Implementation of Conditional Epistemic Planning

Daniel Svendsen

Kongens Lyngby 2013
IMM-M.Sc-2013-21

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-M.Sc-2013-21

Abstract

Automated planning is an area of Artificial Intelligence that studies reasoning about acting, an abstract deliberation process that chooses and organizes action by anticipating outcomes [1]. Classical planning deals with restricted state transition systems. They are deterministic, static, finite and fully observable and with restricted goals and implicit time [1]. In this project we go beyond classical planning by extending on the restrictions of classical planning by considering partial observability (not the entire world is known) and non-determinism (applying the same action to the same state, might not always yield the same result). Recently research [2] has shown that planning under partial observability and non-determinism fits naturally within the theory of dynamic epistemic logic (DEL). In [2], an algorithm for planning under partial observability and non-determinism is provided based on the DEL framework. The primary goal of this project is to implement the algorithm of the paper. Implementation of the planning algorithm of [2] involves parsing of models, computing product updates for states, generating AND-OR-trees via the tree expansion rule and model checking to check for completed goal formulas. In addition to implementing the algorithm, the project will seek to come up with interesting planning domains that fit into the framework and can showcase DEL-based planning and the implementation of it. Furthermore, depending on early successes, implementing the basic steps of the algorithm, several futher avenues can be pursued; these include: Implementation of a model based planner (MBP), plan validation, implementing tools for creating bigger examples in NuPDDL or similar language.

1. Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann (2004)

2. Bolander, T., Birkegaard, A., Holm Jensen, M.: Conditional Espistemic Planning (2012)

Contents

Chapter 1

Introduction

Planning is the act of deliberating about the future with the purpose of achieving a goal. *Automated Planning* is a major classical branch in *Artificial Intelligence* aiming to mimic the human ability of making a plan to solve a task, before commencing *any* action. In *classical automated planning* problems are restricted to be finite, deterministic, fully observable and static [MG04]. Articles [TB12c, TB12b] has been written adressing these restrictions and lifting them, so that uncertainty in the planning domains (partial observability and non-determinism) is possible, meaning that an agent in the domain may not necessarily know the exact outcomes and state of affairs in the domain, to more closely ressemble real life scenarios.

The aforementioned articles describes how the notion of *knowledge* from Dynamic Epistemic Logic (DEL) is used in order to create plans which are conditional in nature, using `if-then-else` structures to ensure a working plan for the agent, even in domains with partial observability or non-determinism. The purpose of this project is to *implement in practise the theory behind the creation of conditional plans in the DEL framework and to show working examples of the planning process with this implementation.* In order to do this, first we must define what the DEL framework is, as according to the articles [TB12c, TB12b]. The definition of the DEL framework and its components will be addressed in detail in the first part of this report (Section Dynamic Epistemic Logic), where working examples will be used to visualize the theory. After this the actual implementation of conditional plans in DEL will be discussed, again accompanied by examples where applicable. In the *Planning* section of the implementation, pseudo-code will be given to illustrate the algorithms used in the *program* – the product of the project. The chapter "Examples and Results" (Chapter 5), will display the examples inputted into the program and the entire process from input to output will be detailed to show the end result, namely a visualization of the planning tree in order to make a final plan. Lastly section Future Work will

describe possible additions to the program that was not implemented because of the time frame.

To reiterate, *the focus of this project is the implementation of the conditional epistemic planning framework as described above.*

As mentioned, throughout the report, examples will be given to support the theory and to visualize the concepts given. In the beginning, the example in Figure 1.1 will be used, however later, in both the theory and implementation sections, more complex examples will be given when needed in order to showcase the concepts specifically.

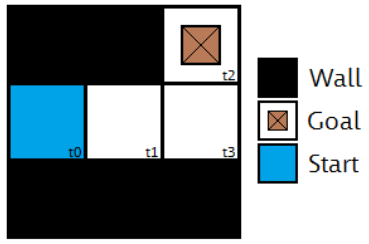


Figure 1.1: A simple finding-goal domain SIMPLE.

Example 1: The agent starts at the left-most tile (t_0), and wants to move to the goal located at t_2. The agent will have to use his available *actions* in order to move to the goal location.

We want to model the states of the domain in this example using DEL in order to later formulate a plan for the domain. The initial model of the domain can be seen below.

The domain in Figure 1.1, which will be elaborated upon in the next section, is shown as an epistemic model in Figure 1.2. The world w is represented as a conjunction of propositional symbols, where an underlined proposition p indicates that p does *not* hold in w. For visual simplicity, reflexive and transitive edges are omitted in the model.

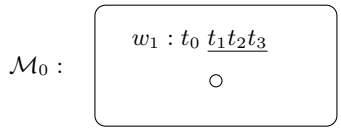


Figure 1.2: The initial situation. The agent is at location t_0.

Chapter 2

Dynamic Epistemic Logic

The first step in order to model the domain is to define the language used to this end. Dynamic epistemic logic lays the foundation on which this project is built, and in order to progress we first need to define the entire framework with proper definitions, starting with the epistemic language, to be able to use this later in the planning process.

2.1 Epistemic Language

Let P be a finite set of atomic propositions (propositional symbols) then the language of dynamic epistemic logic $\mathcal{L}_{DEL}(P)$ is given by the following BNF.

$$\phi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi \wedge \phi \mid K\phi$$

where the proposition $p \in P$. The construct $K\phi$ is to be read as "ϕ is known". The semantics of $\mathcal{L}_{DEL}(P)$ is defined through Kripke structures, from this point on referred as *epistemic models*.

The definition of the language is essential for the understanding of worlds and formulas in the project, as they are both built on this language $\mathcal{L}_{DEL}(P)$.

2.2 Epistemic Models

As with the epistemic language, it is necessary to define what an epistemic model is, in order to lay the foundation for the implementation. An epistemic model

of the language $\mathcal{L}_{DEL}(P)$ is defined as the triple $\mathcal{M} = (W, R, V)$ where W is a finite set of worlds representing the domain of the model $D(\mathcal{M})$, $R \to 2^W$ is a equivalence relation and $V : P \to 2^W$ assigns a valuation for each propositional symbol, i.e. the set of worlds where $p \in P$ holds.

The equivalence relation determines if two worlds are distinguishable to the agent, more specifically, if the agent has *knowledge* of a proposition or not (more on this in Section 2.4.1). A world is always indistinguishable to itself, that is, for visual simplicity reflexive edges are not drawn. The same holds for transitive edges, if world $w_1 \in W$ is indistinguishable to world $w_2 \in W$ and w_2 is indistinguishable to world $w_3 \in W$, then w_1 is indistinguishable to w_3. Two worlds w_1 and w_2 connected by an equivalence relation is said to be in the same *equivalence class*. An equivalence class in a model $\mathcal{M}$ is denoted $[w]_R$, and is to be read as *the set of worlds related to w by R*.

The valuation V defines which propositional symbols hold in any world. For example, in the model in Figure 1.2, the valuation assigns w_1 to t_0. In order to see if a valuation in any given world holds, we need to define the truth in epistemic models.

2.2.1 Truth in Epistemic Models

$\mathcal{M}, w \models$	$\top$		always
$\mathcal{M}, w \models$	$\bot$		never
$\mathcal{M}, w \models$	p	iff	$w \in V(p)$
$\mathcal{M}, w \models$	$\neg\phi$	iff	$\mathcal{M}, w \not\models \phi$
$\mathcal{M}, w \models$	$\phi \wedge \psi$	iff	$\mathcal{M}, w \models \phi$ and $\mathcal{M}, w \models \psi$
$\mathcal{M}, w \models$	$K\phi$	iff	for all $v \in W$, if w R v then $\mathcal{M}, w \models \phi$
$\mathcal{M} \models$	ϕ	iff	$\mathcal{M}, w \models \phi$ *for all* $w \in D(\mathcal{M})$

The pair $\mathcal{M}, w$ represents a specific world w in the epistemic model $\mathcal{M}$. This pair is called either an *epistemic state* or a *pointed epistemic model*.

The structure $\mathcal{M}, w \models K\phi$ depicts the notion of knowledge which will be elaborated in the section on Partial Observability (Section 2.4.1), and is to be read as $K\phi$ is satisfied in $\mathcal{M}, w$ if and only if ϕ is satisfied in all worlds $v \in [w]_R$.

2.3 Event Models

In order to model dynamism or the notion of updating or advancing a state in the domain, we need to introduce the event models. With the event models, the agent will be able to change an epistemic state or model in a given situation. An event model is a tuple $\mathcal{E} = (E, Q, \text{pre}, \text{post})$, where: ([TB12c])

- E, the domain, is a finite non-empty set of events.
- $Q \subseteq E \times E$ assigns an accessibility relation. All accessibility relations are equivalence relations.
- $pre : E \to \mathcal{L}_{DEL}(P)$ assigns to each *event* a precondition.
- $post : E \to (P \to \mathcal{L}_{DEL}(P))$ assigns to each *event* a postmapping. A postmapping is a conjunction of propositional symbols (atomic propositions, including $\top$ and $\bot$) and their negations, mapping or modifying existing propositional symbols in the model.

The domain of an event model is a finite set of possible events or actions within the event model. If we think of a specific action in the domain of the example in Figure 1.1, the action *GoUp* would represent moving from t_3 to t_2. This event model is shown in Figure 2.1.

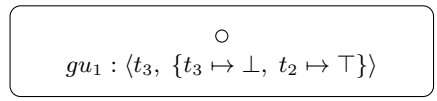


Figure 2.1: The *GoUp* action in the domain of the example in Figure 1.1. *(reflexive and transitive edges are omitted).*

The propositional symbols specified are the tile names t_0, t_1, t_2 and t_3, and the precondition of event gu_1 (first event of the *GoUp* event model) in $\mathcal{E}_0$ specifies that the agent has to be at tile t_3 in order for the event to be applicable (more on applicability in Section 2.4), whereas the postconditions of the event, is a set of propositional mappings that assigns a new valuation to a specific propositional symbol, in this case assigning t_3 to $\bot$ and t_2 to $\top$ (and thus in this domain updating the location of the agent from t_3 to t_2).

Now, if an action has more than one application, we will need to define all of the applications within the event model. In the case of *GoUp* there was only one application since the only place from which the agent could *GoUp* was from

the tile t_3 to t_2. Consider the action *GoRight* instead. Here we are dealing with two different applications, namely that of t_0 to t_1 and t_1 to t_3. The *GoRight* event model can be seen in Figure 2.2.

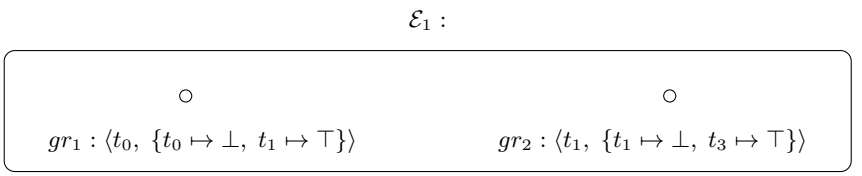


Figure 2.2: The *GoRight* action of example $\mathcal{M}_0$ *(reflexive and transitive edges are omitted).*

In this event model, the preconditions determine which event is used, since there are two different possibilities for the agent to *GoRight*. Depending on whether the agent is standing at tile t_0 or t_1 the event used will be gr_1 or gr_2 respectively.

The accessibility relation (equivalence relation for events), Q, is a set of edges between events in the domain (E), defining wether or not the agent can distinguish between the events in the domain. As can be seen in Figure 2.2, no edges exists between the nodes, as all of the actions are distinguishable to the agent. For a fully observable domain, this specification is complete, however if the domain is partially observable, extra information will need to be added to reflect notion of *discovery*. More on that in Section 2.4.1.

2.4 Product Update

In order for the agent to change states in the domain, the models need to be *updated* with the event models defined earlier. A product update is the application of an event model on an epistemic model. An epistemic model can be updated with an event, yielding as product a new epistemic model. This new model will reflect the action executed on the old model[TB12b].

Let $\mathcal{M} = (W,\ R,\ V)$ be an epistemic model and $\mathcal{E} = (E,\ Q,\ pre,\ post)$ be an event model on $\mathcal{L}_{DEL}(P)$. The product update of $\mathcal{M}$ with $\mathcal{E}$ is the updated epistemic model denoted $\mathcal{M} \otimes \mathcal{E} = (W',\ R',\ V')$, where

- $W' = \{(w,\ e) \in W\ \times\ E \mid \mathcal{M}, w \models pre(e)\}$,
- $R' = \{((w, e), (v, f)) \in W'\ \times\ W' \mid w\ R\ v\ and\ e\ Q\ f\}$,

- $V'(p) = \{(w, e) \in W' \mid \mathcal{M}, w \models \text{post}(e)(p)\}$ *for each* $p \in P$.

The new epistemic model will contain:

- W' : Each of the worlds in $\mathcal{M}$ where the *precondition* is satisfied
- R' : A new equivalence relation, defining edges between two worlds (w, e) and (v, f); if w were connected to v before the update (by R) and if the events e and f updating the worlds were connected also (by Q).
- V' : A new valuation as a product of the *postmapping* of $\mathcal{E}$ and the propositional symbols of the original world.

First let us consider a short example to show the process of product update. Take the domain shown in Figure 1.1 and consider the model, where the agent is located at tile t_3 about to reach the goal.

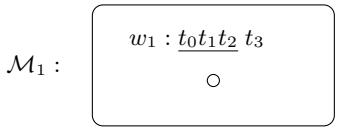


Figure 2.3: Simple model of the agent located on t_3. *(reflexive edges are omitted)*

The action that the agent want to execute is *GoUp*. Figure 2.4 shows the *GoUp* action executed on model $\mathcal{M}_1$.

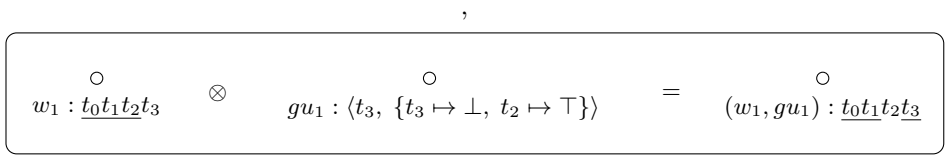


Figure 2.4: Product update of $\mathcal{E}_0$ on $\mathcal{M}_1$. *(reflexive edges are omitted)*

If a world w satisfies the precondition of an event e the product update of w with e is written (w, e). The satisfaction of the precondition is specified below.

Applicability: Given a world $\mathcal{M}, w$ and an event $\mathcal{E}, e$ the event e is applicable in the world w iff $\mathcal{M}, w \models \mathit{pre}(e)$. This is to ensure that the precondition of the event e taking place in w is fulfilled. If not, the resultant world would be empty.

An example of a product update, where the precondition is not met, is when the agent executes the event model *GoRight* (see Figure 2.5). Since *GoRight* has

two different events in the event model, the preconditions determine which of the two events are applied. In this case one of the updated worlds will be empty, since the precondition of the event is not met in that world. Another interesting point here, is that at all times only one of the events in the event model will be applicable. This is because the event model is *globally deterministic* as according to the definition[TB12c] given later in Section 2.4.2, along with examples that are not globally deterministic but non-deterministic.

