Imperial College
London

Imperial College London
Department of Computing

MEng Individual Project

Planning for CTL*-Specified Temporally Extended Goals via Model Checking

Michael Ewedokun Akintunde

Supervisor:
Prof. Alessio Lomuscio

Second Marker:
Dr. Mark Wheelhouse

June 22, 2017

Abstract

Planning is an active research field based on intelligent decision making for autonomous systems. To *plan* is to have an agent reach a goal state given an initial situation. However, we may want to specify temporally extended goals on the *execution structure* of a plan.

In this project, we specify temporally extended goals with a much greater degree of expressivity than what is currently possible. We do this by specifying goals using the full branching-time temporal logic CTL* (Full Computation Tree Logic).

We initially specify a formal correspondence between a representative model of a planning problem and an *interpreted system.* We then document the design and construction of a compiler to translate between the two models. Along with this, we investigate the challenges involved in planning for multiple agents and planning under partial observability.

Acknowledgements

I would firstly like to thank my supervisor, Prof. Alessio Lomuscio, for his support throughout the duration of the project.

I would also like to thank my second marker, Dr. Mark Wheelhouse, for interviewing me for a place at Imperial, as well as being a fantastic personal programming tutor in my first year.

I wish to also make a special thanks to:

- Dr. Fariba Sadri, my personal tutor, for her support and guidance throughout my four years at Imperial.
- My family, for their unconditional love.
- My friends, who have made the past four years slightly more palatable.

Contents

1 Introduction **11**

1.1 Primary Objectives 12

1.2 Challenges 12

1.3 Contributions 13

2 Background **15**

2.1 Model Checking 15

2.2 Symbolic Model Checking 16

2.2.1 Binary Decision Trees 17

2.2.2 Binary Decision Diagrams 18

2.2.3 Reduced Ordered Binary Decision Diagrams 18

2.2.4 Validity and Satisfiability 19

2.3 Temporal Logics 19

2.4 Linear-time Temporal Logic 20

2.4.1 Syntax 20

2.4.2 Semantics 20

2.5 The Branching-Time Logic CTL 21

2.5.1 Syntax 21

2.5.2 Semantics 22

2.6 The Full Branching-Time Logic CTL* 23

2.6.1 Syntax 23

2.6.2 Semantics 24

2.7 Multi-Agent Systems 24

2.8 Interpreted Systems 25

2.9 MCMAS 26
2.9.1 Architecture 26
2.10 Interpreted Systems Programming Language (ISPL) 27
2.11 Introduction to Planning 30
2.12 Planning via Model Checking 32
2.13 Buchi Automata 33
2.13.1 Transition Systems 34
2.13.2 Automata on Infinite Words 34
2.14 Planning via Automata 35
2.15 Relation Between Planning and Reinforcement Learning 36
2.16 Existing Planning Problem Description Languages 36
2.16.1 Non-deterministic Agent Domain Language 36
2.16.2 PDDL 37
2.16.3 The Multi-Agent Extension of PDDL 3.1 39
2.16.4 NuPDDL 39
2.16.5 EaGLe 40
2.17 Existing Planners 41
2.17.1 The Model Based Planner 41
2.17.2 Universal Multi-agent OBDD-based Planner 41
2.17.3 SATplan 41
2.17.4 Existing MCMAS Extensions 42
2.18 Other Existing Logics 42
2.18.1 ATL* 42
2.18.2 Strategy Logic 42
2.18.3 α-CTL 43

3 From Planning Problems to Interpreted Systems 45
3.1 The Dinner Domain 45
3.1.1 Domain 46
3.1.2 Problem 46
3.1.3 Planning Domain Specification 46
3.1.4 Planning Problem Specification 47

3.1.5 Translation to Interpreted Systems 48
3.1.6 Kripke Model Translation of Dinner Domain 51
3.2 The Gripper Domain – Towards a More General Translation 51
3.2.1 Problem 53
3.2.2 Planning Problem Specification 54
3.2.3 Translation to Interpreted System 56
3.2.4 Translation of Local States 56
3.2.5 Actions 57
3.2.6 Protocol function 58
3.2.7 Evolution function 59
3.2.8 Valuation function 60
3.2.9 Conversion from planning problem to corresponding Kripke model 60
3.2.10 Conversion from IS to Kripke model 61
3.3 Correctness of Translation 61
3.3.1 Corollary 64
3.4 Summary 64

4 Implementation 65

4.1 Architecture 66
4.2 Parser 66
4.3 Code Generator 67
4.4 Plan-printing in MCMAS 70
4.5 Summary 71

5 Planning for Temporally-Extended Goals 73

5.1 Using CTL for Goal Specification 73
5.1.1 Using the Gripper Domain 73
5.2 Using CTL* for Goal Specification 74
5.2.1 Using the Dinner domain 74
5.2.2 Using the Gripper domain 74
5.3 Types 75
5.4 Non-Determinism 76
5.4.1 Uncertainty in the Initial State 76

5.4.2 Conditional Effects 77
5.5 Fluents 77
5.5.1 Durative Actions 77
5.6 Planning Under Incomplete Information 77
5.7 Summary 78

6 Evaluation 81

6.1 Experimental Evaluation 81
6.1.1 Classical Domains 81
6.1.2 Temporally-Extended Goals 82
6.1.3 Infinite Plans 83
6.1.4 Large State Spaces 84
6.1.5 Blocks World Domain 85
6.1.6 Multi-Agent Planning 85
6.1.7 Uniform Strategies 88
6.1.8 Performance Test Setup 89
6.2 Testing 89
6.2.1 The Automatic Validation Tool VAL 89
6.2.2 Differential Testing 90
6.2.3 Unit Testing 91
6.3 Summary 91

7 Usage 93

7.1 Translation of Planning Problems to Interpreted Systems 93
7.2 Plan-Finding via MCMAS 93
7.2.1 Dinner domain 94
7.2.2 Gripper domain 94
7.2.3 Blocks World domain 96

8 Project Evaluation 97

8.1 Theory 97
8.1.1 Strengths 97
8.1.2 Weaknesses 97

8.2 Implementation 98

8.2.1 Strengths 98

8.2.2 Weaknesses 98

9 Conclusions and Future Work 101

9.1 Summary of Work 101

9.2 Future Work 101

A Plans generated for Gripper domain problems 109

A Robot Navigation Domain (Full Observability) in ISPL 113

Chapter 1

Introduction

Model checking is a method used for verifying the correctness of certain properties of a system, specified in some modal language. It is possible to construct a representative model $\mathcal{M}$ of a real system using states, transitions and Boolean propositions. Given a specification represented as a formula ϕ, one can algorithmically verify that the property expressed by the formula holds true in the model. This is of great value in mission-critical systems, especially in the case where human health is at risk, taking the case of the Therac-25 disaster [30] as an example.

Planning is one of the ways in addressing one of the key problems of intelligent behaviour. To find a *plan* is to find a sequence of executable actions in a *planning domain* which can lead an agent from a initial situation to a desired goal. In current literature, we find that a direct correspondence between solving planning problems and model checking can be drawn, allowing us to efficiently plan for goals expressed in a rich variety of ways.

We now wish to extend the current work in the field of planning via model-checking, by exploring methods of planning for more complex goals, where we specify properties about the *execution structure* of a plan. We examine the branching-time logic Computation Tree Logic (CTL), which is commonly used for specifying temporally-extended goals, and study how we can instead exploit the expressive power of the more general full branching-time logic CTL*, which supersedes CTL, to formulate more complex goals.

One example of the expressivity of CTL* can be seen in the formula $E\,[GFp]$ for some atomic formula p, which can be interpreted as "there is a path along which p is infinitely often true. We examine how this cannot be expressed in either CTL or the Linear-Time Logic LTL. We find that CTL* gives us much greater flexibility in the properties we wish to express for a goal, compared to what is currently possible with the tools and logics seen in the current state-of-the-art.

1.1 Primary Objectives

This project presents a formal correspondence between planning problems with temporally extended goals expressed in CTL* to model checking input. A tool will then be built allowing for planning with such properties, which is currently not possible in existing planners. Support will also be provided for partially observable domain specifications. From the perspective of an agent in the system, the agent will have incomplete information about the planning domain during the execution of the plan. To facilitate this, a compiler will be built, translating domains specified in a standard planning language into model checking input.

The planning language we will use is the *Planning Domain Definition Language* (PDDL). A non-deterministic version of this language also exists, called NPDDL. This can be used for planning with incomplete information. We will be using a model checking toolkit for the verification of Multi Agent Systems (MCMAS) [32].

The input of the model checker is based on Interpreted Systems (ISs), which are described in detail in the Background section. The IS is encoded in an ISPL file. The outline of the project is as follows:

- Build the first planner for temporally extended goals specified in CTL* with incomplete information.
- Draw a formal correspondence between such domains expressed in PDDL, and interpreted systems.
- Build a compiler to translate planning domain descriptions in PDDL to model checking input. The model checking input will be written in ISPL, the input language to the MCMAS model checker.

1.2 Challenges

- There is currently very limited material for our specific use case of planning with CTL*, as well as planning with incomplete information.
- There is a potentially high computational cost of synthesizing such complex plans along with planning incomplete information. Optimisations may have to be made directly in MCMAS and/or in the parsing phase of compilation.
- We must effectively work with a large legacy codebase (MCMAS) which does not have an existing test suite already set up. Any changes to the source code need to have a guarantee of not altering existing behaviour. Acceptance tests need to be written to check the end-to-end behaviour of the solution.
- Testing the output of the generated plans may have to be done manual inspection, which is not a scalable solution. A separate tool which will read the plan may

have to be created in order to deal with this. (MCMAS)

1.3 Contributions

The main contributions of our work are as follows:

- Develop algorithms for planning for temporally extended goals with incomplete information via model checking.
- Prove the correctness of such planning algorithms, with attention given to their computational complexities.
- Implement the algorithms in the MCMAS model checker (if necessary).
- Prove the correctness of the output of the PDDL–ISPL compiler. It must be possible to prove a bisimulation between the planning domain input into the compiler and the interpreted system represented by the output of the compiler.
- Evaluate the system on a variety of scalable planning domains with various temporally-extended goals. Such planning domains should be easy to scale in order to check the robustness of the models generated by the compiler for large state spaces.

Chapter 2

Background

We will now give an overview of the relevant work in the area of Planning and Model Checking, study the planners and model checkers that are already in existence, and examine the contexts in which certain logics are used for expressing goals in planning problems.

2.1 Model Checking

Model checking is the process of determining whether a formula is true in a model [19]. Model checking is based on the following fundamental ideas:

1. A domain of interest is described by a semantic model. Examples of these are a computer program or a reactive system.

2. A desired property of the domain is described by a logical formula. An example of this is a safety requirement for a reactive system.

3. The fact that a domain satisfies a desired property is determined by checking whether the formula is true in the model.

The way in which we formalise domains will be with Kripke Structures.

Definition 2.1 (Kripke Structure)**.** *We define a Kripke Structure K as a 4-tuple $\langle W, W_0, T, L\rangle$, where*

1. W is a finite set of states.

2. $W_0 \subset W$ is a set of initial states.

3. $T \subset W \times W$ is a binary relation on W, the transition relation, which gives the possible transitions between states. We require T to be ***left-total****, i.e. for every state $w \in W$ there exists a state $w' \in W$ such that $(w, w') \in T$. This means that one can model deadlock by having a state with a single outgoing edge arriving back* *onto itself. Note that although we have initial states, we do not require the notion of terminating states.*

4. *$L : W \mapsto 2^{\mathcal{P}}$ is a labelling function, where $\mathcal{P}$ is a set of atomic propositions. L assigns to each state the set of atomic propositions true in that state.*

We shall now formulate a way to encode the possible evolution of the domain.

Definition 2.2 (Path)**.** *We define a path as an infinite sequence $w_0w_1w_2\dots$ of states in W such that, for each i, $(w_i, w_{i+1}) \in T$. We require that paths begin from an initial state $w_0 \in W_0$. Due to the totality requirement of T from Definition 2.1, it follows that all paths are infinite.*

Example. *The corresponding Kripke Model as depicted in in Figure 2.1 is as follows:*

- $W = \{1, 2, 3\}$
- $W_0 = \{1\}$
- $T = \{(1,1), (1,2), (1,3), (2,2), (2,3), (3,3), (3,2)\}$
- $L(1) = \{p, q\}, L(2) = \{p\}, L(3) = \{q\}$

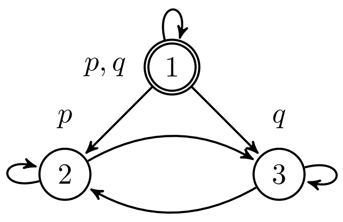

Figure 2.1: An example Kripke model.

2.2 Symbolic Model Checking

Using *symbolic methods* to perform model checking is a breakthrough technology based on the idea of encoding sets of states in a model as bit vectors and using *boolean functions* to encode the transitions present in the model. Ordered Binary Decision Diagrams [3] can then be used as a way of manipulating such boolean functions.

Definition 2.3 (Boolean function)**.** *We define boolean functions to be function which take booleans as inputs and produce booleans as output.*

Observe that for a function with n boolean arguments, there are 2^n different possible inputs, since each argument can take either the value 0 or 1. We also see that since the output of the function must either be 0 or 1, we have 2 outcomes for each input, meaning that we have a total of 2^{2^n} different functions altogether.

2.2.1 Binary Decision Trees

Before we continue to defining a binary decision *diagram*, we must first explore the concept of a binary decision *tree*. A binary decision tree (BDT) is similar to a binary trie. Let the input to a function be the boolean variables $x_1, \ldots, x_n$.

At the root of the tree, we test one of the variables, which produces two subtrees. One subtree corresponds to the case where $x_i = 0$ and other corresponds to $x_i = 1$ for some i where $1 \leq i \leq n$. We continue testing variables in each subtree to create more subtrees, until we eventually find ourselves at the leaf nodes where we have either 0 or 1. This will be the output of the function. Traversing a root–leaf path through the tree is analogous to evaluating the value of each variable of the function's input.

Let us consider the boolean function $x_1 \wedge x_2$ as an example. We can interpret this as the following function in familiar C-style syntax:

```
bool and(bool x1, bool x2) {
    return x1 && x2;
}
```

The corresponding binary decision tree would be that represented in Fig. 2.2. We will use a certain convention for the trees when testing the variable x_i for some i, $1 \leq i \leq n$. The convention is as follows: if x_i is true, we proceed into the right subtree of the corresponding node in the tree (represented as a solid line). If x_i is false, we proceed into the left subtree of the corresponding node (representation as a dashed line).

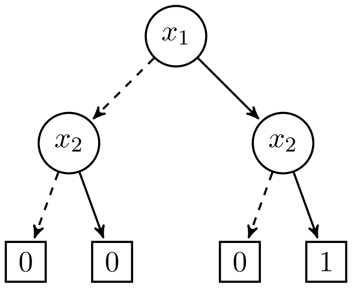

Figure 2.2: Binary decision tree for the function $x_1 \wedge x_2$.

Observe that binary decision trees have the following properties:

- **Large size**: a formula of n variables leads to $2^n - 1$ nodes in the tree, with 2^n leaf nodes in the lowest level. Clearly, the size of the tree is exponential in n, where n is the number of variables in the formula.
- **Canonicity**: The BDT is unique only if we test variables in a fixed order $x_1, \ldots, x_n$. This means that if we want to test whether two boolean functions are logically equivalent, we can simply perform an equality test between two trees.
- **Decision trees can be ordered**: If we fix a variable ordering, we denote the corresponding decision tree as *ordered*.

We now desire to obtain a more compact representation of boolean functions while preserving the canonicity property, in order to make it easier to test the equality of a pair of boolean functions. This leads us to *binary decision diagrams*.

2.2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) as proposed in [12], represent Boolean functions as directed acyclic graphs. Performing operations of boolean functions can then be efficiently done through the use of graph algorithms on such data structures.

It can be shown that BDDs provide the advantage of testing properties such as satisfiability and equivalence, which are of great importance in the field of model checking. Using BDDs for symbolic model checking is currently used in MCMAS and many other model checkers.

We find that the problem of checking if a formula φ is satisfied by a model is reduced to the problem of checking whether the initial state of the model belongs to the set where φ is satisfied. This can be implemented as a BDD test.

We find that BDDs differ from BDTs in two fundamental ways:

- **The redundant testing of boolean variables can be omitted**: Observe in Fig. 2.2 that testing the value of x_2 is pointless after we already see that x_1 is false. This means that we can instead construct a BDT like that seen in Fig. 2.3.
- **Identical subtrees can be shared**: This reduces the number of nodes present in the BDD significantly, where for some boolean formula we may see many subtrees which cannot be distinguished from another.

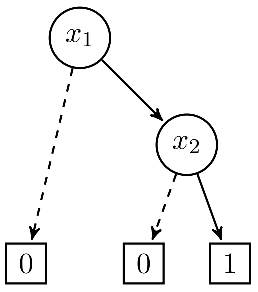

Figure 2.3: An equivalent binary decision diagram corresponding to the formula $x_1 \wedge x_2$.

We find that because of these two properties, the potential for the size of the diagram to be exponential in the number of variables of the formula are significantly reduced. Observe that in the worst case, the size of the BDT will still be exponential in n.

2.2.3 Reduced Ordered Binary Decision Diagrams

If we fix the ordering of the variables, we can obtain a reduced ordered binary decision diagram (ROBDD). ROBDD's are also *reduced*, meaning that they have the following properties:

- **Irredundancy**: The left and right subtrees of each node are distinct
- **Uniqueness**: There are no two distinct nodes testing the same variable with the same subtree(s).

Observe also that for a fixed variable ordering, each boolean function corresponds to a unique ROBDD [31]. This is the *canonicity* property. Two boolean function can therefore be tested to be equivalent by checking if the corresponding ROBDDs are the same. As seen in Fig. 2.4 and in general, ROBDDs have at most two leaf nodes, 0 and 1.

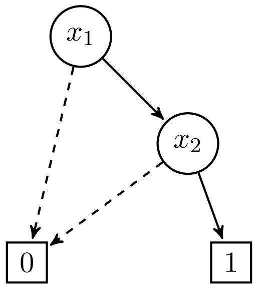

Figure 2.4: ROBDD corresponding to the formula $x_1 \wedge x_2$.

This now brings us to being able to check if a boolean formula is *valid* or *satisfiable.*

2.2.4 Validity and Satisfiability

The concepts of validity and satisfiability are fundamental to the field of model checking, as they allow the efficient determination of properties of a formula given a model.

Definition 2.4 (Validity)**.** *We say that a boolean formula is* ***valid*** *if it is equivalent to true. This mean that we may assign any combination of values to the variables of a formula, but it will inevitably evaluate to* `true`*. In the context of* ROBDD*s, this mean that regardless of where we traverse in the diagram, we always end up at* `1`*.*

Definition 2.5 (Satisfiability)**.** *We say that a boolean formulate is* ***satisfiable*** *if there is some assignment of variables which makes the formula evaluate to* `true`*. In the context of* ROBDD*s, by looking at the diagram, we can check whether the formula is equivalent to* `false` *in a similar way to checking validity. If this is not the case, this means that there must exist some assignment of truth values for the variables which makes the formula evaluate to* `true`*.*

2.3 Temporal Logics

Model checking is completely based on *temporal logic.* Temporal logics are used when formulae are not inherently statically true or false in a model. The difference that we have here is that formulae can be true in some states of a model but false in others. We are essentially replacing the static notion of truth with a dynamic one.

Definition 2.6 (Temporal logic). *Temporal logics have a* ***dynamic*** *aspect to them, where the truth of a formula depends on the time-point inside the model. This contrasts with the static nature of predicate or propositional logic, where the truth of a formula is fixed.*

In the following sections, we will discuss three temporal logics, CTL, LTL and finally CTL*. All of these logics can be used to reason about properties involving time in Kripke models.

2.4 Linear-time Temporal Logic

Linear-time Temporal Logic (LTL) is a temporal logic which includes connectives allowing us to refer to the future. Time is represented as a sequence of states evolving infinitely into the future. We refer to these sets of states as a *computation path.* We define the set *Atoms* to be a set of atomic formulas, for example, p, q, r, or 'Main valve on espresso machine 4 is leaking'.

2.4.1 Syntax

We can now define the syntax of LTL.

Definition 2.7 (Syntax). LTL *has a syntax defined by the following BNF grammar:*

$$\phi ::= \top | \bot | (\neg\phi) \,|\, (\phi \wedge \phi) \,|\, (\phi \vee \phi) \,|\, (\phi \rightarrow \phi) \,|\, (X\phi) \,|\, (F\phi) \,|\, (G\phi) \,|\, (\phi U \phi)$$

The abbreviations $\phi_1 W \phi_2$, 'weak until' and $\phi_1 R \phi_2$ 'release' also exist, but we will not making use of them for the purposes of this project.

