number of action descriptions learned is plotted against the percentage of failing actions in the environment. The average number of actions learned increases from 0.00 to 2.84 as action failure increases. The relationship between the average number of action descriptions learned and the percentage of failing

actions begins to show an exponential pattern, as the rate at which new action descriptions are learned slows down as the action failure rate gets closer to 100%.

7.5.2 Hallway

Summary A summary of action failure rates for hallway environments is shown in Table 7.5. A total of 50 tests were performed for each action failure category, as in the

Action Failure (%)	Average Completion (%)	Average Actions Learned
0	100	0
7	94	0.27
14	86	0.37
21	72	1.02
29	62	1.31
36	58	1.51
43	42	2.00
50	32	1.84
57	34	2.37
64	12	2.39
71	18	2.88
79	12	3.08
86	22	3.24
93	10	3.73
100	8	3.88

Table 7.5: Summary of Action Failure Rates for Hallway Environments

previous examples, resulting in 750 tests overall. There were 14 levels of failure in this domain, as there were 14 available actions that could be set to fail, with one additional action failing for each level.

The effect of a different domain can be clearly noticed in the summary table; the average completion percentage falls considerably with every additional action failure. Despite this, the agent still completes the route at least 8% of the time, even at 100% failure, showing the agent can still find viable solutions to complete the route even when all actions are taking the agent to unexpected waypoints. This is also illustrated in Figure 7.18, where the average success rate of each agent over 50 tests is plotted against the percentage of failing actions in the environment.

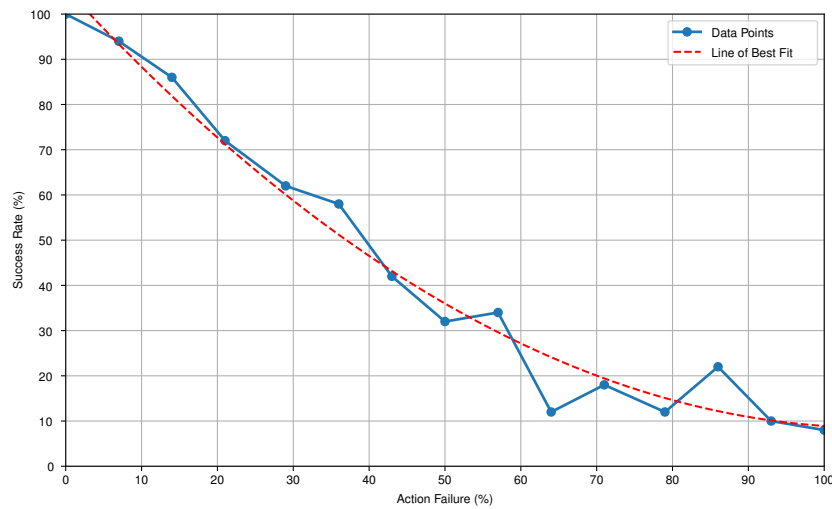


Figure 7.18: Action Failure Rate (%) against Success Rate in Hallway Environments

In the hallway domain, described in Section 7.2.1, the availability of some actions is more important for route completion than others: only two actions are responsible for moving the agent to and from the hallway to the site P , where the agent is required to take a picture. If either of these two actions were selected during the random allocation of failing actions, the agent would have no route to complete the mission. The lack of flexibility in the plan library is manifested in heavily increased failure rates.

In Figure 7.19, the average number of action descriptions learned is plotted against the percentage of failing actions in the environment. There is a range of an average of 0 actions learned at 0% action failure, to an average of 3.88 actions learned on routes at 100% action failure. When this is compared to the equivalent experiment data from the 4-node and 9-node waypoints environments, where the maximum range of average actions learned was a minimum of 0 actions and a maximum of 2.84, it can be seen that the agent is required to learn more action descriptions in a less connected environment.

Directly comparing the Average Actions Learned against the Action Failure percentages for the 9-node waypoints environment and the hallway environment (Tables 7.17 and 7.19), the rate at which action descriptions are being learned in the 9-node environment decreases as the action failure rate approaches 100%, whereas the rate that action descriptions are being learned in the hallway environment consistently increases at a steady rate up to 100% action failure. This suggests that the agent is encountering comparatively more failures on routes in the hallway environment as the action failure rate increases above 50% action failure. This is likely due to the AI

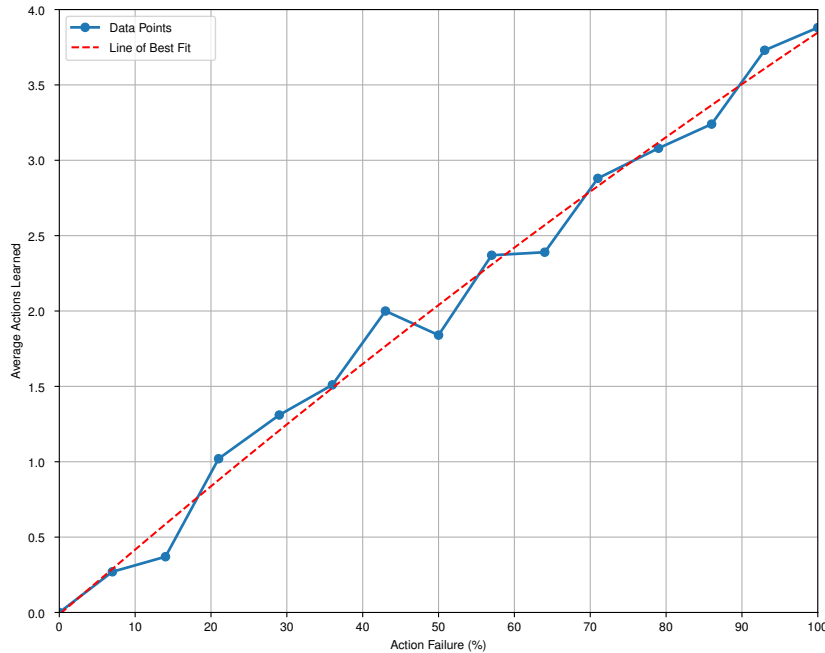


Figure 7.19: Action Failure Rate (%) against Average Action Descriptions Learned in Hallway Environments

planner having access to fewer direct routes to the desired goal state when patching the plans containing the newly learned action description. Consequently, the plans contain more actions, which increases the likelihood of encountering another action failure.

7.5.3 Reconfiguration Timing

4 Node Map

Table 7.6 presents the timing costs associated with re-planning in the context of varying failure rates, ranging from one failure to 100% failure. The values shown in rows one and two are identical, showing that there is no additional planning time cost for patching a single plan when up to 50% of the plan library contains failing actions (that have different post-conditions to the actions in the original plans)

The varying failure rates presented in Table 7.6 are based on the failure rates of the experiment data that was used to generate the inputs for the planning algorithm.

Specifically, the table compares the time required for finding a plan to patch a single failing action (as required by the system proposed in this theses) with the time required

Experiment Type	Time(s) for Patch (ms)	Time(s) for Full Replan (ms)	Scaled
1 Failure	0.071	0.93	2.93
50% Failure	0.071	0.93	2.93
100% Failure	0.074	0.97	3.06

Table 7.6: Replanning Timing Costs for 4-node Waypoint Environments

for re-planning all plans from scratch (described as the optimal agent in the experiment description). It should be noted that in both cases, even when only one action has been affected, the planner is called upon to maintain a correct plan library. The difference between the two costs in the experiment is that in one case only one plan needs to be found by the planner to patch a failing action, whereas in the other case, the whole contents of the plan library are re-planned.

The ‘Scaled’ column in the reconfiguration cost tables provides normalised values to compare costs between the two differently sized plan libraries (4-node and 9-node). This normalisation accounts for differences in scale by calculating a per-plan cost and extrapolating it to the size of the opposite library. Specifically, the scaled value is calculated as follows:

$$\text{Scaled Cost} = \left(\frac{\text{Total Re-planning Cost}}{\text{Plans in Original Library}} \right) \times \text{Plans in Opposite Library}$$

For example, if the 4-node library contains 12 plans and the 9-node library contains 40 plans, the scaled cost for the 4-node library would estimate its total cost as if it had 40 plans. This allows a direct comparison of reconfiguration efficiency, independent of library size.

For the first two experiments on the 4-node map size, the time needed for replanning a single plan remains constant at 0.071 seconds, as does the time for a full replan, which stays at 0.93 seconds. These consistent results lead to a scaled timing cost of 2.93, the time for finding a patch marginally increases to 0.074 seconds, and the time for a full replan rises to 0.97 seconds, resulting in a slightly elevated scaled timing cost of 3.06 seconds.

Whilst the AI planner handles increases in failure well in this scenario, it can be seen that only re-planning a single patch takes less time than re-planning full plan libraries. It can be argued that the experiment domain is very limited, although the cost of re-planning a patch remains the same for larger domains, as it is still only a single plan that requires re-planning, even when the domain is larger. Plan libraries

do not have a size limit, and the larger a plan library grows, the longer it will take to re-plan all of the plans in the library. This does not bode well for agents that require a full re-plan of their plan library in a large and complex domain.

9 Node Map

Table 7.7 showcases the replanning timing costs in a larger context, specifically for 9-node waypoint environments, across the same three experiments: 1 failure, 50% failure, and 100% failure.

Experiment Type	Time(s) for Patch (ms)	Time(s) for Full Replan (ms)	Scaled
1 Failure	0.075	3.09	0.98
50% Failure	0.073	3.00	0.95
100% Failure	0.074	3.05	0.97

Table 7.7: Replanning Timing Costs for 9-node Waypoints Environments

This table gives a comparison for replanning timing costs detailed in Table 7.6. The ‘Scaled’ column in each table calculates the value of the full replanning time on the opposite domain.

In the 9-node environment, the time required for replanning a patch for the failing action remains relatively stable across all failure levels, ranging from 0.073 to 0.075 seconds. Similarly, the time required for replanning from scratch demonstrates a slight increase compared to Table 7.6, with values from 3.00 to 3.09 seconds. Consequently, the scaled timing costs in Table 7.7 also display a narrow range, fluctuating between 0.95 and 0.98 seconds.

Comparing the results of Table 7.6 to Table 7.7, it is evident that the AI planner is able to maintain efficient replanning timing costs even when scaling up to a larger 9-node waypoint environment.

7.5.4 Reconfiguration Cost

4 Node Map

Table 7.8 shows the memory costs for re-planning 4-node waypoint plans, comparing the cost to find a patch for a failing action against the costs for replanning a full plan library.

Experiment Type	Patch(KB)	Full Replan(KB)	Usage Rate(KB/s)	Scaled(KB)
1 Failure	10536	136670	148394	431036
50% Failure	10536	136670	148394	431036
100% Failure	10536	137252	142378	432872

Table 7.8: Memory Cost for Replanning 4 Node Waypoint Plans

It is important to reiterate that the system being evaluated only needs to re-plan one plan using an AI planner, which is different from systems that are not using incremental planning algorithms or those that choose to re-plan the full plan library.

In this experiment, the memory costs for finding a patch and replanning a full plan library do not increase over the first two experiment types: 1 failure and 50% failure, with values of 10,536 KB and 136,670 KB. This shows that an increase in failure from a single failure to 50% failure on a 4-node map size does not increase the memory costs for replanning. The usage rate also remains the same at 148,394 KB/s, resulting in a scaled memory cost of 431,036 KB. However, at 100% failure in the capability library, there is a small increase in the memory cost for replanning all of the library from scratch, reaching 137,252 KB. At the same time, the usage rate decreases to 142,378 KB/s, leading to a slightly higher scaled memory cost of 432,872 KB.

9 Node Map

Table 7.9 shows the memory costs for re-planning 9-node waypoint plans.

Experiment Type	Patch(KB)	Full Replan(KB)	Usage Rate(KB/s)	Scaled(KB)
1 Failure	10536	429166	140480	136077
50% Failure	10536	431973	144328	136967
100% Failure	10536	433398	142378	137418

Table 7.9: Memory Cost for Replanning 9 Node Waypoint Plans

In this experiment, the memory cost for finding a patch plan remained constant across all three experiment types: 1 failure, 50% failure, and 100% failure, with a value of 10,536 KB. It is worth noting that the full replan memory costs and usage rates fluctuate as the failure rate varies, which impacts the resulting scaled memory costs.

A key observation from Table 7.9 is the increase in memory costs for fully re-planning when compared to the 4-node waypoint plans (Table 7.8), highlighting the

challenges that emerge with larger domains. Whilst the increase is marginal, plan libraries can increase in size considerably when more nodes are required, with 4 nodes requiring 13 plans and 9 nodes requiring 41 plans. In a fully connected map, 13 plans increases to 73 plans for 4 to 9 node maps. Nonetheless, the AI planner efficiently handles the increased complexity of a capability library for a 9-node domain. It remains clear that the size of the plan library is the largest unavoidable factor responsible for increased planning cost.

7.5.5 Criteria Satisfaction

Revisiting the evaluation criteria from Section 7.1.3, the complete system's performance can be judged against the factors discussed in the original specification: success rate, robustness, scalability, time cost, and memory cost.

Success Rate

The success rate is the average completion percentage, measuring successful executions over the total number of executions. The success rate revealed the system's performance when faced with increasing action failure rates.

In both waypoint environment variants, the system continues to complete a significant percentage of tests even when the action failure rate increases, highlighting its adaptability to varying conditions. However, it was noted that in less connected domains like the hallway environment, which contained actions that could not be re-configured, the success rate declined significantly regardless of the agent's attempts to continue. Whilst this was not an ideal outcome, the agent still completed routes, albeit not many with an 8% completion rate being achieved at 100% failure rate, shown in Figure 7.18.

Robustness

Robustness is assessed as the ability to manage reduced domain connectivity, and high failure rates. In the series of experiments performed in the evaluation, the system was subjected to varying levels of domain connectivity and high failure rates. The system managed this increased complexity well, although there was no evaluation of its performance when handling noise.

Further experimentation is required here, ideally with real sensor data from a physical deployment. From the behaviours exhibited in the existing evaluation, if noise

resulted in an action description and plan update, the system would right any ‘wrong’ action descriptions and plans when the agent’s failure threshold is breached for the affected action(s). If the noise is consistent over many action executions, then it would be accounted for in the action description after being learned, and this would remain until any changes occur. This would be the desired behaviour in a dynamic environment.