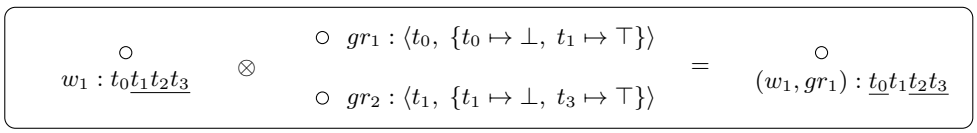


Figure 2.5: The application of *GoRight* ($\mathcal{E}_1$) on $\mathcal{M}_0$. *(reflexive and transitive edges are omitted)*

Since event gr_2 has precondition t_1, this event will not be taken into consideration during the product update. The event gr_1 is the only applicable event in the event model $\mathcal{E}_1$ on $\mathcal{M}_0$.

Now, we have shown that if the agent knows where the goal is and has available the necessary actions, he can apply these actions to his belief state (epistemic model) and get a new belief state where he is closer to the goal. But what happens if the agent is not all-knowing, and does not know where the goal is? Or if there are more than one possible goal location?

In the comings sections the domain SIMPLE will be expanded in order to show how DEL deals with partial observability and non-determinism.

2.4.1 Product Update in a Partial Observable Domain

The examples used until now has been without the notion of partial observability [pom] to indroduce the different components of dynamic epistemic logic in a simple and straight forward fashion. But what if, the agent did not know the actual location of the goal. Imagine a domain, where an agent is put into a maze (see Figure 2.6), the agent can see in a straight line, but not around corners. The agent need to be in a straight line of sight of the goal locations in order to ascertain if the goal is located on that tile or not.

In a *partial observable domain* like the one mentioned, we need to add the notion of *discovery* to the event models, in order for the agent to *learn* how the domain looks. Specifically in the domain shown in Figure 2.6, if the agent had available

only the event models we have seen so far ($\mathcal{E}_0$ and $\mathcal{E}_1$) he could never discover on which of the goal locations the goal is actually located. This is due to the (so far missing) update in the equivalence relation taking as precondition the difference between the worlds.

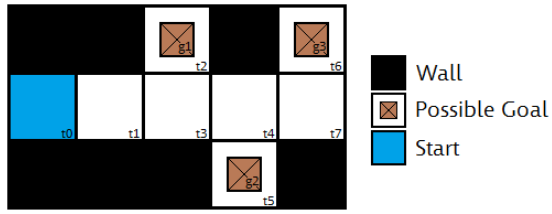


Figure 2.6: A partial observable domain PARTIAL.

The model shown in Figure 2.7 has been visually simplified by not showing the *negated* propositional symbols, and as before all reflexive and transitive edges has been left out.

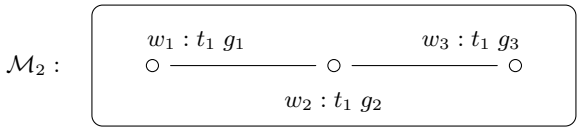


Figure 2.7: A model of the domain where the agent has already executed *GoRight* once. *(reflexive and transitive edges are omitted)*

An interesting observation with this model, however, is that with this model in contrast to the ones we have seen previously, the equivalence relation is in employ. The edges between the worlds w_1 and w_2 and between w_2 and w_3 signify that the agent cannot distinguish between these worlds. The agent does not know where the goal is located[1].

Now in order to make the agent able to discover the goal as mentioned before, we need to update the event model. Because the worlds in Figure 2.7 are indistinguishable on the variables g_1, g_2 and g_3, we need to make the agent able to seperate these. This is done by letting the events have specific preconditions, pairing up the unknowns with a negated and a non-negated precondition for each of the unknown propositional symbols.

We want the agent, when located at t_3 to be able to tell if the goal is located at tile t_2 or not. Taking a closer look at event gr_{11} and gr_{12} in Figure 2.8, the

[1]The goal location is specified by the presense of one of the propositional symbols g_1, g_2 or g_3, and is chosen at random by the environment at the beginning of the simulation.

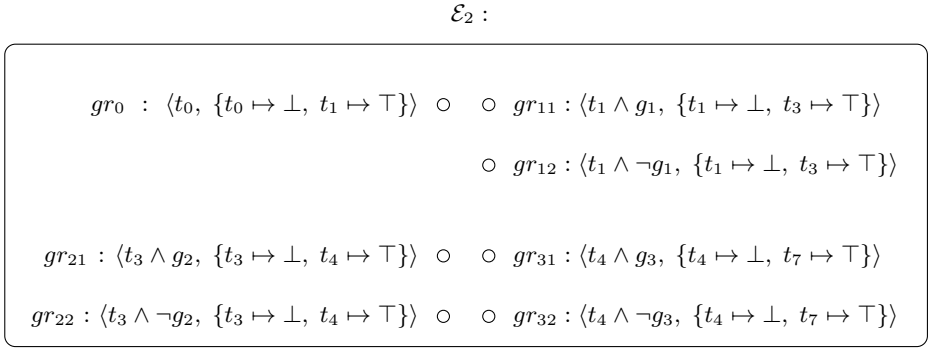


Figure 2.8: The updated event model *GoRight*. *(reflexive and transitive edges are omitted)*

two events display contradictory preconditions in regards to the goal propositional symbols g_1 and $\neg g_1$ in order to give knowledge to the agent regarding the propositional symbol g_1. The same is true for the other two goal tiles g_2 and g_3, when the agent arrives at tile t_4 and t_7 respectively, he will gain the knowledge of g_2 and g_3 respectively.

Application of the action *GoRight* on epistemic model $\mathcal{M}_2$ is shown in Figure 2.9.

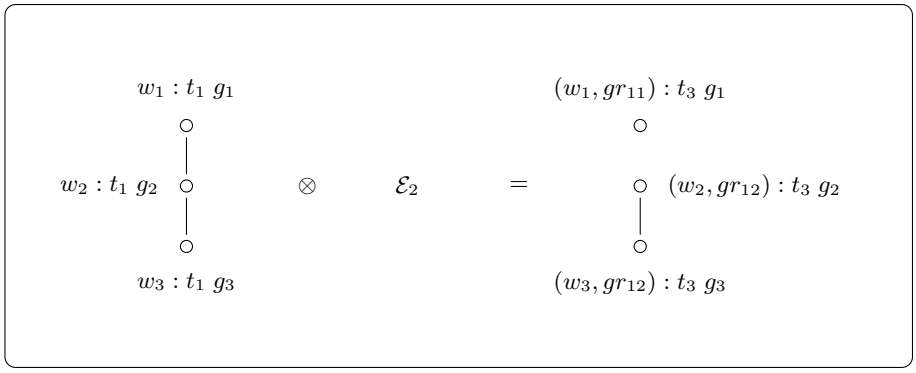


Figure 2.9: The application of *GoRight* ($\mathcal{E}_2$) on $\mathcal{M}_2$. *(reflexive and transitive edges are omitted)*

The outcome of this product update, as can be seen in the far right of Figure 2.9 is that the agent are now able to seperate the worlds (w_2, gr_{12}) and (w_3, gr_{12}) from the first world (w_1, gr_{11}) in the sense that the agent either knows that g_1

holds or knows that it does not. $Kg_1 \vee K\neg g_1$.

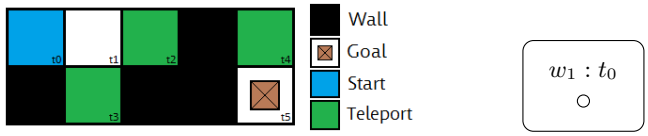
When the environment returns to the agent the simulation of whether or not g_1 holds, the agent is able to make a choice from this, to either go up towards the tile t_2 (if g_1 holds) or to continue right (if g_1 does not hold). More on this in the planning section (see Section 3).

2.4.2 Product Update in a Non-Deterministic Domain

Before going through the second aspect of uncertainty mentioned, namely non-determinism, we first need to specify what non-determinism is. How does it differ from the partial observability? When an agent encounters a new world, for example different instances of the domain given in Figure 2.6 each instance might have the goal on different locations which is non-deterministically decided; So what exactly is meant by non-determinism?[non]

An event model $\mathcal{E} = (E, Q, \text{pre}, \text{post})$ is called *globally deterministic* if all pre-conditions are mutually inconsistent, meaning that $\models pre(e) \wedge pre(f) \rightarrow \bot$ for all distinct pairs $e, f \in E$, that is, at all times only *one* event is applicable for any *one* world.

So far, by definition, all of the event models given, have been globally deterministic as given above and in [TB12c]. That means that within each of the event models, at any point only *one* event at a time, can fulfill the precondition. In the event model shown in Figure 2.8, seven events were given in order to *GoRight*. All of these seven events each had different preconditions and thus at no point would more than one event correspond to a certain world-state. In other words, in order for an event model to be globally deterministic, the conjunction of the event elements must always be a falsum.



(a) The domain NonDet

(b) The epistemic model of NonDet *(reflexive edges are omitted)*

Figure 2.10: The domain NonDet which has a component of non-determinism represented by *teleports*.

In non-determinism, this is no longer the case. An event model can have more

than one event with the same preconditions, making the outcome depend not on a compiletime generation of environment, but rather a runtime stokastic process. This process will at each non-deterministic action at random choose the outcome. In order to model a world with non-determinism we will have to create an environment which has a non-determinic component. An updated domain with the *teleport* component has been created in Figure 2.10(a).

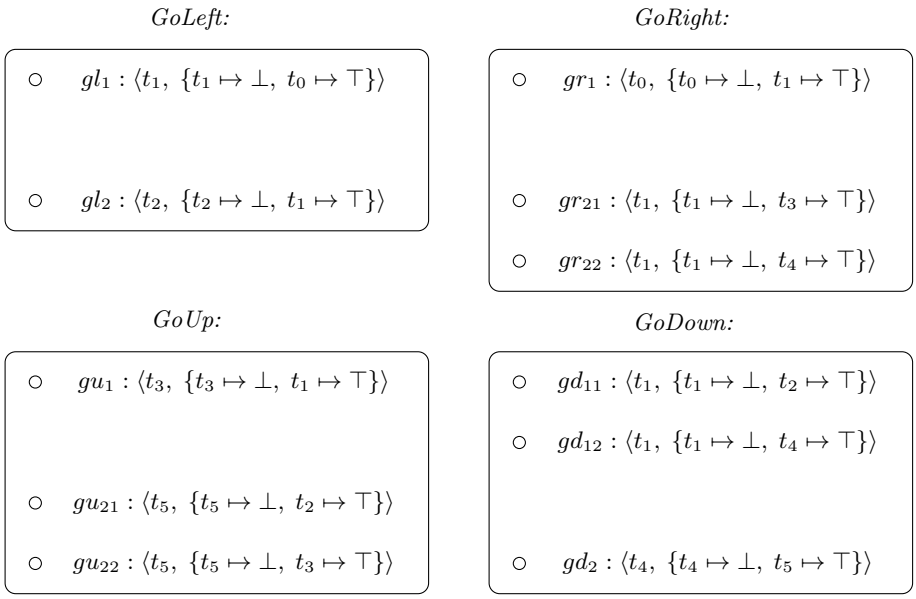


Figure 2.11: The event models *GoRight* and *GoLeft* for the domain *Non-Det*.

In this world, the agent starts in the upper-left corner and has to maneuver to the goal located in the bottom-right corner. The agent knows where the goal is (there is only one) and the domain is fully observable, however the path to the goal is not straight forward as it were in the domain SIMPLE or PARTIAL. In order to reach the goal, the agent has to make use of the *teleports*[2] in the environment, however, since there are multiple of these teleports (green tiles), the agent does not know in advance where the teleport will take him (t_3 or t_4, in the scenario, where the agent enters the teleport at t_2) [3].

[2]A teleport tile instantly moves the agent to another teleport tile. If more than two teleports exist, the destination tile is chosen at random by a stochastic process.

[3]In a world such as this, a strong plan is not possible, since the agent cannot be certain to reach the goal in a finite number of actions. In order to make a plan that will always succeed (under the assumption of fairness), one will have to make a strong *cyclic* plan. More on that in Section 3.7

The event models used to model the behaviour of the teleports can be seen in Figure 2.11.

In the four event models displayed in Figure 2.11, the non-determinism is implemented by having multiple events within the event model with the same preconditions. For example, in the $GoUp$ event model above, the events gu_{21} and gu_{22} are paired because both accept states in which the precondition t_5 is satisfied. Thus whenever an event model with multiple feasible events are applied to an epistemic model, *only one* of the feasible events will be applied by the stochastic process. Thus in the case of the domain NonDet in Figure 2.10(a), when the agent enters a teleport tile (green tiles at t_2, t_3 or t_4) he will appear on *one* of the other two tiles.

Chapter 3

Planning

In order to make conditional plans in DEL to be implemented, we must first define what planning is. As stated in the introduction, planning is the act of deliberating about the future with the purpose of achieving a goal before acting.

3.1 Classical Planning

A planning domain in classical planning according to [MG04] is a restricted state-transition system $\Sigma = (S, A, \gamma)$, where S is a set of states, A a set of *actions* or *events* and γ is a state-transition function such that $\gamma(s, a) \in S$ for $s \in S$ and $a \in A$.

A classical planning problem can then be defined as a triple (Σ, s_0, S_g), where Σ is a restricted state-transition system, s_0 is the initial state, and S_g is the set of goal states. Thus a plan in classical planning, solving a problem (Σ, s_0, S_g), is a series of *actions* or *events* $e_1,\ e_2, \ldots,\ e_n$ such that ([TB12c])

$$\gamma(\gamma(\ldots\gamma(s_0, e_1), e_2), \ldots, e_{n-1}), e_n) \in S_g$$

3.2 Epistemic Planning

Then in epistemic planning, which is a special case of classical planning with the notion of *knowledge* as seen earlier, when given a finite set of propositions P, we can define the *epistemic planning domain* as a restricted state-transition

system $\Sigma = (S, A, \gamma)$, where S is a finite set of epistemic states of $\mathcal{L}_{DEL}(P)$ and A a finite set of actions on $\mathcal{L}_{DEL}(P)$, and

$$\gamma(s, e) = \begin{cases} s \otimes e & \textit{if } e \textit{ is applicable in } s \\ \text{undefined} & \textit{otherwise} \end{cases}$$

An *epistemic planning problem* is then defined as a triple (Σ, s_0, ϕ_g), where

- $\Sigma = (S, A, \gamma)$ *is an epistemic planning domain on* P.
- s_0, *the* initial state, *is a member of* S.
- ϕ_g *is a formula in* $\mathcal{L}_{DEL}(P)$ *called a* goal formula. The *set of* goal states is $S_g = \{s \in S \mid s \models \phi_g\}$.