2.4.2 Semantics

We can now define the semantics of LTL. The semantics of LTL are defined in terms of *paths*, which are defined in the following way:

Definition 2.8 (Path). *A path in a model $\mathcal{M} = (S, R, L)$ is an infinite sequence of states $s_0, s_1, s_2, \ldots$ in S such that for each $i \geq 1$, $R(s_i, s_{i+1})$.*

Definition 2.9 (Semantics). *Let $\mathcal{M} = (S, R, L)$ be a model and $\pi = s_0 s_1 s_2 \ldots$ a path in $\mathcal{M}$. We define the notion of π satisfying an* LTL *formula ϕ, i.e. that $\phi \models_{\mathcal{M}} \phi$ in the following way:*

- $\pi \models \top$

- $\pi \not\models \bot$
- $\pi \models p \Leftrightarrow p \in L(s_0)$
- $\pi \models \neg\phi \Leftrightarrow \pi \not\models \phi$
- $\pi \models \phi_1 \wedge \phi_2 \Leftrightarrow \pi \models \phi_1$ *and* $\pi \models \phi_2$
- $\pi \models \phi_1 \vee \phi_2 \Leftrightarrow \pi \models \phi_1$ *or* $\pi \models \phi_2$
- $\pi \models \phi_1 \rightarrow \phi_2 \Leftrightarrow \pi \models \phi_2$ *whenever* $\pi \models \phi_1$
- $\pi \models X\phi \Leftrightarrow \pi[1] \models \phi$
- $\pi \models G\phi \Leftrightarrow$ *for all* $i \geq 0, \pi[i] \models \phi$
- $\pi \models F\phi \Leftrightarrow$ *there exists some* $i \geq 0 : \pi[i] \models \phi$
- $\pi \models \phi U \psi \Leftrightarrow$ *there exists some* $i \geq 0 : \pi[i] \models \psi$ *and for* $j = 0, \ldots, i-1$ *we have* $\pi[j] \models \phi$

2.5 The Branching-Time Logic CTL

Computation Tree Logic (CTL) is a branching-time logic, where time is interpreted as a tree-like structure, and the future is not determined. In this logic, we are able to formulate properties such as *liveness* or *safety*. It has uses for specifying fairness conditions, such as that used in [14]. Unlike CTL*, where operators can be freely mixed, we are only permitted to group operators into two in CTL – one path operator followed by a state operator. In the sense of temporally extended goals, we can use CTL to express weak, strong and cyclic planning problems.

Referring to the definitions provided in [14], we will now proceed to define the syntax and semantics of CTL.

2.5.1 Syntax

We can now proceed to define the syntax of well-formed formulae in CTL. Using the approach taken in [14], firstly let AP be a set of *atomic propositions*. We can now define the grammar for CTL formulas over AP as follows:

- If $p \in AP$, then p is a CTL formula.
- If ϕ_1, ϕ_2 are CTL formulas, then so are

  $$\neg\phi_1, \quad \phi_1 \wedge \phi_2, \quad AX\,\phi_1, \quad EX\,\phi_1, \quad A[\phi_1\,U\,\phi_2], \quad \text{and } E[\phi_1 U \phi_2],$$

  where:

  - The $\wedge$ and $\neg$ symbols have their usual meanings.

– X is the *nexttime operator*. $AX\phi_1$ intuitively means that ϕ_1 holds in every immediate successor of the current state, while $EX\phi_1$ means that ϕ_1 holds only in some immediate successor state.

– U is the *until* operator; $A[\phi_1 U \phi_2]$ intuitively means that for every computation path, there is an initial prefix of the path such that ϕ_2 holds at the last state of the prefix, with ϕ_1 holding at all other states of the prefix. $E[\phi_1 U \phi_2]$ has a similar meaning, but for *some* computation path.

Note that this syntax is in reduced form, and all other CTL operators can be defined from these primitive operators.

2.5.2 Semantics

We can now proceed to define the semantics of CTL formulae. In order to gain an intuition from the semantics, it is sometimes helpful to think of a Kripke model as a literal tree, where taking a path in the model corresponds to performing a traversal down the corresponding tree. An example of this can be seen in Fig. 2.5.

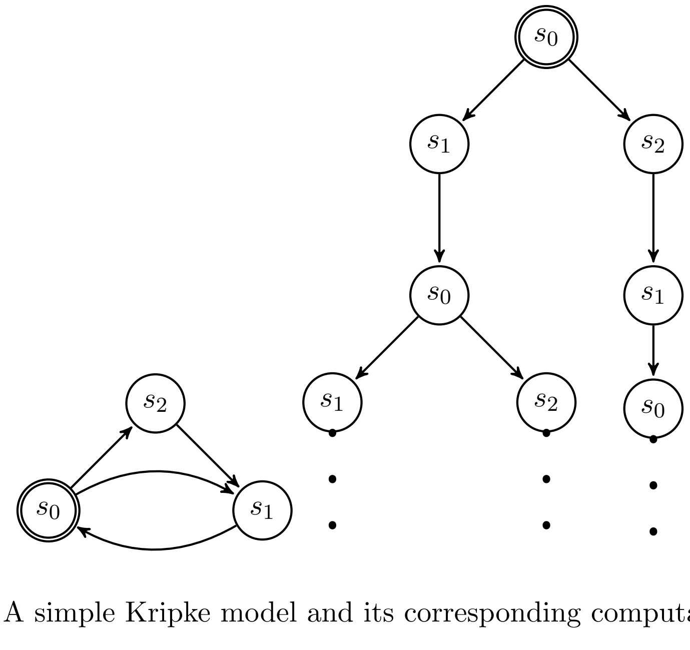

Figure 2.5: A simple Kripke model and its corresponding computation “tree”.

Let $\mathcal{M}$ be a Kripke structure and s be a state in $\mathcal{M}$. If a state s_0 of the Kripke structure $\mathcal{M}$ satisfies a CTL formula ϕ it is denoted $s_0 \models \phi$. We inductively define the $\models$ relation as follows.

$$\begin{aligned}
s_0 \models p &\iff p \in P(s_0) \\
s_0 \models \neg\phi &\iff \text{not } (s_0 \models \phi) \\
s_0 \models \phi_1 \wedge \phi_2 &\iff s_0 \models \phi_1 \text{ and } s_0 \models \phi_2 \\
s_0 \models AX\,\phi &\iff \text{for all states } t \text{ such that } (s_0, t) \in R,\ t \models \phi \\
s_0 \models EX\,\phi &\iff \text{for some state } t \text{ such that } (s_0, t) \in R,\ t \models \phi \\
s_0 \models A[\phi_1 U \phi_2] &\iff \text{for all paths } (s_0, s_1, \dots), \\
&\qquad \exists i\,[i \geq 0 \wedge s_i \models \phi_2 \wedge \forall j[0 \leq j < i \implies s_j \models \phi_1]] \\
s_0 \models E[\phi_1 U \phi_2] &\iff \text{for some paths } (s_0, s_1, \dots), \\
&\qquad \exists i\,[i \geq 0 \wedge s_i \models \phi_2 \wedge \forall j[0 \leq j < i \implies s_j \models \phi_1]]
\end{aligned}$$

Using the above definition, we can now define the abbreviations $AF\,\phi$, $EF\,\phi$, $AG\,\phi$, $AF\,\phi$ in the following way:

- $AF\,\phi \equiv A[\top U \phi]$: ϕ holds at some point in the future along every path from s_0.
- $EF\,\phi \equiv E[\top U \phi]$: There exists some path from s_0 at which ϕ eventually holds.
- $EG\,\phi \equiv \neg AF \neg\phi$: There exists some path from s_0 at which ϕ holds at every state.
- $AG\,\phi \equiv \neg EF \neg\phi$: ϕ holds at every state on every path from s_0; ϕ is *globally* true.

2.6 The Full Branching-Time Logic CTL*

CTL*, as proposed in [21], is a superset of Computational Tree Logic (CTL) and Linear Temporal Logic (LTL) [23]. The expressive powers of CTL and LTL are combined in CTL*. In computer science, the main uses of CTL* are for checking the correctness of complex reactive systems [39]. Recall that from the CTL definition, every temporal operator was required to be combined with an associated path quantifier (A, E). This constraint is relaxed in CTL*. We are now permitted to write formulas such as:

- $E[(rUp) \vee (rUq)]$: There is a path along which either r is true until p or r is true until q.
- $A[p \Rightarrow Xp]$: Along all paths, if p is true at the current state, p must be true at the next state also.
- $A[GFp]$: Along all paths, p is infinitely often true.

2.6.1 Syntax

We will now define the context-free grammar used to generate the language of well-formed CTL* formulae, which is composed of two classes:

- *State formulas*, which are evaluated in states:

$$\Phi ::= \bot \,|\, \top \,|\, p \,|\, (\neg\Phi) \,|\, \Phi \wedge \Phi \,|\, \Phi \vee \Phi \,|\, \Phi \Rightarrow \Phi \,|\, (\Phi \Leftrightarrow \Phi) \,|\, A\phi \,|\, E\phi,$$

  where p is an arbitrary atomic formula, and ϕ is a path formula.

- *Path formulas*, which are events along paths:

$$\phi ::= \Phi \,|\, (\neg\phi) \,|\, (\phi \wedge \phi) \,|\, (\phi \vee \phi) \,|\, (\phi \Rightarrow \phi) \,|\, (\phi \Leftrightarrow \phi) \,|\, X\phi \,|\, F\phi \,|\, G\phi \,|\, [\phi U \phi],$$

  where Φ is any state formula.

2.6.2 Semantics

We define CTL* semantics in terms of Kripke structures. We inductively define satisfaction on state formulae as follows:

Definition 2.10 (CTL* semantics for state formulae)**.** *If a state s of the Kripke structure satisfies a state formula Φ it is denoted $s \models \Phi$.*

- $(\mathcal{M}, s) \models A\phi \Leftrightarrow \pi \models \phi$ *for all paths π starting in s*
- $(\mathcal{M}, s) \models E\phi \Leftrightarrow \pi \models \phi$ *for some path π starting in s*
- $(\mathcal{M}, s) \models \Phi$ *is defined as in* LTL.

Definition 2.11 (CTL* semantics for path formulae)**.** *Denote the path π as $s_0, s_1, \ldots$. If a path π satisfies a path formula ϕ, it is denoted $\pi \models \phi$. Denoting $\pi[n]$ as the sub-path $s_n, s_{n+1}, \ldots$, we inductively define the semantics on path formulae as follows:*

- $\pi \models \Phi \iff (\mathcal{M}, s_0) \models \Phi$
- $\pi \models X\phi \iff \pi[1] \models \phi$. *$\phi$ is true in the next state of the path π of the model.*
- $\pi \models F\phi \iff \exists n \geq 0 : \pi[n] \models \phi$. *$\phi$ holds at some future point in the path π of the model.*
- $\pi \models G\phi \iff \forall n \geq 0 : \pi[n] \models \phi$. *$\phi$ holds at every point in the path π of the model.*
- $\pi \models [\phi_1 U \phi_2] \iff \exists n \geq 0 : (\pi[n] \models \phi_2 \wedge \forall 0 \leq k < n, \pi[k] \models \phi_1)$. *$\phi_1$ holds at every point in the path π up until some point, where ϕ_2 then becomes true.*

The other logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ have their usual meaning.

2.7 Multi-Agent Systems

We will now define the notion of an *agent*, which is largely used in the field of model checking.

Definition 2.12 (Agent). *An agent is a computer system capable of autonomous action on behalf of its user or owner. It is able to decide the steps which need to be done to satisfy provided design objectives.*

One particularly desirable property which we want from agents is the ability for them to collaborate together and engage in complex communication with other agents in order to help to accomplish a common goal. This leads us to define the notion of a *multi-agent system.*

Definition 2.13 (Multi-agent system). *A multi-agent system consists of a number of interacting agents. To successfully interact, they will require the ability to cooperate and coordinate with each other.*

We will now discuss a formalism used to describe the *computations* carried out by a multi agent system. This formalism is called an *interpreted system* and is described in the next section. In the case of adversarial planning, we may run into the case of agents competing with each other, with potentially conflicting goals in mind. Some agents may even have the task of preventing the achievement of another agent's goals.

2.8 Interpreted Systems

Interpreted systems formally describe the *computations* carried out by a set of agents [33]. We can define them in the following way:

Definition 2.14 (Interpreted system). *For a set of agents $\Sigma = \{1, \ldots, n\}$, an interpreted system IS is a tuple:*

$$IS = \langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h \rangle,$$

where

- Each agent $i \in \Sigma$ is characterised by a finite set of *private local states* L_i
- Act_i is a finite set of actions that may be performed for agent i.
- $P_i : L_i \to 2^{Act_i}$ is a *protocol* for agent i. The actions of the agent must be performed in compliance with the protocol, which allows for *non-determinism* in the system.
- E is a special "agent", referred to as the *environment*, of which agents reside. It has its own set of local state, set of actions and protocol
- $t_i : L_i \times L_E \times Act_1 \times \cdots \times Act_n \times Act_E \to L_i$ is a function describing the *evolution* of the agents' local states. It gives the next local state as a function of the current local state of the agent, the environment, and all the other agents' actions.
- I is a set of *initial* global states
- $h : AP \to 2^G$ is a valuation function, where

– AP is a set of atomic propositions

– G is a set of reachable global states

It is also necessary to provide a means of defining the semantics of an interpreted system IS. This can be done by associating a Kripke model M_{IS} to the interpreted system IS. We call M_{IS} the *associated model*. We define the associated model $M_{IS} = (G, t, (\sim_i)_{i\in\Sigma}, L)$ for an interpreted system $IS = \langle (L_i, Act_i, P_i, t_i)_{i\in\Sigma\cup\{E\}}, I, h\rangle$ in the following way:

- We define the set of possible worlds G to be the set of reachable states of IS. One can obtain G from the set of initial states I by iterating the evolution function t as previously defined.
- We define the temporal relation $t \subseteq G \times Act \times G$ to relate two worlds by means of a joint action.
- We define the *epistemic accessibility relation* $\sim_i$ in terms of the equality of local components of two given global states.

  $$g \sim_i g' \text{ iff } l_i(g) = l_i(g'),$$

  i.e. the local states of agent i in global states g and g' are the same.
- The labelling function $L : AP \to 2^G$ is equivalent to the previously defined evaluation function h.

2.9 MCMAS

MCMAS is a Model Checker for Multi-Agent systems (MAS). It takes in input a specification and a set of formula the user wishes to be verified, and it evaluates the truth value of the formulae using algorithms based on Ordered Binary Decision Diagrams (OBDDS). MCMAS is able to generate counterexamples for false formula and witnesses for true formula whenever possible. We will use the witness generation mechanism as a means of finding the set of actions which will constitute a plan for a given planning problem.

MCMAS allows for the reasoning about correct behaviour and strategies specified by formulae in CTLK and ATLK. Several extensions to MCMAS exist, including MCMAS*, which we will be using to find temporally-extended goals specified in CTL*.

2.9.1 Architecture

In order to use MCMAS, the user specifies a model in the Interpreted Systems Programming Language (ISPL). `lex` and `yacc` are used to parse the input. OBDDs are then build using the CUDD library. After parsing any formulae specified by the user, MCMAS then computes a set of states in which the formula(e) holds. This is compared with the set of states reachable from the initial situation, and `true` or `false` is lastly output by MCMAS with either a witness or counterexample if desired by the user. A graphical representation of the process can be seen in Fig. 2.6

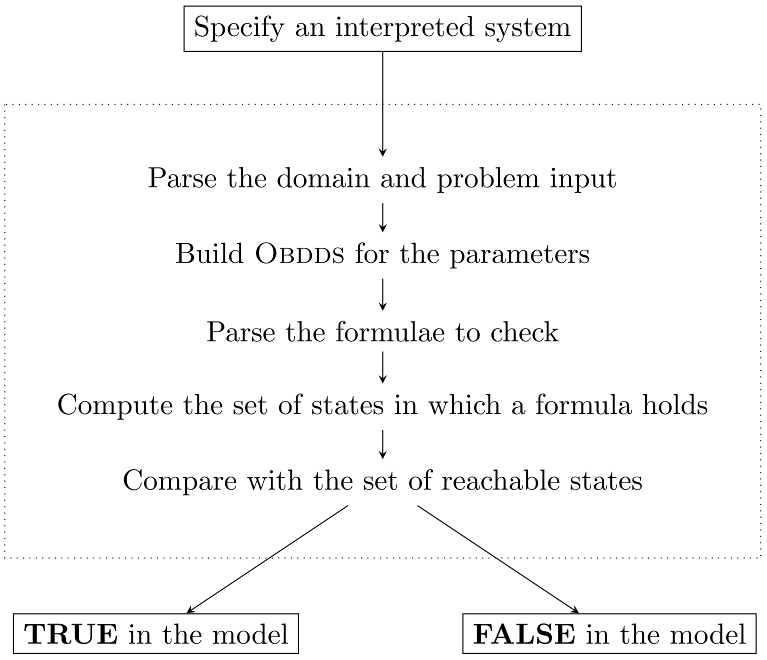
Specify an interpreted system
Parse the domain and problem input
Build OBDDS for the parameters
Parse the formulae to check
Compute the set of states in which a formula holds
Compare with the set of reachable states
TRUE in the model
FALSE in the model

Figure 2.6: Architectural overview of MCMAS.

2.10 Interpreted Systems Programming Language (ISPL)

Descriptions of multi-agent systems are given by means of ISPL (Interpreted Systems Programming Language) programs. ISPL is an agent-based, modular language inspired by interpreted systems, a popular semantics in multi-agent systems. It is the input language to MCMAS. We will formulate a translation between planning problems specified in PDDL to ISPL, in order to be fed into MCMAS.

ISPL programs are defined using the following syntax:

```
Agent Environment
  Obsvars:
  ...
  end Obsvars
  Vars:
  ...
  end Vars
  RedStates:
  ...
  end RedStates
  Actions = {...};
  Protocol:
  ...
  end Protocol
  Evolution:
  ...
  end Evolution
end Agent

Agent TestAgent
  Lobsvars = {...};
  Vars:
  ...
  end Vars
  RedStates:
  ...
  end RedStates
  Actions = {...};
  Protocol:
  ...
  end Protocol
  Evolution:
  ...
  end Evolution
end Agent

Evaluation
...
end Evaluation

InitStates
...
end InitStates

Groups
...
end Groups

Fairness
...
end Fairness

Formulae
```

```
...
end Formulae
```

The purpose of each language construct is as follows:

- **Definition of variables**: The `Vars` section allows for the three types of variables: Boolean, enumeration and bounded integer. In our case, we use booleans to store the information about the state of action-performing agents in the planning domain, and enumerations and bounded integers for fluents.
- **The definition of local observable variables**: We can store variables in agents which were originally declared in the environment agent using `Lobsvars`. The special section `Obsvars` is used to define variables in the environment agent which is observable by all agents. This construct is used in our case, when planning under full observability, as we assume that all action-performing agents have full state information about the environment.
- **Definition of red states**: We are allowed to define `RedStates`, which is a boolean formula defined over a standard agent's local and locally observable variables and an environment agent's local variables. Local states that satisfy the formula are red, while other states are green.
- **Definition of actions**: We define the actions of each agent in the `Actions` section.
- **Definition of protocol function**: We define the protocol function with a condition specified as a boolean function over an agent's local states, followed by a list of actions allowed to be performed in the local states satisfying the boolean function. Note that we can model non-determinism in agents by specifying conditions that are not mutually-exclusive. In this case all actions in both conditions are considered possible by MCMAS. This is of use for strong planning with non-deterministic domains.
- **Definition of the evolution function**: A line in an evolution function specifies a set of assignments of local variables and an *enabling condition*, which is a Boolean formula over all variables and actions of all agents. We say that an item has been *enabled* in a state if its enabling condition has been satisfied in that state.

  We specify the variables being assigned to a new value on the left hand side of an assignment, and the enabling condition as Boolean formula on the right hand side of the environment.

  We use the evolution function to specify the postconditions (effects) of a given PDDL action.
- **Definition of evaluation function**: We define an evaluation function as a group of atomic propositions defined over global. The proposition is true in all global states that satisfy the Boolean formula. We use the evaluation function to specify reachability goals for a planning problem.

- **Definition of initial states**: The `InitStates` construct is used to define the global initial states of the interpreted system as a Boolean function over variables. We use this specify the initial state of the plannig problem. We can model non-determinism in the initial state by specifying Boolean formulae in the form

  ```
  Agent.x=1 or Agent.x=2 or Agent.x=3
  ```

  for some local variable `x` for some action-performing agent `Agent`.

- **Definition of groups**: We can specify group modalities by using `Groups`. For example we can specify groups such as

  ```
  g1 = { Agent1, Agent2, Agent3, Environment };
  ```

  and may want to achieve the goal `<g1> F goal;` (find a plan such that the coalition of agents contained in group `g1` can collaborate to eventually achieve `goal`).

- **Definition of fairness formulae**: We can define fairness formulae in the `Fairness` section in order to avoid undesireable behaviour. We do not use fairness conditions in our case.

- **Definition of formulae to be checked**: We define formulae to be verified over atomic propositions. This is used in our case for specifying reachability and temporally-extended goals.

2.11 Introduction to Planning

In the field of Artificial Intelligence, we aim to address one of the most central problems, the problem of *selecting the action to do next* [18]. We find that there have been three main approaches of tacking this:

1. *Programming based approach.* This involves having a controller which prescribes which action to take next, which is provided by the programmer.

2. *Learning-based approach.* Here, the controller is induced from experience. An example of this is seen in reinforcement learning.