Scalability

Scalability is assessed as the system’s ability to remain functional and maintain performance when handling larger domains. In this case, scalability is measured by comparing the success rate, time cost and memory cost between differently sized domains. As this system only needs to find a single plan to patch all of the plans for the failing action in order to continue with routes, scalability is already improved against systems that need to re-plan the entire plan library to continue. This advantage is particularly evident when comparing the 4-node and 9-node waypoints environments, where the system still exhibits strong performance despite some decrease in success rate and an increase in the number of actions learned.

Time Cost

Time cost is measured as the amount of time, measured in milliseconds, taken by the AI planning algorithm to generate a solution. The system as a whole maintains consistent and efficient time costs across various action failure rates for the 4-node and 9-node waypoint environment experiments. When compared to the optimal agent that replans the full library from scratch, an average of 2.98 seconds is saved every time the agent encounters failure in a 9-node waypoints environment, and 0.87 seconds is saved in a 4-node waypoints environment.

Memory Cost

Memory cost is measured as the overall amount of computer memory, in Kilobytes, (KB), used by the AI planning algorithm. The system efficiently manages memory costs by reducing the re-planning workload for the AI planner. Memory costs remain lower compared to the costs of re-planning a full-plan library, even as complexity grows. When compared to the optimal agent that replans the full library from scratch, there is an average reduction of 420,976 kilobytes of memory used every time the agent

encounters failure in a 9-node waypoints environment, and a reduction of 126,976 kilobytes in a 4-node waypoints environment.

Chapter 8

Conclusions

In this chapter, the research questions and objectives from Chapter 1 are revisited and the contributions made to adaptable BDI reasoning research are detailed. The strengths, weaknesses, and limitations are also discussed. Areas of future work are identified, including suggested approaches for each area. Lastly, a final summary concludes the research presented in this thesis.

8.1 Contributions, Research Questions, and Objectives

Returning to the initial research questions and objectives outlined in Chapter 1 helps to reflect on the significance and impact of the contributions made in this thesis. In this section, the contributions are presented by chapter, showing how they address the four research questions and eight objectives.

8.1.1 Chapter 5: Durative Actions

This chapter introduces the first novel contribution of the thesis:

An extended formal semantics for durative actions with explicit success and failure conditions in BDI languages. An extension of existing BDI action theory was developed to consider actions with explicit success, failure, and abort conditions, and durations, along with the addition of an *action log* that acts as a record of all action executions. The semantics are general to BDI languages that have actions with explicit pre- and post-conditions.

Research Question 1 asked how BDI languages can handle actions that have pre-conditions, post-conditions, durations, and terminating conditions. This prompted *Objective 1.a* to develop a theory of durative actions for BDI languages and *Objective 1.b* to evaluate the developed theory through an implementation in an existing BDI language.

The requirements to implement actions that have durations into BDI languages were thoroughly explored in this chapter. Additionally, a theory of actions with pre-conditions, post-conditions, durations, and terminating conditions for BDI languages was developed, which firmly addresses *Objective 1.a*.

The process for modifying existing action handling mechanisms in the GWENDOLEN programming language to support actions with pre-conditions, post-conditions, durations, and terminating conditions was documented in Chapter 5. The process was then used to implement a complete system capable of running in simulated dynamic environments. The implementation was subjected to experiments hosted in the simulated environments, allowing the extended action theory to be recognised in practice, and subsequently addressing *Objective 1.b*.

In response to *Research Question 1*, it can be concluded that BDI languages can effectively handle actions that have durations. This was demonstrated through both the theoretical extension of BDI semantics to support pre-conditions, post-conditions, durations, and terminating conditions presented in Chapter 5 and its successful implementation and experimental validation in the GWENDOLEN agent programming language.

8.1.2 Chapter 6: Detecting Failures and Learning New Action Descriptions

Two more novel contributions were developed in this chapter:

BDI Agent failure detection. A theory and implementation for detecting failures during execution for BDI agents was developed and analysed.

Learning new action descriptions. A mechanism for learning updated action descriptions was developed and implemented into an existing BDI programming language.

The content and contributions of this chapter address both *Research Question 2* and *Research Question 3*.

Research Question 2 asked how additional action theory for BDI languages could enable plan library reconfiguration. *Objective 2.a* suggests the development of an “Action Log” to allow the system to track the performance of actions throughout their life-cycle. *Objective 2.b* aims to use the “Action Log” with a method for detecting recurrent failures.

The concept of the “Action Log” was introduced in Chapter 5 and further developed in Chapter 6, addressing both *Objective 2.a* and the failure detection mechanism’s requirement of a system to track the performance of actions. The Action Log uses the extended action theory proposed in Chapter 5 to keep a record of the action outcomes and the change in beliefs before and after execution. The log enables the detection of failure and provides the necessary inputs for learning new action descriptions.

A method of detecting persistent action failures was developed using the Action Log data, addressing *Objective 2.b*. The method is primitive, relying upon fixed thresholds to trigger failure, although the evaluation experiments show that this simple method is effective in the scenarios considered, and does not require any greater complexity.

In answer to *Research Question 2*, the extended action theory proposed in this thesis enabled a method for tracking the performance of actions, detecting failures, and learning new action descriptions, that could be used to inform a framework for reconfiguring agent plans. These extensions required extended action theory to operate and they provided the necessary additions to perform agent plan reconfiguration.

Research Question 3 asks how machine learning can be used to update action descriptions. *Objective 3.a* aims to directly address this question by calling for the development of a machine learning algorithm to construct action descriptions. *Objective 3.b* aimed to build upon this algorithm, integrating the action description learning method back into the reconfiguration framework for which the failure detection method was developed.

Objective 3.b also called for the integrated system to preserve safety properties during the learning and reconfiguration process. Concerning this requirement, it was identified that action failure caused by unforeseen circumstances might have already violated safety properties. Whilst this objective was not entirely satisfied, the decision to prevent potentially unsafe and unverified explorative behaviour whilst learning updated action descriptions maintains the theme of prioritising safety, and encourages

efforts made in future works to continue with this theme.

In answer to *Research Question 3*, machine learning can be used to learn new action descriptions, using extended BDI action theory, a historical log of action outcomes, and triggered by a method for detecting persistent failure.

8.1.3 Chapter 7: Evaluation

The extensions proposed in this thesis were integrated into a cohesive system, detailed in Chapter 4, allowing for a meaningful evaluation. The fully integrated system formed the fourth novel contribution of this thesis:

Integration of action failure detection and a mechanism for learning updated action descriptions with a plan repair framework. An implementation of an action failure detection method and a mechanism for learning updated action descriptions was integrated with Cardoso et al. (2019)'s plan library reconfiguration framework that can repair agent plans from existing plans.

This contribution represents the culmination of all of the extensions proposed in this thesis, and the evaluation of the resulting system directly addresses *Research Question 4*.

Research Question 4 asks how a system with the proposed extensions in this thesis would perform. *Objective 4.a* states that experiments should be performed in simulated environments to measure the performance, and *Objective 4.b* seeks to use the performance measurements to evaluate the system against selected criteria.

In the results of Chapter 7, the proposed system was evaluated using simulated environments against five criteria to measure its performance across success rate, time cost, memory cost, robustness, and scalability. The experiments performed in Chapter 7 generated the measurements of performance required by *Objective 4.a*. The proposed system demonstrated an increase in the operational lifetime of agents after encountering failure, with an overall average of 55% of agents still managing to complete their assigned missions after experiencing failure on every available action. The experiment results were then used to evaluate the performance of the system against the five selected criteria, satisfying *Objective 4.b*. The challenges, weaknesses and limitations of the system were identified during this evaluation and are discussed in greater detail in Sections 8.1.4 and 8.2.

8.1.4 Challenges

During the course of this research, three main challenges for current and future work in adaptable BDI reasoning were identified. Specifically, finding a consistent definition of failure, writing and implementing extended operational semantics to be applicable for all BDI languages, and verifying machine learning algorithms for safety-critical systems.

Defining Failure Whilst the term “failure” has maintained a strong link to its etymological origin (primarily used to describe a lack of success), it could be considered unwise to use such a term to define a problem in a system. This becomes clear when a “failure” could merely describe an inconvenience to an agent whilst simultaneously representing another problem in the same system that is a danger to human life if left unchanged.

Integrating with BDI languages As part of the research in this thesis, an operational semantics was developed to extend current BDI action theory to handle actions that have explicit durations, pre-conditions, post-conditions, and terminating conditions. It should be stated that the operational semantics were intended to be widely applicable to BDI languages, yet the semantics assumed an action representation with explicit pre-conditions and post-conditions as a foundation, which is not the case for some BDI languages. However, this assumed action representation enabled the failure detection method to detect recurrent action failure, the machine learning algorithm to learn new action descriptions, and the AI planner to reconfigure the plan library using the PDDL action specification. It is clear that there was a necessary balance between supporting more BDI languages and expanding the system to support more extensions. It cannot be guaranteed that an extended operational semantics integrates fully with all existing BDI languages. Although future integration can be aided by clearly defining the requirements needed for a compatible language, and remaining as general to most BDI theory as possible when defining new semantics.

Verifying Machine Learning The verification of agent programs generated by machine learning algorithms was not evaluated in the scope of this work. Nevertheless, the semantics and implementation proposed in this thesis were applied to GWENDOLEN, a BDI language that is integrated with a framework that is capable of verifying agent programs. Accordingly, research into the verification of agent programs produced by

machine learning algorithms is considered as a logical next step for future work. However, the introduction of increasingly complex machine learning algorithms presents great difficulty for verification as some algorithms cannot (currently) be directly verified (Van Wesel & Goodloe 2017). As a consequence, it should be noted that solutions with complex machine learning algorithms could be unsuitable for scenarios where learning from failure is not safe (e.g., autonomous drones), where it would be safest to execute a controlled stop to the system rather than attempting a recovery. For example, assume an autonomous drone is experiencing persistent rotor failures that are causing a navigation action to lose an extra 10 metres in altitude. Triggering an exploratory machine learning algorithm to learn a new action description for this action could result in the drone losing all of its altitude and causing significant damage to itself and its surroundings when it hits the ground.

8.2 Future Work

A major aspect of future work will be adapting the framework to enable the usage of action descriptions containing variables. A key feature of many BDI languages is the use of variables and unification in plans, to enable one plan to apply in many different situations depending upon the instantiation of its parameters. There are two aspects to this challenge. Firstly, when an action is executed in a BDI language, it is almost always the case that its variable parameters are instantiated (Dennis et al. 2007). Meaning that although an action description of the form

$$\{at(X)\}_{\text{move}(X, Y)} \{-at(X), +at(Y)\}$$

where X and Y are variables, it is only ever called as, $\text{move}(0, 1)$ or $\text{move}(1, 2)$. Consequently, the process of synthesising new descriptions from the action log will need to make use of generalisation techniques to identify the relevant abstract action description from the instantiated action descriptions recorded in the action log so that the updates to the action description will be applied to all of the instantiated versions. It may also be necessary to split action descriptions by synthesising new pre-conditions indicating that, in some situations, the action still behaves as originally assumed, but in others, it does not.

Secondly, STRIPS-type planners, while they frequently use action descriptions that

contain variable parameters, do not generally plan using initial and goal states that contain variables. This includes the AI planner embedded in the implementation from Cardoso et al. (2019) — therefore, this planner would need to be replaced with one capable of handling variables in initial states and goals.

It should be noted that synthesising entirely new actions is considered a separate endeavour from splitting action descriptions and synthesising their pre-conditions. However, future work could investigate action description synthesis by safely incorporating exploratory learning algorithms into the system proposed in this thesis.

A further improvement to the integration of Cardoso et al. (2019)’s framework would be to implement caching for plans that have been translated to the AI planner’s domain definition language. Whilst the individual memory time and cost for translating a single plan library might not be significant, the translation process is invoked every time an action breaches its failure threshold and there is no limit to the number of times this can happen.

The implementation of the *Action Log*, discussed in Chapter 5, currently relies on a fixed-size array list to record action outcomes. This approach is straightforward and performed effectively in the scenarios presented in the evaluation. However, the fixed length of the array results in older entries being discarded when new entries are added, potentially losing valuable historical data. To address this, a dynamic array list structure could be used instead, allowing for more comprehensive logging, though if actions are executed with high frequency this could generate a very large list which will affect memory time and costs. Alternatively, a fixed-size list for each action, with the size determined by the failure threshold, could be implemented. This approach keeps the log manageable and efficient whilst increasing the amount of historical data that can be used for learning new action descriptions.

The algorithm for learning new action descriptions, presented in Chapter 6, would benefit from more sophistication. At present the algorithm treats all changes in belief after an action execution as one group. For example, consider a situation where two robots are both working in an area. Sometimes, after moving between waypoints (e.g., from waypoint 0 to waypoint 1) the agent also perceives the presence of the second robot. In this case, the current action log would sometimes record

$$\{+at(1), -at(0), +second_robot\}$$

as the belief change and sometimes record

$$\{+at(1), -at(0)\}.$$

Currently, Algorithm 2 treats these two sets of post-conditions entirely separately and is unable to recognise that $+at(1)$ and $-at(0)$ occur in both sets. This could be addressed by weighting each term appearing in the set of belief changes individually, rather than as a group, which could enable the construction of post-conditions that better reflected the actual action behaviour.

At present the planning problem that is sent to the STRIPS planner is formulated from the description of the failed action alone and does not account for the context in which the action appears. As stated in Section 2.1 of Chapter 2, many BDI plans are expressed in terms of some *guard*, which can be considered a pre-condition for the whole plan, and a *goal* which can be considered a goal state for the plan (Cardoso et al. 2019, Dennis & Farwer 2008). Techniques such as regression planning (Russell & Norvig 2020) could be used to infer from the plan’s guard and goal, and the pre-conditions and post-conditions of any other actions in the plan, what the actual state of the agent is likely to be at the point the failed action was executed and which of the failed action’s post-conditions were necessary to achieve the goal of the plan. This could be computed offline and stored as part of the plan description which would introduce more flexibility into the patching mechanism, allowing plans to be patched even if an exact replacement for the failing action could not be found. It will also reduce the risk that the computed patch might contain additional post-conditions that will break the plan — for instance, suppose the failed action is:

$$\{pr_1\}a_1\{+po_2\}$$

and the computed patch is a_2, a_3 where a_3 ’s post-condition is:

$$\{+po_2, +po_3\}$$

Now consider a plan:

$$e:\text{guard} \leftarrow a_1, a_4$$

where the description of a_4 is:

$$\{\neg po_3\}a_4\{+po_4\}.$$

If a_1 is replaced in this plan with the patch then a_4 will no longer be applicable and the plan will break. More context-sensitive construction of the planning problem should be able to account for this and avoid creating a patch that will break the plan. There is existing work on context-sensitive plan repair in the AI planning domain (der Krogt & de Weerd 2005), although this was not considered within the scope of this thesis.