Since we at all times are only concerned with the single-agent aspect of the planning domain, this problem is called a single-agent epistemic planning problem.

As with the *epistemic planning domains*, a *plan* in epistemic planning is a special case of a solution to a classical planning problem. More specifically a finite sequence of events $e_1, e_2, \ldots, e_n$ such that $\gamma(\gamma(\ldots\gamma(\gamma(s_0, e_1), e_2), \ldots, e_{n-1}), e_n) \in S_g$ that is, $s_0 \otimes e_1 \otimes e_2 \otimes \cdots \otimes e_n \models \phi_g$.

With the epistemic planning domains, problems and plans defined, we are ready to commence making the structure for conditionals plans in Dynamic Epistemic Logic.

3.3 Conditional Planning

The reason to extend epistemic planning with conditionals (if-then-else constructs) is to be able to plan under uncertaincy. We want to be able to model the worlds which are not straight forward in essence. At this point we are not able to make a plan for the domains given in figures 2.6 and 2.10(a) due to their uncertainty aspects (partial observability and non-determinism respectively).

We want to formulate a planning language, such that the agent, when sufficient *knowledge* has been acquired, makes a choice and are able to find the goal-state even if the domain contains aspects of uncertainty.

3.3.1 The internal and external perspective

When talking about specific worlds in an epistemic model, we have only been talking about *pointed* epistemic models. This is because from the external perspective we are able to *point out* a specific world in an equivalence class in the model. However from the perspective of the planner (i.e. the agent), when modelling his own knowledge or ignorance, he will *not* able to point out the actual world, hence a *non-pointed* epistemic model. We call this the internal perspective [TB12c, Auc10]. Consider the model given in Figure 2.7, containing the three indistinguishable worlds w_1, w_2 and w_3. The model shown here is from the perspective of the agent and he is of course not able to point out the real world. Ergo, it is only natural that this model is a non-pointed epistemic model. Worlds within a non-pointed epistemic model related via the equivalence relation R is said to be in the same equivalence class (also called *information cell* [TB12b]). All worlds in the same equivalence class share the same K-formulas (of the form $K\phi$) and therefore representing the same situation seen from the view of the agent modelling the situation. Each equivalence class represents a possible state of knowledge of the agent [TB12b].

3.3.2 Plan Language

In order to formulate a plan, we need to define the language in which the plans will be given. Given a finite set A of event models on $\mathcal{L}_{DEL}(P)$, the plan language $\mathcal{L}_P(P, A)$ is given by: ([TB12b])

$$\pi ::= \mathcal{E} \mid \text{skip} \mid \text{if } K\phi \text{ then } \pi_1 \text{ else } \pi_2 \mid \pi_1 \ ; \ \pi_2$$

where $\mathcal{E} \in A$ is an event model, ϕ a formula in $\mathcal{L}_{DEL}(P)$ and the if-then-else construct is to be read as *"if* ϕ is *known* do π_1 else do π_2*"*. Note that ϕ is required to be *known* to the agent, as he can only make choices based on propositions he *knows*. Also note that "$K\phi$ then π" is short for "$K\phi$ then π else skip".

Returning to the example of Figure 2.6 the agent will need a plan to go from the start tile (t_0) to the goal tile located on either t_2, t_5 eller t_6. In order to formulate questions such as *does plan* π *achieve* ϕ*?*, the notion of translation is introduced in the next section.

3.3.3 Translation

A strong translation $[\![\cdot]\!]_s\cdot$ respectively weak translation $[\![\cdot]\!]_w\cdot$ is defined as functions from $\mathcal{L}_P(P,A) \times \mathcal{L}_{DEL}(P)$ into $\mathcal{L}_{DEL}(P)$ by ([TB12b]):

$$
\begin{aligned}
&[\![\mathcal{E}]\!]_s\ \phi := \langle\mathcal{E}\rangle\top \wedge [\mathcal{E}]\, K\phi \\
&[\![\mathcal{E}]\!]_w\ \phi := \langle\mathcal{E}\rangle\top \wedge \neg K\langle\mathcal{E}\rangle K\phi \\
&[\![\text{skip}]\!].\ \phi := \phi \\
&[\![\text{ if } \phi' \text{ then } \pi \text{ else } \pi']\!].\ \phi := (\phi' \to [\![\pi]\!].\ \phi) \wedge (\neg\phi' \to [\![\pi']\!].\ \phi) \\
&[\![\pi;\pi']\!].\ \phi := [\![\pi]\!].\ ([\![\pi']\!].\ \phi)
\end{aligned}
$$

By using translation we can interpret a plan π relative to a formula ϕ and answer the question *does plan π achieve ϕ*. If we want to see if the plan is a weak solution to a planning problem $\mathcal{P}$ we use the weak translation $[\![\mathcal{E}]\!]_w\phi$, whereas we want to see if the plan is a strong solution to $\mathcal{P}$ the strong translation $[\![\mathcal{E}]\!]_s\phi$ is used.

Note that the notion of Applicability is built into the translation by the conjunct $\langle\mathcal{E}\rangle\top$ in both the strong translation $[\![\mathcal{E}]\!]_s\phi$ and weak translation $[\![\mathcal{E}]\!]_w\phi$, and that the difference between the two translations is in the *robustness* of the plans: $[\![\mathcal{E}]\!]_s\phi$ respectively $[\![\mathcal{E}]\!]_w\phi$ means that each step of π is applicable and that executing plan π *always* leads, respectively *may* lead to the goal state ϕ as seen in [TB12a].

3.3.4 Planning Problems and Solutions

Now that the plan language for conditional epistemic planning has been given, we can look at formulating planning problems on $\mathcal{L}_P(P,A)$.

Let P be a finite set of propositional symbols, a planning problem on P is a triple $\mathcal{P} = (\mathcal{M}_I, A, \phi_g)$, where $\mathcal{M}_I$ is an non-pointed epistemic model on $\mathcal{L}_{DEL}(P)$ called the initial state, A is a finite set of event models on $\mathcal{L}_{DEL}(P)$ called the action library and $\phi_g \in \mathcal{L}_{DEL}(P)$ is the goal formula[TB12b].

We say that a plan $\pi \in \mathcal{L}_P(P,A)$ is a *strong solution* to $\mathcal{P}$ if $\mathcal{M}_I \models [\![\pi]\!]_s\phi_g$, a weak solution if $\mathcal{M}_I \models [\![\pi]\!]_w\phi_g$, and not a solution otherwise.

Looking at the Partial domain in Figure 2.6 we might formulate a goal as $\phi_g = Kg_1 \vee Kg_2 \vee Kg_3$, namely that the goal is to achieve *knowledge* as to where the goal is located (g_1, g_2 or g_3). Since the *GoRight* event *models* the

agent moving from t_1 to t_3, from t_3 to t_4 and from t_4 to t_7, it *adds* knowledge to the the agent's understanding of the world (his world represented by an epistemic model). In order to achieve the goal ϕ_g the agent needs to make all the above mentioned moves, that is, *GoRight* until reaching t_7. Intuitively looking down on the problem from above, this will solve the problem, as the agent, having moved all the way to the right, will *know* where the goal is, by having acquired knowledge regarding all the possible goal tiles.

Illustrated in the planning language, the above mentioned plan will look like plan π_1 below. Furthermore three additional plans are shown in order to illustrate the differences between weak and strong planning and how the different formulations translate from the view of the agent.

$\pi_1 =$ GoRight ; GoRight ; GoRight ; GoRight

$\pi_2 =$ GoDown ; GoRight ; GoUp

$\pi_3 =$ GoRight ; GoRight

$\pi_4 =$ GoRight ; GoRight ; **if** Kg_1

then GoUp

else GoRight ; **if** Kg_2

then GoDown

else GoRight ; GoUp

Consider again the domain PARTIAL and the model $\mathcal{M}_3$ as shown below, where the agent is located at tile t_0 with no knowledge of where the goal is located.

$\mathcal{M}_3$:

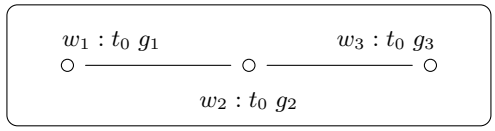


Figure 3.1: A model of the domain PARTIAL where the agent is located at the initial position t_0. *(reflexive and transitive edges are omitted)*

We consider two problems, $\mathcal{P}_1 = (\mathcal{M}_3,\ A,\ (t_2 \wedge g_1) \vee (t_5 \wedge g_2) \vee (t_6 \wedge g_3))$ and $\mathcal{P}_2 = (\mathcal{M}_3,\ A,\ Kg_1 \vee Kg_2 \vee Kg_3)$. In the first problem, the objective for the agent is to *go* to the goal tile, whichever one it may be. In the second problem the goal is to merely obtain *knowledge* of where the goal is located. Consider the plan π_2. Let $\pi_2' =$ GoRight ; GoUp and note that $\pi_2 =$ GoDown ; π_2'. Using strong translation of π_2 we get that $\mathcal{M}_3 \models [\![\pi_2]\!]_s \phi_g$ iff $\mathcal{M}_3 \models \langle \text{GoDown} \rangle \top \wedge$

$[\text{GoDown}]\,[\![\pi_2']\!]_s\phi_g$. Since $\mathcal{M}_3 \models \langle\text{GoDown}\rangle\top$ does not hold, π_2 is not a solution to either $\mathcal{P}_1$ or $\mathcal{P}_2$. This is correct since the agent cannot move down from the initial position. Checking if π_3 is a strong solution to $\mathcal{P}_2$ is equal to checking if $\mathcal{M}_3 \models [\![\pi_3]\!]_s\ (Kg_1 \vee Kg_2 \vee Kg_3)$ which translates to

$$\mathcal{M}_3 \models \langle\text{GoRight}\rangle\top \wedge [\text{GoRight}]\,(\langle\text{GoRight}\rangle\top \wedge [\text{GoRight}]\,(Kg_1 \vee Kg_2 \vee Kg_3))$$

In the same way we can see that π_3 is not a solution to $\mathcal{P}_1$ and that π_4 is a strong solution to both $\mathcal{P}_1$ and $\mathcal{P}_2$.

Now that planning problems has been formulated in an epistemic planning domain we can start looking into how a plan is made. To this end, we first need to specify how to structure the epistemic models (belief states of the agent) produced by product updating the initial state with event models from the action library A.

3.4 Planning Trees

When creating plans the new belief states (epistemic models) constructed are stored in a labelled AND-OR tree, a well known model for planning under uncertaincy [TB12b, MG04]. These AND-OR trees related to planning are called *planning trees*.

3.4.1 Planning Tree - Definition

A planning tree is a finite labelled AND-OR tree, in which each node n is labelled by an epistemic model $\mathcal{M}(n)$, and each edge (n, m) leaving an OR-node is labelled by an event model $\mathcal{E}(n, m)$.

The planning tree T for planning problem $\mathcal{P} = (\mathcal{M}_I, A, \phi_g)$ is then constructed by letting the initial state $(\mathcal{M}_I)$ be the root of the tree T, where the root $root(T)$ is an OR-node. Until expanded, the tree only consists of the root. In order to expand the tree, the following Tree Expansion Rule [TB12b] must be adhered to.

3.4.2 Tree Expansion Rule

Let T be a planning tree for a planning problem $\mathcal{P} = (\mathcal{M}_I, A, \phi_g)$. The tree expansion rules are then defined as follows. Pick an OR-node n in T and an event model $\mathcal{E} \in A$ applicable in $\mathcal{M}(n)$ with the proviso that $\mathcal{E}$ does not label any existing outgoing edges from n. Then:

1. Add a new node m to T with $\mathcal{M}(m) = \mathcal{M}(n) \otimes \mathcal{E}$, and add an edge (n, m) with $\mathcal{E}(n, m) = \mathcal{E}$.
2. For each equivalence class $[w]_R$ in $\mathcal{M}(m)$, add an OR-node m' with $\mathcal{M}(m') = [w]_R$ and add the edge (m, m').

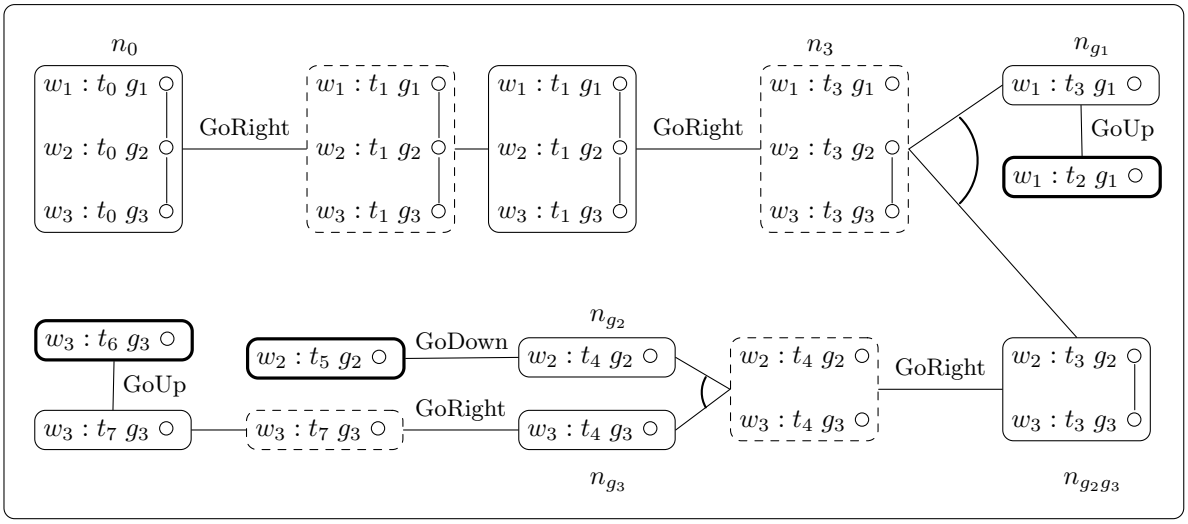


Figure 3.2: Planning tree for the initial domain in Figure 1.1 with the agent starting in t_1.

As can be seen in Figure 3.2, each time the agent has moved next to a goal tile, his belief about the world changes and he is able to distinguish between the goal states. The second *GoRight* action of the agent puts him in tile t_3, where he can see if the goal is located on tile t_2. In the planning tree the posibility of the goal being located on t_2, is noted by the splitting from the AND-node into two new OR-nodes, namely the model where t_2 *is* a goal state marked in bold, and the model where the goal state is located on either t_5 or t_6. This is done by the use of the second part of the expansion rule in the definition,

because world w_1 and worlds w_2, w_3 are in different equivalence classes . More specifically, in the initial state $\mathcal{M}(n_0) \models \neg Kg_1 \wedge \neg K\neg g_1 \wedge \neg Kg_2 \wedge \neg K\neg g_2 \wedge \neg Kg_3 \wedge \neg K\neg g_3$, whereas after the second *GoRight* action the agents belief is $\mathcal{M}(n_3) \models (K(g_1) \vee K(\neg g_1)) \wedge \neg Kg_2 \wedge \neg K\neg g_2 \wedge \neg Kg_3 \wedge \neg K\neg g_3$.