3. *Model-based approach.* The controller here is automatically derived from a model of actions, sensors and goals.

We find that planning uses the *model-based* approach to select the action to do next. This is the method we will focus on for this project.

Planning is the problem of devising a sequence of actions to satisfy a high-level goal. Formally, we aim to map the initial state (or set of initial states) into a goal state via these actions.

In this project, we will eventually look into *conformant* planning, where the initial state isn't completely known. In *conformant* planning, we aim to find a sequence of actions that guarantee to achieve the goal regardless of any uncertainty in the initial condition of the system and in any non-deterministic effects of actions.

As suggested in [8], we can segment planning domains into three different types. These have differing levels of observability in the planning domain, ranging from an agent having perfect information of the domain to it having no information. These are defined in the following way:

1. *Fully observable*: The state of the world is assumed to be completely observable at run-time
2. *Null-observable*: No information is available at run-time
3. *Partially observable*: Only part of the domain information is available at run-time.

For the purposes of this project, we will begin with fully observable planning domains, and then move on to explore plans for partially-observable domains. We see in the literature that methods of tackling problems in the area of partial observability are very commonly suggested to be researched further.

We can distinguish between plans which may or may not be guaranteed to achieve the goal.

Definition 2.15 (Weak plan)**.** *A weak plan is a set of state-action pairs which may achieve the goal.*

We see the potential of goals not being achieved by a sequence state-action pairs if there is non-determinism in the planning domain. For example, executing a certain action may have more than one effect, leading to a state without a direct way to transition into the goal state, or requiring extra transitions than without non-determinism. We can express this in CTL as $EF\mathcal{G}$ for a propositional formula $\mathcal{G}$ representing the set of goal states.

Definition 2.16 (Strong plan)**.** *A strong plan is a set of state-action pairs which is guaranteed to achieve the goal.*

We can express the notion of a strong plan as the CTL formula $AF\mathcal{G}$ for a propositional formula $\mathcal{G}$ representing the set of goal states. Intuitively, the formula can be interpreted as "The goal $\mathcal{G}$ will eventually be reached"

Definition 2.17 (Strong cyclic plan)**.** *A strong cyclic plan is a plan whose executions always have a possibility of terminating and, when they do, they are guaranteed to achieve the goal.*

We can phrase the notion of a strong cyclic plan as "for each possible execution, *always* during the execution, there *exists* the possibility of *eventually* achieving the goal. In CTL this sentence can be expressed by the formula $AGEF\mathcal{G}$, where $\mathcal{G}$ is a propositional formula representing the set of goal states. More detail about the syntax and semantics of CTL will be given in Section 2.5.

Plans of the above sort can be used to achieve a subset of *temporally-extended goals.*

Definition 2.18 (Temporally-extended goal)**.** *These are goals that express conditions on the whole execution associated to the solution plan, rather that the conditions on the final states. One can formulate these as* CTL *formulae. We aim to formulate these as* CTL* *formulae. We see that there is limited progress in literature in the case of having temporally-extended goals under partial observability.*

2.12 Planning via Model Checking

Planning via the use of model checking is based on generating plans by *determining whether formulas are true in model* [19]. For this to be possible, we need to be provided with a *planning domain*, the *planning problem* and the method used for *plan generation*, defined respectively as follows:

Definition 2.19 (Planning domain)**.** *We describe the domain of the plan by a semantic model. It is used to define the states of the domain, available actions and the state transitions which occur as a result of executing the respective actions. A planning domain D is a 4-tuple $\langle F, S, A, R\rangle$, where*

1. *F is a finite set of fluents*
2. *$S \subset 2^F$ is a finite set of states*
3. *A is a finite set of actions*
4. *$R : S \times A \mapsto S$ is a transition function. We define an action a to be executable in $s \in S$ if $R(s, a) \neq \emptyset$.*

The *planning problem* is the problem of *finding plans of actions* provided a planning domain, initial and goal states. We generate a plan by exploring the state space of the semantic model. We can translate the model depicted in Figure 2.1 into a planning domain by simply labelling the edges with actions, as seen in Figure 2.7.

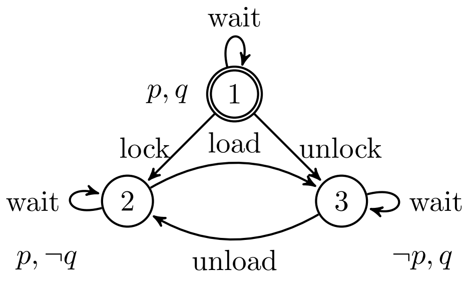

Figure 2.7: An example Planning Domain.

The corresponding planning domain as depicted in the diagram is as follows:

- $F = \{p, q\}$
- $S = \{\{p, q\}, \{p, \neg q\}, \{\neg p, q\}\}$
- $A = \{lock, unlock, load, unload, wait\}$
- $R = \{(\{p, q\}, wait, \{p, q\})$
  $(\{p, q\}, lock, \{p, \neg q\})$
  $(\{p, q\}, unlock, \{\neg p, q\})$
  $(\{p, \neg q\}, wait, \{p, \neg q\})$
  $(\{p, \neg q\}, load, \{\neg p, q\})$
  $(\{\neg p, q\}, wait, \{\neg p, q\})$
  $(\{\neg p, q\}, unload, \{p, \neg q\})\}$

We will now proceed to formally define a *planning problem.*

Definition 2.20 (Planning problem)**.** *A planning problem P for Planning Domain $D = \langle F, S, A, R\rangle$ is a 3-tuple $\langle D, I, G\rangle$, where $I = \{s_0 \subset S\}$ is the initial state and $G \subset S$ is the set of goal states.*

This allows us to now define a *plan*, which can be thought of as a sequence of actions being executed in a set of states, taking an agent from the initial state to the goal.

Definition 2.21 (Plan)**.** *A plan π for a planning problem $P = \langle D, I, G\rangle$ with planning domain $D = \langle F, S, A, R\rangle$ is defined as*

$$\pi = \{\langle s, a\rangle : s \in S, a \in A\}$$

Definition 2.22 (Executable plan)**.** *We say that a plan is executable if we have that*

$$\pi = \{\langle s, a\rangle : s \in S, a \in A, R(s, a) \neq \emptyset\}.$$

We observe from [19] that the problem of planning can be reduced to model checking EFp with p being a propositional CTL formula if we base our planning domain on a Kripke structure.

2.13 Buchi Automata

It is possible to synthesise plans by using an approach based on certain properties of *Buchi Automata* [43] on infinite words. We find that the approach can also be generalised to work in the case of planning with *incomplete information*, where we have uncertainty in the initial situation of the planning domain and successive states are partially observable. We will begin by defining a *transition system*, and examining methods used to plan under *complete information.*

2.13.1 Transition Systems

Definition 2.23 (Transition system). *We define a (finite) transition system $\mathcal{T}$ as the tuple*

$$\mathcal{T} = (W, W_0, Act, R, Obs, \pi)\,.$$

where

- *W is a finite set of possible states*
- *$W_0 \in W$ is the finite set of possible initial states*
- *Act is the set of possible actions*
- *$R : W \times Act \to W$ is the transition function. It can be thought of as a function which returns the next state, given a state and an action.*
- *Obs is the finite set of possible observations. This models the observable part of states. Under complete information, we have this as the set of all states.*
- *$\pi : W \to Obs$ is the observability function. It returns the observable part of the current state. Under complete information this will be the identity function, since the observable part of each state will be the state itself.*

We will now proceed to define the notion of an *execution* of a transition system, the *trace* of an execution, and the *observable behaviour* of a transition system. This will allow us to reason about Buchi automata

Definition 2.24 (Execution of a transition system). *For a transition system $\mathcal{T}$, we define the execution to be an infinite set of states $w_0, w_1, w_2, \ldots$ such that $w_0 \in W_0$ and $w_{i+1} = R(w_i, a)$ for some $a \in Act$.*

Definition 2.25 (Trace of an execution). *We define the trace to be what we can observe from an execution. For example, for the execution $w_0, w_1, w_2, \ldots,$ the trace is $\pi(w_0), \pi(w_1), \pi(w_2) \ldots.$ Observe that under complete information, the trace will be equal to the execution, since every state of it can be observed.*

Definition 2.26 (Observable behaviour). *We define the observable behaviour of a dynamic system to be the set of all possible traces of the transition system.*

2.13.2 Automata on Infinite Words

With the preliminaries from the previous section, we may now proceed to study Buchi automata, using [6] as reference.

Firstly, we will define an *infinite word.*

Definition 2.27 (Infinite word). *Given a finite nonempty alphabet Σ, an infinite word is an element of Σ^ω. It is an infinite sequence $a_0, a_1, \ldots, a_n$ of symbols from Σ.*

Definition 2.28 (Buchi Automaton)**.** *A Buchi automaton is a tuple*

$$\mathcal{A} = (\Sigma, S, S_0, \rho, F)$$

where:

- Σ *is the alphabet of the automaton*
- S *is the finite set of possible states*
- $S_0 \subset S$ *is the set of possible initial states. Under complete information, this will be a singleton set, as we know with certainly which state the automaton will begin with.*
- $\rho : S \times \Sigma \to 2^S$ *is the transition function of the automaton*
- $F \subset S$ *is the set of accepting states.*

We can now explore various properties of a Buchi automaton $\mathcal{A}$.

Definition 2.29 (Input words)**.** *The input words of* $\mathcal{A}$ *are infinite words* $a_0, a_1, a_2, \dots \in \Sigma^\omega$*. This can be thought of as a sequence of actions of a transition system.*

Definition 2.30 (Run)**.** *A run of* $\mathcal{A}$ *on an infinite word* $a_0, a_1, a_2, \dots$ *is an infinite sequence of states* $s_0, s_1, s_2, \dots \in S^\omega$ *such that* $s_0 \in S_0$ *and* $s_{i+1} \in \rho(s_i, a_i)$*. We say that a run* r *is accepting iff* $lim(r) \cap F \neq \emptyset$*, where*

$$lim(r) = \{s | s \text{ occurs in } r \text{ infinitely often}\}.$$

This means that there is at least one state $s_f \in F$ *that is visited infinitely often.*

Definition 2.31 (Language)**.** *The language accepted by a, denoted* $L(\mathcal{A})$ *is the set of words for which there is an accepting run.*

Definition 2.32 (Nonemptiness problem)**.** *The non-emptiness problem for an automaton is to decide given an automaton* $\mathcal{A}$ *whether* $L(\mathcal{A}) \neq \emptyset$*, i.e. if the automaton accepts at least one word.*

2.14 Planning via Automata

We discover in [16] that we are able to synthesize a plan by checking for the non-emptiness of a Buchi automaton. Observe that the algorithm used to check for non-emptiness can be modified to return a plan if a plan exists.

We also find that the power of Buchi automata can also be used for both sequential and conditional planning with *incomplete information.*

2.15 Relation Between Planning and Reinforcement Learning

It is noted in [5] that Reinforcement Learning (Rl) can be thought of as a type of universal planning. Here, the goal is represented as a "reward" function in a *Markov Decision Process* [20] (Mdp) model of the domain.

Definition 2.33 (Markov Decision Process)**.** *A Markov Decision Process (*Mdp*) model contains:*

- *A set of possible world states S*
- *A set of possible actions A*
- *A real valued reward function $R(s, a)$*
- *A description of each action's effects in each state*

We normally expect non-deterministic (Nd) planners to be able to handle domains with a larger state space than in reinforcement learning, since we observe that the Rl domain representation is in general more complex than in a non-deterministic planner.

2.16 Existing Planning Problem Description Languages

We will now discuss the description languages used to specify planning tasks, highlight their individual strengths and weaknesses, and define our motivations for using our language of choice, pddl.

2.16.1 Non-deterministic Agent Domain Language

The *Non-deterministic Agent Domain Language* (Nadl) was introduced in [24] in an attempt to describe multi-agent planning domains. It has a significantly different syntax compared to pddl. It is similar to pddl in the sense that agents are given a set of actions which have their own preconditions and effects.

In general, there is a rather simple model of interactions among concurrent actions, since it is not possible for two actions to concurrently assign different values to a numeric fluent. Each specified agent was initially required to share the same goal, but constraint was later lifted in a later version, to allow agents to have potentially different goals. We however see that no accompanying description language was provided [29]. Nadl is the input language for the Umop planner.

2.16.2 PDDL

The Planning Domain Definition Language (PDDL) [34], is an attempt to standardise planning domain and problem description languages. It is the standard language used to encode classical planning problems. The introduction of such a standardised language was of great benefit for the International Planning Competition (IPC) series.

In the IPC, performance of planning systems are compared on sets of benchmark problems. Therefore, it is essential that a common language for specifying problems should be used. The components of a PDDL planning task are:

- **Objects**: The things in the world which are of interest
- **Predicates**: Properties of the objects which are of interest.
  - Note that these have no intrinsic meaning per se. The meaning of a predicate is determined by what combination of arguments make it true, and the relationships to other predicates, which is determined by effects that actions have on predicates and what instances of the predicate are listed as true in the initial state of the problem definition.
- **Initial state**: The state of the world in which we start in. We define the initial state as a set of predicates which must hold true in the initial situation. All predicates which are not explicitly said to be true in the initial conditions are assumed to be false.
- **Goal specifications**: A specification of a property we wish to be true. A *solution* to a planning problem is thus a series of actions such that
  1. The action sequence is feasible starting from the given initial situation
  2. The **goal** is true in the situation resulting from executing the action sequence
- **Actions/Operators**: The protocols in which the state of the world is changed

Actions

The actions which are to be ran during the execution of the plan are defined in terms of an *action schema*, as introduced in [40]:

Definition 2.34 (Action schema)**.** *The action schema represents a number of different actions that can be derived by instantiating the variables used in the action's parameter list to different constants. The action schema consists of three parts; the action name and parameter list, the precondition, and effect.*

1. *The list of* ***parameters*** *lists the variables upon which the specific action operates on. One can think of them as the "arguments" of the action.*

2. *The* ***Precondition*** *is an optional goal description which must be satisfied before the action is applied.*

3. ***Effects*** *list the changes which the action imposes on the current state of the world. They may be universally quantified and conditional, but full first order sentences (e.g., disjunction) are not allowed, unless we use extensions of the language.*

For the purposes of this project, we will focus on only the standard PDDL definition of effects. All variables must be bound. If a predicate P is not mentioned in an action's effects then the truth of P is assumed to be unchanged by an instance of the action.

An important assumption to make about how we reason about the literals which hold true in a given state during the execution of a plan is the STRIPS assumption.

Definition 2.35 (STRIPS Assumption)**.** *Every literal* ***not mentioned*** *in the effect of an action schema* ***remains unchanged****.*

The consequence of this assumption is to avoid the *representational frame problem.* We must also note that having a positive effect from an action which is already present in a given state does not lead to the effect being added twice. On the other hand, if a negative effect is not specified in a given state, that part of the effect is ignored.

Goal Descriptions

Goal descriptions are used to specify the goals that are desired in the planning problem. These are also used for specifying the precondition for an action. We are allowed to use function-free first-order predicate logic for goal descriptions. One must be sure to have occurrences of predicates in goal descriptions agreeing with its domain declaration in terms of its number of arguments.

Specifying Planning Tasks

Tasks specified in PDDL are split into a *domain file* and a *problem file.* We define domain files in the following form:

```
(define (domain <domain name>)
    <PDDL code for predicates>
    <PDDL code for first action>
      ...
    <PDDL code for last action>
)
```

The name of the planning domain is specified by the string `<domain name>`. The **problem** defines what the planner tries to solve.

We define problem files in the following format:

```
(define (problem <problem name>)
    (:domain <domain name>)
    <PDDL code for objects>
    <PDDL code for initial state>
    <PDDL code for goal specification>
)
```

Here, we require that `<domain name>` matches the name of the domain of which was specified in the corresponding domain file. `<problem name>` is a string identifying the planning task.

We note that we are also able to specify *fluents* in order define metrics to measure the quality of a plan.

2.16.3 The Multi-Agent Extension of PDDL 3.1

The *Multi-Agent Extension of* PDDL *3.1* (MA-PDDL) was proposed in [29] with the BNF grammar given in [28]. The purpose of the language was to give an additional, optional extension to the existing PDDL language. MA-PDDL only provides minimalistic changes and is backwards compatible with every existing extension in the official language. An informal semantics of the language is given in [29].

One drawback that we however see with MA-PDDL is that it does not include support for partial observability. By default, the planning environment is fully observable, meaning that any observation is a complete description of new states and action-combinations that produced them.

Nevertheless, we will continue using PDDL for use in our project, due to its reputation on being the standard language for use in International Planning Competitions. A multi-agent planning track is also proposed in [29] for upcoming IPCS, which will make it more convenient to benchmark out solution.

We also observe that it is noted that in MA-PDDL it is "non-trivial" to represent *strategies* in a compact manner. We believe that this will be one of the strengths of out solution, since MCMAS has the built-in ability to efficiently reason about the multiple strategies involved in systems of multiple agents [2].

2.16.4 NuPDDL

NuPDDL is a extension of PDDL for planning in non-deterministic domains [7]. It models this with a variety of new constructs.

- We are enabled to specify temporally-extended goals using the temporal logic CTL, which can be used by specifying a `:ctlgoal`. For example, one can seek to obtain a plan in the form `af(..)`, `ag(..)`, `ef(..)`,....

- We can also have uncertainty in the initial state of the planning problem. This uncertainty can be specified using the `oneof` and `unknown` keywords, for example

  `(:init (and P1) (oneof (and (P2) (P3)) (P4)) unknown(P5))`, where:

  - The `oneof (..)` keyword allows the state to be set non-deterministically to one of a set of states
  - The keyword `unknown (f)` is the same as specifying a `oneof` statement, but with the type of `f` taking any possible value.
- We are allowed to have non-deterministic action effects
- We can have partial observability of the state of the domain.

We note that most classes of non-deterministic problems can be captured by a temporal logic, since most of the classes identify constraints over the *execution* of the plan, rather than the goal of the plan itself.

For example, we may wish to ensure that some goal has a guarantee of being reached ($AF\,goal$), or if there is only a possibility ($EF\,goal$) of it being reached. It is crucial to have these types of goals distinguished, as is may be the case that a given action could have a non-deterministic action, with the option of either staying in the same state or transitioning to another one.

The language is used as the input of the Mbp planner. Although the Mbp planner offers a powerful selection of strengths, we see that it cannot currently plan for temporally extended goals under partial observability. We also do not see an attempt to plan for multiple agents in Mbp. We aim to tackle both of these problems with a combination of MCMAS and our compiler.

We also find that Mbp does not offer support for non-deterministic and/or faulty sensors. Having this functionality would be a further step into significantly more realistic planning problems, but is beyond the scope of this project, and would have to be a topic for further study.

2.16.5 EaGLe

EaGLe, the “Extended Goal Language” is a goal language proposed in [15] used to specify an even richer set of extended goals. CTL is used to specify extended goals allowing us to distinguish between temporal requirements on “all the possible executions” and on “some executions” of a plan.

In contrast, EaGLe is introduced in an attempt to extend CTL with the possibility of expressing classes of goals that are typical of real world applications, but can’t be expressed in CTL or any other existing temporal logics. For example, one may wish to express “Try to achieve a goal whenever possible” or “If you fail to achieve a goal, recover by trying a different goal”.

The language can also be used as the input of the MBP planner.

2.17 Existing Planners

We will now study the planners which are currently in use today. We will examine their strengths and shortcomings in order to see how our solution will compare to the state-of-the-art.

2.17.1 The Model Based Planner

The *Model Based Planner* (MBP) is used for planning for non-deterministic domains with temporally-extended goals and several degrees of observability. The input language of MBP is the NPDDL language as discussed previously.

MBP also has capabilities for the *validation* of plans, where it uses model checking to validate plans.

2.17.2 Universal Multi-agent OBDD-based Planner

The *Universal Multi-agent OBDD-based planner* (UMOP) [24] is based on model checking algorithms and uses BDDs to encode the planning domain symbolically using the BuDDy BDD package. It supports five planning algorithms:

1. Strong planning
2. Strong cyclic planning
3. Optimistic planning
4. Strong cyclic adversarial planning
5. Optimistic adversarial planning

NADL, the *Nondeterministic Agent Domain Language*, is the language used for representing the agent domain in the input of the UMOP planner.

2.17.3 SATplan

SATplan [25] is a method for *automated planning.* It is based on the method of Planning as Satisfiability [26]. The planning problem is converted into a Boolean satisfiability problem (also known as a SAT problem), which is then solved using a method for establishing satisfiability.

2.17.4 Existing MCMAS Extensions

In a previous MSc Thesis, we see two algorithms proposed for planning for epistemic goals in the Alternative-Time Temporal *Epistemic* Logic (ATEL) [42]. ATEL extends ATL by adding knowledge modalities. An example of a formula expressible in ATEL is $K_i\langle\langle i\rangle\rangle \circ p$, which means that "agent i knows that he can make p true in the next state". In the thesis, the Planning via Model Checking method [19] is used for constructing the two algorithms:

1. Planning during the model checking phase, constructing universal strong plans
2. Planning after model checking

A compiler was then constructed, translating the Umop specification language to Ispl in order to be fed into the mcmas model checker. We see that the planner was used on a variety of domains, including the "Rugby", "Muddy children" and "Gripper" domain. Evaluation is on the correctness of output of the planner. We will perform a similar evaluation in our case, but for the logic ctl*.