The use of the GWENDOLEN language which is linked to the AJPF model-checking tool and the MCAPL framework (Dennis 2018), opens the possibility of verifying the patched plans produced by the framework. However, the AJPF model-checker ordinarily performs verification slowly. If the agent existed in an environment where there were periods of inactivity, then it would be possible for re-verification to take place to ensure that the agent's plans continued to adhere to any specified properties, but in an environment where patching needs to occur quickly then this may not be feasible. If the reconfiguration mechanism was adapted to be sensitive to the context in which an action was invoked then it should be possible to establish idealised results about the safety of patches, at least in environments where the only things changing the environment are the agent's own actions. It might also be possible to treat actions appearing in plans as sequences of abstract actions of length up to l , with the abstract actions having no specified behaviour during verification, forcing the verification to consider all possible action outcomes. This is how the verification of abstract actions is handled by the AJPF model-checking tool used in the MCAPL distribution of GWENDOLEN. If the verification is successful, this would allow plans to be patched with any sequence of actions of length less than l , but the resulting state space for verification is likely to be unwieldy and include consideration of many action outcomes that are either unlikely or impossible, forcing, in turn, the inclusion of fail-safe plans within the agent to handle behaviour that can never occur, resulting in "crufty" code. A solution to this could be to pre-verify a set of possible repairs based on anticipated failure modes, and if possible, the approach should choose one of these pre-verified repairs.

A physical implementation of the waypoints environment would enable testing with exposure to realistic noise, complementing the evaluations performed in the simulated environments. The framework presented in this research is designed to cope with unplanned changes to action descriptions in dynamic environments. Whilst simple inputs used in the example reflect the same inputs expected from sensors mounted on a robot, it would be naive to assume that a comprehensive evaluation has been performed without considering the effect of noise in a real-world environment. It is important for a system designed for failure recovery to remain robust in all conditions.

The extent to which long-term autonomy can be achieved through the generation of amended action descriptions and the patching of plans is an open question. Scenarios similar to the evaluation scenarios in Chapter 7, involving navigation around waypoints linked in a graph structure, are relatively common (Roh et al. 2020), and it is reasonable to suppose that over time paths between waypoints might alter. Similarly, it is easy to imagine some actions, over time, exhibit partial success. For example, not as much material is moved, or things are not moved as far, or as fast when compared to the original action descriptions. Therefore modified action descriptions may be useful in patching plans by repeating actions or combining them with other compensatory actions. Case studies could be undertaken to assess how common it is for action degradation to result in mission failures, and if patching plans or combining actions with compensatory actions could improve the chances of mission success. The framework presented in this research could be combined with mechanisms for weakening mission specifications (D’Ippolito et al. 2015, Ingrand & Ghallab 2017), for instance, by dropping some goals that were no longer obtainable, while continuing to pursue others.

Conducting experiments in a physical environment would have provided a more realistic benchmark for agent performance, where the effect of noisy inputs could have been evaluated. However, progress was hampered by the restrictions made during a global pandemic. Physical experimentation could have unveiled bugs, weaknesses, and limitations earlier in the process, allowing more time for debugging and improvements.

8.2.1 Limitations

The limitations of the approach followed in this thesis are: the domain specificity of the resulting implementation; the requirement to manually program initial action descriptions; the inheritance of safety guarantees; and finally, the inherent uncertainty of agent perception from undetected environmental changes. Each of these limitations and their implications were identified and do not present major issues.

Domain Specificity When compared to exploratory agent programs, designed to navigate and map unknown domains, this system would not be able to operate with the same behaviour in its current implementation. The environment must be programmed to include the expected pre- and post-conditions for each action, as these are used in the specification of success, failure and abort conditions. This may not be a significant limitation, as actions for BDI agent programs are already required to be programmed

before execution, although this additional requirement has to be considered in the design of agent programs that make use of these extensions. Actions could be specified differently to address this limitation. For example, instead of defining actions using static beliefs, where pre-conditions and post-conditions rely on fixed absolute values, such as:

$$\{at(0,0)\}\{move(5,5)\}\{at(5,5)\}[5s]$$

Pre-conditions and post-conditions could be constructed using relational expressions, where the same action could be redefined with variables representing the agent's starting position (X, Y) , enabling the action to describe the effect of multiple similar actions at once by removing the static reference to the agent's starting location:

$$\{at(X,Y)\}\{move(5,5)\}\{at((X+5),(Y+5))\}[5s]$$

This relational specification avoids hardcoding static values, making action descriptions reusable for multiple scenarios without reprogramming.

Scalability Due to the environment requiring manually pre-programmed actions and action descriptions, the size of the environment would be limited by the availability of development time. This is a problem that applies generally to hierarchical AI planning algorithms (Silver et al. 2023), as they require domain-specific abstractions to operate. These abstractions are usually manually programmed and therefore this limitation cannot be avoided without using an alternative method of planning. Silver et al. (2023) tackled this limitation by performing bilevel planning combined with a method for learning predicates in the domain during other objectives.

Inheriting Safety Guarantee Caveats The safety guarantees of the proposed system are based on several assumptions made about the relationship between AI planning, action descriptions, and the learning process.

Concerning the generated patch from the AI planner, if the action descriptions are correct at the time of creating the domain for the AI planner, the system will generate plans that achieve the desired goal. This assumption stems from the principle of AI planning where plans are created based on accurate models of actions and their effects.

In this system, as long as the number of action failures remains below the action failure threshold for each action, the algorithm will not modify the action description. This requires failure thresholds and failure conditions to be set appropriately for each

action, so that unexpected behaviours are acknowledged, and so that the action descriptions can be kept accurate to the action's effect on the environment. If the failure thresholds and conditions are not set properly, this could lead to situations where the system continues to use an outdated action description, potentially impacting safety. The opposite case can also cause issues: stating thresholds that are too low, or conditions that are too close to the success conditions, could trigger failure detection to learn new action descriptions for actions that aren't truly exhibiting patterns of persistent failure.

Whilst the assumptions inherited from the AI planning domain cannot be entirely circumvented, the assumptions and weaknesses associated with defining failure thresholds and conditions could be addressed by using more intelligent methods for defining them, with scaled and dynamic values and conditions.

Inherent Uncertainty Failure detection heavily depends on agent perception. If actions are failing due to unknown factors, undetected by the agent, then new action descriptions cannot be learned. It is likely impossible to predict the origin of all potential failures, resulting from the lack of appropriate sensors. Even when equipped with all of the available sensors for all foreseeable circumstances, sensor failures can occur. If sensor failure occurred in the current implementation, the agent would just learn that the action no longer achieves any post-conditions. Which could still be viewed as learning a new action description. However, this would likely make the reconfiguration of plans more difficult, and possibly affect the success rate of the resulting plan library as the updated action description(s) would be using perceptions reported by a failing sensor rather than from a working sensor that would report what is actually happening in the environment.

8.2.2 Weaknesses

The weaknesses of this research are minor in contrast to the benefits. These include a reliance on accurate failure detection; having a limited scope for learning; and its susceptibility to noise.

Incomplete failure detection Agents using this system rely on the accurate detection of failure to operate successfully. This concerns the accuracy of both detecting what has happened and then detecting failure as soon as it has happened. The use of action durations limits how frequently potential failure can be detected, meaning failures will

not immediately trigger detection unless additional ‘abort’ conditions are stated. The accuracy of failure detection is directly tied to the accuracy of the perceptions received by the agent, which is reliant upon accurate and reliable sensor data that cannot be directly improved by extending the agent’s underlying software. However, the method for detecting failures proposed in this thesis could be extended to let agents flag action executions as failures immediately after the failure conditions are believed, instead of after the action’s duration has lapsed.

Limited learning scope The post-conditions in action descriptions can only be discovered if the agent experiences them during the execution of the action. The exploration of new actions is not encouraged in this system, due to the implications for safety and possible violations of the verified program specification. Due to this, the learning behaviour of this system is not entirely comparable to other methods of machine learning, e.g. reinforcement learning, where exploration is actively incentivised to maximise the reward potential. The extent of the safety implications introduced by using exploratory machine learning algorithms has not been fully explored in the scope of this thesis, and is considered to be part of future work.

Fixed failure thresholds There is considerable reliance on the pre-programmed failure thresholds for actions - which are not updated alongside the process of learning action descriptions. This could result in a disparity between the confidence in the action and how often it should be allowed to fail before triggering the learning process. To address this concern, failure thresholds could become dynamic: updating as part of the action’s description. If a level of confidence can be calculated for the solution found by the learning process, using a comparison of the weights assigned to the candidate post-conditions, this could be used to inform a new failure threshold.

It is important to note that these weaknesses and limitations impact only the proposed extensions and do not affect the core functionality of the base agent architecture, which continues to operate effectively without disruption from these issues. Nevertheless, the extensions proposed in this research, whilst bearing weaknesses and limitations, only present improvements to the performance of agents programmed using the original implementation.

8.3 Conclusion

The primary focus of this research was to enable autonomous agents to adapt to changes in dynamic environments. The research found that agent programming languages can be extended to enable this. When tested in simulated dynamic environments, the extensions demonstrated an increase in the operational lifetime of agents after encountering failure, with an overall average of 55% of agents still managing to complete their assigned missions after experiencing failure on every available action. As part of the language extensions, the research confirmed during the evaluation presented in Chapter 7, that BDI languages can effectively handle actions with durations, and existing mechanisms can be adapted to consider action durations without major reconstruction as presented in Chapter 5. Additional mechanisms were required for detecting failures from action execution logs, although only a simple implementation of this was required to allow the machine learning algorithm to operate effectively. The algorithm used these additional mechanisms to create weighted collections of post-conditions, favouring recency and frequency, to rank and select the most appropriate action description from a list of action post-conditions, as presented in Chapter 6. The experimentation results confirmed that this implementation was an effective solution that avoids the use of more complex machine-learning algorithms.

In conclusion, the feasibility and efficacy of extending agent programming languages to handle online adaptation to dynamic environments has been explored. Three extensions to BDI programming languages were developed; theory for the underlying mechanisms of each extension was created; a methodology for the implementation of the extensions was detailed; and the theory was evaluated by subjecting the resulting implementation to a large number of inputs using simulated dynamic environments. Finally, the evaluation results highlighted a clear increase in the operational lifetime of BDI agents when using the extensions developed in this research, compared to agents operating without them. This thesis provides an important foundation for creating more adaptive and resilient autonomous agents, capable of operating effectively in dynamic and unpredictable environments.

Bibliography

- Adam, C. & Gaudou, B. (2016), ‘BDI agents in social simulations: a survey’, *The Knowledge Engineering Review* **31**(3), 207–238.
- Aitken, J. M., Veres, S. M., Shaukat, A., Gao, Y., Cucco, E., Dennis, L. A., Fisher, M., Kuo, J. A., Robinson, T. & Mort, P. E. (2018), ‘Autonomous Nuclear Waste Management’, *IEEE Intelligent Systems* **33**(6), 47–55.
- Alloghani, M., Al-Jumeily, D., Mustafina, J., Hussain, A. & Aljaaf, A. J. (2020), *A Systematic Review on Supervised and Unsupervised Machine Learning Algorithms for Data Science*, Springer International Publishing, pp. 3–21.
- Archibald, B., Calder, M., Sevegnani, M. & Xu, M. (2024), ‘Quantitative modelling and analysis of BDI agents’, *Software and Systems Modelling* **23**(2), 343–367.
- Arora, A., Fiorino, H., Pellier, D., Métivier, M. & Pesty, S. (2018), ‘A review of learning planning action models’, *The Knowledge Engineering Review* **33**, e20.
- Bahoo, S., Cucculelli, M., Goga, X. & Mondolo, J. (2024), ‘Artificial intelligence in Finance: a comprehensive review through bibliometric and content analysis’, *SN Business & Economics* **4**, 1–46.
- Boissier, O., Bordini, R. H., Hubner, J. & Ricci, A. (2020), *Multi-agent oriented programming: programming multi-agent systems using JaCaMo*, Intelligent Robotics and Autonomous Agents, MIT Press.
- Bolander, T. (2017), A gentle introduction to epistemic planning: The DEL approach, in S. Ghosh & R. Ramanujam, eds, ‘Proceedings of the Ninth Workshop on Methods for Modalities’, Vol. 243 of *EPTCS*, pp. 1–22.
- Bordini, R. H. & Hübner, J. F. (2010), Semantics for the Jason Variant of AgentSpeak (Plan Failure and some Internal Actions), in H. Coelho, R. Studer & M. J.

- Wooldridge, eds, 'Proceedings of the 19th European Conference on Artificial Intelligence', Vol. 215 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, pp. 635–640.
- Bordini, R. H., Hübner, J. F. & Vieira, R. (2005), Jason and the Golden Fleece of Agent-Oriented Programming, in R. H. Bordini, M. Dastani, J. Dix & A. E. F. Seghrouchni, eds, 'Multi-Agent Programming: Languages, Platforms and Applications', Vol. 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, Springer, pp. 3–37.
- Bordini, R. H., Seghrouchni, A. E. F., Hindriks, K. V., Logan, B. & Ricci, A. (2021), Agent Programming in the Cognitive Era, in F. Dignum, A. Lomuscio, U. Endriss & A. Nowé, eds, 'Proceedings on the 20th International Conference on Autonomous Agents and Multiagent Systems', ACM, pp. 1718–1720.
- Bratman, M. E. (1987), *Intention, Plans, and Practical Reason*, Vol. 10, Harvard University Press.
- Cardoso, R. C., Dennis, L. A. & Fisher, M. (2019), Plan Library Reconfigurability in BDI Agents, in L. A. Dennis, R. H. Bordini & Y. Lespérance, eds, 'Proceedings of the 7th International Workshop on Engineering Multi-Agent Systems', Vol. 12058 of *Lecture Notes in Computer Science*, Springer, pp. 195–212.
- Cardoso, R. C., Farrell, M., Luckcuck, M., Ferrando, A. & Fisher, M. (2020), Heterogeneous Verification of an Autonomous Curiosity Rover, in R. Lee, S. Jha, A. Mavridou & D. Giannakopoulou, eds, 'Proceedings of the 12th International Symposium on NASA Formal Methods', Vol. 12229 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 353–360.
- Cardoso, R. C. & Ferrando, A. (2021), 'A Review of Agent-Based Programming for Multi-Agent Systems', *Computers* **10**(2), 16.
- Celorrío, S. J., de la Rosa, T., Fernández, S., Fernández, F. & Borrajo, D. (2012), 'A review of machine learning for automated planning', *Knowledge Engineering Review* **27**(4), 433–467.
- Chen, W., Liu, T., Lan, Y., Ma, Z. & Li, H. (2009), Ranking Measures and Loss Functions in Learning to Rank, in Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I.