The next step, in order to ensure that the tree expansion terminates, the following *saturation rule* will be defined as in [TB12b].

β_1 The tree expansion rule may not be applied to a node n for which there exists an ancestor node m with $\mathcal{M}(m) \leftrightarrows \mathcal{M}(n)$[1].

When expanding the tree and adhering to the above specified saturation rule β_1, the resulting planning tree *will* be finite, as according to the proof in [TB12b]. Furthermore, since we in the planning tree are only looking for solved nodes, we can extend the β_1 saturation rule with the following argument [TB12b].

β_2 The tree expansion rule may not be applied to a node n if one of the following holds: 1) n is solved; 2) n has a solved ancestor; 3) n has an ancestor node m with $\mathcal{M}(m) \leftrightarrows \mathcal{M}(n)$.

A solved node n is defined by:

- $\mathcal{M}(n) \models \phi_g$ (the goal formula is satisfied in n).
- n is an OR-node, having at least 1 solved child.
- n is an AND-node, having all of its children solved.

In the implementation in order to give a view of the fully expanded planning tree, the second rule of β_2 will be relaxed.

To give a more practical view of AND-OR trees, think of a game of chess against a computer opponent. The OR-nodes in the tree is the human player making a move (the agent doing an action). The AND-nodes are the computer making a counter move, an action you have no control over (the environment picking one of the equivalence classes of the AND-node).

[1] Here $\mathcal{M}(m) \leftrightarrows \mathcal{M}(n)$ denotes that $\mathcal{M}(m)$ and $\mathcal{M}(n)$ are bisimilar according to definitions of bisimulation [DH08].

3.4.3 Making the Plan

Given a solved node in the planning tree we can create a plan for the agent by the following algorithm [TB12b].

- if $\mathcal{M}(n) \models \phi_g$, then $\pi(n) = \text{skip}$.
- if n is an OR-node and m its solved child, then $\pi(n) = \mathcal{E}(n, m); \pi(m)$.
- if n is an AND-node with children $m_1, \ldots, m_k$, then $\pi(n) =$ **if** $\delta_{\mathcal{M}(m_1)}$ **then** $\pi(m_1)$ **else if** $\delta_{\mathcal{M}(m_2)}$ **else** $\ldots$ $\delta_{\mathcal{M}(m_k)}$ **then** $\pi(m_k)$, where $\delta_{\mathcal{M}(m_i)}$ is the characterising formula defined below [TB12b].

Definition (Characterising formula): Let $\mathcal{M} = (W, R, V)$ denote an a model with a single equivalence class on $\mathcal{L}_{DEL}(P)$. For all $w \in W$ we define a formula ϕ_w by: $\phi_w = \bigwedge_{p \in V(w)} p \wedge \bigwedge_{p \in P - V(w)} \neg p$. The characterising formula for $\mathcal{M}$, $\delta_{\mathcal{M}}$, is defined as: $\delta_{\mathcal{M}} = K(\bigwedge_{w \in W} \neg K \phi_w \wedge K \bigvee_{w \in W} \phi_w)$.

The characterising formula is used in the respect, that when the environment simulates which situation turns out to be true (cf. Figure 2.6, if g_1, g_2 or g_3 holds in the given model) the characterising formula will *describe* that world exactly by listing the propositional symbols that needs to be true and listing the propositional symbols that needs to be false. This makes it excellent for the conditional of the if-statement of the plan algorithm, to choose the right sub-plan to execute.

When wanting to extract the plan for a solved node with the goal of the agent to *go* to the actual goal tile, the above definition can be used. We see that $\pi(n_0) =$ GoRight ; GoRight ; **if** $\delta_{\mathcal{M}(n_{g_1})}$ **then** GoUp **else** GoRight ; **if** $\delta_{\mathcal{M}(n_{g_2})}$ **then** GoDown **else** GoRight ; GoUp. It is easy to see that this is a strong solution for getting to the goal state $(t_2 \wedge g_1 \vee t_5 \wedge g_2 \vee t_6 \wedge g_3)$ from the initial state as seen earlier.

3.5 Strong Planning Algorithm

At this point we are ready to give the algorithm for synthesising strong plan solutions for planning problems. The algorithm is given below [TB12b].

StrongPlan($\mathcal{P}$)

1. Let T be the planning tree only consisting of *root(T)* labelled by the initial state of $\mathcal{P}$.

2. Repeatedly apply the tree expansion rule of $\mathcal{P}$ to T until the tree is β_2-saturated.
3. If *root(T)* is solved, return $\pi(\textit{root(T)})$, otherwise return FAIL.

Lets say, in the environment the goal is located on g_2. Using StrongPlan($\mathcal{P}$) we want a plan for the agent. The first step in the algorithm has T being the planning tree consisting only of a single state, namely the initial state seen on Figure 3.1. This model is the root of T. The recursive application of the tree expansion rule results in the β_2-saturated tree seen in Figure 3.2 (The action *GoLeft* has been omitted in the tree, as the resultant nodes would be bisimilar and thus go against β_2.

The third step is to extract the plan from the tree, and traversing the tree gives us the plan that was shown earlier, namely $\pi(n_0)$ = GoRight ; GoRight ; **if** $\delta_{\mathcal{M}(n_{g_1})}$ **then** GoUp **else** GoRight ; **if** $\delta_{\mathcal{M}(n_{g_2})}$ **then** GoDown **else** GoRight ; GoUp.

When the agent tries to use the plan to find the goal, the first two *GoRight* actions will be executed. Then the agent will have perceived that the goal does *not* exist on tile t_2 and therefore the first **if**-statement is false, leading to another *GoRight* action. The second conditional (**if** $\delta_{\mathcal{M}(n_{g_2})}$) is true however, due to the fact that $\delta_{\mathcal{M}(n_{g_2})}$ describes the goal model containing g_2 *exactly*, leading to the agent ending with a *GoDown* and successfully reaching the goal.

3.6 Weak Planning Algorithm

The weak plan serves to give us an alternative if no strong plans are possible. In order to make use of the weak planning algorithm, we need to make just a few changes to the algorithm already in place. Firstly, a weakly solved node in opposition to a strongly solved node, is any node that either satisfies $\mathcal{M} \models \phi_g$ or has as child another weakly solved node.

To extract a weak plan for a planning tree $\mathcal{P} = (\mathcal{M}_I,\ A,\ \phi_g)$, do this recursively by:

- if $\mathcal{M}(n) \models \phi_g$, then $\pi_w(n) =$ **skip**.
- if n is an OR-node and m its weakly solved child, then $\pi_w(n) = \mathcal{E}(n, m)$; $\pi_w(m)$
- if n is an AND-node and m its weakly solved child, then $\pi_w(n) = \pi_w(m)$

The WeakPlan($\mathcal{P}$) algorithm then looks like:

1. Let T be the planning tree only consisting of *root(T)* labelled by the initial state of $\mathcal{P}$.
2. Repeatedly apply the tree expansion rule of $\mathcal{P}$ to T until the tree is β_2-saturated.
3. If *root(T)* is weakly solved, return $\pi_w(\textit{root(T)})$, otherwise return FAIL.

Following the same example as given with the StrongPlan($\mathcal{P}$) algorithm, we reach the planning tree on Figure 3.2. Extracting the plan for the agent in the weak plan mode, yields $\pi(n_0)$ = GoRight ; GoRight ; GoUp.

The reason for this, is that since all the nodes in the tree are weakly-solved as per above definition of weakly-solved nodes, the agent just needs to reach a node where a *possible* goal is located. Since the closest goal is at t_2, the shortest weak plan is $\pi(n_0)$ = GoRight ; GoRight ; GoUp.

3.7 Strong Planning for Non-Determinism

Epistemic planning domains with a component of non-determinism cannot be strongly solved in the normal regard, because of the aspect of randomness in the stochastic process. In NonDet domain given in Figure 2.10(a) with the three teleporters, one cannot be sure to end up at the correct teleporter (tile t_4) for reaching the goal (tile t_5).

In these cases a strong plan cannot be made, as the plan might not work every time due to the non-determinism and thus defies the definition of a strong plan. In this case we will need to relax the strong planning algorithm in order to accomodate an aspect of looping an action or a sequence of actions, until a condition (the characterising formula for the target model) has been fulfilled. However, this has not been implemented into the program and is on the list of possible extensions in the section "Future Work".

Chapter 4

Implementation

In order to implement dynamic epistemic logic and conditional planning within DEL, we must first consider the important aspects of an implementation. How are we going to load in example domains to the program. How is the data going to be stored in the program (data structure) and how is the data going to be represented (outputted). In this and the following sections the implementation will be described in detail. The first section will describe in depth how the epistemic models, event models and formulas are written in order for the program to be able to read them. The second section will describe how the data structure in the program should be created (not language specific) in order to optimally process the data. The third section on the planning process will describe how the program uses the input and processes this information in order to create a planning tree and a plan. Lastly the fourth section will illustrate the output of the program, namely the display of the planning tree upon succesful completion of the planning process as well as the plan.

4.1 Input

For the program to be able to receive input from a text file, we first need a way to define the textual input. This has been done with the assistance of a parser generator, specifically [ant], which is a left-to-right, leftmost derivation parser generator. This is used because we wanted to define the syntax of the text input in specifics and `antlr` allows for this. The grammar created in antlr is available in its entirety in the appendix (A). An excerpt of the grammar is shown in Figure 4.1(a) and 4.1(b)).

In Figure 4.1(a) the grammar shows that a correct input consists of a title, an optional list of propositional symbols used in the domain, an initial `model`, a number of `event models` and lastly a goal `formula`. The way that the models, event models and formulas has been defined so far has been with mathematical notation, however this notation is not suitable for textual input. In order to input models, event models and formulas into the program, we first need to introduce a textual notation for these. This notation

```
public cep :
        'Title:' title
        PROPOSITIONALSYMBOLS propositionalsymbol+
        model
        eventmodel*
        goal = formula?
        ;
```

(a) The grammar of the text input `cep` (conditional epistemic planner).

```
expr returns [Formula f] :
        (propositionalsymbol
      | NOT expr
      | K expr
      | LPAREN expr RPAREN)
      ((AND expr) |
      (OR expr))*
      ;
```

(b) The grammar of the `formula`, showing that formulas are polymorphic.

Figure 4.1: Two examples of the grammar used to parse input from a text file.

will also be used at a later stage in order to print out the models, event models and formulas from the program.

The reason for this notation is that non-ascii symbols like $\neg$, $\wedge$, $\top$, $\bot$ are not possible to type in ascii as text input. The following table shows what ascii-chars are being used in place for the mathematical symbols.

$\mathcal{MATH}$	$\vee$	$\wedge$	$\top$	$\bot$	$\neg$
Ascii	|	&	T	F	$\sim$

Table 4.1: The textual ascii representation of the used mathematical symbols.

In the next section the string representation of epistemic models will be described in detail.

4.2 String Representation of Models

In the program a representation of the epistemic models were needed, not only to quickly be able to uniquely identify the worlds, but also in order to have a good visual overview when printing out the resultant planning tree. In order to compare models, equivalence classes, worlds and propositional symbols a representation was needed, adheering to the definition of bisimilarity. In the following will be defined the string representation of the entire epistemic model, split up in the components constituting the model.

Below is a grammar outlining the string representation of the models. For each non-terminal γ in the BNF we define $\mathcal{L}(\gamma)$ as the language of the subexpression in the BNF.

The `$data` symbol in the last line of the grammar, refers to any propositional symbols given in the textual input, that is, `$data` $\in P$ (see Section 2.2), although from now on

$$
\begin{array}{rcl}
\textit{model} & ::= & \texttt{'('} + \textit{equivalence classes} + \texttt{')'} \\
\textit{equivalence classes} & ::= & \textit{ec} \mid \textit{ec} + \texttt{','} + \textit{equivalence classes} \\
\textit{ec} & ::= & \texttt{'<'} + \textit{worlds} + \texttt{'>'} \\
\textit{worlds} & ::= & \textit{world} \mid \textit{world} + \texttt{','} + \textit{worlds} \\
\textit{world} & ::= & \texttt{'['} + \textit{propositional symbols} + \texttt{']'} \\
\textit{propositional symbols} & ::= & \textit{ps} \mid \textit{ps} + \texttt{','} + \textit{propositional symbols} \\
\textit{ps} & ::= & \texttt{\$data}
\end{array}
$$

Figure 4.2: The BNF for the string representation of the epistemic model, equivalence classes and propositional symbols.

we assume that P is a set of propositional symbols that can be displayed as strings, allowing the check for bisimilarity, and following the standard notation for naming variables, i.e. consisting of only digits and lower- and uppercase letters, with the first character being a non-digit.

Below is then given the definition for the string representation of a model. **Definition:**

- The canonical string representation of a world $w \in W$ is the string *str(w)* given by $\texttt{[}p_1, p_2, \ldots, p_n\texttt{]} \in \mathcal{L}(world)$, where $p_1, p_2, \ldots, p_n$ is the set $\{p \in P \mid w \in V(p)\}$ ordered lexicographically.
- The canonical string representation of an equivalence class $[w]_R$ (see [ecs]) in $\mathcal{M}$ is the string *str($[w]_R$)* given by $\texttt{<}str(w_1), str(w_2), \ldots, str(w_m)\texttt{>} \in \mathcal{L}(ec)$, where $[w]_R$ are the elements of the set $\{str(v) \mid v \in [w]_R\}$ ordered lexicographically.
- The canonical string representation of an epistemic model $\mathcal{M} = (W, R, V)$ is the string *str($\mathcal{M}$)* given by $\texttt{(}str([w_1]_R), str([w_2]_R), \ldots, str([w_k]_R)\texttt{)}$, where $[w_1]_R$, $[w_2]_R$,..., $[w_k]_R$ is the set of equivalence classes in $\mathcal{M}$ and $str([w_1]_R)$, $str([w_2]_R)$, ..., $str([w_k]_R)$ are ordered lexicographically. Recall here that $[w]_R$ is interpreted as the set of worlds related to w by R.

Lemma 1: Bisimulation on string representation
Given two epistemic models $\mathcal{M}_1$ and $\mathcal{M}_2$, $str(\mathcal{M}_1) = str(\mathcal{M}_2)$ *if* $\mathcal{M}_1 \leftrightarroweq \mathcal{M}_2$.

Proof-Sketch:
In the simple case, if two models $\mathcal{M}$ and $\mathcal{M}'$ are bisimilar and consist of only one equivalence class with one world each, these model will be represented with the same string due to the lexicographical ordering of the worlds within the model and propositional symbols within the world.