2.18 Other Existing Logics

We will now examine other logics used in the current state-of-the art for expressing goals for planning problems, and evaluate their benefits and shortcomings.

2.18.1 ATL*

Atl* was proposed in [4] as a generalisation of ctl*. Instead of using the path quantifiers 'there exists' (E) and 'for all' (A), we instead use *strategic modalities* of the form $\langle\langle\, E\,\rangle\rangle$ and $\langle\langle\, A\,\rangle\rangle$ referring to a group of agents A. The cooperation and also competition of agents can be expressed using the modalities in order for them to achieve temporal goals.

There are some problems with planning for goals specified in Atl* [36, 41]. There a lack of support for binding strategies explicitly to various agents, or the same agents in different contexts. To overcome these difficulties, we have Strategy Logic which was proposed in [37].

2.18.2 Strategy Logic

Strategy Logic (SL) was introduced in [13] in an attempt to overcome some of the shortcomings seen in logics such as Atl*. It can express game-theoretic properties such as Nash equilibrium. Nash equilibrium occurs in a non-cooperative game of two or more players where each player is assumed to know the equilibrium strategies of the other players, and no player has anything to gain by changing their own strategy.

The main obstacle in using strategy logic in a planner is that model checking is very expensive. Other fragments of SL exist, for example SL[1G] and SL[BG], introduced in [38], with more efficient model checking methods, but we find that they are not as expressive as the 'full' SL.

2.18.3 α-CTL

As proposed in [17], α-CTL was introduced in an attempt to specify more complex temporally-extended goals. We however find that we are unable to specify a coalition of agents using this logic. This logic differs from CTL in the sense that CTL is used for formulating properties with respect to the *execution structure* of a plan, whereas $\alpha-$CTL allows semantic properties to be specified about the *planning domain* directly.

α-CTL allows us to specify properties such as "Try its best to achieve the goal g" for some agent. We find that the property can in fact be specified in another similar extension to CTL called P-CTL*, which we will not discuss further in this project, but the model checking procedure is less computationally expensive when using α-CTL.

Chapter 3

From Planning Problems to Interpreted Systems

In this chapter, we will discuss the method in which we can compile from a planning problem specified in a PDDL file to a semantically interpreted system specified in an ISPL file. We will do this by going through different examples of standard planning problems, and then proving the bisimularity of the generated model compared to the original planning problem.

3.1 The Dinner Domain

We will now consider a simple example domain in order to study the way in which we can perform the translation to an interpreted system. The planning problem is described in [44], where we plan to prepare a surprise date for a sleeping partner. We wish to achieve the goal of having `dinner` cooked, a `present` wrapped, and for the `garbage` to be taken out. We have four possible actions that we can perform: `cook`, `wrap`, `carry`, and `dolly`.

In order to `cook`, which results in `dinner` being cooked, we require that our hands are clean. To `wrap`, which results in a `present`, we require that it is `quiet`, since we do not want to wake up our significant other. To `carry` means to take out the garbage. This requires that the garbage has not already been taken out, and results in the garbage being taken out but leaves our hands unclean. We note that none of the actions are parametrised. The `dolly` action is another means of taking out the garbage, which leaves our hands clean, but creates noise. We start with the initial state of having `garbage`, `clean` hands, and `quiet` in the room.

A formal summary is as follows:

3.1.1 Domain

Recall that the domain is defined to be a set of predicates, along with a set of actions with their own respective preconditions and effects:

- **Predicates**: The predicates are the following:

  `clean, dinner, quiet, present, garbage`

- **Actions**: The actions are as follows:
  - `cook`
    - Precondition: `clean`
    - Effect: `dinner`
  - `wrap`
    - Precondition: `quiet`
    - Effect: `present`
  - `carry`
    - Precondition: `garbage`
    - Effect: `not(garbage), not(clean)`
  - `dolly`
    - Precondition: `garbage`
    - Effect: `not(garbage), not(quiet)`

3.1.2 Problem

We can now define our planning problem in terms of its initial and goal state. We define a *state* to be the conjunction of a number of predicates.

- **Initial state**: The initial state is simply the case when the following three fluents hold true:

  `garbage, clean, quiet`

- **Goal state**: The goal is when the following three fluents hold:

  `dinner, present, not(garbage)`

3.1.3 Planning Domain Specification

As described in Section 2.12, we can now define the Dinner domain via a semantic model M_P.

- **Fluents**: We can define our fluents as the following set:

$$F = \{\texttt{quiet}, \texttt{clean}, \texttt{dinner}, \texttt{present}, \texttt{garbage}\} .$$

- **Set of states**: We now need to define the set of states S that can possibly be visited in the domain. For convenience, we will let S be the set of all subsets of F, namely 2^F.

- **Set of actions**: The set of actions will be

$$A = \{\texttt{cook}, \texttt{wrap}, \texttt{carry}, \texttt{dolly}\} .$$

- **Transition function**: We must finally define our transition function $R : S \times A \mapsto S$. We will make the assumption that the domain is deterministic, i.e. that it is not possible to arrive in more than one state after the execution of an action.

  We can now begin to write the function from the initial state of the system starting as follows:

$$\begin{aligned}
R\left(\{\texttt{garbage}, \texttt{clean}, \texttt{quiet}\}, \texttt{cook}\right) &= \{\texttt{garbage}, \texttt{dinner}, \texttt{quiet}, \texttt{clean}\} \\
R\left(\{\texttt{garbage}, \texttt{clean}, \texttt{quiet}\}, \texttt{wrap}\right) &= \{\texttt{garbage}, \texttt{present}, \texttt{quiet}, \texttt{clean}\} \\
R\left(\{\texttt{garbage}, \texttt{clean}, \texttt{quiet}\}, \texttt{carry}\right) &= \{\texttt{quiet}\} \\
R\left(\{\texttt{garbage}, \texttt{clean}, \texttt{quiet}\}, \texttt{dolly}\right) &= \{\texttt{clean}\} \\
&\vdots
\end{aligned}$$

  This can be written more generally as

$$R\left(s, a\right) = \left(s \setminus a_{neg}\right) \cup a_{pos}, \quad \text{for } a \in A, s \in S, \text{ if } a_{pre} \subseteq s,$$

  where the action a_{pre} denotes the positive preconditions of action a (recall that negative preconditions are ignored), a_{pos} denotes the positive effects of a, and a_{neg} denotes the negative effects of a.

  Note that the cardinality of R, namely $|R|$, is of the order of $|A||S| = 2^{|F|}|A|$ in size, which grows exponentially with the number of fluents in the planning domain.

We will now proceed to formally define the planning *problem.*

3.1.4 Planning Problem Specification

Recall from Section 2.12 that the planning problem P for a planning domain $D = \langle F, S, A, R\rangle$ is a 3-tuple $\langle D, I, G\rangle$, where $I = \{s_0\} \in S$ is the initial state, and $G \subseteq S$ is the set of goal states. We will now attempt to specify the planning problem P for the Dinner domain.

- **Initial state**: We define the initial state as

$$I = \{\texttt{garbage}, \texttt{clean}, \texttt{quiet}\}$$

- **Goal states**: We define the set of goal states as the set corresponding to all states where $\texttt{dinner} \wedge \texttt{present} \wedge \neg\texttt{quiet}$ holds true, namely:

$$G = \left\{ \begin{array}{c} \{\texttt{dinner}, \texttt{quiet}, \texttt{present}\}, \{\texttt{dinner}, \texttt{present}, \texttt{clean}\}, \\ \{\texttt{dinner}, \texttt{present}\}, \{\texttt{dinner}, \texttt{present}, \texttt{quiet}, \texttt{clean}\} \end{array} \right\}$$

Construction of Corresponding Kripke Model

This concludes the specification of our planning problem $P = \langle D, I, G\rangle$. From [19], we find that it is possible to construct a Kripke model $K = \langle W, W_0, T, L\rangle$ corresponding to the planning problem P. We will denote the corresponding Kripke model by $\mathcal{M}_P$.

The Kripke model corresponding to the planning problem P with domain $D = \langle F, S, A, R\rangle$ can be constructed as follows:

1. $W = S = 2^F$, as previously defined, where F is the set of fluents defined for the planning domain.
2. $W_0 = I$
3. $T \subset W \times W$ is such that

   $$T(w, w') \text{ iff } \exists a \in A : w' = R(w, a),$$

   where R is the transition function as previously defined for the planning domain D.
4. $L = \text{id}$, the identity function. This is the case because the states of the planning domain are determined by the set of fluents which are true in a given state.

We will now discuss the construction of an interpreted system $\mathcal{M}_{IS}$ for the planning problem, which we later show is semantically equivalent to $\mathcal{M}_P$.

3.1.5 Translation to Interpreted Systems

We will now attempt to translate the semantic model for the planning domain M_P into an Interpreted System (IS). Since the actions in this domain do not have any parameters, we are able to make a one-to-one mapping between the four PDDL actions and the actions required for the IS. Note that this will not be the case for $n-$ary predicates (for $n > 0$), as we will see that we must *ground* the predicates in order to fully specify the state-space.

- **Local States** L: To define our local states, let us firstly define a set S containing all predicates of the planning domain which will act as *atoms* in our IS:

$$S = \{\texttt{quiet}, \texttt{clean}, \texttt{dinner}, \texttt{present}, \texttt{garbage}\}.$$

  Each of the atoms can either be true or false for a given state. All of the possible truth values of the atoms can be encoded by using the *power set* of S. Define 2^S to represent the *power set* of S, i.e. the set of all subsets of S. Our interpreted system will have a single agent, the environment agent. We can now define our set of local states as

$$L = L_E = \left\{ \bigwedge_{i=1}^{|s|} s_i : s_i \in s, s \in 2^S \right\}.$$

  Let $\wedge_{i=1}^{|s|} s_i = l$ for some $s \in 2^S$. We interpret $S \setminus s$ as the set of atoms in the local state l which are *false*.

- **Actions**: Our actions are identical to the actions defined in the planning domain. The actions that we wish to be able to perform are to cook dinner, wrap a gift, carry out the trash manually, or to carry out the trash using the dolly. Formally,

$$Act = \{\texttt{cook}, \texttt{wrap}, \texttt{carry}, \texttt{dolly}\}$$

- **Protocol function**: We aim to be able to express the states which are possible as a result of performing an action. We are enabled to cook dinner only if our hands are clean. If there is garbage, we should be enabled to use the dolly or carry out the trash. If it is quiet, we should be enabled to wrap the gift. We can naturally express this in our protocol function P, where $P : L \to 2^{Act}$ is defined as follows:

$$P(l) = \begin{cases} \{\texttt{cook}\} & \text{if } \texttt{clean} \in s \\ \{\texttt{dolly}, \texttt{carry}\} & \text{if } \texttt{garbage} \in s \\ \{\texttt{wrap}\} & \text{if } \texttt{quiet} \in s, \end{cases} \tag{3.1}$$

  where $s \in S$ is such that $l = \wedge_{i=1}^{|s|} s_i : s_i \in s$.

- **Evolution function**: We wish to formally describe how the system will evolve through the execution of actions. We can read off an evolution function from the precondition–effect description of the actions defined in the planning domain description.

  If we perform the `cook` action, we wish to attain a state where the `dinner` atom is true. If the `dolly` action is performed, we wish to attain a state where the `garbage` and `quiet` atoms are both false. If `carry` is performed, we wish to attain a state where the atoms `garbage` and `clean` are both false. If we perform `wrap`, we should attain a state where `present` is true. Note that the evolution function is independent to the local state that the agent is in at the time of performing an action.

Let $t(l, a) = l'$ for some action a and some local state l. Let S be the set of all atoms as defined before. Let s be the corresponding element of the power set 2^S for the resulting local state l'. Atoms contained in the set $S \setminus s$ can non-deterministically be true or false. The evolution function $t : L \times \mathit{Act} \to L$ can be defined for some action $a \in A$ as follows:

$$t(\wedge_\alpha : s_\alpha \in s, a) = \wedge_\alpha s_\alpha : s_\alpha \in \begin{cases} s \cup \{\texttt{dinner}\} & \text{if } a = \texttt{cook} \\ s \setminus \{\texttt{garbage}, \texttt{quiet}\} & \text{if } a = \texttt{dolly} \\ s \setminus \{\texttt{garbage}, \texttt{clean}\} & \text{if } a = \texttt{carry} \\ s \cup \{\texttt{present}\} & \text{if } a = \texttt{wrap} \end{cases}$$

In general,

$$t\left(\wedge_\alpha s_\alpha : s_\alpha \in s, a\right) = \wedge_\alpha s_\alpha : s_\alpha \in (s \setminus a_{neg}) \cup a_{pos}, \quad \text{where } s \in 2^S, a \in A,$$

where a_{pos} and a_{neg} respectively denote the positive and negative effects of an action a from some planning problem P.

- **Initial global states**: In our case, we have only one initial global state. We define our initial state as $i = \texttt{garbage} \wedge \texttt{clean} \wedge \texttt{quiet}$. Every other atomic proposition must be false. Our set of initial global states is therefore the singleton set

$$I = \{\texttt{garbage} \wedge \texttt{clean} \wedge \texttt{quiet}\}.$$

- **Valuation function**: The valuation function $h : AP \to 2^G$ of the interpreted system IS_P is defined to map a set of reachable global states to each atomic proposition. One may wish to define a valuation function $h : AP \to 2^G$, which assigns to each state $g \in G$ the set of atomic propositions which are assumed to be true at that state. Since we only have one agent, observe that $L = G$.

Since the states of the IS are themselves are determined by the set of atomic propositions true at each state, it is natural to define

$$h\left(\underbrace{\wedge_{i=1}^{|s|} s_i}_{g}\right) = s, \quad \text{for } s_i \in s, s \in 2^S \tag{3.2}$$

3.1.6 Kripke Model Translation of Dinner Domain

We must now define the Kripke model equivalent to our interpreted system. We will denote the corresponding Kripke model as $\mathcal{M}_{IS}$.

Let $\mathcal{M}_{IS} = \langle W, W_0, T, L \rangle$ be such that

- $W = L$, the set of all possible global states in the IS, as previously defined.
- $W_0 = I$, the singleton set of initial states, as previously defined.
- $T \subseteq W \times W$ such that

  $$T(w, w') \text{ iff } \exists a \in Act : w' = t(w, a),$$

  where t is the evolution function as previously defined for the interpreted system and where the action a is appropriately enabled for a given local state by the protocol function defined in (3.1).
- $L = h$, the interpreted system's valuation function as defined in expression 3.2.

3.2 The Gripper Domain – Towards a More General Translation

We will now discuss the process of compilation for a more general planning problem, where we have parametrised actions and predicates in the planning domain. We will reference the "Gripper" domain, used in the first (1998) International Planning Competition as a way to gain an intuition about the translation.

The Gripper domain consists of a robot with two grippers, n balls, and two rooms. In the initial situation of the planning problem, one of the rooms contains all n balls.

Our goal is to eventually realise a state of all balls eventually being in the other room. The robot has the task of using its grippers to facilitate this state being reached.

The robot is constrained in a way which only allows each gripper to carry one item at any given time, and must drop an item before picking up another.

We define the predicates of the domain as the following:

```
room(r), ball(b), gripper(g), at-robby(r), at(b, r), free(g),
carry(o, g)
```

These have the following interpretations:

- `room(r)`: `r` is a room.
- `ball(b)`: `b` is a ball.

- `gripper(g)`: `g` is a gripper.
- `at-robby(r)`: The robot is located at `r`.
- `at(b,r)`: `b` is located at `r`.
- `free(g)`: `g` is free.
- `carry(o,g)`: the object `o` is being carried by `g`.

We combine these predicates together via conjunction and negation in order to produce preconditions and postconditions for actions. In PDDL the postconditions of an action are its *effects*. The preconditions must hold true before an action is performed, and the effect is guaranteed to hold true after the action is performed. In the Gripper domain, we have actions which take several parameters, which is not the case in the "Dinner" domain.

Having parametrised actions and predicates leads to an increase in the amount of states necessary to reason about after discovering the possible predicate–action combinations through the process of grounding.

In the Gripper planning problem, we have the following actions:

Move

- **Parameters**: There are only two parameters are the following:

  `from, to`
- Precondition: `room(from)`, `room(to)`, `at-robby(from))`
- Effect: `not(at-robby(from))`

Pick

- Parameters: `obj, room, gripper`
- Precondition:

  ```
  ball(obj), room(room), gripper(gripper), at(obj, room),
  at-robby(room), free(gripper)
  ```
- Effect: `carry(obj, gripper), not(at(obj, room)), not (free(gripper))`

Drop

- Parameters: `obj, room, gripper`
- Precondition:

`ball(obj), room(room), gripper(gripper), carry(obj, gripper), at-robby(room)`

Interpretation: The robot is in a room and a ball is being carried by a gripper of the robot.

- Effect:

  `and (at(obj, room), free(gripper), not(carry(obj, gripper)))`

  Interpretation: The ball is now in the room, the gripper is free, and the ball is no longer being carried by the gripper.

Note: It is forbidden to use the dash (`'-'`) character in variable names in ISPL files. We however generate variables in MCMAS from the predicate names. To solve this issue, when parsing the PDDL file, we currently perform some preprocessing in order to remove occurrences of dashes in names. For example, `at-gripper` is converted to `at_gripper`.

3.2.1 Problem

To define the planning *problem*, we specify a set of initial states of which the model begins, and the goal which should be realised by some plan, such that for a group of agents g_1, we have that $\langle g_1 \rangle F\, goal$ holds, where *goal* is a logical formula made up of a conjunction of predicates. In our case, the group g_1 consists only of a single agent.

In our case, we have a finite set *objects*, of which the predicates are quantified over.

Initial State

Our initial situation is the conjunction of the predicates shown in Fig. 3.1.

```
room(rooma), room(roomb)
ball(ball4), ball(ball3), ball(ball2), ball(ball1)
at-robby(rooma),
free(left), free(right),
at(ball4, rooma), at(ball3, rooma), at(ball2, rooma), at(ball1, rooma)
gripper(left), gripper(right)
```

Figure 3.1: Initial situation of the Gripper domain

Goal state

Our goal is to have all four balls inside room *b*, which is expressed by the conjunction of the following predicates:

```
at(ball4, roomb), at(ball3, roomb), at(ball2, roomb), at(ball1, roomb)
```

3.2.2 Planning Problem Specification

The planning problem specification for domains with parametrised actions and predicates relies on the process of *grounding.*

Definition 3.1 (Grounding)**.** *The process of obtaining all state variables and actions for a planning problem.*

A single predicate can represent a large number of state variables, and a single PDDL action can similarly represent a large number of ISPL actions.

We will now define the Gripper domain using a semantic model $D = \langle F, S, A, R \rangle$.

Fluents

Our fluents are the grounded set of all predicates applied to each of the PDDL `objects`. Let O be the set of all objects, and P the set of all predicates in the planning problem. Let $argcount(p)$ denote the number of arguments for predicate $p \in P$. Let $groundPredicate(p, O)$ be a function which combines a predicate p with a set of objects to obtain a set of fluents representing the predicate applied to $argcount(p)$ objects as parameters taken from the set O. Then,

$$F = \bigcup_{p \in P} groundPredicate(p, O). \tag{3.3}$$

For example, for the predicate $p = same$, where $argcount(p) = 2$ and $O = \{ball1, ball2\}$, we have that

$$groundPredicate(name, O) = \{same(ball1, ball2), same(ball2, ball1)\}.$$

Note that that different orderings of the arguments represent different groundings. Note also that one can draw an isomorphism between the *groundPredicate* function and the function $P(n, k)$ which returns the k-permutations of n, which is defined to be the different ordered arrangements of a k-element subset of an n-set.

States

As before, we define the set of states S as the set of all subsets of the set of fluents, namely $S = 2^F$. By intuition, We know that some states will not be reached during the execution of the plan, but we must first enumerate over all possibilities first.

Actions

In a similar fashion to before, let O denote the set of all objects in the domain. Let A_P be a set containing the names of all actions of the planning domain. Let $groundAction(a, O)$

be a function taking an action with name $a \in A_P$ and a set of objects O which returns a set of grounded actions for each permutation of objects in O of size $argcount(a)$, where $argcount(a)$ is a function returning the number of arguments taken by the action with name a. Then,

$$A = \bigcup_{a \in A} groundAction(a, O) \tag{3.4}$$

Transition function

We will now define the transition function $R : S \times A \mapsto S$. As before, we will require that R will be deterministic. R is defined as follows:

$$R(s, a) = (s \setminus a_{neg}) \cup a_{pos}, \quad \text{for } a \in A, s \in S, \text{ if } a_{pre} \subseteq s \tag{3.5}$$

where s denotes an appropriately grounded state contained in S, and a denotes a grounded action contained the set defined in 3.4. Define a_{neg} to be the set of grounded negative effects, a_{pos} to be the set of grounded positive effects, and a_{pre} the set of grounded preconditions of the action a. Then, to compute the result of $R(s, a)$ for some state s and action with name a, we remove the negative effects from the current state and union the result with the positive effects of a. This is conditional on the set a_{pre} being contained in s. In the case that this condition does not hold, the function is not defined for such a (s, a) pair.