- Williams & A. Culotta, eds, 'Proceedings of the 23rd Annual Conference on Advances in Neural Information Processing Systems', Vol. 22, Curran Associates, Inc., pp. 315–323.
- Cirillo, M., Karlsson, L. & Saffiotti, A. (2010), 'Human-aware task planning: An application to mobile robots', *ACM Transactions on Intelligent Systems and Technology* 1(2), 15:1–15:26.
- Clarke, E. M. (1997), Model Checking, in S. Ramesh & G. Sivakumar, eds, 'Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science', Vol. 1346 of *Lecture Notes in Computer Science*, Springer, pp. 54–56.
- Cohen, P. R. & Feigenbaum, E. A., eds (1982), *The Handbook of Artificial Intelligence*, Vol. 3, William Kaufmann.
- Cunningham, P., Cord, M. & Delany, S. J. (2008), Supervised Learning, in M. Cord & P. Cunningham, eds, 'Machine learning techniques for multimedia: Case Studies on Organization and Retrieval', Cognitive Technologies, Springer, pp. 21–49.
- Dastani, M., van Riemsdijk, M. B. & Meyer, J. C. (2005), Programming Multi-Agent Systems in 3APL, in R. H. Bordini, M. Dastani, J. Dix & A. E. F. Seghrouchni, eds, 'Multi-Agent Programming: Languages, Platforms and Applications', Vol. 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, Springer, pp. 39–67.
- De Silva, L., Meneguzzi, F. R. & Logan, B. (2020), BDI Agent Architectures: A Survey, in C. Bessiere, ed., 'Proceedings of the 29th International Joint Conference on Artificial Intelligence', IJCAI, pp. 4914–4921.
- Dennis, L. A. (2017), Gwendolen Semantics: 2017, Technical Report ULCS-17-001, University of Liverpool, Department of Computer Science.
URL: <https://personalpages.manchester.ac.uk/staff/louise.dennis/pubs/ulcs-17-001.pdf>
- Dennis, L. A. (2018), 'The MCAPL Framework including the Agent Infrastructure Layer and Agent Java Pathfinder', *Journal of Open Source Software* 3(24), 617.

- Dennis, L. A. & Farwer, B. (2008), Gwendolen: a BDI language for verifiable agents, *in* ‘Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning’, AISB, pp. 16–23.
- Dennis, L. A., Farwer, B., Bordini, R. H., Fisher, M. & Wooldridge, M. J. (2007), A Common Semantic Basis for BDI Languages, *in* M. Dastani, A. E. F. Seghrouchni, A. Ricci & M. Winikoff, eds, ‘Proceedings of the 5th International Workshop on Programming Multi-Agent Systems’, Vol. 4908 of *Lecture Notes in Computer Science*, Springer, pp. 124–139.
- Dennis, L. A. & Fisher, M. (2014), Actions with Durations and Failures in BDI Languages, *in* T. Schaub, G. Friedrich & B. O’Sullivan, eds, ‘Proceedings of the 21st European Conference on Artificial Intelligence’, Vol. 263 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, pp. 995–996.
- Dennis, L. A. & Fisher, M. (2023), Autonomous Satellite Control, *in* ‘Verifiable Autonomous Systems: Using Rational Agents to Provide Assurance about Decisions Made by Machines’, Cambridge University Press, chapter 8, p. 104–126.
- Dennis, L. A., Fisher, M., Webster, M. P. & Bordini, R. H. (2012), ‘Model checking agent programming languages’, *Automated Software Engineering* **19**(1), 5–63.
- Dennis, L., Fisher, M., Lisitsa, A., Lincoln, N. & Veres, S. (2010), ‘Satellite control using rational agent programming’, *IEEE Intelligent Systems* **25**(3), 92–97.
- der Krogt, R. V. & de Weerd, M. (2005), Plan Repair as an Extension of Planning, *in* S. Biundo, K. L. Myers & K. Rajan, eds, ‘Proceedings of the 15th International Conference on Automated Planning and Scheduling’, AAAI, pp. 161–170.
- d’Inverno, M., Hindriks, K. V. & Luck, M. (2000), A formal architecture for the 3apl agent programming language, *in* J. P. Bowen, S. Dunne, A. Galloway & S. King, eds, ‘Proceedings of the First International Conference of B and Z Users’, Vol. 1878 of *Lecture Notes in Computer Science*, Springer, pp. 168–187.
- d’Inverno, M., Kinny, D., Luck, M. & Wooldridge, M. J. (1997), A formal specification of dmars, *in* M. P. Singh, A. S. Rao & M. J. Wooldridge, eds, ‘Proceedings of the 4th International Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages’, Vol. 1365 of *Lecture Notes in Computer Science*, Springer, pp. 155–176.

- D'Ippolito, N., Braberman, V. A., Sykes, D. & Uchitel, S. (2015), Robust degradation and enhancement of robot mission behaviour in unpredictable environments, *in* A. Filieri & M. Maggio, eds, 'Proceedings of the 1st International Workshop on Control Theory for Software Engineering', ACM, pp. 26–33.
- Doan, T. T., Yao, Y., Alechina, N. & Logan, B. (2014), Verifying heterogeneous multi-agent programs, *in* A. L. C. Bazzan, M. N. Huhns, A. Lomuscio & P. Scerri, eds, 'Proceedings of the 13th International conference on Autonomous Agents and Multi-Agent Systems', IFAAMAS/ACM, pp. 149–156.
- Evertsz, R., Lucas, A., Smith, C., Pedrotti, M., Ritter, F. E., Baker, R. & Burns, P. (2014), Enhanced behavioural realism for live fire targets, *in* 'Proceedings of the 23rd Annual Conference on Behaviour Representation in Modeling and Simulation', Curran Associates, Inc.
- Ferber, J. & Müller, J.-P. (1996), Influences and reaction: a model of situated multi-agent systems, *in* 'Proceedings of the Second International Conference on Multi-Agent Systems', AAAI, pp. 72–79.
- Ferrando, A. & Cardoso, R. C. (2022), Safety Shields, an Automated Failure Handling Mechanism for BDI Agents, *in* P. Faliszewski, V. Mascardi, C. Pelachaud & M. E. Taylor, eds, 'Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems', International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), pp. 1589–1591.
- Ferrando, A., Cardoso, R. C., Fisher, M., Ancona, D., Franceschini, L. & Mascardi, V. (2020), ROSMonitoring: A Runtime Verification Framework for ROS, *in* A. Mohammad, X. Dong & M. Russo, eds, 'Proceedings of the 21st Conference on Towards Autonomous Robotic Systems', Vol. 12228 of *Lecture Notes in Computer Science*, Springer, pp. 387–399.
- Fikes, R. E. & Nilsson, N. J. (1971), 'STRIPS: A new approach to the application of theorem proving to problem solving', *Artificial Intelligence* **2**(3), 189–208.
- Fox, M. & Long, D. (2003), 'PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains', *Journal of Artificial Intelligence Research* **20**, 61–124.
- Georgeff, M. P. & Ingrand, F. F. (1989), Decision-making in an embedded reasoning system, *in* N. S. Sridharan, ed., 'Proceedings of the 11th International Joint Conference on Artificial Intelligence', Morgan Kaufmann, pp. 972–978.

- Georgeff, M. P. & Lansky, A. L. (1987), Reactive Reasoning and Planning, in K. D. Forbus & H. E. Shrobe, eds, ‘Proceedings of the 6th National Conference on Artificial Intelligence’, Morgan Kaufmann, pp. 677–682.
- Goldberg, D. E. (1989), *Genetic Algorithms in Search Optimization and Machine Learning*, Addison-Wesley.
- Grounds, M. J. & Kudenko, D. (2007), Combining reinforcement learning with symbolic planning, in K. Tuyls, A. Nowé, Z. Guessoum & D. Kudenko, eds, ‘Proceedings of the 5th, 6th, and 7th European Symposium on Adaptive and Learning Agents and Multi-Agent Systems’, Vol. 4865 of *Lecture Notes in Computer Science*, Springer, pp. 75–86.
- Guerra-Hernández, A., Seghrouchni, A. E. F. & Soldano, H. (2004), Learning in BDI Multi-agent Systems, in J. Dix & J. A. Leite, eds, ‘Proceedings of the 4th International Workshop on Computational Logic in Multi-Agent Systems’, Vol. 3259 of *Lecture Notes in Computer Science*, Springer, pp. 218–233.
- Harland, J., Morley, D. N., Thangarajah, J. & Yorke-Smith, N. (2014), ‘An operational semantics for the goal life-cycle in BDI agents’, *Autonomous agents and multi-agent systems* **28**(4), 682–719.
- Harland, J., Morley, D. N., Thangarajah, J. & Yorke-Smith, N. (2017), ‘Aborting, suspending, and resuming goals and plans in BDI agents’, *Autonomous Agents and Multi-Agent Systems* **31**(2), 288–331.
- Helleboogh, A., Vizzari, G., Uhrmacher, A. & Michel, F. (2007), ‘Modeling dynamic environments in multi-agent simulation’, *Autonomous Agents and Multi-Agent Systems* **14**(1), 87–116.
- Hindriks, K. V. (2009), Programming Rational Agents in GOAL, in A. El Fallah Seghrouchni, J. Dix, M. Dastani & R. H. Bordini, eds, ‘Multi-Agent Programming: Languages, Tools and Applications’, Springer, Boston, MA, pp. 119–157.
- Hindriks, K. V. (2021), Programming cognitive agents in GOAL, Technical report, Delft University of Technology.
URL: <https://goalapl.dev/GOALProgrammingGuide.pdf>

- Hindriks, K. V., De Boer, F. S., Van der Hoek, W. & Meyer, J.-J. C. (1999), ‘Agent programming in 3APL’, *Autonomous Agents and Multi-Agent Systems* **2**(4), 357–401.
- Holland, J. H. (1992), *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, MIT Press.
- Howden, N., Rönnquist, R., Hodgson, A. & Lucas, A. (2001), Jack intelligent agents - summary of an agent infrastructure, in ‘Proceedings of the 5th International Conference on Autonomous Agents’, Vol. 6, Association for Computing Machinery, pp. 1–27.
- Hu, G., Miller, T. & Lipovetzky, N. (2022), ‘Planning with perspectives - decomposing epistemic planning using functional STRIPS’, *Journal of Artificial Intelligence Research* **75**, 489–539.
- Ingrand, F. F., Georgeff, M. P. & Rao, A. S. (1992), ‘An architecture for real-time reasoning and system control’, *IEEE expert* **7**(6), 34–44.
- Ingrand, F. & Ghallab, M. (2017), ‘Deliberation for autonomous robots: A survey’, *Artificial Intelligence* **247**, 10–44.
- Kuutti, S., Bowden, R., Jin, Y., Barber, P. & Fallah, S. (2021), ‘A survey of deep learning applications to autonomous vehicle control’, *IEEE Transactions on Intelligent Transportation Systems* **22**(2), 712–733.
- Lee, J., Katz, M., Agravante, D. J., Liu, M., Tasse, G. N., Klinger, T. & Sohrabi, S. (2022), ‘Hierarchical Reinforcement Learning with AI Planning Models’, IBM Research AI.
URL: <https://arxiv.org/abs/2203.00669>
- Li, J., Qiu, L., Tang, B., Chen, D., Zhao, D. & Yan, R. (2019), Insufficient data can also rock! learning to converse using smaller data with augmentation, in ‘Proceedings of the 33rd AAAI Conference on Artificial Intelligence’, AAAI Press, pp. 6698–6705.
- Logan, B. (2018), ‘An agent programming manifesto’, *International Journal of Agent-Oriented Software Engineering* **6**(2), 187–210.