Now, consider the case that two models $\mathcal{M} = (W, R, V)$ and $\mathcal{M}' = (W', R', V')$ has a single equivalence class each, $[w]_R$, $[w']_{R'}$ respectively, but with multiple worlds present in the equivalence classes. If $\mathcal{M} \leftrightarroweq \mathcal{M}'$ this means that $[w]_R \leftrightarroweq [w']_{R'}$. If the two equivalence classes are bisimilar, then for each world $w \in [w]_R$ there exists a

world $w' \in [w']_{R'}$ so that $str(w) = str(w')$ and for each world $w' \in [w']_{R'}$ there exists a world $w \in [w]_R$ so that $str(w') = str(w)$. By the lexicographical ordering of the worlds within $[w]_R$ and $[w']_{R'}$, this means that $str([w]_R) = str([w']_{R'})$.

In the extended case, where two models $\mathcal{M}$ and $\mathcal{M}'$ are bisimilar and consist of more than one equivalence class, for each equivalence class in $\mathcal{M}$ there will be an equivalence class in $\mathcal{M}'$ represented with the same string and for each equivalence class in $\mathcal{M}'$ there will be an equivalence class in $\mathcal{M}$ represented with the same string. Since comparing the models are now reduced to comparing two sets of strings, these can be ordered lexicographically and string comparison gives us that if the two models are bisimilar, their string representation will be equal.

4.2.1 Epistemic Models

Using the above definition of string representation of the epistemic models, equivalence classes and worlds, the example below has been given. In the example, made from model $\mathcal{M}_2$ in Section 2.4.1, the propositional symbols will be assembled first, then the worlds, classes and lastly the model. This is done in order to be able to take the example one step at a time.

1. Propositional Symbols

The atomic form of an epistemic model comes down to the propositional symbols present in the worlds of the model. Using the example epistemic model $\mathcal{M}_2$, the world w_1 consists of only two symbols, namely t_1 and g_1. In the string representation, each of the propositional symbols will be written, seperated by commas, ordered lexicographically. `g1,t1`.

2. Worlds

Worlds, containing the propositional symbols (or being empty) will add to the string representation a set of square brackets "`[`" "`]`", enclosing the propositional symbols belonging to the specific world. Multiple worlds in an epistemic model are comma seperated just like with the propositional symbols and again ordered lexicographically. Thus the three worlds in model $\mathcal{M}_2$ can be written: `[g1,t1],[g2,t1],[g3,t1]`.

3. Equivalence Classes

Recall that worlds connected with an equivalence relation are indistinguishable to the agent. Take as example the updated model $\mathcal{M}_2 \otimes \mathcal{E}_2$ in Figure 2.9, the agent has executed the action *GoRight* twice and has discovered whether or not the goal exists at location t_2. Now the world (w_1, gr_{11}) is not related to the worlds (w_2, gr_{12}) and

(w_3, gr_{12}) by the equivalence relation anymore and is as result not in the same equivalence class. Each equivalence class in the epistemic model will be adding a pair of angled brackets to the string representation "<" ">" enclosing worlds related via the equivalence relation. Multiple equivalence classes in a model are comma seperated and ordered lexicographically. Thus $\mathcal{M}_2 \otimes \mathcal{E}_2$ is written as: `<[g1,t3]>,<[g2,t3],[g3,t3]>`, and of course the original model $\mathcal{M}_2$ will be written with only one equivalence class: `<[g1,t1],[g2,t1],[g3,t1]>`

4. Epistemic Models

Lastly, even though each epistemic model is well isolated from one another in different nodes, a pair of parentheses '(' ')' will be added to the string represention to complete the string.

$\mathcal{M}_2$: `(<[g1,t1],[g2,t1],[g3,t1]>)`
$\mathcal{M}_2 \otimes \mathcal{E}_2$: `(<[g1,t3]>,<[g2,t3],[g3,t3]>)`

And we have the full textual representation of the models $\mathcal{M}_2$ and $\mathcal{M}_2 \otimes \mathcal{E}_2$.

4.2.2 Storing

After the input has been given to the program, the next logical step is to determine how this information should be stored. How should the models be stored? The planning tree? The following sections will define for each of the important aspects of conditional espistemic planning exactly how the different parts are stored in the program and why these choices has been made. For most of this discussion the methods of storing are language *independant*, however in the case of formulas it is restricted to the set of languages including the aspect of polymorphism[pol].

4.3 Data Structure

To ensure that the program has the optimal conditions for a fast plan calculation, the information necessary to the plan need to be stored in data structures optimal to the process. In the following sections the different components to an epistemic planning domain will be described in detail with regards to the data structures. The runtime operation cost of the different procedures using the data structures will be detailed later in the section "Algorithms" and just briefly mentioned here.

4.3.1 Epistemic Models

When planning the data structure for the epistemic models, we need to consider how these models will be used, which operations are important and need priority on diminishing runtime cost. The models will first and foremost be a part of the product update functionality, model checking[] and model comparison (checking for bisimilar models when updating). As explained in the previous section, the latter is achieved by having a string representation of the model which can be compared to other models by string comparison. The runtime of the string representation of the node can be further enhanced by caching[cac] of the model's string represention. More on this in the section "Future Works". In terms of product update functionality we need to consider which parts of the model will be used when updating.

An initial thought regarding the data structure of the models, is that it would be nice to be able to access any model in instant or close to instant time. Immediately hashing[THC01] of the string representation of the models and storing in hash-maps comes to mind, which would allow for just that. However, as algorithms later will show, the nature of the product update is such that we will not have use for accessing any particular model at any one point. When updating, all nodes in the planning tree, which are not yet expanded, are updated at the same time, and thus we have no use for the intricate setup of hash-maps and storing of the models, as we have no need to access them directly.

Instead, the epistemic models in the program are stored within the planning tree (planning trees will be discussed in later in this chapter). The models will need to contain the worlds in the models as well as the relationship between the worlds (equivalence relation). This is done by treating the model as a set of equivalence classes, and in turn have the equivalence classes be sets of worlds. Each model can then be considered a set of sets, much like the string representation shown earlier. Just like with the models, we have no need to access a specific world in an equivalence class, nor a specific equivalence class in a model. Therefore we just need to argue that each product update does indeed warrant an iteration over all the worlds and that no world is accessed twice when applying a specific event.

```
          epistemic model  =  list ( equivalence classes )
        equivalence class  =  list ( worlds )

              event model  =  list ( event equivalence classes )
  event equivalence class  =  list ( events )
```

Figure 4.3: The pseudo code for the data structure of the epistemic models and event models.

The argument that a model as a set of sets in terms of the global product update function does *not* increase the calculation time from an intricate hash-mapping system,

is given on the basis that in a product update between an epistemic model $\mathcal{M} = (W, R, V)$ and an event model $\mathcal{E} = (E, Q, pre, post)$, *all* worlds $w \in W$ and *all* events $e \in E$ will have to be iterated through and paired to check if an update between w and e is applicable. This is the case in whichever approach is chosen. Since w can only belong in one equivalence class and the iteration happens over the worlds in each equivalence class, w will at any time only be accessed *once* per world-event-update.

Lastly we need to consider the model checking of a model, as this functionality will be used extensively when the agent plans.

4.3.2 Formulas

The data structure of the formulas has been chosen very specifically, as this is one of the functions that will be executed a lot. Every time model checking happens, either because a world is tested to see if a goal formula is satisfied or because a world is having an event applied, the checking of a model with a formula will occur. Another thing to keep in mind is that the search depth of the formulas can be any size and thus it is important to keep in mind, how model checking is done within the data structure. This being said, model checking is a vast area and will only be detailed to the extend used in the implementation. In the program, the formulas, as well as the

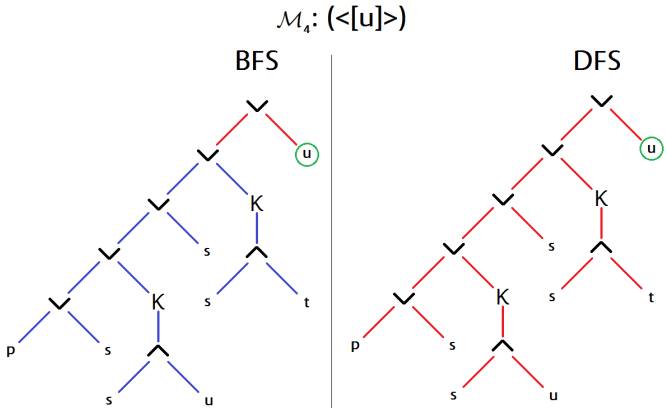


Figure 4.4: Example 1: The formula tree is primed towards a BFS solution, finding that $\mathcal{M}_4$ holds in 2 steps, while the DFS takes 16 steps to find same outcome.

nodes in the planning tree (see Figure 3.2) are built on a polymorphic data structure.

Each formula derives[1] from a parent class and each formula must override[2] the parent method to check if the formula holds in a certain world. An example of this is given in Figures 4.4 and 4.5. On top of this data structure a queue system is built, which will be loaded on specific search conditions (DFS or BFS, can be extended with other search algorithms). This means that the search method of the model checking of a model is not predetermined to be either BFS or DFS, but is given by the user. Below is shown an example of a model checking on two different models with the same goal formula $g_1 = ((((p \vee s) \vee K(s \wedge u)) \vee s) \vee K(s \wedge t)) \vee u$.

As can be seen in the binary formula tree in Figure 4.4, the model $\mathcal{M}_4$ is being checked for the formula g_1 given earlier.

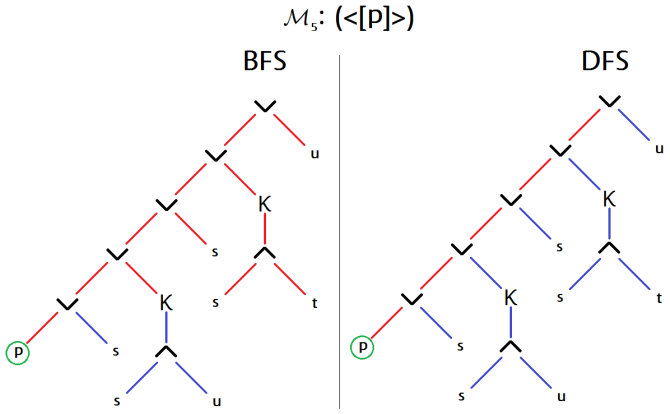


Figure 4.5: Example 2: The formula tree is less optimal for a BFS solution, finding that $\mathcal{M}_5$ holds in 12 step, while the DFS finds same solution in the optimal 5 steps.

To summarize, the data structure chosen for the formulas within the program has been chosen because it offers the greatest range of customization for the user as well as being the optimal in terms of number of computations. This gives the user the possibility of trying out several search algorithms for finding the fastest solution for extensive problems.

[1]In Object Oriented Programming (OOP), a derived class *inherits* functionality from a base class. In this case functionality common to all formulas, namely a `Check()` procedure is inherited along with any sub-formulas in the data structure.

[2]In OOP, overriding a derived procedure replaces the functionality of said procedure with functionality specific to the derived class.

4.3.3 Planning Tree

As with the argument for having models stored as sets of sets (sets of equivalence classes) that no specific world in a model is interesting outside of the product update queue of the update algorithms, a similar argument can be made for the models in the planning tree. Each of the models stored in the nodes of the planning tree is only considered when its node is expanded in accordance to the planning algorithm. However these models are not referenced by any function or method outside of the planning algorithm, and thus no reference to any specific model is necessary except for accessing the goal-nodes when found. This means that all nodes in the planning tree will only be reference via the root of the tree (again, except the goal-nodes, which will be the intial nodes in the **ExtractPlan(**$\cdots$**)** algorithm).

The nodes in the planning tree, can be either AND-nodes or OR-nodes. Each of these will contain a list of child nodes which will be added and accesses during expansion, evaluation and extracting of the plan. Furthermore each node will also have a string representation similar to that of the string representation of the epistemic model. The string representation of a node will merely add an A or and O to the grammar for the string representation of the model, as seen, in the extention of the grammar, below:

$$\begin{array}{rcl} \textit{node} & ::= & \textit{and-node} \mid \textit{or-node} \\ \textit{and-node} & ::= & \texttt{'A'} + \textit{model} \\ \textit{or-node} & ::= & \texttt{'O'} + \textit{model} \\ \textit{model} & ::= & \texttt{'('} + \cdots + \texttt{')'} \end{array}$$

Figure 4.6: The extended BNF for including the string representation of the nodes in the planning tree to the BNF seen earlier.

The main reason for this is when expanding a node n with the available event models from $D(\mathcal{E})$ in the **Evaluate(**$\cdots$**)** algorithm, if new childode n_c already exists (is bisimilar with an existing node m in the planning tree) then n_c need not be expanded as m will already be in the queue for expansion. The foundation for this check for already existing nodes is done by storing the string representations of the nodes in a hash-map[THC01], in order to supply a reference to m given its string representation (in this case equal to the string representation of n_c).

4.3.4 Plans

Once the tree has been expanded (the algorithm for this is shown in Section 4.4.2) we are either left with a set of goal nodes in which the goal formula holds, or FAIL is returned. If the former, we are then left with a set of goal nodes in which the goal formula holds. From these goal nodes, in order to extract the plan, each expanded node has a reference to its parent node (included both model and event model). As

examples later will show, this enables us to label the tree with the event models from the goal nodes to the root of the tree and extracting the sequence of events that completes the plan.

4.4 Planning

Having defined the data structure and way of storing models, formulas and plans, we can now progress to the actual planning. The planning process in the program is a major part of the functionality in the program. In the following sections it will be described in detail how an agent starting with an initial model, expands the planning tree using the algorithms given in the theory (and [TB12b]), model checks on each iteration to check if the goal formula is satisfied, and lastly when the tree has been fully expanded[3], extracts the plan using the plan-extraction algorithm.

Each of the algorithms given in this chapter are language independent however they are designed towards an object oriented[oop] language. Function calls like `default(<class>)` has been included for instantiating the `<class>`, which most object oriented languages requires, but beyond that has no effect other than making the type of the variable known.

The algorithms in this chapter will be displayed in the order in which they are employed.

4.4.1 Global Planning

After the list of propositional symbols, initial model, set of event models and goal formula has been input into the program, the agent can start the planning process of how to complete the problem. This process is divided into multiple smaller tasks which will be detailed next. Specifically these tasks are the expansion of a node, whether AND-node (splitting) or OR-node (product updating), evaluation of a given node or subnodes and lastly the extraction of the plan, if one exists, when the planning tree has been fully expanded.

As seen in algorithm **Plan(**$\cdots$**)** shown in Algorithm 1, in order for the agent to start planning he first needs to initialize the planning tree, that is, creating the root of the AND-OR-tree and setting the model of the root to the given initial model $\mathcal{M}_I$. Once this has been completed, the evaluation of the tree starts. After the tree has been fully expanded and the goal nodes are found (if any), the agent proceeds to the extraction of the plan. This method is the main method of the program, meaning that once the

[3] As mentioned earlier, in oder to show the entire planning tree of the problems given, the second part of the saturation rule β_2 has been relaxed. In an runtime-optimized version of the program where the full plannning tree is not important, this restriction can be restored.