Initial state

The initial state I is the set of grounded states representing the predicates defined in the `init` construct in the PDDL file describing the planning problem. It is necessary in PDDL to specify the values of each parameter of each predicate used for the `init` construct. Let I_P be the set of predicates used in `init` for a planning problem P. Then,

$$I = \{s : s \equiv i_P, s \in S, i_P \in I_P\} \tag{3.6}$$

In the case of the Gripper domain, I will be the grounded predicates of Fig. 3.1.

Goal state

We define the goal state G to be set of grounded states representing the set of predicates in the `goal` construct in a PDDL definition of a planning problem P. Let G_P be the set of predicates used in the `goal` construct of a planning problem P. Formally,

$$G = \{s : s \equiv g_P, s \in S, g_P \in G_P\} \tag{3.7}$$

In the case of the Gripper domain, G will be the set of states $s \in S$ representing the set of predicates

$$\{\texttt{at(ball4, roomb)}, \texttt{at(ball3, roomb)}, \texttt{at(ball2, roomb)}, \texttt{at(ball1, roomb)}\}.$$

This concludes the construction of the planning problem $P = \langle D, I, G\rangle$ where

$$D = \langle F, S, A, R\rangle.$$

We will now discuss the translation to a corresponding interpreted system IS_P which will have a Kripke model simulating $\mathcal{M}_P$.

3.2.3 Translation to Interpreted System

We will now discuss our attempt to encode the semantic model of a general planning problem as an Interpreted System.

3.2.4 Translation of Local States

We will now discuss the translation from the PDDL predicates to local states in ISPL. When parsing the PDDL file, we are able to obtain a set of predicates and a set of objects. In our Gripper domain, the predicates are:

```
room/1, ball/1, gripper/1, atrobby/1, at/2, free/2, carry/2
```

We denote `<predicate-name>/x` to mean "the predicate with name `predicate-name` takes `x` arguments".

Our objects are:

```
rooma, roomb, ball4, ball3, ball2, ball1, left, right
```

This defines the objects in our domain; our two rooms, four balls and two (left and right) grippers respectively.

ISPL supports three types of variables, boolean, enumeration and bounded integer [32]. We choose to represent the possible predicates of a planning domain as `boolean`s in ISPL, storing them in the `Obsvars` section of the environment agent.

To make this translation, for each predicate, we can compute the permutations of all objects of size x_i, with x_i being the number of arguments for predicate i. We may also maintain a hash table of these variables and their truth value in the compilation stage. This will be of use when encoding the initial states of the planning problem.

We ensure that the agents that interact with the environment do not have their own state, but instead change the observable state of the environment through the execution of actions. We do not require that the environment performs any actions.

For the `room` predicate, we know that it has one argument, and there are eight objects, so we know that we will have eight different states that are generated from this predicate to be used as local states, namely:

```
room(rooma), room(roomb), room(ball4), room(ball3), room(ball2),
room(ball1), room(left), room(right)
```

We can then create the variables

$$\begin{aligned}\texttt{room\_rooma} &: \texttt{boolean},\\ \texttt{room\_roomb} &: \texttt{boolean},\\ &\vdots\end{aligned}$$

where we substitute the corresponding object as the predicate's argument.

For predicates with two arguments, we use a similar method, ending up with 6 local states for each predicate with two arguments. In the general case, for a planning problem with m objects and a predicate i with r_i arguments, we will generate $\frac{m!}{(m-r_i)!}$ new local states. This leads to $\sum_{i=1}^{n} \frac{m!}{(m-r_i)!}$ local states for an arbitrary planning domain with n predicates. Therefore, for the set L_E of local states of the environment,

$$|L_E| = \sum_{i=1}^{n} \frac{m!}{(m-r_i)!},$$

where L_E is the conjunction of elements of 2^S, as previously defined using the *groundPredicate* function.

Define

$$S = \bigcup_{p\in P} groundPredicate(p, O). \tag{3.8}$$

Then,

$$L_E = \bigcup_{s\in 2^S} \left\{ \bigwedge_{\alpha\in I} s_\alpha : s_\alpha \in s \right\}, \tag{3.9}$$

where I is some index set for elements in each set s. We assume that all states contained L_E are observable by all agents in the planning domain.

3.2.5 Actions

Denote the set of actions for agent i of an interpreted system as Act_i. We find that for every action defined in a planning domain, we may need to create several corresponding actions for the IS. The amount of actions generated depends on the number of parameters of a given action.

For every PDDL action, at the parsing stage we already know the number of arguments each action will use. In a similar way of generating local variables, we again generate combinations of the objects in the planning domain of size k_j, where k_j is the number of arguments of the action j.

For every action, we therefore generate a total of $\frac{m!}{(m-k_j)}$ IS actions, so $\sum_{j=1}^{n_{act}} \frac{m!}{(m-k_j)}$ total actions for an arbitrary planning domain with m objects and n_{act} actions.

In a similar fashion to how the planning problem P was defined, we have that the set of actions for agent i of the IS are

$$Act_i = \bigcup_{a \in A} groundAction(a, O), \tag{3.10}$$

where O denotes the set of objects in the planning domain and *groundAction* is a function which, given an action defined in the planning domain and a set of objects, returns a set of grounded actions with respect to the set of objects and the number of arguments of the action a.

For the environment, we require that the set of actions are empty, i.e.

$$Act_E = \{\} \tag{3.11}$$

3.2.6 Protocol function

We use the protocol function $P_i : L_i \to 2^{Act_i}$ as a way of defining the actions that are enabled when in some local state $l_i \in L_i$. We achieve this by examining the preconditions which hold in a given state for a given action.

Let *groundPreconditions*(a) be a function which finds all preconditions of a given PDDL action a and returns a set grounded predicates representing all possible preconditions of the action with respect to the number of objects in the set O taken as parameters of the action.

$$P_i(l) = \{a : groundPreconditions(a) \subseteq s, a \in Act\}, \tag{3.12}$$

where l is a conjunction of fluents contained a set $s \in S$, and $a \in Act$ is a grounded action which represents the name and the specific parameter values of the action to be executed.

For the environment, we have an empty protocol function, as no actions are specified for it. We therefore have that:

$$P_E(l) = \{\} \text{ for all } l \in L_E \tag{3.13}$$

3.2.7 Evolution function

The evolution function is defined to give the next local state as a function of the current local state of the agent, environment, and all the other agent's actions. In our case, we will only change the observable state of the environment as a function of a separate agent's actions, which has knowledge of the actions from the planning problem.

We can encode the effects of the actions of the planning domain by using an evolution function. In our case, the next local (but globally observable) state of the environment is determined by the action of an action-performing agent i and the current local state of the environment, so we aim to define the function

$$t_E : L_E \times Act_i \to L_E.$$

This is encoded in ISPL as an `Evolution` construct.

For each grounded action $a \in Act$, we need to be able to obtain its grounded positive and negative preconditions. The positive preconditions are those which do not contain any negations.

Define the following functions for a given grounded PDDL action a:

- $groundNegatives(a)$: returns the set of grounded negative effects of a with respect to the parameter count of a and objects in O
- $groundPositives(a)$: returns the set of grounded positive effects of a with respect to the parameter count of a and objects in O

Take for example the action `drop`, which takes a single argument and a singleton set of objects $O = \{ball\}$.

Let the effects of the action be `on-ground(ball) and not(elevated(ball))`. Then,

$$groundNegatives(\texttt{drop(ball)}) = \{\texttt{elevated(ball)}\},$$

and

$$groundPositives(\texttt{drop(ball)}) = \{\texttt{on-ground(ball)}\}.$$

For the environment agent E, we can now proceed to define

$$t_E\left(l_E, a_i\right) = \wedge_\alpha s_\alpha : s_\alpha \in (s \setminus groundNegatives(a_i)) \cup groundPositives(a_i), \quad (3.14)$$

where $l_E = \wedge_\alpha s_\alpha : s_\alpha \in s, s \in 2^S, a_i \in Act_i$.

Each local state is expressed as a conjunction of elements of some state $s \in S$ and some action. We then compute the resulting set of atomic propositions that remain true after removing the negative effects and adding additional positive ones, and compute the conjunction of such a set. The preconditions of the effects are handled in the protocol function, so we do not need to add another conditional statement as seen in the transition function defined in 3.5.

3.2.8 Valuation function

It is now necessary to define the valuation function $h : AP \to 2^G$ which assigns to each propositional atom a set of states at which the propositional atom is true.

In our case, our set AP will only contain propositions of the form $\wedge_{\alpha \in I} s_\alpha$ for $s_\alpha \in s$, $s \in S$, i.e. a conjunction of atoms which are true. If we decompose $\wedge_\alpha s_\alpha$ back into a set s of elements of the conjunction, we then need to find the set of subsets of S that the set s is a subset of. If we then obtain the conjunction of each of these individual sets, we obtain the set of states at which the atomic proposition is true. Formally,

$$h(p) = \left\{ \wedge_{j=1}^{|s'|} s_j : s_j \in s', s \subseteq s', s' \in S \right\}, \tag{3.15}$$

where $p = \wedge_{i=1}^{|s|} s_i, s_i \in s$.

This concludes the translation from the planning problem P to an interpreted system IS_P.

3.2.9 Conversion from planning problem to corresponding Kripke model

We can construct the Kripke model $\mathcal{M}_P = \langle W, R, h \rangle$ in the following way:

1. We define the set of worlds W as the set of conjunctions of elements of 2^F. Namely,

   $$W = \left\{ \wedge_{i=1}^{|s|} s_i : s_i \in s, s \in 2^F \right\}, \tag{3.16}$$

   where 2^F is the set of all subsets of fluents defined in expression 3.3.

2. Define the set of initial states W_0 to be the singleton set containing the conjunction of all fluents required to be true in the initial state of P.

3. Define the function *compose*(w), which returns the conjunction of a set of states w and the function *decompose*(w), which returns the set of states that w is a conjunction of. Then,

   $R \subseteq W \times W$ is such that

   $$R\left(w, compose(w')\right) \text{ iff } \exists a \in A : w' = R_P(decompose(w), a), \tag{3.17}$$

   where:

   - $a \in A$ is a grounded action,
   - R_P is the transition relation for the planning problem P.

4. We now must define the valuation function $h : W \mapsto 2^{\mathcal{P}}$ which assigns to each world a set of atomic propositions which are true at that world. For the valuation function $h : W \mapsto 2^{\mathcal{P}}$, we exploit the fact that worlds are expressed as conjunctions of fluents. We can then decompose the fluents that make up the world w as a set s. We can then compute the set of subsets of the set s. The conjunction of each element of this set will be the set of atomic propositions true at the world w. Formally,
$$h(w) = \left\{ \wedge_{i=1}^{|s'|} s_i : s_i \in s', s' \in 2^s \right\},$$
where s is such that $w = \wedge_{i=1}^{|s|} w_i, w_i \in s$.

3.2.10 Conversion from IS to Kripke model

We will now attempt to convert the interpreted system IS_P into a Kripke model $\mathcal{M}_{IS_P}$.

Let $\mathcal{M}_{IS_P} = \langle W, R, h \rangle$ be such that

- $W = L$, the set of all possible global states in IS.
- W_0 be a singleton set containing the conjunction of fluents representing the initial state of P.
- $R \subseteq W \times W$ such that
$$R(w, w') \text{ iff } \exists a_i \in Act_i : w' = t(w, a_i),$$
where t is the evolution function as defined in 3.14 and a_i is a grounded action defined for the agent i. Note that the transition will only be defined if the protocol function P for IS defined in 3.12 has enabled the corresponding action.
- We define the valuation function $h : W \mapsto 2^{\mathcal{P}}$, in a similar way to that of $\mathcal{M}_P$, as we can still exploit the fact that worlds are expressed as conjunctions of fluents.

  We therefore have also that
$$h(w) = \left\{ \wedge_{i=1}^{|s'|} s_i : s_i \in s', s' \in 2^s \right\},$$
  where s is such that $w = \wedge_{i=1}^{|s|} w_i, w_i \in s$.

This concludes our translations to a planning problem P and interpreted system IS and the construction of the corresponding Kripke models $\mathcal{M}_P$ and $\mathcal{M}_{IS_P}$.

3.3 Correctness of Translation

We must now prove the correctness of the translation by showing that we can obtain a bisimulation between the interpreted system $\mathcal{M}_{IS}$ and the planning model $\mathcal{M}_P$. We wish to show that $\mathcal{M}_{IS}$ can simulate $\mathcal{M}_P$ and vice-versa.

Definition 3.2 (Bisimulation)**.** *Let $\mathcal{M} = (W, R, h)$ and $\mathcal{M}' = (W', R', h')$ be Kripke models. Let $t \in W$, $t' \in W'$. A **bisimulation** between $(\mathcal{M}, t)$ and $(\mathcal{M}', t')$ is a relation $B \subseteq W \times W'$ satisfying:*

- *$B(t, t')$,*

and for every $u \in W$ and $u' \in W'$ such that $B(u, u')$:

- *For all atoms p: $\mathcal{M}, u \models p$ iff $\mathcal{M}', u' \models p$.*
- ***forth**: If $v \in W$ and $R(u, v)$, then there is a $v' \in W'$ with $R'(u', v')$ and $B(v, v')$*
- ***back**: If $v' \in W'$ and $R'(u', v')$, then there is a $v \in W$ with $R(u, v)$ and $B(v, v')$*

We can now define what it means for two models to be *bisimular*.

Definition 3.3 (Bisimular)**.** *We say $(\mathcal{M}, t)$ and $(\mathcal{M}, t')$ are **bisimular** if there exists a bisimulation between $(\mathcal{M}, t)$ and $(\mathcal{M}', t')$.*

Remark. *Let:*

- *W_P be the set of all possible states of the Kripke model $\mathcal{M}_P$.*
- *W_{IS} be the set of all possible states of the Kripke model $\mathcal{M}_{IS}$.*
- *I_P be the set of initial states of $\mathcal{M}_P$.*
- *I_{IS_P} be the set of initial states of $\mathcal{M}_{IS_P}$.*
- *$w_P \in W_P$. $w_{IS} \in W_{IS}$.*
- *R_P and R_{IS_P} be the transition functions of $\mathcal{M}_P$ and $\mathcal{M}_{IS_P}$ respectively.*
- *h_P be the valuation function of $\mathcal{M}_P$.*
- *h_{IS} be the valuation function of $\mathcal{M}_{IS}$.*

the relation $B \subseteq W_P \times W_{IS}$ between $(\mathcal{M}_P, w_P)$ and $(\mathcal{M}_{IS}, w_{IS})$ is a bisimulation.

In order to prove this, we wish to show that the following properties hold for an arbitrary $w_P \in W_P$ and $w_{IS} \in W_{IS}$ if $B(w_P, w_{IS})$:

1. All initial states of $\mathcal{M}_P$ and $\mathcal{M}_{IS}$ can be related by B,
2. Assuming $B(w_P, w_{IS})$,
   - For an arbitrary atomic proposition p, $\mathcal{M}_P, w_P \models p$ iff $\mathcal{M}_{IS}, w_{IS} \models p$
   - **forth**: If $w'_P \in W_P$ and $R_P(w_P, w'_P)$, then there is a $w'_{IS} \in W_{IS}$ with $R_{IS}(w_{IS}, w'_{IS})$ and $B(w'_P, w'_{IS})$
   - **back**: If $w'_{IS} \in W_{IS}$ and $R_{IS}(w_{IS}, w'_{IS})$, then there is a $w'_P \in W_P$ with $R_P(w_P, w'_P)$ and $B(w'_P, w'_{IS})$.

Proof. (1). We have defined the initial states of $\mathcal{M}_P$ as I_P, the singleton set containing the conjunction of fluents which are true in the initial state of the planning problem.

We have defined the initial states of $\mathcal{M}_{IS}$ as I_{IS_P}, the singleton set containing a local state of the interpreted system IS representing the initial state of the planning problem.

I_P and I_{IS} are clearly isomorphic, as I_{IS}, a set containing a local state of IS, is simply a conjunction of the grounded fluents representing the initial state of P, which by definition is I_P. One can therefore define a bijection between elements of I_P and I_{IS_P}. This bijection will act as the relation B. □

(2). Assume $B(w_P, w_{IS_P})$ for arbitrary worlds $w_P \in W_P, w_{IS_P} \in W_{IS_P}$.

Let p be an arbitrary atom in F, the set of fluents as previously defined.

The functions h_P and h_{IS} are isomorphic as worlds are identically defined in both $\mathcal{M}_P$ and $\mathcal{M}_{IS_P}$, and through their respective valuation functions, h_P and h_{IS}, they will produce the same set of atomic propositions for worlds w_P, w_{IS_P}. Therefore, $h_P \simeq h_{IS}$, and so $p \in h_P(w_P)$ iff $p \in h_{IS}(w_{IS_P})$ and so $\mathcal{M}_P, w_P \models p$ iff $\mathcal{M}_{IS}, w_{IS} \models p$.

For some state $w_P \in W_P$, $w_{IS} \in W_{IS}$, we have that

- $R_P(w_P, compose(w'_P))$ iff $\exists a \in A : w'_P = R(decompose(w_P), a)$, where R is the transition function defined for the planning problem P, a is an action in the set of actions Act, the functions *compose* and *decompose* are as previously defined.
- $R_{IS}(w_{IS}, w'_{IS})$ iff $\exists a_i \in Act_i : w'_{IS} = t(w_{IS}, a_i)$, where a_i is appropriately enabled by the protocol function of the IS, and a_i is an action of the i-th action-performing agent of the IS.

We can therefore define the sets $next(w_P)$ and $next(w_{IS})$ corresponding to the next states that w_P and w_{IS} are respectively related to. Specifically,

- $next(w_P) = \{w'_P : R_P(w_P, w'_P), w'_P \in W_P\}$
- $next(w_{IS}) = \{w'_{IS} : R_{IS}(w_{IS}, w'_{IS}), w'_{IS} \in W_{IS}\}$

By construction, R_P will give a successor state only if the set of positive preconditions required for a given action are contained in the set of fluents representing the current state. Similarly, R_{IS} will produce a successor state only if an action can be found which is enabled by the protocol function.

Observe that the protocol function enables actions based on the preconditions which hold in a given state of the environment agent, which occurs in an identical fashion to R_P. This means that we can construct a bijection between $next(w_P)$ and $next(w_{IS})$.

We can therefore draw a relation B between all states of $next(w_P)$ and $next(w_{IS})$. This proves the back and forth properties required for a bijection and hence concludes the proof. □

□

3.3.1 Corollary

A corollary of the above proof is that any formula which holds true in the interpreted system will also hold true for the interpreted system IS_P. This is of great relevance for attempting to specify reachability goals. If we want to find a plan to ensure that the goal is eventually, we simply need to check if the formula $F\,goal$ is satisfied in the model IS_P. This can be verified by the use of model checking.

Due to the bisimular property, if the formula holds in IS_P, the formula must hold in the original planning problem P. We can then find a witness which achieves the truth of the formula, which will correspond to a plan in the planning problem P.

A crucial benefit of this is that we can now specify formulas of any degree of complexity using CTL or CTL* to represent temporally-extended goals. By the bisimular property, again, if the formula holds in IS_P, it must also hold in P. This gives a lot more power of the goals that we can find plans for without substantial additional cost.

It is also possible to generalise the proof to multiple agents, which will allow us to specify properties involving coalitions of agents. There has however not been enough time to formalise this within the constraints of this project.

3.4 Summary

In this chapter, we discussed the formalisation of the correspondence between a planning problem and an interpreted system, which can be input into MCMAS to find a plan via finding the existence of a witness. This is true because any formula true in the interpreted system will be true in the planning problem. We discussed this in terms of two of the ex ample domains, Dinner and Gripper.

In the next chapter, we will discuss how the theoretical results can be applied on the machine level in the physical implementation of the compiler.

Chapter 4

Implementation

In this chapter, we will discuss the implementation details of the PDDL–ISPL compiler. The operation of the compiler, from a given PDDL file to an ISPL file is completed from 3 main stages:

1. **Parsing**: The PDDL is parsed and tokenised, supporting a subset of the syntax of the PDDL 1.0 and some NPDDL extensions, as mentioned in Section 2.16.4. This takes care of both the lexer and syntax analysis stages, where an error will be thrown in the case of any violation of the standard PDDL syntax.

2. **Semantic analysis**: After the domain and problem are tokenised, semantic analysis is then performed to obtain meaning from the tokens.

3. **Intermediate code generation**: Code is generated internally to the compiler, where 'candidates' are created with the potential to be added to the final output.

4. **Code generation**: Code is finally outputted by the compiler, ready to be ran with MCMAS. An debug mode is also available to the compiler, where the user can instead see a representation of symbol table after the parsing stage instead.

It is known that the library Python-Lex-Yacc (PLY) is an implementation of the lex and yacc parsing tools for Python, used normally for the construction of compilers. It was initially considered to used it, but since the PDDL syntax is relatively simple, we thought the overhead of using any other external libraries were not worth the overhead. If the PDDL language were however to be extended in drastic ways in the future, it may become more necessary to use an existing parsing library rather than build it from scratch.