- Mascardi, V., Demergasso, D. & Ancona, D. (2005), Languages for Programming BDI-style Agents: an Overview, *in* F. Corradini, F. D. Paoli, E. Merelli & A. Omicini, eds, ‘Proceedings of the 6th AI*IA/TABOO Joint Workshop ”From Objects to Agents”: Simulation and Formal Analysis of Complex Systems’, Pitagora Editrice Bologna, pp. 9–15.
- Mausam & Weld, D. S. (2008), ‘Planning with Durative Actions in Stochastic Domains’, *Journal of Artificial Intelligence Research* **31**, 33–82.
- Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D. & Wilkins, D. (1998), PDDL - The Planning Domain Definition Language, Technical Report TR-98-003, Yale Center for Computational Vision and Control.
- Meneguzzi, F. & de Silva, L. (2015), ‘Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning’, *Knowledge Engineering Review* **30**(1), 1–44.
- Meneguzzi, F. R. & Luck, M. (2008), Leveraging New Plans in AgentSpeak(PL), *in* M. Baldoni, T. C. Son, M. B. van Riemsdijk & M. Winikoff, eds, ‘Proceedings of the 6th International Workshop on Declarative Agent Languages and Technologies’, Vol. 5397 of *Lecture Notes in Computer Science*, Springer, pp. 111–127.
- Meyer, J.-J., Broersen, J. & Herzig, A. (2015), BDI logics, *in* H. van Ditmarsch, J. Y. Halpern, W. van der Hoek & B. Kooi, eds, ‘Handbook of Logics for Knowledge and Belief’, College Publications, chapter 10, pp. 453–493.
- Millican, P. & Wooldridge, M. J. (2014), Them and us: Autonomous agents in vivo and in silico, *in* A. Baltag & S. Smets, eds, ‘Johan van Benthem on Logic and Information Dynamics’, Springer, pp. 547–567.
- Mitchell, T. M. (1997), *Machine learning, International Edition*, McGraw-Hill Series in Computer Science, McGraw-Hill.
- Mugan, J. & Kuipers, B. (2011), ‘Autonomous learning of high-level states and actions in continuous environments’, *IEEE Transactions on Autonomous Mental Development* **4**(1), 70–86.
- Ng, A. Y., Harada, D. & Russell, S. (1999), Policy invariance under reward transformations: Theory and application to reward shaping, *in* I. Bratko & S. Dzeroski, eds,

- ‘Proceedings of the 16th International Conference on Machine Learning’, Morgan Kaufmann, pp. 278–287.
- Nguyen, T. T., Nguyen, N. D. & Nahavandi, S. (2020), ‘Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications’, *IEEE Transactions on Cybernetics* **50**(9), 3826–3839.
- Ølmheim, J., Landre, E., Spillum, Ø. & Hepsø, V. (2010), Decision Support and Monitoring Using Autonomous Systems, in ‘SPE Intelligent Energy Conference and Exhibition’, OnePetro, pp. SPE–128202.
- Omitaomu, O. A. & Niu, H. (2021), ‘Artificial intelligence techniques in smart grid: A survey’, *Smart Cities* **4**(2), 548–568.
- Pan, S. J. & Yang, Q. (2010), ‘A survey on transfer learning’, *IEEE Transactions on Knowledge and Data Engineering* **22**(10), 1345–1359.
- Pasula, H. M., Zettlemoyer, L. S. & Kaelbling, L. P. (2007), ‘Learning symbolic models of stochastic domains’, *Journal of Artificial Intelligence Research* **29**, 309–352.
- Phung, B. T., Winikoff, M. & Padgham, L. (2005), Learning Within the BDI Framework: An Empirical Analysis, in R. Khosla, R. J. Howlett & L. C. Jain, eds, ‘Proceedings of the 9th International Conference on Knowledge-Based Intelligent Information and Engineering Systems’, Vol. 3683 of *Lecture Notes in Computer Science*, Springer, pp. 282–288.
- Rao, A. S. (1996), AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, in W. V. de Velde & J. W. Perram, eds, ‘Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: Agents Breaking Away’, Vol. 1038 of *Lecture Notes in Computer Science*, Springer, pp. 42–55.
- Rao, A. S. & Georgeff, M. P. (1991), Modeling Rational Agents within a BDI-Architecture, in J. F. Allen, R. Fikes & E. Sandewall, eds, ‘Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning’, Morgan Kaufmann, pp. 473–484.
- Rao, A. S. & Georgeff, M. P. (1992), An Abstract Architecture for Rational Agents, in B. Nebel, C. Rich & W. R. Swartout, eds, ‘Proceedings of the 3rd International

- Conference on Principles of Knowledge Representation and Reasoning’, Morgan Kaufmann, pp. 439–449.
- Rao, A. S. & Georgeff, M. P. (1995), BDI Agents: From Theory to Practice, in V. R. Lesser & L. Gasser, eds, ‘Proceedings of the First International Conference on Multi-agent Systems’, The MIT Press, pp. 312–319.
- Remote control: remotely piloted air systems - current and future UK use* (2014), HC 772, Houses of Parliament, St Margaret St, London SW1A 0AA.
- Ricci, A., Piunti, M. & Viroli, M. (2011), ‘Environment programming in multi-agent systems: an artifact-based perspective’, *Autonomous Agents and Multi-Agent Systems* **23**(2), 158–192.
- Ricci, A., Santi, A. & Piunti, M. (2010), Action and Perception in Agent Programming Languages: From Exogenous to Endogenous Environments, in R. W. Collier, J. Dix & P. Novák, eds, ‘Proceedings of the 8th International Workshop on Programming Multi-Agent Systems’, Vol. 6599 of *Lecture Notes in Computer Science*, Springer, pp. 119–138.
- Roh, J., Mavrogiannis, C. I., Madan, R., Fox, D. & Srinivasa, S. S. (2020), Multi-modal trajectory prediction via topological invariance for navigation at uncontrolled intersections, in J. Kober, F. Ramos & C. J. Tomlin, eds, ‘Proceedings of the 4th Conference on Robot Learning’, Vol. 155 of *Proceedings of Machine Learning Research*, Proceedings of Machine Learning Research, pp. 2216–2227.
- Russell, S. & Norvig, P. (2020), *Artificial Intelligence: A Modern Approach (4th Edition)*, Pearson.
- Salfner, F., Lenk, M. & Malek, M. (2010), ‘A survey of online failure prediction methods’, *ACM Computing Surveys (CSUR)* **42**(3), 1–42.
- Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstić, S., Lourenço, J. M. et al. (2019), ‘A survey of challenges for runtime verification from advanced application domains (beyond software)’, *Formal Methods in System Design* **54**(3), 279–335.
- Sardiña, S. & Padgham, L. (2007), Goals in the context of BDI plan failure and planning, in E. H. Durfee, M. Yokoo, M. N. Huhns & O. Shehory, eds, ‘Proceedings

- of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems’, IFAAMAS, p. 7.
- Sardina, S. & Padgham, L. (2011), ‘A BDI agent programming language with failure handling, declarative goals, and planning’, *Autonomous Agents and Multi-Agent Systems* **23**(1), 18–70.
- Sierhuis, M. (2001), Modeling and Simulating Work Practice. BRAHMS: a multiagent modeling and simulation language for work system analysis and design, PhD thesis, SWI, University of Amsterdam, SIKS Dissertation Series No. 2001-10.
- Sierhuis, M., Clancey, W. J. & Van Hoof, R. J. (2007), ‘BRAHMS: a multi-agent modelling environment for simulating work processes and practices’, *International Journal of Simulation and Process Modelling* **3**(3), 134–152.
- Silver, T., Chitnis, R., Kumar, N., McClinton, W., Lozano-Pérez, T., Kaelbling, L. P. & Tenenbaum, J. B. (2023), Predicate Invention for Bilevel Planning, in B. Williams, Y. Chen & J. Neville, eds, ‘Proceedings of the 37th AAAI Conference on Artificial Intelligence, 35th Conference on Innovative Applications of Artificial Intelligence, and 13th Symposium on Educational Advances in Artificial Intelligence’, AAAI Press, pp. 12120–12129.
- Stocker, R., Sierhuis, M., Dennis, L., Dixon, C. & Fisher, M. (2011), A Formal Semantics for Brahms, in ‘Proceedings of the 12th International Workshop on Computational Logic in Multi-Agent Systems’, Vol. 6814 of *Lecture Notes in Computer Science*, Springer, pp. 259–274.
- Stringer, P., Cardoso, R. C., Dixon, C. & Dennis, L. A. (2021), Implementing Durative Actions with Failure Detection in Gwendolen, in N. Alechina, M. Baldoni & B. Logan, eds, ‘Proceedings of the 9th International Workshop on Engineering Multi-Agent Systems’, Vol. 13190 of *Lecture Notes in Computer Science*, Springer, pp. 332–351.
- Stringer, P., Cardoso, R. C., Dixon, C., Fisher, M. & Dennis, L. A. (2023a), Adaptive Cognitive Agents: Updating Action Descriptions and Plans, in V. Malvone & A. Murano, eds, ‘Proceedings of the 20th European Conference on Multi-Agent Systems’, Vol. 14282 of *Lecture Notes in Computer Science*, Springer, pp. 345–362.
- Stringer, P., Cardoso, R. C., Dixon, C., Fisher, M. & Dennis, L. A. (2023b), Updating Action Descriptions and Plans for Cognitive Agents, in N. Agmon, B. An, A. Ricci

- & W. Yeoh, eds, 'Proceedings of the 22nd International Conference on Autonomous Agents and Multiagent Systems', ACM, pp. 2370–2372.
- Stringer, P., Cardoso, R. C., Huang, X. & Dennis, L. A. (2020), Adaptable and Verifiable BDI Reasoning, *in* R. C. Cardoso, A. Ferrando, D. Briola, C. Menghi & T. Ahlbrecht, eds, 'Proceedings of the First Workshop on Agents and Robots for reliable Engineered Autonomy', Vol. 319 of *Electronic Proceedings in Theoretical Computer Science*, pp. 117–125.
- Stringer, P., Cardoso, R. C., Huang, X. & Dennis, L. A. (2021), Adaptable and Verifiable BDI Reasoning, *in* F. Dignum, A. Lomuscio, U. Endriss & A. Nowé, eds, 'Proceedings of the 20th International Conference on Autonomous Agents and Multiagent Systems', ACM, pp. 1835–1836.
- Sutton, R. S. & Barto, A. G. (2018), *Reinforcement Learning: An Introduction*, second edn, The MIT Press.
- Tidhar, G., Selvestrel, M. C. & Heinze, C. (1995), Modelling Teams and Team Tactics in Whole Air Mission Modelling, *in* G. F. Forsyth & M. Ali, eds, 'Proceedings of the 8th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems', ACM, pp. 373–381.
- Traldi, A., Bruschetti, F., Robol, M., Roveri, M. & Giorgini, P. (2022), Real-Time BDI Agents: A Model and Its Implementation, *in* L. D. Raedt, ed., 'Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence', International Joint Conferences on Artificial Intelligence Organization, pp. 511–517.
- Troquard, N. & Vieu, L. (2006), Towards a Logic of Agency and Actions with Duration, *in* G. Brewka, S. Coradeschi, A. Perini & P. Traverso, eds, 'Proceedings of the 17th European Conference on Artificial Intelligence', Vol. 141 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, pp. 775–776.
- Van Wesel, P. & Goodloe, A. E. (2017), Challenges in the verification of reinforcement learning algorithms, Technical Report NASA/TM-2017-219628, NASA Langley Research Center.
URL: <https://ntrs.nasa.gov/api/citations/20170007190/downloads/20170007190.pdf>
- Vikhorev, K., Alechina, N. & Logan, B. (2011), Agent programming with priorities and deadlines, *in* L. Sonenberg, P. Stone, K. Tumer & P. Yolum, eds, 'Proceedings of

- the 10th International Conference on Autonomous Agents and Multiagent Systems', IFAAMAS, pp. 397–404.
- Weiss, K. R., Khoshgoftaar, T. M. & Wang, D. (2016), 'A survey of transfer learning', *Journal of Big Data* **3**(1). Article 9.
- Wong, S. C., Gatt, A., Stamatescu, V. & McDonnell, M. D. (2016), Understanding data augmentation for classification: When to warp?, in 'Proceedings of the 2016 International Conference on Digital Image Computing: Techniques and Applications, DICTA', IEEE, pp. 1–6.
- Wooldridge, M. (1999), Intelligent agents, in G. Weiss, ed., 'Multiagent systems: A modern approach to distributed artificial intelligence', Vol. 1, MIT Press, chapter 1, pp. 27–73.
- Wooldridge, M. (2002), *An Introduction to Multiagent Systems*, John Wiley & Sons.
- Wooldridge, M. J. (2001), Intelligent agents: The key concepts, in V. Marík, O. Stepánková, H. Krautwurmová & M. Luck, eds, 'Multi-Agent-Systems and Applications II', Vol. 2322 of *Lecture Notes in Computer Science*, Springer, pp. 3–43.
- Wooldridge, M. & Rao, A., eds (1999), *Foundations of Rational Agency*, Applied Logic Series, Kluwer Academic Publishers.
- Yao, Y., Logan, B. & Thangarajah, J. (2016), Robust Execution of BDI Agent Programs by Exploiting Synergies Between Intentions, in D. Schuurmans & M. P. Wellman, eds, 'Proceedings of the 30th AAAI Conference on Artificial Intelligence', AAAI Press, pp. 2558–2565.
- Younes, H. L. S. & Simmons, R. G. (2004), Solving Generalized Semi-Markov Decision Processes Using Continuous Phase-Type Distributions, in D. L. McGuinness & G. Ferguson, eds, 'Proceedings of the 19th National Conference on Artificial Intelligence', AAAI'04, AAAI Press, pp. 742–748.
- Yu, Q., Moghani, M., Dharmarajan, K., Schorp, V., Panitch, W. C., Liu, J., Hari, K., Huang, H., Mittal, M., Goldberg, K. & Garg, A. (2024), 'ORBIT-Surgical: An Open-Simulation Framework for Learning Surgical Augmented Dexterity', *Clinical Orthopaedics and Related Research* **abs/2404.16027**.

Zhang, T. & Zhu, Z. (2019), Interpreting adversarially trained convolutional neural networks, in K. Chaudhuri & R. Salakhutdinov, eds, 'Proceedings of the 36th International Conference on Machine Learning', Vol. 97 of *Proceedings of Machine Learning Research*, Proceedings of Machine Learning Research, pp. 7502–7511.

Appendix A

GWENDOLEN Semantics

This chapter duplicates the operational semantics for GWENDOLEN presented in (Denis 2017).

A.1 Intentions

Intentions are crucial to understanding GWENDOLEN. BDI languages use intentions to store the *intended means* for achieving goals – this is generally represented as some form of *deed stack* (deeds include actions, belief updates, and the commitment to goals). Intention structures also maintain information about the (sub-)goal they are intended to achieve or the event that triggered them. GWENDOLEN aggregates this information: an intention becomes a stack of tuples of an event, a deed, and a unifier. This tuple is most simply viewed as a matrix structure consisting of three columns in which we record events (new perceptions, goals committed to and so forth), deeds (a plan of future actions, belief updates, goal commitments, etc.), and unifiers. These columns form an event stack, a deed stack, and a unifier stack. Rows associate a particular deed with the event that has caused the deed to be placed on the intention, and a unifier. New events are associated with an empty deed, ϵ .