Algorithm 1: Plan(m, ems, g, pt) - Given a planType *pt* (weak or strong), an initial model *m* and a set of event models *ems*, execute the planning process on the given domain searching for the goal formula *g*.

input : m, ems, pt, g, the model m on which to start the pt-typed (weak/strong) planning process to achieve the goal g, using ems.
output: plan, the plan for the agent.

```
1 queue ← default(Queue);
2 tree ← CreateTree(m);
3 Evaluate(tree, ems, g, pt);
  // At this point all the goalnodes has been found.  A plan is
     achieved by walking up the tree, starting from the goal
     nodes.
4 plan ← ExtractPlan(goalNodes, tree, ems, pt);
5 return plan;
```

program has received the input, the program starts and ends with this method. It is also responsible for calling the functionality of all the subsequent algorithms shown in this section. The order of the call stack is shown in Figure 4.7.

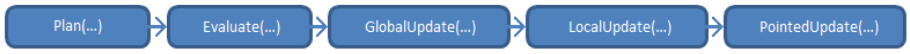


Figure 4.7: Call order for the methods mentioned: Plan, Evaluate, GlobalUpdate, LocalUpdate, PointedUpdate.

The reason for this call order, is that the agent when starting his planning, calls **Plan(**$\cdots$**)**. In the **Plan(**$\cdots$**)** method, the call to **Evaluate(**$\cdots$**)** happens when commencing the planning process on the planning tree. The three update functions are different in the regard that **GlobalUpdate(**$\cdots$**)** applies all possible event models in the domain to the current model when expanding a node. **LocalUpdate(**$\cdots$**)** applies a specific event model to the current model, and **PointedUpdate(**$\cdots$**)** pairs a specific world in the model, with a specific event in the event model in order to update the world.

The function calls of line 1 and 2 in Algorithm 1 are responsible for initializing the planning process. Line 1 initializes the static queue from which all the **Next(**$\cdots$**)** calls of the **Evaluate(**$\cdots$**)** function dequeues from. Line 2 intializes the planning tree, that is, creates the root node with the model *m* on which **Evaluate(**$\cdots$**)** function subsequently will be called.

4.4.1.1 Runtime

The runtime of **Plan(···)** depends on the runtime of the call to **Evaluate(···)** as well as the runtime of the **ExtractPlan(···)** algorithm. The runtimes of these two algorithms will be discussed in the corresponding sections. A summary of the runtimes will be given towards the end of the section when all the sub-runtimes has been computed.

4.4.2 Evaluation

The evaluation of a node in the planning tree is a recursive function as shown in the pseudo code in Algorighm 2 and 3 (due to its size it has been split into two parts in order to fit on the page). This method contains the logic for walking the planning tree as according to the set plantype (weak or strong planning).

Algorithm 2: Evaluate(n, ems, g, pt) - Recursively evaluate the model of the current node n in regards to the goal g and the planType pt, using the event models *ems* to update. *Continues in Algorithm 3.*

input : n, ems, g, pt, the node n, with its containing model, which is to be evaluated for g. Recusive check using event models ems to expand the pt-typed tree.
output: bool, true if the node n is solved, false otherwise.

```
1  if n is null then return true;
2  if n.IsSaturated then
3  |  Evaluate(Next(), ems, g, pt);
4  |  return n.Value;
5  end
6  if n.Model.Check(g) then
   |  // The model is satisfying the goal condition
7  |  n.IsSaturated ← true;
8  |  n.Value ← true;
9  |  GoalNodes.Add(n);
10 |  Evaluate(Next(), ems, g, pt);
11 |  return true;
12 end
13 subNodes ← default( list of nodes );
14 if n is AndNode then subNodes.AddAll(Split(n.Model, ems));
15 else if n is OrNode then subNodes.AddAll(GlobalUpdate(n.Model,
   ems));
16 Add(subNodes);
```

Besides the mandatory *null* check, the **Evaluate(···)** method iterates each node in a recursive fashion, checking if a node is saturated. If the node is saturated the (already) computed `Value` is returned. This check is to ensure that a node and containing model is not model checked more than once. Next step (line 6) is the actual model checking on the model of the node. The process of model checking behaves as the model checking explained in Section 4.3.2 "Formulas", namely that each formula behave as a parse tree which can be assigned a specific search method (BFS or DFS) for the model check.

If the model checking of line 6 returns true, the node is marked as saturated, because no further exploration of the sub-tree of the node is necessary. Furthermore the `Value` of the node is set to true for future reference. Lastly the the function returns true, as the goal formula is satisfied in the node.

4.4.2.1 Queue

So far lines 3 and 10 has been skipped, and the reason for this is that both of these contains the method call `Next()`, which is a part of the queue system for the nodes. The system is built up by having an external queue, in which nodes can be added and the next node can be dequeued, whenever needed. The order of the nodes in the queue depends on the search type set statically within the queue system, whether DFS, BFS or A*. This order is determined when the nodes are added to the queue (line 16).

Algorithm 3: Evaluate(n, ems, g, pt) - *Continued from Algorithm 2.*

```
   // At this point, nodes that are not yet saturated, expanded
      or satisfying g will be evaluated
17 Evaluate(Next(), ems, g, pt);
   // At this point all the subnodes of n has returned and the
      n.Value can be set accordingly
18 n.Value ← (n is OrNode || pt is Weak) && HasSolvedSubNode(subNodes)
19          || n is AndNode && not HasNoSolvedSubNode(subNodes) ;
20 return n.Value;
```

The recursive call of **Evaluate(···)** also happens in line 17 at which point the current node is neither saturated nor satisfying the goal formula `g`, and the last option for the node to be considered `Solved`, is by having a solved child node. In the last lines 18 and 19 the node is assigned a solved value based on the values of its children. This means, that if the node is an OR-node and at least 1 of the children is marked as `Solved` or if the plan type is set to *weak* meaning that even if an AND-node has only one solved child, the node will still be marked as `Solved`. Similarly, if the node is an AND-node, and *all* of its children are marked as `Solved`, the node itself will also be marked as `Solved`.

Another significant line in the **Evaluate(···)** algorithm that so far has been overlooked

is line 9, which *adds* the newly discovered goal node to the list of already existing goal nodes. This list of goal nodes is maintained outside of the method, and will be used upon completion of the tree-traversal in order toe generate the plan. More on this in Section 4.4.7. The two unreferenced functions **HasSolvedSubNode(**$\cdots$**)** and **HasNoSolvedSubNodes(**$\cdots$**)** are function to test if a node has any solved children and will not be further elaborated. If an OR-node has no solved children, or if an AND-node has a subnode that is not solved, the value of the current node will be set to false.

4.4.2.2 Runtime

The runtime of the **Evaluate(**$\cdots$**)** algorithm is dependent upon the size of the planning tree. The evaluate method is called once on each node in the tree, model checking and possibly expanding the node. If k is the total number of OR-nodes in the tree (and thus also the number of AND-nodes), $|\texttt{GU}|$ the runtime of the **GlobalUpdate(**$\cdots$**)** algorithm, $|\texttt{MC}|$ the runtime of a single model check and $|\texttt{SP}|$ the runtime of the **Split(**$\cdots$**)** algorithm, then the runtime of **Evaluate(**$\cdots$**)** can be given by:

Algorithm	Alias	Runtime
Evaluate($\cdots$**)**	$\lvert\texttt{Eval}\rvert$	$O(k \cdot \lvert\texttt{GU}\rvert \cdot \lvert\texttt{MC}\rvert + k \cdot \lvert\texttt{SP}\rvert)$

Table 4.2: Runtime of the **Evaluate(**$\cdots$**)** algorithm.

In the runtime analysis of **Evaluate(**$\cdots$**)** shown in Table 4.2 the runtime has been calculated on the basis of the sum of the runtime cost of an expansion of a OR-node *and* an AND-node, because the number of OR-nodes and AND-nodes are the same but the two operations have very different costs.

The runtimes of $|\texttt{SP}|$ $|\texttt{GU}|$ and $|\texttt{MC}|$ will be given in the coming sections.

4.4.3 Global Update

The **GlobalUpdate(**$\cdots$**)** function is called on each iteration of the **Evaluate(**$\cdots$**)** method, whenever an OR-node is expanded. This method is responsible for applying all the given event models to the model of the node being expanded. This is done by a simple iteration of the event models and for each event model calling the **LocalUpdate(**$\cdots$**)** function to apply that particular event model to the model of the node.

Each resultant model from the **LocalUpdate(**$\cdots$**)** function call is stored in a list which is returned at the termination of the **GlobalUpdate(**$\cdots$**)** function. If a resultant model is null, meaning no applicable event models were available on the model, this model will not be returned.

Algorithm 4: GlobalUpdate(m, ems) - Updates a model m with all possible event models *ems*.

input : m, ems, the event models in ems to be applied to the epistemic model m.
output: newModels, the list of new epistemic models after ems has been applied.

```
1  newModels ← default(Model[ ]);
2  index ← 0;
3  foreach EventModel em in ems do
4  |  newModel ← LocalUpdate(m,em);
5  |  if newModel is not null then
6  |  |  newModels [index ] ← newModel;
7  |  |  index ← index + 1;
8  |  end
9  end
10 return newModels;
```

4.4.3.1 Runtime

The runtime of the **GlobalUpdate(⋯)** algorithm is dependant on the size of the action set available ($|A|$), as well as on the runtime of the **LocalUpdate(⋯)** ($|\texttt{LU}|$) algorithm given in the next section. In overall terms, the runtime of the algorithm can be given by:

Algorithm	Alias	Runtime
Evaluate(⋯)	$\lvert\texttt{Eval}\rvert$	$O(k \cdot \lvert\texttt{GU}\rvert \cdot \lvert\texttt{MC}\rvert + k \cdot \lvert\texttt{SP}\rvert)$
GlobalUpdate(⋯)	$\lvert\texttt{GU}\rvert$	$O(\lvert A\rvert \cdot \lvert\texttt{LU}\rvert)$

Table 4.3: Runtime of the **GlobalUpdate(⋯)** algorithm.

4.4.4 Local Update

The function **LocalUpdate(⋯)** is named thusly, because the function applies *one* event model to an epistemic model, hence a local update of the model. The function contains four nested `foreach`-loops, which are responsible for iterating the equivalence classes of both event model and epistemic model.

This may seem expensive in terms of number of runtime operations, however, in any product update event between an epistemic model and an event model, all *pointed* models (i.e. *worlds*) in the epistemic model needs to be paired with all the events in

the event model, in order to test the precondition of the event and if successful the application of the event on the pointed model.

Algorithm 5: LocalUpdate(m, em) - Local update of an epistemic model with an event model.

input : m, em, the event model em to be applied to the epistemic model m.
output: newModel, the new epistemic model after em has been applied.

```
   // Setting parent relationship
 1 newModel ← default(Model);
 2 newModel.ParentModel ← m;
 3 newModel.ParentEventModel ← em;
 4 newModel.Title ← m.Title + em.Title;
 5 foreach EquivalenceClass ec in m.EquivalenceClasses do
 6 |   foreach EventEquivalenceClass eec in em.EquivalenceClasses do
   |   |   // Creating new Equivalence Class
 7 |   |   newEC ← default (EquivalenceClass);
 8 |   |   foreach World w in ec do
 9 |   |   |   foreach Event e in eec do
   |   |   |   |   // Creating new World and adding if not null
10 |   |   |   |   newWorld ← PointedUpdate(w, e, eec, m);
11 |   |   |   |   if newWorld is not null then
12 |   |   |   |   |   newWorld.Title ← w.Title + e.Title;
13 |   |   |   |   |   newEC.Worlds.Add(newWorld);
14 |   |   |   |   end
15 |   |   |   end
16 |   |   end
   |   |   // Adding Equivalence Class if not empty
17 |   |   if newEC is not ∅ then
18 |   |   |   newModel.EquivalenceClasses.Add(newEC);
19 |   |   end
20 |   end
21 end
22 return newModel;
```

The order in which the `foreach`-loops are executed takes care of preserving the equivalence relation between the worlds, that is, if a world w_1 is being applied an event e_1 in the same equivalence class as a world w_2 having applied an event e_2, and the events e_1 and e_2 are related via the event equivalence relation R from the definition of event models, the new worlds (w_1, e_1) and (w_2, e_2) will also be in the same equivalence class. Due to the fact that a world w is always indistinguishable to itself, if w is being applied two related events e_1 and e_2, the updated worlds (w, e_1) and (w, e_2) will also be related, that is, be in the same equivalence class.

In line 7 of Algorithm 5, the new equivalence classes for the updated models are made. This happens once for each pair of epistemic equivalence classes and event equivalence classes, which also determines the number of new equivalence classes in the upated world (less any empty equivalence classes, due to preconditions not being fulfilled).

4.4.4.1 Runtime

The runtime of algorithm **LocalUpdate(⋯)** is again dependant on a sub-call to the algorithm **PointedUpdate(⋯)** which will be described in the next section. Furthermore the runtime of **LocalUpdate(⋯)** is dependant on the number of worlds and event in the epistemic model $\mathcal{M} = (W, R, V)$ respectively the event model $\mathcal{E} = (E, Q, pre, post)$. This yields the runtime seen in Table 4.4.

Algorithm	Alias	Runtime
Evaluate(⋯)	$\lvert\texttt{Eval}\rvert$	$O(k \cdot \lvert\texttt{GU}\rvert \cdot \lvert\texttt{MC}\rvert + k \cdot \lvert\texttt{SP}\rvert)$
GlobalUpdate(⋯)	$\lvert\texttt{GU}\rvert$	$O(\lvert A\rvert \cdot \lvert\texttt{LU}\rvert)$
LocalUpdate(⋯)	$\lvert\texttt{LU}\rvert$	$O(\lvert W\rvert \cdot \lvert E\rvert \cdot \lvert\texttt{PU}\rvert)$

Table 4.4: Runtime of the **LocalUpdate(⋯)** algorithm.

4.4.5 Pointed Update

The last of the three update methods, **PointedUpdate(⋯)** as seen in Algorithm 6, is the core of the product update process. First of all this is where the precondition check happens. The precondition formula of the event e is checked against the world w being updated. If the precondition formula holds in w, the method will move on to create a new world (w, e) by way of copying all of the propositional symbols of $V(w)$ into $V((w, e))$. Then, from the postmapping of e each of the propositional symbols will be copied to (w, e) if not present in (w, e) already. Any negated propositional symbols of the postmapping of e will be remove from (w, e) the symbol, if present.

Algorithm 6: PointedUpdate(world, event) - Pointed update of a world w with an event e.

input : w, e, the event e to be applied to the world w.
output: newWorld, the new world after e has been applied.

```
1  if not e.Precondition.Check(w) then
2  |  return null;
3  end
4  newWorld ← default(World);
5  newWorld.Title ← w.Title + e.Title;
   // Add all the symbols from w
6  foreach PropositionalSymbol ps in w.Symbols do
7  |  newWorld.AddSymbol(ps);
8  end
   // then add the symbols from the postmapping of e if not
      present, or remove if negated
9  foreach PropositionalSymbol ps in e.PostEventSymbols do
10 |  newWorld.ModifySymbol(ps);
11 end
12 return newWorld;
```

4.4.5.1 Runtime

The runtime of the **PointedUpdate(⋯)** is determined by two factors, namely the runtime of the model check that happens in relation to the precondition check of e on w and the number of propositional symbols $p \in V(w)$ and the postmapping of e. Because the propositional symbols in $V(w)$ and the postmapping of e is moved to the new world (w, e) the runtime of this is determined by both.