Instead of using PLY, the compiler is written using the open source `pddl-parser` written by Felipe Meneguzzi, an Artificial Intelligence researcher at the Pontifical Catholic University of Rio Grande do Sul (PUCRS). The parser initially had basic tools for parsing domains and problems written with the PDDL 1.0 syntax. A basic propositional planner was also included in the original project.

Several extensions however had to be made to allow for the more expressive syntax with the first change being made in order to parse predicates. The parser is now enabled to parse types, fluents and extended goals.

Using a combination of Java and ANother Tool for Language Recognition (ANTLR) was also considered, but the trade-off of time taken to set up ANTLR with an already structured language did not seem worth the time. Having a compiler built and working with as little external dependencies as possible made more logical sense. In the future, it would probably better to use a language with static types, as there is much potential for the compiler to grow in size drastically as further extensions are made to PDDL.

4.1 Architecture

The architecture of the compiler can be seen in Fig. 4.1. The domain and problem definition files are provided as input arguments to the compiler. The files are then parsed and a symbol table is built. If the PDDL file contained any temporally-extended goals, they are parsed and converted internally into a format recognisable by MCMAS. The final ISPL code is lastly created in the code generation stage.

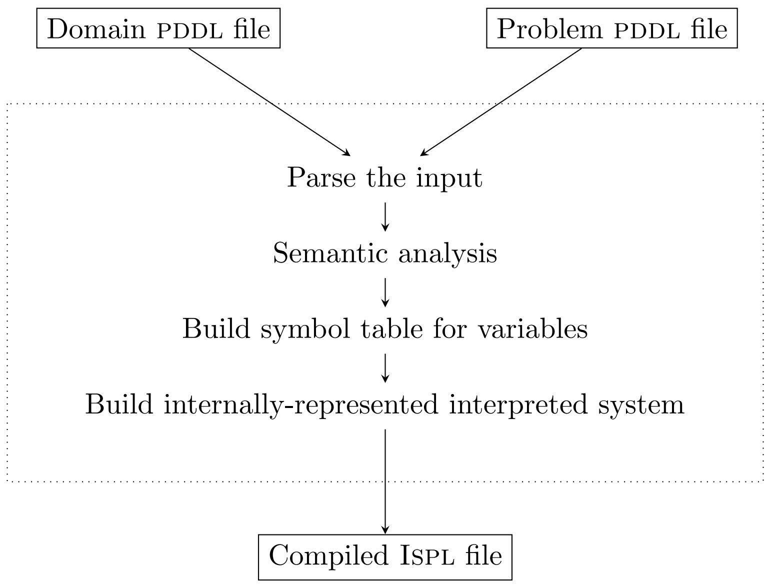

Figure 4.1: Architectural overview of MCMAS.

4.2 Parser

The syntax of PDDL is fortunately rather simple, all constructs are surrounded by well-balanced parentheses with no exceptions. This means that we can tokenise the entire syntax using a simple regex and use a standard parsing algorithm to start building the structure of the syntax tree representing the PDDL-specified domain and problem.

Algorithm 1: Generation of a syntax tree given a tokenised PDDL file.

```
Result: A syntax tree of a tokenized PDDL file, represented as a nested list.
stack = [];
list = [];
for t in tokens do
    if t == '(' then
        stack.append(list);
        list = [];
    end
    else if t == ')' then
        if stack is not empty then
            l = list;
            list = stack.pop();
            list.append(l);
        end
        else
            raise Exception('missing open parentheses');
        end
    end
    else
        list.append(t);
    end
end
if stack is not empty then
    raise Exception('Missing close parentheses');
end
if list has a length not equal to 1 then
    raise Exception('Malformed expression')
end
```

4.3 Code Generator

The `code_generator` is responsible for generating all ISPL code, storing the lines as a simple list of strings. All code is printed using the following simple procedure:

Algorithm 2: Print all lines of a fully compiled ISPL file.

```
code_generator = generate_code();
for line in code_generator do
    print line;
end
```

The `generate_code()` function is defined as in the following algorithm:

Algorithm 3: Build the symbol table and add symbolic representation of all components of interpreted system.

```
initialise_variable_map();
prepare_actions();
add_environment_agent();
for agent in action_performing_agents do
    add_action_performing_agent(agent);
end
add_evaluation(environment);
add_initial_states(environment);
add_groups(action_performing_agents);
add_fairness_conditions();
add_formulae();
```

Here we see that we firstly initialise a map of variables acting as the symbol table to be used throughout all stages of compilation, prepare the actions, and add an environment agent. After this, we loop through each agent specified in the planning domain, and add *action-performing* agents to the corresponding interpreted system. We denote agents that are different to the environment to be *action-performing*, as we assume that the environment will not be required to perform any actions. After this, we can then add the evaluation, initial states, any groups of agents, fairness conditions, and any formulae needed to express reachability goals.

The `prepare_actions()` function does the bulk of the work for this function, as it also initialises several other data structures to be used in the interpreted system's protocol and evolution functions.

The function loops through each action specified in the planning domain and performs the following algorithm if typing is not being used:

Algorithm 4: Construct a mapping from each parameter to a unique integer and call a function to get preconditions and effects.

```
combinations = permutations(objects, action.parameters.length());
parameter_map = dict();
for i = 0 to action.parameters.length() do
    parameter_map[action.parameters[i]] = i;
end
get_preconditions_and_effects(action, combinations, parameter_map);
```

Here, `permutations(objects, n)` is a function taking a collection of `objects` and an integer n and returns a list of permutations of `objects` of size n. In our implementation we do not store the list directly in memory, due to its potential to become very large. We instead create an iterator which yields each value only when required. The reason for the function is to ground each action in order to find every combination of parameters possible given the objects present in the domain.

The `get_preconditions_and_effects()` is defined in the following way:

Algorithm 5: Algorithm for populating the **effects** and **combinations** hash maps. Note: **action** is a PDDL action of a domain, **combination** is an element of a list of permutations, **param_map** is a hash map mapping from a parameter to a unique integer.

```
for (i, combination) in enumerate(combinations) do
    candidates = set();
    negatives = set();
    positives = set();
    get_all_candidates(action, candidates, combination, parameter_map,
      negatives, positives);
    if negatives == positives then
        continue;
    end
    next_combination = ‘ and ’.join(candidate + ‘=true’ for candidate in
      candidates);
    with_effects = [p + ‘=true’ for p in positives] + [n + ‘=false’ for n in
      negatives];
    next_effect = ‘ and ’.join(with_effects);
    action_name = ‘_’.join((action.name,) + comb);
    effects[action_name] = next_effect;
    combinations[action_name] = next_combination;
end
```

Dealing With Typed Predicates and Actions

As discussed previously, using types in planning problems leads to significant reductions in the time taken to produce a plan for a given planning problem. In order to generate the code for typed predicates, we follow the following algorithm:

1. Store a hash map, mapping from the name of the predicate to another map, mapping each argument to the corresponding type.
2. Using another hash map used to store the types of existing objects and constants in the planning domain, we loop through the argument's types, and produce a list for each argument, containing possible objects in the domain which have the same type as the current argument.
3. Compute the Cartesian product of the lists, giving an ‘argument list’ containing grounded arguments, but with the correct type.

4. Iterate through the argument list and populate the `Vars` section of the interpreted system as before. Observe that we will be required to create far less `Vars`, since we do not redundantly use every object in the planning domain. We only generate a new variable if the variable did not already exist in the variable map. We also initialise the variable map with a `False` value by default, as we do not consider the possibility of it being true in the initial state yet.

5. For each grounded predicate generated, produce a physical line in the ISPL file in the form `grounded_predicate_name : boolean;`.

Taking the Gripper domain as an example, the map of predicate names to types is of the following form:

```
(at_robby: {'?r': 'room'}),
(at: {'?b': 'ball', '?r': 'room'}),
(free: {'?g': 'gripper'}),
(carry: {'?o': 'ball', '?g': 'gripper'})
```

For the `at_robby` predicate, we know that the only objects with the type `room` are `rooma` and `roomb`. For this predicate we therefore only generate the singleton set of a set containing the two rooms, namely

$$\{\{\texttt{rooma}, \texttt{roomb}\}\}\,.$$

When the argument list contains multiple arguments of different types, we arrive in cases such as the in `at` predicate, where we have the set

$$\{\{\texttt{ball1}, \texttt{ball2}, \texttt{ball3}, \texttt{ball4}\}, \{\texttt{rooma}, \texttt{roomb}\}\}\,.$$

The Cartesian product of the elements of the set will be of the form

$$\{(\texttt{ball1}, \texttt{rooma}), (\texttt{ball1}, \texttt{roomb}), (\texttt{ball2}, \texttt{rooma}) \ldots\},$$

which creates the grounded arguments for the `at` predicate.

We also use a similar method to pre-process actions in the planning domain to create an internal representation of the possible grounded actions. This representation is then used to generate code in the same way as for untyped actions, but again, we have less internal actions being generated, leading to a reduction in the size of the final ISPL file.

4.4 Plan-printing in MCMAS

We can print the plan simply by reading from the witness generated from ECTLK formulae, and can output each action executed in the witness in the following way:

```
void
print_plan(bdd_parameters * para, vector< vector< transition * >*> *cextr)
{
  cout << "Found Plan:" << endl;
  for (unsigned int ac = 0; ac < (int) cextr->size(); ac++) {
    for (unsigned int j = 0; j < (int) cextr->at(ac)->size(); j++) {
      cextr->at(ac)->at(j)->print_plan(para->a);
    }
  }
}
```

Here, we are iterating through a witness generated in MCMAS. The j variable is used to iterate through each action-performing agent. We then print the actions simultaneously performed by each agent from the initial state to the goal. The option of printing the plan is toggled though a command-line argument. The print_plan method is defined as follows:

```
void
transition::print_plan(vector<BDD> *a)
{
  for (unsigned int i = 0; i < agents->size(); i++) {
    cout << " " << (*agents)[i]->action_to_str(*label, *a);
  }
  cout << endl;
}
```

4.5 Summary

In this chapter we discussed the implementation of the compiler. We discussed the methods taken to build the parser, and the components of the code generator. We also looked into how typed variables were handled and the difference this makes in the number of ISPL variables generated for the corresponding interpreted system.

We also discussed the way in which plans are printed in MCMAS, highlighting the limitations our method has for formulae other than those contained in ECTL*.

Chapter 5

Planning for Temporally-Extended Goals

In this chapter, we will examine the extensions to classical planning that we support with the compiler. This includes temporally-extended goals specified in CTL and CTL*, allowing us to find more complex plans for the Dinner and Gripper planning domains.

We also look into using types in planning domains to reduce the state-space explosion seen in untyped domains, as well as non-determinism in the initial state of a planning problem and in the effects of actions.

Along with this, We examine how fluents are represented in MCMAS, and also the possibility of planning under incomplete information.

5.1 Using CTL for Goal Specification

Goals for planning problems are traditionally viewed as a set of desirable final states being reached. The plan returned by a planner is correct if and only if it can translate the current state to one satisfying the desirable states.

As discussed in the background section, we aim to view goals as a desirable *sequence* of states instead. Here, the plan is correct if its execution yields one of the desirable sequences. We will denote a *classical goal* to represent the standard definition of a goal, i.e. having a plan to solely reach some final state.

5.1.1 Using the Gripper Domain

Recall the description of the Gripper domain from Section 3.2. Initially, we wanted to only satisfy the condition that the Gripper could move balls from one room to another, regardless of the means in which the left and right grippers coordinated to achieve this goal.

Using CTL we can now express the condition that that goal is reached within n steps, which can be expressed in the following way:

$$\underbrace{EX\,(EX\,(\ldots(goal)\ldots))}_{n\,EX's}.$$

Another property which can be specified in CTL is "Eventually reach the goal and permanently have the right gripper free thereafter". From a practical point of view, this can be useful if in general a property is desired to remain true after the goal is reached, for example, a notification signal. This is expressed in CTL as:

$$EF\,(goal) \wedge EG\,(free(right)).$$

5.2 Using CTL* for Goal Specification

We are now in a position to use the model checker MCMAS-DYNAMIC, which was proposed in [27] to plan for temporally-extended goals using the logic CTL*. This allows plans to be generated which cannot be expressed in the language CTL or LTL alone.

5.2.1 Using the Dinner domain

Observe that we can use MCMAS* to check the satisfaction of formulae of the form $EFG\,p$ for some model, which is not CTL or LTL-expressible. This means that we can now produce a plan which satisfies EFG `goal`, meaning "Find a way to eventually achieve `goal` infinitely often", where `goal` is true if `dinner` $\wedge$ `present` $\wedge\ \neg$`garbage` is true for some state in the planning domain.

5.2.2 Using the Gripper domain

We use the Gripper domain and aim to specify properties about the execution of plans in this domain. For this domain, let `goal` be the classical conjunction of predicates

`at(ball1, roomb)` $\wedge$ `at(ball2, roomb)` $\wedge$ `at(ball3, roomb)` $\wedge$ `at(ball4, roomb)`,

and let `free(right)` denote a predicate representing the truth of the robot's right gripper being free.

We find that it is now possible to express properties such as:

1. $E\,(GF\,$`free(right)`$) \rightarrow G(E(F(G\,$`goal`$))))$ – Find a plan where if the right gripper is permanently free, it is always eventually possible for the goal to be permanently true.

2. $E\,(GF\,\texttt{goal}) \wedge (G\,\texttt{free(right)})))$ – Infinitely often reach the goal without ever using the right gripper.

3. $E(F\,\texttt{goal}) \wedge (G\,\texttt{free(right)})$ – Find a way to eventually achieve the goal and never use the right gripper.

4. $E(G(F(\texttt{goal} \wedge (E(X\neg\texttt{goal})))))$ – Whenever I achieve the goal, I can find a way to negate it after executing the next action.

All of these goals produce a witness in MCMAS-DYNAMIC, meaning that a plan can be generated. In general, any formula contained in ECTL* should produce a witness and formula in ACTL* will not find a witness, but can the prove the non-existence of a plan in the case that one does not exist. By construction, We are able provide support for the arbitrary nesting of temporal operators for CTL and CTL* formulae.

5.3 Types

Observe that predicates and all arguments of actions specified in a planning domain can take on any arbitrary value. This means that during the grounding process, we have no choice but to have each possible argument value as a potential candidate. This will happen regardless of if the grounded action or predicate can every be reached from the initial state of the planning problem.

One method of reducing the combinatorial explosion caused by this is by allowing the arguments predicates and the arguments of actions to have types. We also observe that starting from PDDL 2.1, typing is a compulsory requirement, so it is necessary to have types supported by the compiler.

Including types requires that the user specifies `:typing` in the `:requirements` section of the domain. We can now move to a typed version of the Gripper domain, where:

- `room`, `ball` and `gripper` are `:types`
- `rooma` and `roomb` are of type `room`
- `ball1`, `ball2`, `ball3` and `ball4` are of type `ball`
- `left` and `right` are of type `gripper`

When using the typed domain compared to the untyped domain, we find that the code generated by the compiler reduces from $\sim$ 1800 lines of ISPL code to 150 lines, and the time taken for model checking reduces from 60 seconds to 0.25 seconds, which is a very significant reduction.

We also observe that we still have the same number of reachable states in both models when input into MCMAS, but the typed version uses about 1/4 of the memory that the untyped version uses. This highlights the crucial importance of reducing the state space of the domain. The experimental results of the difference between solving planning problems in typed and untyped domains can be found in Section 6.1.1.

5.4 Non-Determinism

5.4.1 Uncertainty in the Initial State

We can encode uncertainty in the initial state of the domain by using the `oneof` keyword introduced in NPDDL. For example, we may want to find a way to achieve the same goal for the Gripper domain, but with one of the balls non-deterministically being in room *a* or room *b* and the robot non-deterministically starting either room. In PDDL, this is specified by:

```
(:init (oneof (at-robby rooma) (at-robby roomb))
       ..
       (oneof (at ball4 rooma) (at ball4 roomb))
       ..
)
```

In general, for each `oneof` $(p_1, p_2, \ldots, p_n)$ statement in PDDL, we generate the following code in ISPL:

$$\bigvee_{i=1}^{n} \left(p_i\texttt{=true} \wedge \left(\bigwedge_{j \neq i} p_j\texttt{=false} \right) \right), \tag{5.1}$$

which takes the following form in MCMAS:

```
InitStates
    (p1=true and p2=false and p3=false and ... and pn=false) or
    (p2=true and p1=false and p3=false and ... and pn=false) or


    ...
    (pn=true and p1=false and p2=false and ... and p_n-1=false);
end InitStates
```

MCMAS will now non-deterministically choose one of the values contained in the set $\{p_1, p_2 \ldots, p_n\}$ to be true for the initial state.

Although it is possible to check for the existence (or non-existence) of a plan, the plan itself is currently not generated. The reason for this is because MCMAS does not generate a witness in such a non-deterministic case. Adding the functionality to also produce the plan in non-deterministic situations will also be a topic for further work.

5.4.2 Conditional Effects

We also provide support for planning with conditional effects being specified in the planning domain. We may want to have an action having a specific effect only if a certain condition holds, along with a precondition. An example of this is in the `robot_navigation` domain, where we see that the precondition is a disjunction of clauses, and the `effects` section specifies the specific state change that will occur when a given disjunct holds.

The way in which this is implemented is by creating a new action for each specific case of the conditional, and append the condition to the action's initial precondition.

5.5 Fluents

The specification of fluents are supported. They are implemented by creating an enumeration of values in the `Vars` section of an ISPL file, given that an appropriate range of values are given in the original PDDL specification.

5.5.1 Durative Actions

Durative actions and the solving of minimisation problems during the execution of a plan are currently not supported in this project. It is known that linear programming methods along with the use of fluents should be used.

5.6 Planning Under Incomplete Information

One of the most realistic uses for solving planning problems is planning with incomplete information. That is, having action-performing agents possessing only partial observability of the domain.

Referring to an example described in [9, 10], one such domain is a robot in a maze with the goal of moving from one specific room of the maze to another.

The robot only has information regarding whether there is a wall in front, behind, to the left, or to the right of itself, but it does not know its precise position. We can think of this information being communicated to the robot thought the use of sensors.

The example is illustrated in Figure 5.1. There are four rooms, labelled s_0, s_1, s_2 and s_3. It is possible to move between two connected rooms. s_0 is connected with s_1, s_0 with s_2, and s_2 with s_3. However, if the robot moves away from s_0, due to an oil spillage, there is a chance that the robot will slip and end up in s_2. Due to the robot's limited sensing, it is not able to distinguish between states s_1 and s_3.

We see an approach being taken to plan under partial observability by classical replanning in [11]. They propose a method to convert the partially observable planning

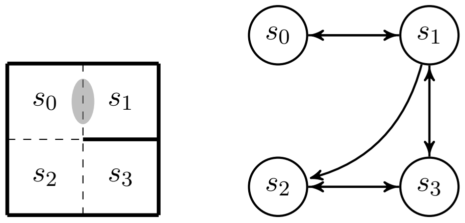

Figure 5.1: The domain as described in [9, 10], with the corresponding state transition diagram.

problem P into a fully-observable problem obtained by the translation $K'(P)$, with the restriction that the problem P is *simple* and *solvable*. A planning problem P is *solvable* means that the problem can be solved by a classical planner. They define the problem to be *simple* using the following definition:

Definition 5.1. *A (deterministic) partially-observable problem $P = \langle F, O, I, G, M \rangle$ is* ***simple*** *if the non-unary clauses in the initial condition I are all invariant, and no hidden fluents appear in the body of a conditional effect.*

An example of a non-unary clause is one which contains uncertainty, for example, assigning $\mathit{oneof}(x_1, \ldots, x_n)$ to a variable.

An *invariant* in a planning problem is a formula which remains true during the execution of a plan. An example of this, also as given in [11], is a robot which can move an object to n different locations. Invariant are in this case formulas such as $at(o, l_1) \vee \cdots \vee at(o, l_n) \vee \mathit{hold}(o)$. The reason for this is because the object can be at any of the n locations or the robot is currently holding the object.

From [11], we also see that the observable variables should have the ability to update automatically without regards to the action being executed. This means that there is not a direct translation into a bisimular ISPL specification, since the next state is a function of the previous state and the last action being executed, and the next action is a function of the current state. This subtle semantic difference explains the difference in expressing similar properties in ISPL.

An alternative method is to have the action-performing agent execute an `observe` action, which will sense the current state and subsequently update the observable variables of the agent. This however again has different semantics to what is originally proposed in the literature.

5.7 Summary

In this chapter we discussed the way in which temporally extended goals can be encoded using CTL* formula with interpreted systems. We discussed how the goals are encoded in both the Dinner and Gripper domains. We also discussed how types are handled in the compiler and the drastic reduction in the state space generated and consequently, the reduction in the time taken to find a plan.

We see that there are many strengths gained from planning by model-checking with MCMAS, in the sense that expressing complex temporally-extended goals in CTL* are now possible. In the evaluation chapter, we see that the increased expressivity does not come at a significant cost.

Along with the gains in expressivity of goals compared to that of classical planning problems, we also observe that the ability of planning for multiple agents should be feasible, as MCMAS allows for the specification of formulae involving coalitions of agents. Planning for multiple agents have however not been included in this project due to time constraints.