Example The following shows the full structure for a single intention to clean a room. We use a standard BDI syntax: $!g$ to indicate the goal g , and $+\!g$ to indicate the commitment to achieve that goal (i.e., a new goal that g becomes true is adopted). Constants are shown starting with lower case letters, and variables with upper case letters.

event	deed	unifier
+!clean()	+!goto(Room)	Room = room1
+!clean()	+!vacuum(Room)	Room = room1

This intention has been triggered by a goal to clean — the commitment to the goal *clean()* is the trigger event for both rows in the intention. An intention is processed from top to bottom so we see here that the agent first intends to commit to the goal *goto(Room)*, where *Room* is to be unified with *room1*. Once it has committed to that goal it then commits to the goal *vacuum(Room)*. In GWENDOLEN the process of committing to a goal causes an expansion of the intention stack, pushing more deeds on it to be processed. So *goto(Room)* is expanded *before* the agent commits to vacuuming the room and the above intention becomes

event	deed	unifier
+!goto(Room)	+!planRoute(Room, Route)	Room = room1
+!goto(Room)	+!follow(Route)	Room = room1
+!goto(Room)	+!enter(Room)	Room = room1
+!clean()	+!vacuum(Room)	Room = room1

At any moment, we assume there is a *current intention* which is the one being processed at that time. The stacks that form the intention are further paired with two booleans, *suspended*, and *locked* which indicate the intention’s status. A suspended intention is, by default, *not* selected at the intention selection phase of the agent’s reasoning. Typically an intention will remain suspended until some belief condition occurs, normally that a belief is acquired via perception or from the receipt of a message. If an intention is locked, conversely, then it must be selected at the intention selection phase.

A.2 Plans, Applicable Plans and Intentions

A GWENDOLEN agent also has a *plan library* which is an ordered list of plans. Plans are matched against intentions and manipulate them. There are three main components to a plan,

1. A *trigger event* which may match the top event of an intention.
2. A *guard*: the guard is checked against the agent’s state for plan applicability.
3. A *body* which is the new deed stack that the plan proposes for execution.

We use the syntax $trigger : \{guard\} \leftarrow body$ to represent plans.

Plans only match intentions which contain unplanned goals (i.e., those associated with the “no plan yet” deed, ϵ). For instance after a commitment to $goto(Room)$ the above intention might appear as:

event	deed	unifier
+!goto(Room)	ϵ	Room = room1
+!clean()	+!goto(Room)	Room = room1
+!clean()	+!vacuum(Room)	Room = room1

which would match the plan

$$+!goto(Room) : \{upstairs(Room)\} \leftarrow +!goto(stairs); +!goto(Room)$$

This plan says that in order to achieve the goal $goto(Room)$ in the case where the room is upstairs, ($upstairs(Room)$), first the goal $goto(stairs)$ must be achieved and then the goal $goto(Room)$ achieved.

This would transform the intention to:

event	deed	unifier
+!goto(Room)	+!goto(stairs)	Room = room1
+!goto(Room)	+!goto(Room)	Room = room1
+!clean()	+!goto(Room)	Room = room1
+!clean()	+!vacuum(Room)	Room = room1

A.2.1 Applicable Plans

Applicable plans are an interim data structure that describe how a plan from an agent’s plan library changes the current intention. An applicable plan describes the new rows that will replace the top row of the intention. The new rows are generated from an event, a unifier and a stack of deeds. The new intention rows are generated by creating a row for each deed and associating the event and unifier with each of those rows (so the event and unifier are duplicated several times).

Therefore, an applicable plan is a tuple, (p_e, p_{ds}, p_θ) , of an event p_e , a deed stack p_{ds} , and a unifier p_θ . The applicable plan in the example above would be

$$(+!goto(Room), [+!goto(stairs); +!goto(Room)], \{Room = room1\}) \quad (A.1)$$

Applicable plans are used because GWENDOLEN first determines a list of applicable plans and then picks one plan to be applied. The function S_{plan} is used to select *one* applicable plan from a set. By default, this treats the set as a list and picks the first plan, but it may be overridden by specific applications.

Applicable Plan Generation Method The function **appPlans**, generates a set of applicable plans from the current intention, i , and an agent's internal state.

There are two cases. In the first case the top deed on the intention is not ε (i.e., no planning is needed). In this case the set of applicable plans is for continuing to process intention i without any changes (i.e., it represents the top row of the intention). So the set of applicable plans is the singleton:

$$\{(\text{hd}_e(i), \text{hd}_d(i), \theta^{\text{hd}(i)}) \mid \text{hd}_d(i) \neq \varepsilon\} \quad (\text{A.2})$$

where $\text{hd}_e(i)$ is the top event in i , $\text{hd}_d(i)$ is the top deed, and $\theta^{\text{hd}(i)}$ is the top unifier.

In the case where the top deed on the intention is ε , **appPlans** generates the set

$$\{(p_e, p_d, \theta^{\text{hd}(i)} \cup \theta) \mid p_e : \{p_{gu}\} \leftarrow p_d \in P \wedge \text{hd}_e(i)\theta^{\text{hd}(i)} \models p_e, \theta' \wedge ag \models p_{gu}\theta', \theta\} \quad (\text{A.3})$$

where P is the agent's library of plans. $\text{hd}_e(i) \models p_e, \theta'$ means that the plan's trigger event follows from the top event on the current intention returning a unifier, θ' . This allows for Prolog-style reasoning on plan triggers.

The notation $ag \models g, \theta$ means that the guard, g , is satisfied by agent ag given unifier θ . Again this allows Prolog-style reasoning. Plan guards may refer to the agent's belief base, goal base or outbox. For instance $\mathcal{B}b$ means some belief, b should follow by logical inference from the agent's belief base and $\mathcal{G}g$ means that some goal g should follow by logical inference from the goal base.

The notation $t\theta$ indicates the application of unifier θ to term t . So, for instance, $\text{hd}_e(i)\theta^{\text{hd}(i)}$ is the result of applying the unifier $\theta^{\text{hd}(i)}$ to the top event on the intention.

A.3 The Environment

A feature of BDI agent programming languages is that BDI programs do not, in general, stand alone but exist within a computational environment. GWENDOLEN programs expect to interact with environments programmed in JAVA which implement a specific interface. This means the semantics of some rules will depend upon the environment used. Environments offer various functions – executing agent actions, supplying sets of perceptions etc. The execution of these functions may also induce a change in the environment itself according to its own semantics.

We represent the environment as ξ . Table A.1 summarises the functions that all

Notation	Description
$\xi.\mathbf{do}(a)$	Executes an action. Returns a unifier.
$\xi.\mathit{getmessages}(ag)$	Returns a set of new messages for agent ag .
$\xi.\mathbf{Percepts}(ag)$	Returns a set of new perceptions (logical formulae) for agent, ag .
$\xi.\mathit{done}$	True if the environment is incapable of further independent action.

Table A.1: Methods implemented by GWENDOLEN Environments

environments are required to offer by the GWENDOLEN semantics. Some environments only change when one of these functions is called but others may be independently dynamic (e.g., because other agents, not programmed in GWENDOLEN are acting in them). We therefore also allow a transition relation on environments ξ : $\xi \rightarrow_{\xi} \xi'$ and represent the transitions caused by the functions in table A.1 as $\xi \xrightarrow{\mathbf{do}(a)}_{\xi} \xi'$, $\xi \xrightarrow{\mathit{getmessages}}_{\xi} \xi'$ and $\xi \xrightarrow{\mathbf{Percepts}(ag)}_{\xi} \xi'$. done does not change the environment.

A.4 Multi-Agent System Semantics, Scheduling, Reasoning Cycle

A GWENDOLEN agent is executed as part of a multi-agent system which includes an environment and a *scheduler*. The scheduler is specific to the application and so its policy for the order in which agents (and where relevant the environment) are executed varies.

We represent the operational semantics of the multi-agent system a set of transition rules. The first rules, \rightarrow_s operate on tuples of the environment, a set of agents and the scheduler and represents how the scheduler chooses the next agent for execution. The agent then transitions through stages in a *reasoning cycle* (represented with \rightarrow_a). At each stage in the reasoning cycle specific rules are selected which cause transitions on the agent (and sometimes also on the environment).

We assume the existence of the following functions: $\mathit{next_job}(s)$ returns a tuple of an agent (or the environment) and an updated version of the scheduler depending on the scheduler policy; $\mathit{sleeping}(a, s)$ returns true if the scheduler lists a as asleep; $\mathit{sleep}(a)$ returns true if the agent's status is that it has no further reasoning at the moment and $\mathit{sleep}(a, s)$ returns an updated scheduler that lists a as sleeping. \rightarrow_a^* represents the transitive closure of the semantics on an agent's reasoning cycle so $\langle \xi, a, \mathbf{A} \rangle \rightarrow_a^* \langle \xi', a', \mathbf{F} \rangle$

represents the effect of a run of the agent's reasoning cycle (from stage **A** to **F** – see below) on both the agent and the environment. $\xi \rightarrow_{\xi} \xi'$ represents an update of the environment according to its own semantics (not considered here).

The following rules represent the operation of the scheduler.

$$\frac{\neg \xi.done \quad next_job(a) = \langle \xi, s' \rangle \quad \xi \rightarrow_{\xi} \xi'}{\langle \xi, A, s \rangle \rightarrow_s \langle \xi', A, s' \rangle} \quad (A.4)$$

$$\frac{\exists a \in A. \neg sleeping(a, s) \quad next_job(s) = \langle a, s' \rangle \quad \langle \xi, a, \mathbf{A} \rangle \rightarrow_a^* \langle \xi', a', \mathbf{F} \rangle \quad \neg sleep(a')}{\langle \xi, A, s \rangle \rightarrow_s \langle \xi', A[a \setminus a'], s' \rangle} \quad (A.5)$$

$$\frac{\exists a \in A. \neg sleeping(a, A) \quad next_job(s) = \langle a, s' \rangle \quad \langle \xi, a, \mathbf{A} \rangle \rightarrow_a^* \langle \xi', a', \mathbf{F} \rangle \quad sleep(a')}{\langle \xi, A, s \rangle \rightarrow_s \langle \xi', A[a \setminus a'], sleep(a', s') \rangle} \quad (A.6)$$

It should be noted that, among other things, $next_job(s)$ can change the internal state of the scheduler, for instance altering the set of agents marked as sleeping if, for instance, new perceptions are available in the environment that might mean the agent now has something to do.

The GWENDOLEN reasoning cycle is a set consisting of size stages (**A**, **B**, **C**, **D**, **E**, and **F**). Each stage is a list of rules which are discussed in section A.5. The agent reasoning cycle transitions, \rightarrow_a , by picking the first applicable rule, r , from the list in the current reasoning stage, RS , transitioning the agent (and in some cases environment) according to the rule \rightarrow_r and then moving the reasoning cycle on according to the function $next$ (see (A.9)).

$$\frac{\exists r \in rules(RS). \langle \xi, a \rangle \rightarrow_r \langle \xi', a' \rangle}{\langle \xi, a, RS \rangle \rightarrow_a \langle \xi', a', next(a', RS) \rangle} \quad (A.7)$$

$$\frac{\neg \exists r \in rules(RS). \langle \xi, a \rangle \rightarrow_r \langle \xi', a' \rangle}{\langle \xi, a, RS \rangle \rightarrow_a \langle \xi, a, next(a, RS) \rangle} \quad (A.8)$$

A GWENDOLEN agent is a tuple $\langle ag, i, I, P, Pl, B, R, In, Out, S \rangle$ of an identifier, current intention, intention set, plan library, applicable plan set, belief base, rule base, inbox, outbox and sleep flag (more in this in section A.5). The definition of $next$ in (A.9) sometimes uses the current intention, i , and intention set, I , to compute the next reasoning stage. In these cases we represent the agent a as $\langle \dots i \dots \rangle$ or $\langle \dots i, I \dots \rangle$ as appropriate.

$$\begin{aligned}
next(\langle \dots i, I \dots \rangle, \mathbf{A}) &= \begin{cases} \mathbf{E} & i = \perp \wedge \forall i' \in I. \mathbf{is_suspended}(i') \\ \mathbf{B} & i \neq \perp \vee \exists i' \in I. \neg \mathbf{is_suspended}(i') \end{cases} \\
next(a, \mathbf{B}) &= \mathbf{C} \\
next(\langle \dots i \dots \rangle, \mathbf{C}) &= \begin{cases} \mathbf{E} & i = \perp \\ \mathbf{D} & i \neq \perp \end{cases} \\
next(a, \mathbf{D}) &= \mathbf{E} \\
next(a, \mathbf{E}) &= \mathbf{F} \\
next(a, \mathbf{F}) &= \mathbf{A}
\end{aligned} \tag{A.9}$$

where $\mathbf{is_suspended}(i)$ is true if the intention, i , is suspended.

A.5 Stage Rules: The Agent Reasoning Cycle

We represent an agent as a tuple $\langle ag, i, I, P, Pl, B, R, In, Out, S \rangle$ where:

- ag is a unique identifier for the agent (it's name);
- i is the current intention (see section A.1); Note that there can be no current intention which we will indicate with the expression $i = null$.
- I is a stack of intentions $\{i, i', \dots\}$;
- P is an ordered list of the agent's plans (see section A.2);
- Pl is a set of currently applicable plans (see section A.2);
- B is a set of the agent's beliefs which are pairs of ground first-order formulae and a string indicating the *source* of the belief. In GWENDOLEN all beliefs are automatically assigned the source `self` unless they are acquired by perception in which case they are assigned the source `percept`;
- R is a set of Prolog-style rules used in reasoning;
- In is the agent inbox. Elements of inbox have the form $\downarrow^{id, ilf} m$ where id is the identifier of the sender, ilf is the illocutionary force of the message and can be *tell*, *perform*, or *achieve*, and m is the message content, a ground first-order formula.

a	An action.
b	A belief.
$+b$	A belief addition.
$-b$	A belief removal.
$b\{source\}$	A belief, from source $source$.
$!_{\tau}g$	A goal of type τ .
$+!_{\tau}g$	A goal addition.
$-!_{\tau}g$	A goal drop.
$\times!_{\tau}g$	A goal which can't be planned.
lock	An lock.
unlock	An unlock.
$\uparrow^{ag,ilf} m$	A message m sent to ag .
$\downarrow^{ag,ilf} m$	A message m received from ag .
\top	An structure who's logical content is trivially true.
ϵ	A special marker indicating that some event has no plan yet.

Table A.2: Notations for deed type checks

- Out is the agent outbox. Messages in this set have the format $\uparrow^{id,ilf} m$ where id is the identifier of the recipient, ilf is the illocutionary force and m is the message content, a ground first-order formula.
- S is a boolean indicating whether the agent should be *slept* by the scheduler or not.

In its initial state the current intention is *null*, the intention set consists of one intention for each of the initial goals provided by the programmer. These intentions are of the form $(start, +!_{\tau_g}g, \emptyset)$ where $start$ is a special event used for intentions with no specific trigger. Its plan library is a set of plans provided by a programmer. The applicable plans are empty. The belief base and rule base are as defined by the programmer. The inbox and outbox are empty and the sleep flag is false.