Algorithm	Alias	Runtime
Evaluate$(\cdots)$	$\lvert$`Eval`$\rvert$	$O(k \cdot \lvert$`GU`$\rvert \cdot \lvert$`MC`$\rvert + k \cdot \lvert$`SP`$\rvert)$
GlobalUpdate$(\cdots)$	$\lvert$`GU`$\rvert$	$O(\lvert A \rvert \cdot \lvert$`LU`$\rvert)$
LocalUpdate$(\cdots)$	$\lvert$`LU`$\rvert$	$O(\lvert W \rvert \cdot \lvert E \rvert \cdot \lvert$`PU`$\rvert)$
PointedUpdate$(\cdots)$	$\lvert$`PU`$\rvert$	$O(2 \cdot \lvert P \rvert + \lvert$`MC`$\rvert)$

Table 4.5: Runtime of the **PointedUpdate**$(\cdots)$ algorithm.

4.4.6 Splitting the AND-node

When it comes to expanding a node, there are two different ways this can happen. The first, expanding an OR-node, has been explained above. Expanding an AND-node is quite different and much simpler as seen in Algorithm 7. Since AND-nodes occur as every second node in the planning tree, after a OR-node has been updated, the splitting of the AND-nodes will also happen at every second node. However the splitting of the equivalence classes within the AND-node, which is really the result of the splitting, can naturally only happen if the previous product update resulted in more than one equivalence class. As seen in **Split**$(\cdots)$ the algorithm is quite simple, iterating over

Algorithm 7: Split(m, em) - Splitting the epistemic model m of an AND-node into OR-nodes, one for each equivalence class present in m.

input : m, the model to be split.
output: newModels, the list of new models, each containing an equivalence class from m.

```
1 index ← 0;
2 newModel ← default(Model);
3 foreach EquivalenceClass ec in m.EquivalenceClasses do
4 |   newModel.EquivalenceClasses.Add(ec);
5 |   newModel.Title ← m.Title + index;
6 |   newModels[index] ← newModel;
7 |   index ← index + 1;
8 end
9 return newModels;
```

the equivalence classes of the model in the AND-node and creating a new model and OR-node for each equivalence class. In this algorithm the naming convention is that each new model from the splitting of an AND-node, will get the name of the previous epistemic model suffixed with an integer index counter starting from 0.

The result of expanding both AND-nodes and OR-nodes, can be seen in Figure 3.2, which shows the planning tree of an agent in the domain PARTIAL. Notice especially that the splitting of the AND-nodes occur twice, namely after the second *GoRight* creating nodes n_{g_1} and $n_{g_2 g_3}$ and then after the third *GoRight* action, creaing nodes

n_{g_2} and n_{g_3}. An example of expanding an OR-node can be seen in the same figure from the first node (initial node) in the tree, expanding the node by applying all possible event models to the epistemic model in that node. The *GoRight* event was the only applicable event and thus only one new AND-node was created.

4.4.6.1 Runtime

The runtime of the **Split(**$\cdots$**)** algorithm is upper bound by the number of equivalence classes in the model of the node, which again is upper bound by the number of worlds $|W|$ in the model.

Algorithm	Alias	Runtime
Evaluate($\cdots$**)**	$\lvert\texttt{Eval}\rvert$	$O(k \cdot \lvert\texttt{GU}\rvert \cdot \lvert\texttt{MC}\rvert + k \cdot \lvert\texttt{SP}\rvert)$
GlobalUpdate($\cdots$**)**	$\lvert\texttt{GU}\rvert$	$O(\lvert A\rvert \cdot \lvert\texttt{LU}\rvert)$
LocalUpdate($\cdots$**)**	$\lvert\texttt{LU}\rvert$	$O(\lvert W\rvert \cdot \lvert E\rvert \cdot \lvert\texttt{PU}\rvert)$
PointedUpdate($\cdots$**)**	$\lvert\texttt{PU}\rvert$	$O(2 \cdot \lvert P\rvert + \lvert\texttt{MC}\rvert)$
Split($\cdots$**)**	$\lvert\texttt{SP}\rvert$	$O(\lvert W\rvert)$

Table 4.6: Runtime of the **Split(**$\cdots$**)** algorithm.

4.4.7 Extracting Plan

When the goal nodes have been found in the planning tree we are ready to extract the plan from the planning tree. Recall Section 3.4.3 detailing how to extract the plan, given a solved node in the planning tree. The implemented algorithm works in a similar way, however the direction of the extraction has been reversed. This is because while expanding the nodes in the planning tree, each node was given a reference to its parent epistemic model and its parent event model (the event model applied to the parent epistemic model in order to create the node). By having this reference, the plan extraction comes down to walking the tree from bottom (goal nodes) and up to the root, contrary to having to search each node for which child is the solved node in order to progress.

Algorithm 8: ExtractPlan(n, plan, planType) - Recursively extract the plan *plan* given the plan type (weak/strong) *planType* and the node n. The plan step of n is calculated and then **ExtractPlan(**$\cdots$**)** is recursively called on the parent. *Continues in Algorithm 9*

input : n, plan, planType, node, planlink, plantype
output: true if the node is solved, false otherwise.

```
1 if n is null then return plan;
2 if n is OrNode || planType is Weak then
3 |   plan.Add(n.Model.ParentEventModel);
4 |   return ExtractPlan(n.ParentNode, plan, planType);
5 end
```

As with the definition in Section 3.4.3, making each step in the plan is dependant on what kind of node is currently being processed as well as what plantype has been selected (Strong or Weak). The algorithm shown in 8 and 9 is a recursive algorithm, extracting for each node the plan-step and then calling itself on the parent node. The different plan-steps is detailed below.

The algorithm starts after the plan-step for the goalnode has been initialized to `Skip`, moving on to the parent node of the goalnode. For any parent node n_p there are two[4] different cases:

- The parent node n_p is an OR-node, which means an event model $\mathcal{E}_p$ of the parent was applied to the model $\mathcal{M}_p$ of node n_p and the child node is the product update $\mathcal{M}_p \otimes \mathcal{E}_p$. Thus the plan-step returned is $\mathcal{E}_p$.
- The parent node n_p is an AND-node, which means the plan-step returned will be an `if-then-else` construct. This construct will be created in the following way:

[4] Actually a third case exists if cyclic solutions are taken into consideration. More on that in the section *Future Work*.

Algorithm 9: ExtractPlan(m, ems, g, pt) - *Continued from Algorithm 8*

input : n, plan, planType, node, planlink, plantype
output: true if the node is solved, false otherwise.

```
6  if n is AndNode then
7      currentStep ← default( if-then-else );
8      returnStep ← currentStep;
9      index ← 0;
10     while index < n.SubNodes.Length do
11         tempStep ← default( if-then-else );
12         tempStep.Condition ← n.SubNodes[index];
13         tempStep.IfStep ← n.SubNodes[index].ParentEventModel;
14         currentStep.ElseStep ← tempStep;
15         currentStep ← tempStep;
16         index ← index + 1;
17     end
18     currentStep ← skip;
19     plan.Add(returnStep);
20     return ExtractPlan(n.ParentNode, plan, planType);
21 end
```

– Given children $n_{c_1}, n_{c_2}, \ldots, n_{c_n}$ of n_p and the plan for each child made thus far $\pi_{n_{c_1}}, \pi_{n_{c_2}}, \ldots, \pi_{n_{c_n}}$, recall the `if`-statement from Section 3.4.3. The characterising formula $\delta_{\mathcal{M}}$ is implemented by the string representation of the model, since this representation is also unique and characterising (cf. proofscetch Section 4.2 and thus sufficient for the conditional in the `if`-statement. Thus, an `if-then-else` construct is made in the fashion `if` $str(\mathcal{M}(n_{c_1}))$ `then` $\pi_{n_{c_1}}$ `else ( if` $str(\mathcal{M}(n_{c_2}))$ `then` $\pi_{n_{c_2}}$ `else ( if` ... `then` ... `else ( if` $str(\mathcal{M}(n_{c_n}))$ `then` $\pi_{n_{c_n}}$ `else skip`))).

The consecutive creation of the `if-then-else` statements in lines 10 - 17 of Algorithm 8, makes for each child, a nested `if-then-else` so that when all the children has been iterated over, the resultant `if-then-else` statement returned, will be one continuous statement. The recursive call in line 20 ensures that the algorithm will not return the plan before the parent is *null* and terminates in line 1.

This algorithm will run upward in the planning tree from the goal nodes found earlier. For each OR-node it will add an event step to the plan, and for each AND-node a sequence of `if-then-else` statements with length equal to the number of children k in the AND-node, thus $O(|\text{OR}| + |\text{AND}| \cdot |n_c|)$, where $|\text{OR}|$ and $|\text{AND}|$ are the number of OR-nodes and AND-nodes in the final plan respectively.

4.4.7.1 Runtime

The runtime of the **ExtractPlan(···)** algorithm is bounded by the number of steps in the final plan, as each node in the planning tree is walked in the extraction. For each OR-node only a single step is needed to move upwards in the tree, and for each AND-node an iteration over the number of children is necessary. However, since these children are already being touched by their own goal-node-to-root-path, this will not add extra calculations to the runtime.

Algorithm	Alias	Runtime
Evaluate(···)	$\lvert\texttt{Eval}\rvert$	$O(k \cdot \lvert\texttt{GU}\rvert \cdot \lvert\texttt{MC}\rvert + k \cdot \lvert\texttt{SP}\rvert)$
GlobalUpdate(···)	$\lvert\texttt{GU}\rvert$	$O(\lvert A\rvert \cdot \lvert\texttt{LU}\rvert)$
LocalUpdate(···)	$\lvert\texttt{LU}\rvert$	$O(\lvert W\rvert \cdot \lvert E\rvert \cdot \lvert\texttt{PU}\rvert)$
PointedUpdate(···)	$\lvert\texttt{PU}\rvert$	$O(2 \cdot \lvert P\rvert + \lvert\texttt{MC}\rvert)$
Split(···)	$\lvert\texttt{SP}\rvert$	$O(\lvert W\rvert)$
ExtractPlan(···)	$\lvert\texttt{EP}\rvert$	$O(\lvert\pi_f\rvert)$

Table 4.7: Runtime of the **Split(···)** algorithm.

In the runtime analysis above, the expression $|\pi_f|$ is the number of steps in the final plan.

4.4.8 Model Check - |MC|

The algorithm for model checking was described above in Section 4.3.2 where it was visualized that the ultimate runtime of each model check depends on the search formula used as well as complexity of the formula used to check. Recall that the model checking algorithm consisted of a traversal of the tree, and that in worst case, dependant on the search method chosen, the entire tree will be traversed.

4.4.8.1 Runtime

As described above the runtime of the model checking depends on the size of the formula tree. In the example of Figures 4.4 and 4.5 the size of the formula tree for $|\phi|$ is 17 nodes.

This yeilds the final runtime for the runtime table as shown in Table 4.8 below:

In the next section the runtime analysis will be concluded and the different runtimes seen above will be concatenated to give an overview of the runtime for the entire program.

Algorithm	Alias	Runtime
Evaluate$(\cdots)$	$\lvert\texttt{Eval}\rvert$	$O(k \cdot \lvert\texttt{GU}\rvert \cdot \lvert\texttt{MC}\rvert + k \cdot \lvert\texttt{SP}\rvert)$
GlobalUpdate$(\cdots)$	$\lvert\texttt{GU}\rvert$	$O(\lvert A\rvert \cdot \lvert\texttt{LU}\rvert)$
LocalUpdate$(\cdots)$	$\lvert\texttt{LU}\rvert$	$O(\lvert W\rvert \cdot \lvert E\rvert \cdot \lvert\texttt{PU}\rvert)$
PointedUpdate$(\cdots)$	$\lvert\texttt{PU}\rvert$	$O(2 \cdot \lvert P\rvert + \lvert\texttt{MC}\rvert)$
Split$(\cdots)$	$\lvert\texttt{SP}\rvert$	$O(\lvert W\rvert)$
ExtractPlan$(\cdots)$	$\lvert\texttt{EP}\rvert$	$O(\lvert\pi_f\rvert)$
ModelCheck$(\cdots)$	$\lvert\texttt{MC}\rvert$	$O(\lvert\phi\rvert)$

Table 4.8: Runtime of the **Split**$(\cdots)$ algorithm.

4.5 Runtime Conclusion

In the previous sections the different algorithms in the program has been displayed along with a runtime analysis of their execution. The table expanded with each runtime analysis shows the values for the indivial algorithms. In this section the efficiency of the different algorithms will be discussed along with a concatenated analysis for the entire planning part of the program.

The individual algorithms will be referenced with their alias ($\lvert\texttt{Eval}\rvert$, $\lvert\texttt{GU}\rvert$, $\lvert\texttt{LU}\rvert$, $\lvert\texttt{PU}\rvert$, $\lvert\texttt{MC}\rvert$, $\lvert\texttt{SP}\rvert$ and $\lvert\texttt{EP}\rvert$) below.

The algorithms $\lvert\texttt{EP}\rvert$, $\lvert\texttt{SP}\rvert$, $\lvert\texttt{PU}\rvert$ and $\lvert\texttt{LU}\rvert$ has all been implemented optimally, in the regard that all of these algorithms needs to iterate over either the number of propositional symbols in a world ($\lvert\texttt{PU}\rvert$), the number of worlds in an equivalence class ($\lvert\texttt{LU}\rvert$), the number of steps in the final plan ($\lvert\texttt{EP}\rvert$) or the number of equivalence classes in the AND-node to be split ($\lvert\texttt{SP}\rvert$).

The model checking algorithm ($\lvert\texttt{MC}\rvert$) has been implemented in a straightforward way, starting from the root of the formula tree and with each iteration relying on a search method (DFS or BFS) to choose the next node. Another way to implement this, which overall might be a faster solution would be to start with the atomic propositions in the formula tree, instead of the root, and with each step upwards in the tree cache the result in order to prevent repetitions. This method of model checking has been chosen to be an optional expansion of the program and will be described a little more in the "Future Work" section.