Chapter 6

Evaluation

In this chapter, we discuss several ways in which we evaluated the compiler. We firstly examined the way in which we benchmarked classical domains, and then looked into the running time taken to find the plan for temporally-extended goals. We also discussed our testing methods for the correctness of the compiler.

6.1 Experimental Evaluation

6.1.1 Classical Domains

For the classical typed planing domains, we can conduct an experimental evaluation when scaling the domains. For the Gripper planning problem, one method of scaling is by increasing the number of balls the robot is required to move from room a to room b. This also defines the name of the planning problem. In other words, planning problem i corresponds to moving $2i + 2$ balls from room a to room b. The results can be seen in Table 6.1, where we examine the running time for each planning problem, as well as the BDD memory being taken up while model checking is being conducted.

Problem #	Time taken (secs)	# of balls (N)	BDD memory (bytes)
1	0.231	4	9685632
2	0.566	6	12154624
3	1.904	8	16258512
4	3.734	10	28356208
5	6.278	12	28144752
6	10.29	14	34523760
7	30.93	16	49814688
8	63.78	18	48676352

Table 6.1: Experimental results for the typed Gripper domain.

From the table, we see a clear exponential increase in the time taken for the plan to be found via model checking. The BDD memory being taken appears to increase linearly with respect to N, where N is the number of balls being used in the planning domain.

The reason for this is because for every n extra balls being added to the domain, the number of boolean variables declared in the `Vars` section of the action-performing agent will only grow by mn, where m is the number of predicates containing references to balls, for example, `at(ball2, rooma)`. The number of predicates m stays constant as n increases, so we indeed have a linear increase in the size of the state space. The time taken to compile each file is negligible, with all problems taking less than one tenth of a second to compile.

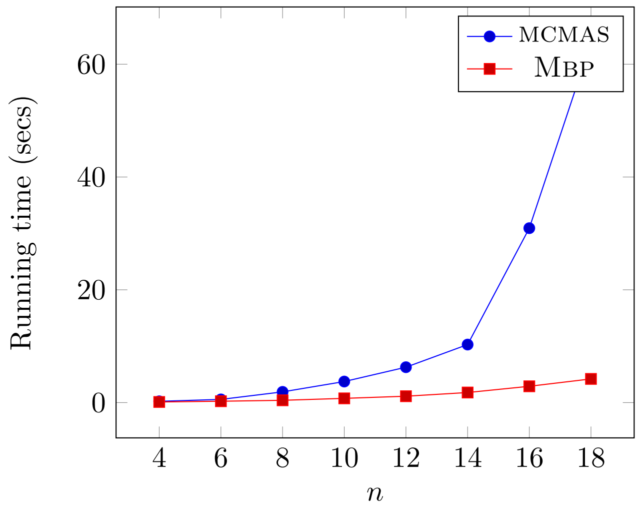

Figure 6.1: Comparison of running times for each planning problem.

6.1.2 Temporally-Extended Goals

We can also test the running time for planning for temporally-extended goals. We can use a sufficiently complicated goal, we will use the goal

$$E(G(F(\texttt{goal} \wedge (E(X\neg\texttt{goal}))))),$$

meaning "Whenever I achieve the goal, I can find a way to negate it after executing the next action."

Again, we can scale the problem by increasing the number of balls required to be moved by the robot. The results can be seen in Table 6.2.

Here, we see the same exponential increase in the time taken to find the existence of a plan, along with the linear increase in the size of the BDD memory taken in encode the domain. We note that having temporally-extended goals expressed in CTL does not lead to any major increases in the running time needed to compute a plan.

It is also possible to express the need to achieve a goal in precisely n steps. In this case, since we are allowed the arbitrary nesting of CTL operators, this can be expressed as

$$\underbrace{EX\,(EX\,(\ldots(goal)\ldots))}_{n\,EX's}.$$

Problem #	Time taken (secs)	# of balls (N)	BDD MEMORY (bytes)
1	0.255	4	11674048
2	0.538	6	16909440
3	1.964	8	19343792
4	4.035	10	33403536
5	7.201	12	38262416
6	11.479	14	43684048
7	28.523	16	58292960
8	61.108	18	57074752

Table 6.2: Experimental results for the typed Gripper domain with the formula $E(G(F(\texttt{goal} \wedge (E(X\neg\texttt{goal})))))$.

For the Dinner domain, as explained in section 3.1, we may want to delay the eventual reaching of the goal, rather than have it solved in the standard three steps. This leads to generated plans in the form seen in Fig. 6.3 for $n = 6$, where an enabled action is repeated a sufficient amount of times in order to achieve the goal in the number of steps given. If no possible actions are enabled, the plan will not exist.

6.1.3 Infinite Plans

It is also possible for infinite plans to be represented in MCMAS. Taking the Dinner domain again as an example, we may want to find a plan to achieve $EFG\, goal$, meaning "Find a way to eventually *maintain* the *goal*", which can be expressed in CTL*, but not CTL or LTL. When finding the plan in MCMAS, the goal is reached, and the final state is simply repeated infinitely often, representing the "G" part of the formula. The plan generated is of the form seen in Fig. 6.2.

We have also included in the appendix the plans generated for the CTL* formulae

$$E(F\, goal) \wedge (G\, \texttt{free(right)})$$

and

$$E(G(F(goal \wedge (E(X\neg goal)))))$$

for the Gripper domain.

$$\begin{tikzcd}
s_0 \arrow[r, "cook"] & s_1 \arrow[r, "cook"] & s_2 \arrow[r, "wrap"] & s_3 \arrow[r, "dolly"] & s_4 \arrow[r, bend left, "cook"] & s_5 \arrow[l, bend left, "cook"]
\end{tikzcd}$$

Figure 6.2: Infinite plan generated for the formula $EFG\, goal$.

$$\begin{tikzcd}
s_0 \arrow[r, "cook"] & s_1 \arrow[r, "cook"] & s_2 \arrow[r, "carry"] & s_3 \arrow[r, "wrap"] & s_4 \arrow[r, "wrap"] & s_5
\end{tikzcd}$$

Figure 6.3: Plan generated for achieving *goal* in 6 steps.

6.1.4 Large State Spaces

One example of a planning problem with a large state space is a domain named "Wumpus world" [40], which leads to a very large output ISPL file. The problem is designed to be a benchmark for the ability for planners to deal with partial observable domains. We incur poor performance even with full observability. The problem is stated as follows:

The world is a grid-based cave. Our hero wants to enter the cave, find some gold, and leave unscathed. The rules of the Wumpus world are as follows:

1. Unfortunately the cave contains a number of pits, which our hero can fall into and die.
2. The cave also contains the Wumpus, who will eat him.
3. The Wumpus is too fat to fall into a pit

Our hero can move around the cave at will and can *perceive* the following:

- In a position near the Wumpus, a *stench* is perceived.
- In a position near a pit, a *breeze* is perceived.
- In a position where the gold is located, a *glitter* is perceived.
- When trying to move into a wall, a *bump* is perceived.
- When killing a Wumpus, a *scream* is perceived.

The hero also has a single arrow, which can be used to kill the Wumpus. The usual benchmark used for this planning problem is to design an agent that can *usually* perform well regardless of the layout of the cave.

As a display of the extremely large state-space explosion, we obtain the following results when trying to find a plan to achieve the goal in MCMAS:

Execution time	No. reachable states	BDD memory in use
5514.6	126	115351568

Table 6.3: Performance results of the WUMPUS domain.

We see that it took 5514.6 seconds, or 1.53 hours and 115 MB of BDD memory to find the plan. This explicitly shows the difficulties involved in planning with untyped domains with large state spaces.

The plan generated by MCMAS is the following:

```
move_agent_sq_1_1_sq_1_2
move_agent_sq_1_2_sq_1_3
take_agent_the_gold_sq_1_3
move_agent_sq_1_3_sq_1_2
move_agent_sq_1_2_sq_1_1
```

n	Time	Reachable states	BDD memory
4	0.245	125	10427056
5	1.561	866	15726432
6	22.583	7057	31699184
7	687.964	65990	108441040
8	11071	695417	458544592

Table 6.4: Performance results for BLOCKS domain

6.1.5 Blocks World Domain

We will now evaluate the compiler for the Blocks World domain, one of the standard benchmark planning problems used for the International Planning Competitions. The domain consists of n 'blocks', which are initially situated on a table. The goal of the planning problem is to find a way to stack the blocks in a single tower where the blocks are arranged in a fixed order. The problems vary in the number n of blocks in the domain and the height of any existing stacks of blocks in the initial state.

For $n = 4$ blocks C, A, B, and D, with the goal to have them stacked in the order $ABCD$, reading from the bottom of the tower to the top, one possible plan (and the plan generated by our tool) is:

$$\mathit{pick\_up}(B),\ \mathit{stack}(B,A),\ \mathit{pick\_up}(C),\ \mathit{stack}(C,B),\ \mathit{pick\_up}(D),\ \mathit{stack}(D,C)$$

We will test our solution with the problem where no blocks are already stacked on any other blocks.

We again see an exponential blow-up in the time taken and the reachable states explored to produce a plan. The blowup in this case is far worse than the GRIPPER domain. The time taken for the $n = 8$ case is well over 3 hours; the time recorded by MCMAS undergoes a loss of precision when the time becomes excessively large. A correct plan was found nevertheless.

Another observation is that the lengths of the plans found are not a direct function of the number of blocks initially in the planning.

6.1.6 Multi-Agent Planning

Planning for multiple agents is a crucial area in the field of artificial intelligence. In order to specify planning domains with multiple agents, we will be using a subset of the MA-PDDL specification language, as discussed in Section 2.16.3.

Although it was possible to translate planning domains for a single agent into an equivalent interpreted system, the case for multiple agents is more complicated. We also note that there are several notions of planning for multiple agents. In order to test our solution, we use the BLOCKS WORLD domain. We take several approaches to this:

- **Planning for multiple agents as if there was only one agent**: In this case, we interpret an agent only as an additional parameter to actions and predicates. Since we specify the agents to have the type `agent`, we are able to ground our actions and predicates using the appropriate types for agents and other typed objects.

  In this case, we end up with a sequential plan, where each agent takes turns to perform an action which modifies the state of the environment. When running the example, the plan takes the following form:

  ```
  unstack_d_a_a2
  unstack_b_d_a3
  unstack_f_b_a4
  put_down_b_a4
  unstack_e_f_a1
  put_down_d_a3
  unstack_c_e_a3
  pick_up_c_a4
  stack_d_c_a4
  pick_up_b_a4
  stack_c_b_a4
  stack_b_a_a2
  stack_a_f_a1
  stack_f_e_a3
  ```

  We see that each agent collaborates in a sequential fashion in order to achieve the goal of having the blocks in the formation $DCBAFE$ (reading from the bottom of the stack to the top).

- **Solving the problem using only one agent**: We can also compare the results to that when attempting to solve the planning problem with only one agent. We see that the plan is noticeably longer than that for planning with multiple agents. Using more agents to collaborate can achieve a shorter plan to achieve the same goal.

- **Solving the planning problem using multiple physical IS agents, in a sequential fashion**: We can also encode each agent as a separate `Agent` in the compiled interpreted system. Ma-pddl allows us to specify both public and private predicated in the planning domain. We interpret the private predicates, which involve the use of a specific agent as local variables, and the global predicates as observable variables (`Obsvars`) in the environment agent of the interpreted system. The environment agent has its state changed as a function of the separate agents. When each agent performs an action, their own respective state is then changed.

  This solution is far more complex as the previous two, as it leads to problems such as race-conditions which may not be taken into account in the definition of the concrete planning problem specification. This may sometimes lead to undesirable behaviour during the execution of the plan.

For example, in the BLOCKS WORLD domain, it is not explicitly stated that one agent can be carrying an agent only if another agent is not carrying the same object. This is not a problem when planning as a single agent, but leads to race-conditions when using multiple physical agents.

Another problem is for each agent to perform exactly the same action as the other, since there may not be a mechanism defined in the planning problem to prevent agents from picking up an item while another agent picks up the same action. This leads to the plan below for the `4-0` configuration[1] of BLOCKS WORLD:

```
Found Plan:
  pick_up_b_a1  pick_up_b_a2
  stack_a_b_a1  stack_a_b_a2
  pick_up_c_a1  pick_up_c_a2
  stack_b_c_a1  stack_b_c_a2
  pick_up_d_a1  pick_up_d_a2
  stack_c_d_a1  stack_c_d_a2
----------------------------------------
done, 1 formulae successfully read and checked
execution time = 8.767
number of reachable states = 1163
BDD memory in use = 39800992
```

In this case, We see that the coalition is specified by the group of agents `g1`, where `g1` is a set containing the agents `a1`, `a2`, `a3` and `a4`.

- **Planning with multiple physical agents using the composition of actions**: This is by far the most complex case, with an extremely large number of observable variables being referred to in the evolution function of the environment agent.

  The reason for this is because we need to search for all possible state changes that can happen to the environment for all agents specified in the planning domain, and perform all state changes as a result of the concurrent execution of all actions performed by the respective agents.

  Along with a very large ISPL output file, the time taken to solve very simple planning problems becomes very unreasonable to be used practically.

- **Planning without explicit specification of a coalition of agents**: In this case, we simply specify CTL formula involving the desired goal, for example,

[1]This configuration refers to the problem of having four blocks initally placed on a table, with no block on top of another block, with the goal of having the blocks stacked in a single tower of some predefined ordering.

EF goal for reachability. This is without reference to any coalition of agents. In this case, we see behaviour involving the concurrent execution of actions of the agents which do not necessarily make direct progress to the goal, but eventually will reach it. An example of this is the following, for the `4-0` configuration of the domain:

```
pick_up_c_a1  pick_up_d_a2
stack_a_c_a1  stack_b_d_a2
pick_up_c_a1  unstack_b_d_a2
stack_c_c_a1  stack_c_d_a2
pick_up_b_a1  unstack_c_d_a2
stack_a_b_a1  stack_d_d_a2
unstack_a_b_a1  unstack_c_d_a2
stack_c_b_a1  put_down_d_a2
pick_up_c_a1  pick_up_d_a2
stack_b_c_a1  stack_d_d_a2
pick_up_d_a1  unstack_b_c_a2
stack_c_d_a1  stack_c_c_a2
```

In this case, we also see that the time taken to find a plan is relatively quick.

In general, we that it is sometimes better to plan for a goal using a single agent to plan for all agents specified in the planning domain. This may be because of a lack of suitable planning domains which make strong use of being able to execute actions concurrently, and also the definition of well-defined domains where race-conditions are taken into account.

We use a multi-agent version of the BLOCKS WORLD domain for our evaluation.

6.1.7 Uniform Strategies

We can also use MCMAS to find out the uniformity in the strategies used to reach a given goal. This can be done by passing the `uniform` flag to MCMAS for a given ISPL file.

The aim of this is to find out if it is possible for a formula to be true when uniform strategies versus the case when we do not. One domain used to test this out is the `robot-navigation` domain, which involves a robot in a maze with five rooms, the `store`, the `lab`, `ne-room`, `sw-room` and the `dep` room.

The robot has the aim of travelling from the `store` to the `dep`. Each of the rooms have a given number of doors open for the robot to freely travel through. The domain is normally used in literature as an example of a partially observable domain, but we will use it in order to demonstrate uniform strategies.

We may aim to deduce whether the formula *AF goal* can be achieved by the robot in the maze, meaning "it is always possible to reach the goal". When checking for uniform strategies, we find that the formula is true, but false when using non-uniform strategies.

We also find that there are 5 reachable states and 25 uniform protocols in general, which can be seen from the MCMAS output.

```
Building partial transition relation...
  processing protocol in agent Environment... done: 1 uniform protocols
  processing protocol in agent Performer... done: 24 uniform protocols
Building BDD for initial states...
Verifying formulae under uniform strategies...
  Strategy 1:  number of reachable states = 5
  Formula number 1: (AF goal), is TRUE in the model
done, 1 formulae successfully read and checked
execution time = 0.003
maximum number of reachable states = 5
number of uniform strategies = 1
BDD memory in use = 8972544
```

6.1.8 Performance Test Setup

For benchmarking the performance of MCMAS and the compiler, we used a "Graphics" machine in the Department of Computing with eight 3.70GHz Intel(R) Xeon(R) CPUs cores and 64GB of RAM, running Ubuntu 16.04.2 LTS (Xenial Xerus). The metrics for the time taken for model checking is independent of the time taken for compilation.

6.2 Testing

We will now discuss the various measures taken to determine the correctness of the compiler's output.

6.2.1 The Automatic Validation Tool VAL

We find that it is possible to test the correctness of plans using the VAL tool [22]. The tool is used as standard in International Planning Competitions to evaluate the plans being generated for planners using PDDL 2.1, and is also fundamental for the development of new planners.

We however did not use VAL for the purposes of the project, and instead compared the generated plans from MCMAS with the output of existing planners and making use of the cloud-based automated planning service offered at [1].

6.2.2 Differential Testing

We were able to provide a form of *differential testing* [35] to evaluate the correctness of our tool's output, which is a method frequently used to evaluate compilers. The way in which we approach this method of testing is by using input PDDL files representing the same planning problem. As discussed earlier, the Gripper domain can be expressed with or without using types. By inputting each problem into the compiler, we see that the plan generated achieves the same goal.

We note that this is not a scalable method of testing, as the generated plan needs be analysed by a human to check the correctness, since like a topological sort, some actions can appear in a different order, but the same goal can still be achieved. Another method of checking correctness is by encoding the plan in a way which can be executed as a finite state machine, and it can then be evaluated if executing the actions of the plan from the initial state can indeed achieve the desired goal.

The plan generated for both versions of the Gripper domain can be seen in Table 6.5.

Untyped	Typed
`pick_ball2_rooma_right`	`pick_ball3_rooma_right`
`pick_ball1_rooma_left`	`pick_ball1_rooma_left`
`move_rooma_roomb`	`move_rooma_roomb`
`drop_ball2_roomb_right`	`drop_ball1_roomb_right`
`drop_ball1_roomb_left`	`drop_ball3_roomb_left`
`move_roomb_rooma`	`move_roomb_rooma`
`pick_ball3_rooma_left`	`pick_ball2_rooma_left`
`pick_ball4_rooma_right`	`pick_ball4_rooma_right`
`move_rooma_roomb`	`move_rooma_roomb`
`drop_ball4_roomb_right`	`drop_ball4_roomb_right`
`drop_ball3_roomb_left`	`drop_ball2_roomb_left`

Table 6.5: Comparison of plans generated by typed and untyped Gripper domain definitions.

Problem	Execution time (secs)	Reachable states	BDD memory in use (bytes)
Typed	0.28	256	10594112
Untyped	60.516	256	40440096

Table 6.6: Performance comparison for typed and untyped Gripper domains

From Table 6.6, we observe that the number of reachable states stays constant between the problems, but we have a 99.5% decrease in the execution time, as well a 73% decrease in the BDD memory taken by the generated model.

By inspecting the generated plans, we see that both plans achieve the same goal. Balls are moved by the grippers in a different order, but this does not make a difference in the possibility of the goal being achieved.

6.2.3 Unit Testing

In order to test for the correctness of individual components compiler, we have introduced a test suite. It is possible to ensure that every component works as expected, since it was built in a modular fashion.

It is significantly more difficult to introduce an automated test suite for MCMAS. Due to the limited time constraints of the project and the small interaction the compiler has with the internals of MCMAS, we have decided to not write unit tests for it, but stick with acceptance tests instead to ensure that the code written to print the action from the witness is correct.

6.3 Summary

In general, we see good performance from the compiler for typed domains, but see problems occurring with untyped domains with large state spaces. The next chapter will focus on the usage of the compiler, with instructions for performing a compilation given a planning domain and problem specification.

Chapter 7

Usage

In this chapter, we will discuss the way in which a user can perform a compilation given a planning domain and problem specification written as PDDL files, and how the user can retrieve an equivalent interpreted system which can then be input into MCMAS, which will use model checking to find a plan.

7.1 Translation of Planning Problems to Interpreted Systems

The compilation of a PDDL domain and problem into an equivalent ISPL file can be done by running the `PDDL.py` script and specifying the domain and problem as command-line arguments. In general, the usage is as follows:

```
> python PDDL.py DOMAIN-NAME.pddl PROBLEM-NAME.pddl
```

Taking the Gripper domain as an example, we compile into an equivalent ISPL file by running the following:

```
> python PDDL.py problems/gripper/domain.pddl problems/gripper/problem.pddl
```

MCMAS will also provide information about the non-existence of a plan, if the given formula does not hold in the generated model.

7.2 Plan-Finding via MCMAS

We are now able to compile specific PDDL files into ISPL files.

7.2.1 Dinner domain

We will attempt to perform the compilation and model checking procedure for the Dinner domain.

We run MCMAS with the successfully compiled `dinner.ispl` using the following command:

```
> ./mcmas -c 4 dinner.ispl
```

We obtain the following output:

```
----------------------------------------
Found Plan:
  cook
  wrap
  dolly
----------------------------------------
done, 1 formulae successfully read and checked
execution time = 0.006
number of reachable states = 12
BDD memory in use = 8974672
```

Here, we see that one sequence of actions to be fired in order to achieve the goal are to cook dinner, wrap the gift, and then use the dolly to take out the trash.