Many of the transition rules make a check on a deed to see what type it is (e.g. the addition of a belief, the deletion of a goal). We represent these checks implicitly using the notation shown in table A.2. Many of the rules also check intentions for various properties and manipulate them. Table A.3 summarises various operations on intentions that are used in the rules.

It is generally unwieldy to present the full agent tuple in the description of a transition rule. As a result we restrict ourselves to presenting only those parts of the intention that are changed by the rule as we did in (A.9).

We now discuss each stage of the reasoning cycle in turn.

Notation	Description
U_θ	Compose a unifier with the top unifier on the intention.
$\text{empty}(i)$	The deed stack of the intention is empty.
$\text{events}(i)$	The stack of events associated with intention i .
$\text{hd}_e(i)$	The top event on the intention.
$\text{hd}_d(i)$	The top deed on the intention.
$\theta^{\text{hd}(i)}$	The top unifier on the intention.
$@$	Add a new event, deed stack, and unifier to the top of the intention.
$;_p$	Add a new event, deed, and unifier as the top row of the intention.
$\text{tl}_i(i)$	Drop the top row of the intention.
$\text{drop}_E(e, i)$	Drop all rows in the intention above and including the first appearance of e as a trigger.
$\text{lock}(i)$	Mark the intention as locked.
$\text{locked}(i)$	The intention is locked.
$\text{suspend}(i)$	Mark the intention as suspended.
is_suspended (i)	The intention is suspended.
$\text{unlock}(i)$	Mark the intention as unlocked.

Table A.3: Operations on Intentions

A.5.1 Stage A

Stage A of the GWENDOLEN reasoning cycle consists of a list of three rules which are focused around managing intention selection:

[select_intention, sleep, drop_intention]

Select Intention (select_intention)

$$\frac{\neg \text{empty}(i) \quad \neg \text{locked}(i) \quad \exists i'' \in I \cup \{i\}. \neg \text{is_suspended}(i'') \quad \mathcal{S}_{\text{int}}(I \cup \{i\}) = (i', I') \quad \text{hd}_e(i') \neq \neg !_{\tau_g} g \vee \text{hd}_d(i') \neq \varepsilon}{\langle \xi, \langle \dots i, I \dots \rangle \rangle \rightarrow_{\text{select_intention}} \langle \xi, \langle \dots i', I' \dots \rangle \rangle} \quad (\text{A.10})$$

$$\frac{\neg \text{empty}(i) \quad \text{locked}(i) \quad \text{hd}_e(i) \neq \neg !_{\tau_g} g \vee \text{hd}_d(i) \neq \varepsilon \quad \exists i'' \in I \cup \{i\}. \neg \text{is_suspended}(i'')}{\langle \xi, \langle \dots i, I \dots \rangle \rangle \rightarrow_{\text{select_intention}} \langle \xi, \langle \dots i, I \dots \rangle \rangle} \quad (\text{A.11})$$

where $\text{empty}(i)$ is true if intention i has an empty deed stack, $\text{locked}(i)$ is true if intention i is locked, and **is_suspended**(i) is true if intention i is suspended. Table A.3 summarises all the operations on intentions.

This rule has two cases, one for when the current intention isn't locked and one

for when it is. When the intention isn't locked the system uses the application specific selection function \mathcal{S}_{int} to pick a new current intention (by default this treats the intention set I as a LIFO queue and selects the first unsuspended intention from the queue). The rule is inapplicable if the current intention is empty or the selected intention's trigger is a drop goal event.

Sleep (sleep)

$$\frac{(i = \text{null} \vee \text{empty}(i) \vee \text{is.suspended}(i)) \quad \forall i' \in I. \text{is.suspended}(i')}{\langle \xi, \langle \dots i, I, \dots S \rangle \rangle \rightarrow_{\text{sleep}} \langle \xi, \langle \dots, i, I, \dots \top \rangle \rangle} \quad (\text{A.12})$$

Table A.3 summarises all the operations on intentions such as empty etc.,

This rule sets an agent's sleep flag if all its intentions are empty or suspended. The agent will then be marked as sleeping by the scheduler once the reasoning cycle is concluded.

Drop Intention (drop_intention)

$$\frac{i \neq \text{null} \quad \text{empty}(i) \quad I \neq \emptyset \quad \mathcal{S}_{\text{int}}I = (i', I')}{\langle \xi, \langle \dots i, I \dots \rangle \rangle \rightarrow_{\text{drop.intention}} \langle \xi, \langle \dots i', I' \dots \rangle \rangle} \quad (\text{A.13})$$

Table A.3 summarises all the operations on intentions such as empty etc.,

This rule drops the intention i if it is empty and selects a new current intention from the intention set. The additional $i \neq \text{null}$ is necessary since a few rules can leave the agent state with no current intention.

A.5.2 Stage B

Stage B of the GWENDOLEN reasoning cycle consists of a list of two rules based on generating a set of applicable plans: [generate_plan, no_plan]

Generate Plan (generate_plan)

$$\frac{\text{appPlans}(i) \neq \emptyset}{\langle \xi, \langle \dots i, Pl \dots \rangle \rangle \rightarrow_{\text{generate.plan}} \langle \xi, \langle \dots i, \text{appPlans}(i) \dots \rangle \rangle} \quad (\text{A.14})$$

appPlans is as described in section A.2.

No Applicable Plans (no_plan)

$$\frac{\text{appPlans}(i) = \emptyset \quad \text{hd}_e(i) = +!_{\tau}g}{\langle \xi, \langle \dots i, Pl \dots \rangle \rangle \rightarrow_{\text{no.plan}} \langle \xi, \langle \dots i, [(x!_{\tau}g, [\varepsilon], \theta^{\text{hd}(i)})] \dots \rangle \rangle} \quad (\text{A.15})$$

$$\frac{\mathbf{appPlans}(i) = \emptyset \quad \neg \text{hd}_e(i) = +!_{\tau}g}{\langle \xi, \langle \dots i, Pl \dots \rangle \rangle \rightarrow_{\text{no_plan}} \langle \xi, \langle \dots i, [(\text{hd}_e(i), [], \emptyset)] \dots \rangle \rangle} \quad (\text{A.16})$$

$\mathbf{appPlans}(i)$ is empty if there is no plan applicable to the current intention. This rule differentiates between whether the intention trigger is a goal commitment (in which case the rule creates an applicable plans consisting of an unplanned “problem goal” event ($x!_{\tau}g$) (which might, for instance, be responded to by suspending the intention until the agent’s beliefs have changed and some plan does become applicable). Otherwise it generates an applicable plan with an empty deed stack. This will have the effect of removing the top row of the intention and replacing it by nothing – i.e., it ignores the event that had no applicable plan for handling it. The reasoning behind this is that such events (notifications of beliefs acquired or dropped generally only require a planning response in special cases and can normally be ignored).

A.5.3 Stage C

Stage C of the GWENDOLEN reasoning cycle consists of a list of a single rule for modifying the current intention according to the applicable plan: [apply_plan]

Apply Plan (apply_plan)

$$\frac{(e, Ds, \theta) = \mathcal{S}_{\text{plan}}(Pl)}{\langle \xi, \langle \dots i \dots Pl \dots \rangle \rangle \rightarrow \langle \xi, \langle \dots (e, Ds, \theta) @ \tau l_i(i) \dots \emptyset \dots \rangle \rangle} \quad (\text{A.17})$$

where $\tau l_i(i)$ represents intention, i , with its top row removed and $(e, Ds, \theta) @ \tau l_i(i)$ represents the applicable plan (e, Ds, θ) expanded and added to the top of the intention, i in place of its top row as described in section A.2. Table A.3 summarises all the operations on intentions such as empty etc.,

This rule selects a plan from the agent’s applicable plans as determined by the application specific $\mathcal{S}_{\text{plan}}$ (by default this is the first applicable plan found in the plan library and, where a unifier is required, this is the first returned by checking against the agents internal state (this lists beliefs and goals, etc., in alphabetical order)). The plan is represented as a tuple of the trigger event, the plan’s deed stack and unifier. The top row of the current intention is dropped and the applicable plan is “glued” in its place.

A.5.4 Stage D

Stage D of the GWENDOLEN reasoning cycle consists of a list of rules for processing the top deed on the current intention:

```
[empty, add_achieve_goal, add_perform_goal, drop_goal, add_belief,
  drop_belief, lock_unlock, wait_for, problem_goal, action, send,
  null]
```

Handle Empty Deed Stack (*empty*)

$$\frac{\text{empty}(i)}{\langle \xi, \langle \dots i \dots \rangle \rightarrow_{\text{empty}} \langle \xi, \langle \dots i \dots \rangle \rangle} \quad (\text{A.18})$$

Table A.3 summarises all the operations on intentions such as *empty* etc.,

This rule does nothing if the current intention's deed stack is empty (which can occur if there is no plan for handling the intention's trigger event). This leaves the intention unchanged and it will be removed during the select intention phase (Stage A).

Handle Add Achieve Goal (*add_achieve_goal*)

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = +!_ag \quad B, R \models g, \theta_g}{\langle \xi, \langle \dots i \dots B, R \dots \rangle \rangle} \quad (\text{A.19})$$

$$\begin{array}{c} \rightarrow_{\text{add_achieve_goal}} \\ \langle \xi, \langle \dots \tau l_i(i) \cup_{\theta} \theta_g \dots B, R \dots \rangle \rangle \end{array}$$

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = +!_ag \quad B, R \not\models g}{\langle \xi, \langle \dots i \dots B, R \dots \rangle \rangle} \quad (\text{A.20})$$

$$\begin{array}{c} \rightarrow_{\text{add_achieve_goal}} \\ \langle \xi, \langle \dots (+!_ag, \epsilon, \theta^{\text{hd}(i)});_p i \dots B, R \dots \rangle \rangle \end{array}$$

where $B, R \models g, \theta_g$ means that the formula g (which is the goal with the top unifier from the intention applied to it) follows using Prolog-style reasoning from the agent's belief base when the additional unifier θ_g is applied. $\tau l_i(i) \cup_{\theta} \theta_g$ indicates the union of unifier θ_g with the unifier on the top of the intention $\tau l_i(i)$. $(e, d, \theta);_p i$ represents the addition of a row (e, d, θ) to the top of an intention i . Table A.3 summarises all the operations on intentions such as *empty* etc.,

GWENDOLEN recognises two types of goal, *achieve* goals and *perform* goals (goal types a and p respectively). This rule handles the commitment to an achieve goal. An achieve goal is one that triggers a plan if it not already believed but does no more

than set a unifier if it is. If it is to trigger a plan, then we register the commitment to planning the goal as an event on the top of the intention stack. In this case the top row of the intention is not dropped so the deed intending a commitment to the goal remains. This means that if, after execution of the plan, the goal is not achieved then it will be replanned.

Handle Add Perform Goal (add_perform_goal)

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = +!_p g}{\langle \xi, \langle \dots i \dots \rangle \rangle} \quad (\text{A.21})$$

$$\xrightarrow{\text{add_perform_goal}} \langle \xi, \langle \dots (+!_p g, \varepsilon, \theta^{\text{hd}(i)});_p (\text{hd}_e(i), \text{null}, \theta^{\text{hd}(i)});_p \text{tl}_i(i) \dots \rangle \rangle$$

where $(e, d, \theta);_p i$ represents the addition of a row (e, d, θ) to the top of an intention i . Table A.3 summarises all the operations on intentions such as $\theta^{\text{hd}(i)}$ etc.,.

Perform goals always trigger planning but are not replanned if they fail to achieve some state of the world. This being the case we replace the top deed (the request to commit to the goal) on the intention with *null* so that this is automatically processed once the system reaches that row of the intention. We then add a new top row with the trigger event of the new goal and a no plan yet deed.

Handle Drop Goal (drop_goal)

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = -!_{\tau_g} g \quad \exists e \in \text{events}(i). \mathbf{unify}(e, +!_{\tau_g} g)}{\langle \xi, \langle \dots i \dots \rangle \rangle \xrightarrow{\text{drop_goal}} \langle \xi, \langle \dots \text{tl}_i(\text{drop}_E(e, i)) \dots \rangle \rangle} \quad (\text{A.22})$$

where $\mathbf{unify}(e_1, e_2)$ indicates that two events can be unified. $\text{drop}_E(e, i)$ is a function that recurses through an intention dropping every row after the first occurrence of e – i.e. it prunes the intention back to the point where the event first occurred. Table A.3 summarises all the operations on intentions such as *events* etc.,.

This rule searches the current intention for the most earliest add goal event that unifies with the goal to be dropped and then deletes all rows on the intention above that. It then deletes the new top row which will be the one that contains the instruction to commit to the goal (if an achieve goal) or *null* (if a perform goal).

Handle Add Belief (add_belief)

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = +b}{\langle \xi, \langle \dots i, I, B \dots \rangle \rangle \rightarrow_{\text{add_belief}} \langle \xi, \langle \dots \text{tl}_i(i) \cup_{\theta} \theta^{\text{hd}(i)}, \mathbf{unsuspend}(I, b) \cup_{\text{new}}(+b, \varepsilon, \emptyset), B \cup \{b\}, \dots \rangle \rangle} \quad (\text{A.23})$$

where $\mathbf{unsuspend}(I, b)$ unsuspends all suspended intentions in I that are waiting for b to become true. $\text{new}(e, d, \theta)$ creates a new intention from an event, deed and unifier. Table A.3 summarises all the operations on intentions such as $\theta^{\text{hd}(i)}$ etc.,

This rule adds new belief to the belief base and a new intention noting the appearance of the new belief. At the same time it unsuspends all intentions which are waiting for b to be achieved as part of their suspend condition.

Handle Drop Belief (drop_belief)

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = -b \quad B^1 = \{b' | b' \in B \wedge \mathbf{unify}(b', b)\}}{\langle \xi, \langle \dots i, I, B \dots \rangle \rangle \rightarrow_{\text{drop_belief}} \langle \xi, \langle \dots \text{tl}_i(i) \cup_{\theta} \theta^{\text{hd}(i)}, I \cup_{\text{new}}(-b, \varepsilon, \emptyset), B \setminus B^1, \dots \rangle \rangle} \quad (\text{A.24})$$

where $\mathbf{unify}(b', b)$ means that b' and b unify with each other. $\text{new}(e, d, \theta)$ creates a new intention from an event, deed and unifier. Table A.3 summarises all the operations on intentions such as $\theta^{\text{hd}(i)}$ etc.,

This rule drops a belief from the belief base. At the same time it generates a new intention containing the event that the belief has been dropped. Appropriate handling of this event can allow the agent to form plans in reaction to it.