7.2.2 Gripper domain

We will now attempt to obtain an ISPL file representing the Gripper planning problem.

We can run MCMAS with the compiled `gripper.ispl` file using the following command:

```
> ./mcmas -c 4 gripper.ispl
```

We obtain the following output, which displays the actions to be executed along with the necessary arguments to be used to achieve the goal of having all four balls being in room *b*:

```
----------------------------------------
Found Plan:
  pick_ball2_rooma_right
  pick_ball1_rooma_left
  move_rooma_roomb
  drop_ball2_roomb_right
  drop_ball1_roomb_left
  move_roomb_rooma
  pick_ball3_rooma_left
  pick_ball4_rooma_right
  move_rooma_roomb
  drop_ball4_roomb_right
  drop_ball3_roomb_left
----------------------------------------
done, 1 formulae successfully read and checked
execution time = 82.247
number of reachable states = 256
BDD memory in use = 38834032
```

Here, we see that the robot must pick up two balls from room a using its left and right grippers, move to room b, drop both balls and move back to room a. This sequence of actions is repeated for each remaining pair of balls left in room a, and the goal state is successfully reached.

The planner will also give feedback if it is not possible to plan for a desired goal. An example is the temporally-extended goal

$$E\left[(free(left) \wedge free(right))\ U\, goal\right],$$

where *goal* is the goal as defined previously, and $free(left)$ and $free(right)$ correspond to the left and right grippers respectively being free. The formula represents the property "Find a plan to reach the goal with using the left and right grippers". This property is clearly not true. Assuming that the compiled file was saved in `until-false.ispl` in a directory named `output`, we end up with the following output in MCMAS:

```
...
output/until-false.ispl has been parsed successfully.
Global syntax checking...
Done
Encoding BDD parameters...
Building partial transition relation...
Building BDD for initial states...
Building reachable state space...
Checking formulae...
Verifying properties...
  Formula number 1: E((free_left && free_right) U goal),
                    is FALSE in the model
done, 1 formulae successfully read and checked
execution time = 0.274
number of reachable states = 256
```

```
BDD memory in use = 9851360
```

Here, we see that a plan could not be found to achieve the goal without using any of the grippers.

7.2.3 Blocks World domain

We will now attempt the same procedure for the Blocks World domain.

Running the compiled ISPL file in MCMAS, we obtain the following output:

```
----------------------------------------
Found Plan:
  unstack_b_c
  put_down_b
  unstack_c_a
  put_down_c
  unstack_a_d
  stack_a_b
  pick_up_c
  stack_c_a
  pick_up_d
  stack_d_c
----------------------------------------
done, 1 formulae successfully read and checked
execution time = 0.254
number of reachable states = 125
BDD memory in use = 10791280
```

Chapter 8

Project Evaluation

In this chapter we will perform an evaluation of the strengths and weaknesses of both the theoretical results and the implementation of the solution.

8.1 Theory

8.1.1 Strengths

The theoretical aspects of the project had the following strengths:

- **Proof of a bisimulation between planning problems and interpreted systems**: We were able to ensure that the planning problem and the equivalent interpreted system are bisimular to each other.
- **Analysis of the semantics of partial observability**: We looked into the semantics of having planning problems with only partial observability of state. We made initial progress in seeing that the semantics differ to an extent that there may not be the possibility of a direct translation.

8.1.2 Weaknesses

We however saw some weaknesses in the theoretical contributions of the project.

- **More generalised formal correspondence**: We were able to define a formal correspondence for planning problems generalised to the extent of having parametrised actions and predicates. It is however not general enough to deal with multiple agents, due to the wide array of semantic differences seen when attempting to reason about planning problems with multiple agents.

  We however note that once the semantic differences are formally laid out and examined in detail, we should be able to exploit the power of interpreted systems and their ability to verify properties about multi-agent systems.

We were also not able to amend the correspondence with the addition of typed objects in the planning. This however should not be too large of a difference, as using types is simply a method to reduce the size of the grounded actions and predicates generated when replacing argument with concrete objects.

- **Proof of a more generalised correspondence**: It is naturally expected for a proof to be provided for the language extensions. This should again be possible for typed objects, but in the case of multi-agent systems, the bisimular property may change depending on the means of which the agents are required to achieve the final goal.

  In the case of finding a correspondence for planning for partial observability, the proof will be far more difficult. The models may again lose the bisimular properties because observations will have to represented in interpreted-system specific properties which may lose the specific properties of an observation when compared directly with the planning problem.

8.2 Implementation

8.2.1 Strengths

In the implementation side of the solution, we saw multiple strengths worth noting:

- **Testing**: Using a test-driven approach when building components of the compiler to deal with new language extensions lead to more modular and inherently more testable code in general.
- **Support for classical planning problems**: We are able to compile classical planning problems specified in PDDL with support for parametrised actions and predicates.
- **First planner for temporally extended goals in CTL***: Along with goals expressible in LTL and CTL, we were able to support CTL* specifications. There is no other planner of our knowledge with the ability to express goals with the same level of expressivity of CTL*. Since the output of the compiler is ISPL, it should be compatible with any further extensions MCMAS.
- **Omission of any external parsing libraries**: We were able to keep the dependency on any external dependencies to a minimum, leading to strongly self-contained code, and ease of use for users to install and begin using the solution.

8.2.2 Weaknesses

- **Testing**: The testing of the solution could have been improved. Unit tests were used for the compiler implementation and acceptance tests were used to check the correctness of the generated plans.

- **Performance of the compiler output**: The output of the compiler generated correct plans in MCMAS, but we see that the time taken to find the plan quickly becomes unreasonable as the size of the state space grows due to an increase in the number of actions, predicates and/or agents in the planning domain.
- **Omission of any external parsing libraries**: Although having the absence of libraries leads to self-contained code without any external dependencies, it does lead to reinventing methods which already exist and may have a more elegant and efficient solution. For example, using the PLY library for parsing the PDDL files would have led to less work on our side to implement new parsing functionality for new extensions to the language.

Chapter 9

Conclusions and Future Work

9.1 Summary of Work

We initially had the goal of defining a formal correspondence between planning problems and interpreted systems and build a compiler to be able to tackle the problem of planning for temporally-extended goals using the temporal logic CTL*. Our overall contributions are the following:

- **Formal correspondence drawn between planning problems and interpreted systems**. We were able to successfully draw a formal correspondence by proving a bisimulation between the two models. The correspondence is sufficient for planning problems where actions and predicates are parametrised.
- **Compiler built to translate PDDL files into ISPL files**: We were able to successfully build a compiler to translate a subset of PDDL and MA-PDDL into ISPL, the input for the model checker MCMAS. The compiler is capable of dealing with both typed and untyped domains, and can plan for the goals to be achieved by multiple agents in different ways.
- **Planning for temporally-extended goals specified in CTL***: We were able to successfully use the extension to MCMAS, MCMAS* to solve planning problems where the goal was specified over the execution structure of the plan rather that only the final states. We investigated the rich set of properties which could be specified in CTL* and analysed the performance for a variety of domains and goal specifications.

9.2 Future Work

Although we managed to investigate a small part of the field of planning through the use of model checking, we still believe that there are several avenues in which the project can be taken further. These are the following:

- **A full investigation into planning under partial observability**: We managed to make progress into how planning with partial observability can be done, but we realised that the difference in the semantics between planning problems with observability and the evolution of an interpreted system caused progress to not be made.

  The investigation itself was also limited by the physical time constraints of the project. With more time, we believe that a formal correspondence should be able to be made. As discussed earlier, this will lead to much more realistic planning problem being solved using model checking in MCMAS.

  Taking this further, it would be ideal to be able to model the possibility of changes of observability of state, in the form of a faulty sensor for a robot, for example.

  It would also be ideal to look further into the differences between planning with uniform versus non-uniform strategies. Mcmas already has functionality to deal with this, but we were only able to make limited progress in this area due to time constraints.

- **Further investigation in planning with multiple agents**: We made initial progress in the investigation of planning with multiple agents, but there is still work to be done. We were able to investigate the different ways in which a plan can be thought of from the point of view of the environment agent or an action-performing one.

  Due to time constraints, we were however not able to investigate further other methods of achieving a goal, for example, adversarial planning, where other agents in the planning domain may have the aim of preventing other agents from achieving their own goals. This also leads into planning under partial observability, where other agents may perform changes to the state of the environment without other agents being aware of it.

- **Optimisations of the compiler and/or MCMAS**: The compiler was built only with minimal optimisations done mainly to reduce the lines of ISPL code outputted. In addition to this, it would also be useful to perform more advanced semantic analysis on the planning domains after the parsing stage of the problem.

  This may be useful in reducing the memory used for allocating BDDs in the model checking process, and will be able to reduce the time taken for finding a plan.

  One concrete example of where this would come into use would be when local variables generated by the predicates of the planning problem may never been modified during any execution of a plan.

  We saw that using heuristics is one of the better ways of achieving much quicker planning times. This should either be done via preprocessing of the PDDL file before it is output to ISPL, or even making planning-specific optimisations in MCMAS itself. We must note that no functionality changes were made in MCMAS apart from plan-printing.

- **Durative actions and constraint solving**: One key area in planning is being able to perform optimisations of variables in some planning domain. Parsing the costs of actions would not be a problem, but the functionality involved in solving the optimisation problem would have to built directly into MCMAS.

Bibliography

[1] An automated planner in the cloud. `http://solver.planning.domains/`. Accessed: 2017-06-21.

[2] MCMAS v1.2.2: User manual. `http://vas.doc.ic.ac.uk/software/mcmas/documentation/`. Accessed: 2017-06-18.

[3] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[4] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, September 2002.

[5] Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22(1):5–27, 1998.

[6] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008.

[7] Piergiorgio Bertoli, Alessandro Cimatti, UD Lago, and Marco Pistore. Extending pddl to nondeterminism, limited sensing and iterative conditional plans. In *Proceedings of ICAPS03 Workshop on PDDL*, 2003.

[8] Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Mbp: a model based planner. In *Proc. of the IJCAI01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[9] Piergiorgio Bertoli, Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'01, pages 473–478, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[10] Piergiorgio Bertoli, Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Strong planning under partial observability. *Artificial Intelligence*, 170(4):337 – 384, 2006.

[11] Blai Bonet and Hector Geffner. Planning under partial observability by classical replanning: Theory and experiments. 2011.

[12] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[13] Krishnendu Chatterjee, Thomas A Henzinger, and Nir Piterman. Strategy logic. In *International Conference on Concurrency Theory*, pages 59–73. Springer, 2007.

[14] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, April 1986.

[15] Ugo Dal Lago, Marco Pistore, and Paolo Traverso. Planning with a language for extended goals. In *AAAI/IAAI*, pages 447–454, 2002.

[16] Giuseppe De Giacomo and Moshe Y. Vardi. *Automata-Theoretic Approach to Planning for Temporally Extended Goals*, pages 226–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[17] Silvio do Lago Pereira and Leliane Nunes de Barros. Using α-CTL to specify complex planning goals. In *International Workshop on Logic, Language, Information, and Computation*, pages 260–271. Springer, 2008.

[18] Hector Geffner. Non-classical planning with a classical planner: The power of transformations. In *European Workshop on Logics in Artificial Intelligence*, pages 33–47. Springer, 2014.

[19] Fausto Giunchiglia and Paolo Traverso. Planning as model checking. In *European Conference on Planning*, pages 1–20. Springer, 1999.

[20] Bob Givan and Ron Parr. An introduction to markov decision processes. *Purdue University*, 2001.

[21] Thilo Hafer and Wolfgang Thomas. *Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree*, page 269279. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.

[22] R. Howey and D. Long. Val's progress: the automatic validation tool for PDDL2.1 used in the international planning competition. In *Proceedings of the ICAPS 2003 workshop on "The Competition: Impact, Organization, Evaluation, Benchmarks"*, pages 28–37, June 2003.

[23] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge university press, 2004.

[24] Rune M Jensen and Manuela M Veloso. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research*, 13:189–226, 2000.

[25] Henry Kautz and Bart Selman. SATPLAN04: Planning as satisfiability. *Working Notes on the Fifth International Planning Competition (IPC-2006)*, pages 45–46, 2006.

[26] Henry A Kautz, Bart Selman, et al. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363. Citeseer, 1992.

[27] Jeremy Kong and Alessio Lomuscio. Symbolic model checking multi-agent systems against CTL*K specifications. In *Proceedings of the 16th Conference on Autonomous Agents and Multi Agent Systems*, AAMAS '17, pages 114–122, Richland, SC, 2017. International Foundation for Autonomous Agents and Multiagent Systems.

[28] Daniel L Kovacs. BNF definition of PDDL 3.1. *Unpublished manuscript from the IPC-2011 website*, 2011.

[29] Daniel L Kovacs. A multi-agent extension of PDDL 3.1. In *Proceedings of the 3rd Workshop on the International Planning Competition (IPC), 22nd International Conference on Automated Planning and Scheduling (ICAPS-2012)*, pages 19–27. ICAPS, 2012.

[30] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[31] Heh-Tyan Liaw and Chen-Shang Lin. On the OBDD-representation of general boolean functions. *IEEE Transactions on Computers*, 41(6):661–664, Jun 1992.

[32] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *International Conference on Computer Aided Verification*, pages 682–688. Springer, 2009.

[33] Alessio Lomuscio and Franco Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 161–168. ACM, 2006.

[34] Drew Mcdermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - the planning domain definition language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[35] William M. McKeeman. Differential testing for software. *DIGITAL TECHNICAL JOURNAL*, 10(1):100–107, 1998.

[36] Fabio Mogavero, Aniello Murano, Giuseppe Perelli, and Moshe Y. Vardi. *What Makes Atl* Decidable? A Decidable Fragment of Strategy Logic*, pages 193–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[37] Fabio Mogavero, Aniello Murano, Giuseppe Perelli, and Moshe Y Vardi. Reasoning about strategies: On the model-checking problem. *ACM Transactions on Computational Logic (TOCL)*, 15(4):34, 2014.

[38] Aniello Murano. Verifying and synthesising multi-agent systems against one-goal strategy logic specifications. In *3rd International Workshop on Strategic Reasoning*, volume 20, 2015.

[39] M. Reynolds. An axiomatization of full computation tree logic. *The Journal of Symbolic Logic*, 66(3):1011–1057, 2001.

[40] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995.

[41] Sven Schewe. *ATL* Satisfiability Is 2EXPTIME-Complete*, pages 373–385. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[42] Wiebe Van Der Hoek and Michael Wooldridge. Tractable multiagent planning for epistemic goals. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*, pages 1167–1174. ACM, 2002.

[43] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1 – 37, 1994.

[44] Daniel S Weld. Recent advances in AI planning. *AI magazine*, 20(2):93, 1999.

Appendix A

Plans generated for Gripper domain problems

We attach the plans generated for the formulae

$$E(F\,goal) \wedge (G\,\texttt{free(right)})$$

and

$$E(G(F(goal \wedge (E(X\neg goal)))))$$

for the Gripper domain.

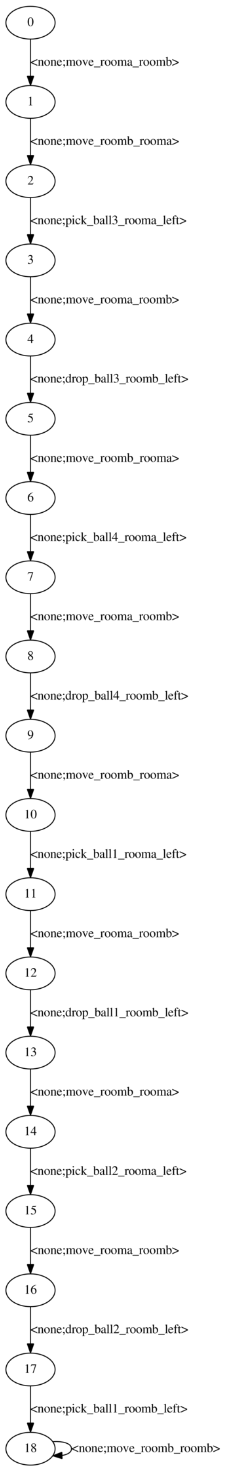

Figure A.1: Plan generated for formula $E(F\,goal) \wedge (G\,\texttt{free(right)})$.

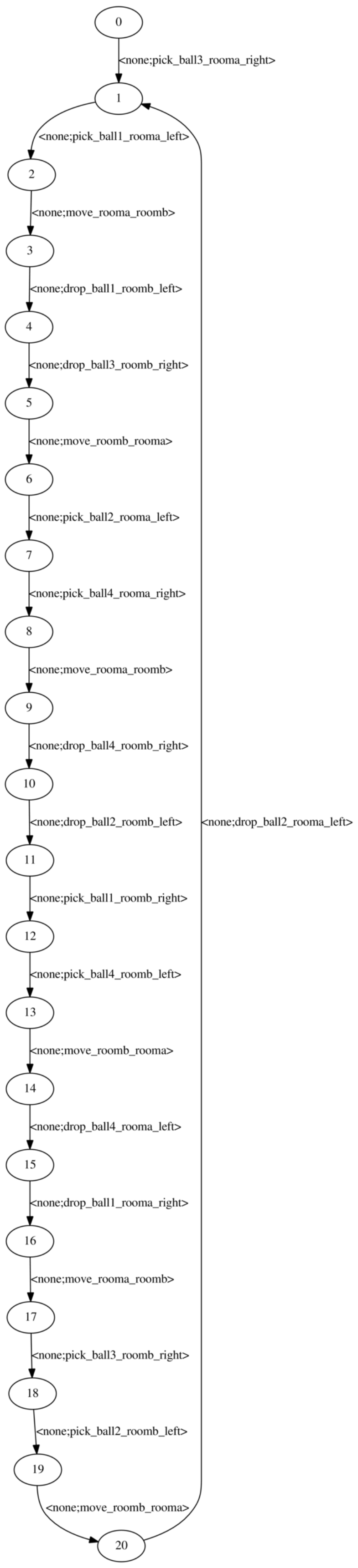
0
<none;pick_ball3_rooma_right>
1
<none;pick_ball1_rooma_left>
2
<none;move_rooma_roomb>
3
<none;drop_ball1_roomb_left>
4
<none;drop_ball3_roomb_right>
5
<none;move_roomb_rooma>
6
<none;pick_ball2_rooma_left>
7
<none;pick_ball4_rooma_right>
8
<none;move_rooma_roomb>
9
<none;drop_ball4_roomb_right>
10
<none;drop_ball2_roomb_left>
<none;drop_ball2_rooma_left>
11
<none;pick_ball1_roomb_right>
12
<none;pick_ball4_roomb_left>
13
<none;move_roomb_rooma>
14
<none;drop_ball4_rooma_left>
15
<none;drop_ball1_rooma_right>
16
<none;move_rooma_roomb>
17
<none;pick_ball3_roomb_right>
18
<none;pick_ball2_roomb_left>
19
<none;move_roomb_rooma>
20

Figure A.2: Plan generated for formula $E(G(F(goal \wedge (E(X\neg goal)))))$.

Appendix A

Robot Navigation Domain (Full Observability) in ISPL

```
Agent Environment
    Obsvars:
        robot_position : {store, lab, ne_room, sw_room, dep};
    end Obsvars
    RedStates:
    end RedStates
    Actions = {
        none
    };
    Protocol:
        Other : { none };
    end Protocol
    Evolution:
        robot_position=store if ( Performer.Action=move_robot_left and robot_position=ne_room) or
                                ( Performer.Action=move_robot_up and robot_position=sw_room );
        robot_position=ne_room if ( Performer.Action=move_robot_right and robot_position=store ) or
                                  ( Performer.Action=move_robot_up and robot_position=dep ) or
                                  (Performer.Action=move_robot_down and robot_position=lab );
        robot_position=lab if ( Performer.Action=move_robot_up and robot_position=ne_room );
        robot_position=sw_room if ( Performer.Action=move_robot_left and robot_position=dep) or
                                  ( Performer.Action=move_robot_down and robot_position=store );
        robot_position=dep if ( Performer.Action=move_robot_right and robot_position=sw_room ) or
                              ( Performer.Action=move_robot_down and robot_position=ne_room );
    end Evolution
end Agent
Agent Performer
    Vars:
        state : { empty };
    end Vars
    Actions = {
        move_robot_up, move_robot_down, move_robot_right, move_robot_left
    };
    Protocol:
        ( Environment.robot_position=sw_room or
          Environment.robot_position=dep or
          Environment.robot_position=ne_room ) : { move_robot_up };
        ( Environment.robot_position=store or
          Environment.robot_position=lab or
          Environment.robot_position=ne_room ) : { move_robot_down };
        ( Environment.robot_position=sw_room or
          Environment.robot_position=store ) : { move_robot_right };
        ( Environment.robot_position=dep or
          Environment.robot_position=ne_room ) : { move_robot_left };
    end Protocol
    Evolution:
        state=empty if state=empty;
    end Evolution
end Agent
Evaluation
    goal if Environment.robot_position=dep;
end Evaluation
InitStates
    Environment.robot_position=store;
end InitStates
Groups
    g1 = { Performer };
end Groups
Fairness
end Fairness
Formulae
    AF goal;
end Formulae
```