Handle Lock and Unlock (lock_unlock)

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = \text{lock}}{\langle \xi, \langle \dots i \dots \rangle \rangle \rightarrow_{\text{lock_unlock}} \langle \xi, \langle \dots, \text{lock}(\text{tl}_i(i) \cup_{\theta} \theta^{\text{hd}(i)}) \dots \rangle \rangle} \quad (\text{A.25})$$

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = \text{unlock}}{\langle \xi, \langle \dots i \dots \rangle \rangle \rightarrow_{\text{lock_unlock}} \langle \xi, \langle \dots \text{unlock}(\text{tl}_i(i) \cup_{\theta} \theta^{\text{hd}(i)}) \dots \rangle \rangle} \quad (\text{A.26})$$

Table A.3 summarises all the operations on intentions such as $\theta^{\text{hd}(i)}$ etc.,

This allows an intention to be “locked” as the current intention, for instance to allow a complete sequence of belief changes be processed before any other reasoning takes place. Once finished the intention has to be unlocked.

Handle Wait For

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = * \dots b \quad B, R \models b, \theta_b}{\langle \xi, \langle \dots i \dots B, R \dots \rangle \rangle \rightarrow_{\text{wait_for}} \langle \xi, \langle \dots \text{tl}_i(i) \cup_{\theta} (\theta^{\text{hd}(i)} \cup \theta_b) \dots B, R \dots \rangle \rangle} \quad (\text{A.27})$$

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = * \dots b \quad B, R \not\models b, \theta_b \quad \exists i' \in I. \neg \text{is_suspended}(i')}{\langle \xi, \langle \dots i, I, B, R \dots \rangle \rangle \rightarrow_{\text{wait_for}} \langle \xi', \langle \dots \text{suspend}(i\theta^{\text{hd}(i)}), I, B, R \dots \rangle \rangle} \quad (\text{A.28})$$

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = * \dots b \quad B, R \not\models b, \theta_b \quad \forall i' \in I. \text{is_suspended}(i')}{\langle \xi, \langle \dots i, I, B \dots In \dots S \rangle \rangle \rightarrow_{\text{wait_for}} \langle \xi', \langle \dots \text{null}, I \cup \{ \text{suspend}(i\theta^{\text{hd}(i)}) \}, B, R \dots \top \rangle \rangle} \quad (\text{A.29})$$

where $B, R \models b, \theta_b$ means that the formula b follows using Prolog-style reasoning from the agent's belief base and Prolog rule-base when the additional unifier θ_b is applied. $\text{tl}_i(i) \cup_{\theta} (\theta^{\text{hd}(i)} \cup \theta_b)$ indicates the union of unifier $(\theta^{\text{hd}(i)} \cup \theta_b)$ with the unifier on the top of the intention $\text{tl}_i(i)$. $\text{suspend}(i)$ suspends an intention. Table A.3 summarises all the operations on intentions such as $\theta^{\text{hd}(i)}$ etc.,

If an intention is waiting for some belief, b , to become true then if that belief is now true the intention continues processing. Otherwise the intention is suspended. If all intentions are suspended then the agent is told to sleep at the next opportunity.

Ignore Unplanned Problem Goal (`problem_goal`)

$$\frac{\text{hd}_e(i) = x! \tau_g g \quad \text{hd}_d(i) = \varepsilon}{\langle \xi, \langle \dots i \dots \rangle \rangle \rightarrow_{\text{problem_goal}} \langle \xi, \langle \dots i \dots \rangle \rangle} \quad (\text{A.30})$$

Table A.3 summarises all the operations on intentions such as $\theta^{\text{hd}(i)}$ etc.,

This rule ignores an unplanned problem goal. It simply does nothing but allows the reasoning cycle of the agent to continue processing on the assumption that planning of the goal may become possible later.

Handle General Action (`action`)

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = a \quad \xi \xrightarrow{\text{do}(a)} \xi' \quad \xi. \text{do}(a) = \theta_a \quad a \neq \uparrow^{ag, if} m}{\langle \xi, \langle \dots i \dots \rangle \rangle \rightarrow_{\text{action}} \langle \xi', \langle \dots \text{tl}_i(i) \cup_{\theta} (\theta^{\text{hd}(i)} \cup \theta_a) \dots \rangle \rangle} \quad (\text{A.31})$$

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = a \quad \xi \xrightarrow{\text{do}(a)} \xi' \quad \neg\xi.\text{do}(a) \quad \text{hd}_e(i) = +!\tau_g g \quad a \neq \uparrow^{ag,ilf} m}{\langle \xi, \langle \dots i \dots \rangle \rangle \rightarrow_{\text{action}} \langle \xi', \langle \dots (x!\tau_g g, \varepsilon, \theta^{\text{hd}(i)} \cup \theta_a); pi \dots \rangle \rangle} \quad (\text{A.32})$$

$$\frac{\text{hd}_d(i)\theta^{\text{hd}(i)} = a \quad \xi \xrightarrow{\text{do}(a)} \xi' \quad \neg\xi.\text{do}(a) \quad \text{hd}_e(i) \neq +!\tau_g g \quad a \neq \uparrow^{ag,ilf} m}{\langle \xi, \langle \dots i \dots \rangle \rangle \rightarrow_{\text{action}} \langle \xi', \langle \dots \text{tl}_i(i) \cup \theta^{\text{hd}(i)} \dots \rangle \rangle} \quad (\text{A.33})$$

where $\xi.\text{do}(a\theta^{\text{hd}(i)}) = \theta_a$ means that the environment successfully executes a returning unifier θ_a . $\text{tl}_i(i) \cup \theta_g$ indicates the union of unifier θ_g with the unifier on the top of the intention $\text{tl}_i(i)$. $(e, d, \theta); pi$ represents the addition of a row (e, d, θ) to the top of an intention i . Table A.3 summarises all the operations on intentions such as $\theta^{\text{hd}(i)}$ etc.,

In this rule, the agent attempts the action (unless it is a send action $-\uparrow^{ag,ilf} m$). If the action succeeds it returns a unifier and the environment updates. Otherwise, if the trigger event at the top of the intention is a goal then this generates a problem goal event for handling by some plan.

Handle Send Action (send)

$$\frac{\xi \xrightarrow{\text{do}(\uparrow^{ag',ilf} m_{ag})} \xi' \quad \text{hd}_d(i)\theta^{\text{hd}(i)} = \uparrow^{ag',ilf} m \quad \xi.\text{do}(\uparrow^{ag',ilf} m_{ag}) = \theta_a}{\langle \xi, \langle ag, i, I \dots Out \dots \rangle \rangle \rightarrow_{\text{send}} \langle \xi', \langle ag, \text{tl}_i(i) \cup \theta (\theta^{\text{hd}(i)} \cup \theta_a), I \cup \{\text{new}(\uparrow^{ag',ilf} m, \varepsilon, \emptyset)\}, \dots Out \cup \{\uparrow^{ag',ilf} m\} \dots \rangle \rangle} \quad (\text{A.34})$$

$$\frac{\xi \xrightarrow{\text{do}(\uparrow^{ag',ilf} m_{ag})} \xi' \quad \text{hd}_d(i)\theta^{\text{hd}(i)} = \uparrow^{ag',ilf} m \quad \neg\xi.\text{do}(\uparrow^{ag',ilf} m_{ag}) \quad \text{hd}_e(i) = +!\tau_g g}{\langle \xi, \langle ag, i, I \dots Out \dots \rangle \rangle \rightarrow_{\text{send}} \langle \xi', \langle ag, (x!\tau_g g, \varepsilon, \theta^{\text{hd}(i)} \cup \theta^{\text{hd}(i)}); pi, I \dots Out \dots \rangle \rangle} \quad (\text{A.35})$$

$$\frac{\xi \xrightarrow{\text{do}(\uparrow^{ag',ilf} m_{ag})} \xi' \quad \text{hd}_d(i)\theta^{\text{hd}(i)} = \uparrow^{ag',ilf} m \quad \neg\xi.\text{do}(\uparrow^{ag',ilf} m_{ag}) \quad \neg\text{hd}_e(i) = +!\tau_g g}{\langle \xi, \langle ag, i, I \dots Out \dots \rangle \rangle \rightarrow_{\text{send}} \langle \xi', \langle ag, \text{tl}_i(i) \cup \theta^{\text{hd}(i)}, I \dots Out \dots \rangle \rangle} \quad (\text{A.36})$$

where $\xi.\mathbf{do}(\uparrow^{ag', ilf} m_{ag})$ is the environment executing the sending of a message, m , from ag to ag' with illocutionary force ilf . $\mathbf{new}(e, d, \theta)$ creates a new intention from an event, deed and unifier (in this case the event is the sending of a message to ag'). $\tau_{1_i}(i) \cup_{\theta} \theta_g$ indicates the union of unifier θ_g with the unifier on the top of the intention $\tau_{1_i}(i)$. Table A.3 summarises all the operations on intentions such as $\theta^{\mathbf{hd}(i)}$ etc.,

This rule behaves much as the rule for handling general actions with the exception that when a send action succeeds a new intention is generated registering the event that a message was sent and the message itself is added to the agent's outbox.

Handle Null (null)

$$\frac{\mathbf{hd}_d(i)\theta^{\mathbf{hd}(i)} = \mathbf{null}}{\langle \xi, \langle \dots i \dots \rangle \rangle \rightarrow_{\mathbf{null}} \langle \xi, \langle \dots \tau_{1_i}(i) \cup_{\theta} \theta^{\mathbf{hd}(i)} \dots \rangle \rangle} \quad (\text{A.37})$$

where $\tau_{1_i}(i) \cup_{\theta} \theta_g$ indicates the union of unifier θ_g with the unifier on the top of the intention $\tau_{1_i}(i)$. Table A.3 summarises all the operations on intentions such as $\theta^{\mathbf{hd}(i)}$ etc.,

The null action is used as a place holder to note, when a perform goal has been committed to, a record of the relevant trigger event in an intention stack. This rule simply ignores the null action when it is encountered and deletes that row from the intention.

A.5.5 Stage E

Stage E of the GWENDOLEN reasoning cycle consists of a list of a single rule for handling perception: [perceive]

Perceive

$$\frac{\begin{array}{c} \xi \xrightarrow{\mathbf{Percepts}(ag)}_{\xi} \xi_1 \quad \xi_1 \xrightarrow{\mathbf{getmessages}(ag)}_{\xi} \xi' \\ P = \xi.\mathbf{Percepts}(ag) \\ OP = \{b \mid b \in B \setminus P \wedge \mathbf{source_of}(b) = \mathbf{percept}\} \\ P \setminus B \cup OP \cup \xi.\mathbf{getmessages}(ag) \neq \emptyset \end{array}}{\begin{array}{c} \langle \xi, \langle \dots I, B \dots In \dots S \rangle \rangle \rightarrow_{\mathbf{perceive}} \\ \langle \xi', \langle \dots \\ I \cup \{\mathbf{new}(\mathbf{start}, +b, \emptyset) \mid b \in P \setminus B\} \cup \{\mathbf{new}(\mathbf{start}, -b, \emptyset) \mid b \in OP\}, \\ B \dots In \cup \xi.\mathbf{getmessages}(ag) \dots \top \rangle \end{array}} \quad (\text{A.38})$$

$$\begin{array}{c}
\xi \xrightarrow{\text{Percepts}(ag)}_{\xi} \xi_1 \quad \xi_1 \xrightarrow{\text{getmessages}(ag)}_{\xi} \xi' \\
P = \xi.\text{Percepts}(ag) \\
OP = \{b \mid b \in B \setminus P \wedge \text{source_of}(b) = \text{percept}\} \\
P \setminus B \cup OP \cup \xi.\text{getmessages}(ag) = \emptyset \\
\hline
\langle \xi, \langle \dots I, B \dots In \dots \rangle \rangle \rightarrow_{\text{perceive}} \langle \xi', \langle \dots I, B \dots In \dots \rangle \rangle
\end{array} \tag{A.39}$$

where $\xi.\text{Percepts}(ag)$ returns a set of new beliefs to the agent which are all annotated as coming from the source percept. $\text{source_of}(b)$ returns the source of a belief b . $\xi.\text{getmessages}(ag)$ returns a set of new messages to the agent. $\text{new}(e, d, \theta)$ creates a new intention from an event, deed and unifier. In this case the event is a special distinguished event `start` which is used to indicate an intention with no trigger.

This rule adds all messages to the inbox. It also creates new intentions, each triggered by the event of acquiring or losing a percept. A key part of the working of the rule depends on AIL's annotation of all beliefs in the belief base with a source and its use of a special annotation for beliefs whose source is perception. If some change is brought about either to the agent's inbox or to its intentions then the agent's sleep flag is set to true (i.e., the agent will not sleep at the end of this reasoning cycle).

Note that in the EASS variant of GWENDOLEN the perception rule also updates the belief base directly, unlike this rule which creates intentions to update the belief base and leaves these to later reasoning cycles for execution.

A.5.6 Stage F

Stage F of the GWENDOLEN reasoning cycle consists of a list of a single rule for processing messages in the inbox: `[messages]`

Handle Messages (`messages`)

$$\begin{array}{c}
\hline
\langle \xi, \langle \dots I \dots In \dots \rangle \rangle \rightarrow_{\text{messages}} \\
\langle \xi, \langle \dots I \cup \{\text{new}(+received(ag, ilf, m), \varepsilon, \emptyset) \mid \downarrow^{ag, ilf} m \in In\} \dots [] \dots \rangle \rangle
\end{array} \tag{A.40}$$

where $\text{new}(e, d, \theta)$ creates a new intention from an event, deed and unifier. In this case the event is a belief that the agent has received message m from agent, ag with illocutionary force, ilf . It is up to the programmer to decide how messages should be handled, there is no default mechanism for handling messages of any particular illocutionary force (unlike many BDI languages which give a specific semantics to

such constructs).

This rule does not poll the environment for messages. It takes all messages currently in an agent's inbox and converts them to intentions (triggered by a perception that the message has been received), emptying the inbox in the process. It should be noted that it does not store the message anywhere once the inbox is emptied. It assumes that some plan will act appropriately to the message received event. If this does not happen then the message content may be lost.