Since whether or not |GU| is optimal depends upon |LU| being optimal, and the argumentation above specifies |LU| being optimal, then |GU| is optimal too. The reason for this is that as explained in the analysis of |GU|, whenever an OR-node is extended (a model is updated), *all* possible event models in the domain will need to be applied to the model, in order to fully expand the node.

$$
\begin{aligned}
O(|\texttt{Problem}|) &= O(|\texttt{Eval}| + |\texttt{EP}|) && (1)\\
&= O((k \cdot |\texttt{GU}| \cdot |\texttt{MC}| + k \cdot |\texttt{SP}|) + |\texttt{EP}|) && (2)\\
&= O((k \cdot |A| \cdot |\texttt{LU}| \cdot |\phi| + k \cdot |W|) + |\texttt{EP}|) && (3)\\
&= O((k \cdot |A| \cdot |W| \cdot |E| \cdot |\texttt{PU}| \cdot |\phi| + k \cdot |W|) + |\texttt{EP}|) && (4)\\
&= O((k \cdot |A| \cdot |W| \cdot |E| \cdot (2 \cdot |P| + |\texttt{MC}|) \cdot |\phi| + k \cdot |W|) + |\texttt{EP}|) && (5)\\
&= O((k \cdot |A| \cdot |W| \cdot |E| \cdot (2 \cdot |P| + |\phi|) \cdot |\phi| + k \cdot |W|) + |\texttt{EP}|) && (6)\\
&= O((k \cdot |A| \cdot |W| \cdot |E| \cdot (2 \cdot |P| + |\phi|) \cdot |\phi| + k \cdot |W|) + |\pi_f|) && (7)\\
&= O(k \cdot |W| \cdot (|A| \cdot |E| \cdot (2 \cdot |P| + |\phi|) \cdot |\phi| + 1) + |\pi_f|) && (8)\\
&= O(k \cdot |W| \cdot (|A| \cdot |E| \cdot (|P| + |\phi|) \cdot |\phi|) + |\pi_f|) && (9)
\end{aligned}
$$

And recall that k is the number of nodes in the planning tree, $|W|$ the maximum number of worlds in *any* model, $|A|$ the size of the set of available event models, $|E|$ the maximum number of events in *any* event model, $|P|$ the total amount of propositional symbols in the given domain, $|\phi|$ the number of nodes in the formula tree and $|\pi_f|$ the number of steps in the resultant plan.

The reason for $|W|$ and $|E|$ to be the *maximum* number of worlds in a *any* model respectively events in *any* event model, is that the number of worlds in the different models varies, just as the number of events varies from event model to event model. Therefore if this runtime calculation are to have any merits, we need to consider the maximum number of worlds in any model and event in any event model.

As can be seen in the final line of the runtime analysis ((9)), the only redundant information used is the goal formula tree in the model checking. This is because both the **Evaluate(**$\cdots$**)** model checks for models satisfying the goal formula ϕ_g and the **PointedUpdate(**$\cdots$**)** checks the precondition of each event to the world being updated.

Chapter 5

Examples and Results

In this section four examples will be shown to illustrate the different features of the theory and implementation. The first three examples are the running examples from previous sections, namely Simple, Partial and NonDet. The last example is a more complex domain, Complex, that utilizes both the aspect of partial observability and non-determinism in the domain. These components are implemented in the same way as seen before, via *discovery* of tiles not yet seen (partial observability) and the *teleports* as seen in Section 2.4.2 (non-determinism).

For each of the examples will be provided the domain of the example as a figure, like the ones seen in the previous running examples. Also the `input`, given to the program as a text based file, will be shown. The input file is based on the grammar as partly seen earlier (see 4.1(a) and 4.1(b)) and will be detailed where appropriate. Lastly the output planning tree for the domain will be shown, as calculated by the program. In the tree-graphs the legend for the node colors and edges will be detailed. As the graphs shown are *trees*, the gray edges that lead upwards in the tree are merely reference edges to show that the resultant child node n_c of a node n has already been encountered in the program. This is for visual simplicity. It is worth mentioning at this point that the planning tree for the final domain Complex is so huge that it does not fit into this section.

The first example will be described a little more detailed than the others, in order to introduce the input syntax as well as planning tree. Also in the last section (Section 5.5 - Runtime of Examples) for each of the examples will be given the benchmark numbers of the runtime for that domain.

5.1 Domain: Simple

The first of the example runs given is the Simple domain as given in the introduction. The layout of the domain is shown again for reference's sake in Figure 5.1. As stated before, in this example the goal of the agent is to *go* to the goal located on tile t_2.

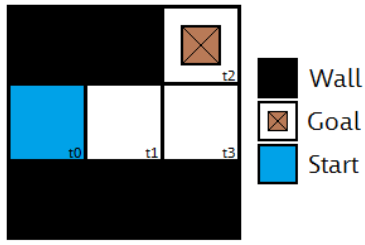


Figure 5.1: The domain Simple as seen in the introduction.

The input for the domain is shown in Figure 5.2 and describes everything about the domain in the syntax specified by the grammar created for this purpose. Initially in the input, line 1 gives the name of the domain which will be shown in the resultant graphviz[gra] tree-graph and line 2 optionally shows a list of the symbols used in the domain, seperated by commas. After this the initial model is specifed, in this case consisting of only a single world, with the proposition t_0, *The agent is at tile* t_0 and an optional title for the world has been given, w_1.

```
1  Title: _sokobansimple
2  Symbols: t0, t1, t2, t3
3
4  Model _init = [ _w1 = t0 ]
5
6  EventModel _GoRight = [
7    _gr1 = t0 ; ~t0 & t1,
8    _gr2 = t1 ; ~t1 & t3]
9  EventModel _GoLeft = [
10   _gl1 = t1 ; ~t1 & t0,
11   _gl2 = t3 ; ~t3 & t1]
12 EventModel _GoUp = [ _gu = t3 ; ~t3 & t2 ]
13 EventModel _GoDown = [ _gd = t2 ; ~t2 & t3 ]
14
15 _goal = t2
```

Figure 5.2: The input text file for the domain Simple.

Next the event models available to the agent in the domain is specified. There are four event models, namely *GoRight*, *GoLeft*, *GoUp* and *GoDown*. Each of these has a number of events given, based on the number of places that particular event is applicable.

The syntax of the events in the event models are specified by the precondition followed by a semicolon, after which the postmapping is stated. The postmapping in the input specification is given with an list of symbols to be modified, for example written `~t0` for $t_0 \mapsto \bot$. In the case of *GoRight* and *GoLeft* there are two possible places (t_0 and t_1 respectively t_1 and t_2), whereas in the case of *GoUp* and *GoDown* these can only be executed at tiles t_3 and t_2 respectively. Lastly the goal formula sought is given, again with an optional name.

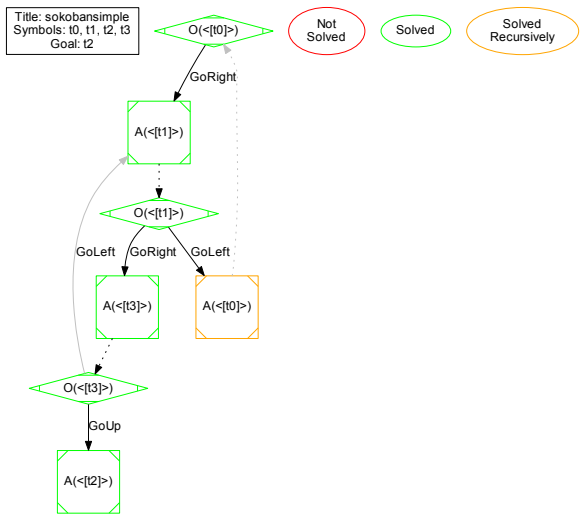


Figure 5.3: The resultant planning tree for the SIMPLE domain.

In the resultant tree-graph visible in Figure 5.3 the planning tree for the domain can be seen. The root node is located in the top, in between the title of the domain and the node-legends showing what each of the node colors mean. It should be mentioned here, that the orange colored nodes (*Solved Recursively*) is merely an indicator that the node *could* be solved, if the subtree of that node were to be expanded. Each node is marked by its unique string representation as discussed in Section 4.3.3 which also gives a simple overview of which propositional symbols holds in the given model and world.

As can be seen in the tree, the agent can from the initial node only choose to *GoRight*. After this action, the resultant AND-node is split into 1 OR-node containing a single equivalence class with a single world, placing the agent at tile t_1. From this node (`((O(<[t1]>)))`) the agent now has two options: Either he can continue right, or he can go back (left). The edges from the OR-node reflects this. If the agent chooses to go left, the protruding edge from this choice (`(A(<[t0]>))`) leads back up to the initial state, and is colored orange, marking the node as a recursive solution.

Lastly the model satifying the goal formula is found in the node `A(<[t2]>)` after going up from tile t_3. This node is a goal node, because the node is solved, and it has no outgoing edges.

5.2 Domain: Partial

In the partially observable domain introduced earlier the input as well as the resultant planning tree is substantially larger and more complex. The domain is, again as a reference, shown in Figure 5.4.

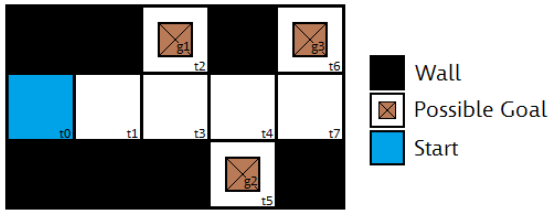


Figure 5.4: A partial observable domain Partial.

Now in addition to the syntax explained in the Simple domain, there is now the extra information at line 8 in the input of Figure 5.5, which makes the initial model differ from the definition in the simple domain. This line details that world w_1 is equivalent to to the world w_2 and furthermore that the world w_2 is equivalent to the world w_3. This is of course the definition of the equivalence relation and merely defines that worlds w_1, w_2 and w_3 belongs in the same equivalence class related via the equivalence relation R.

```
1 Title: _sokobanpartial
2 Symbols: g1, g2, g3, t0, t1, t2, t3, t4, t5, t6, t7
3
4 Model _init = [
5   _w1 = t0 & g1,
6   _w2 = t0 & g2,
7   _w3 = t0 & g3
8   :: w1 = w2, w2 = w3 ]
```

Figure 5.5: First part of the input text file for the domain Partial.

This will also become clear when loooking at the initial model in the planning tree in Figure 5.7 which has the string representation `O(<[g1,t0],[g2,t0],[g3,t0]>)` indicating that indeed the three worlds given by `[g1,t0]`, `[g2,t0]` and `[g3,t0]` is located within the same pair of angled brackets `'<' '>'` as specified in the grammar in Section 4.2.

Another aspect of the text input which is worth mentioning, are the paired events in the event model *GoRight* at lines 12 to 17 in the second part of the input in Figure 5.6, pairing up `_gr11` with `_gr12`, `_gr21` with `_gr22` and so forth. The reason for this, is that when the agent executes these event, he will arrive at two different states, depending of whether or not the goal is located at the location g_1, g_2 or g_3 respectively.

```
9  EventModel _GoRight = [
10   _gr0 = t0 ; ~t0 & t1,
11   _gr11 = t1 & g1 ; ~t1 & t3,
12   _gr12 = t1 & ~g1 ; ~t1 & t3,
13   _gr21 = t3 & g2 ; ~t3 & t4,
14   _gr22 = t3 & ~g2 ; ~t3 & t4,
15   _gr31 = t4 & g3 ; ~t4 & t7,
16   _gr32 = t4 & ~g3 ; ~t4 & t7 ]
17
18 EventModel _GoLeft = [
19   _gl0 = t1 ; ~t1 & t0,
20   _gl1 = t3 ; ~t3 & t1,
21   _gl2 = t4 ; ~t4 & t3,
22   _gl3 = t7 ; ~t7 & t4 ]
23
24 EventModel _GoUp = [
25
26   _gu0 = t3 ; ~t3 & t2,
27   _gu1 = t5 ; ~t5 & t4,
28   _gu2 = t7 ; ~t7 & t6 ]
29
30 EventModel _GoDown = [
31   _gd0 = t2 ; ~t2 & t3,
32   _gd1 = t4 ; ~t4 & t5,
33   _gd2 = t6 ; ~t6 & t7 ]
34
35  _goal2 = (t2 & g1) | (t5 & g2) | (t6 & g3)
```

Figure 5.6: The second part of the input text file for the domain PARTIAL.

This can also be seen in the planning tree in Figure 5.7 where the AND-node are split, namely the first time at node 4 from the root (`A(<[g1,t3]>,<[g2,t3],[g3,t3]>)`), which is split into `O(<[g1,t3]>)` and `O(<[g2,t3],[g3,t3]>)` representing the situations where the goal is located at g_1 and the situation where it is not.

Already now, the resultant planning tree has increased dramatically in size because of the partial observability. In the tree exists three nodes that are satisfying the goal formula. These three are `A(<[g1,t2]>)`, `A(<[g2,t5]>` and `A(<[g3,t6]>)` representing the three goal states possible as given with the inputted goal formula `_goal2 = ( t2 & g1 ) | ( t5 & g2 ) | ( t6 & g3 )`.

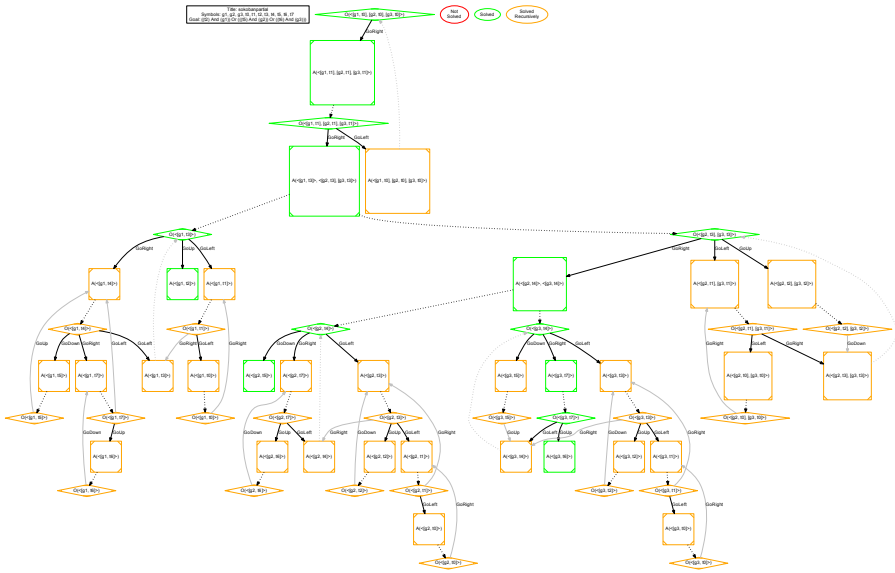


Figure 5.7: The resultant planning tree for the PARTIAL domain.

5.3 Domain: NONDET

The non-deterministic domain NONDET shown in Figure 5.8 (and also earlier in Section 2.4.2) introduces into the input file the same kind of grouping of events as partial observability did.

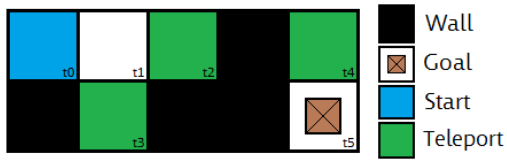


Figure 5.8: The domain NONDET which has a component of non-determinism represented by *teleports*.

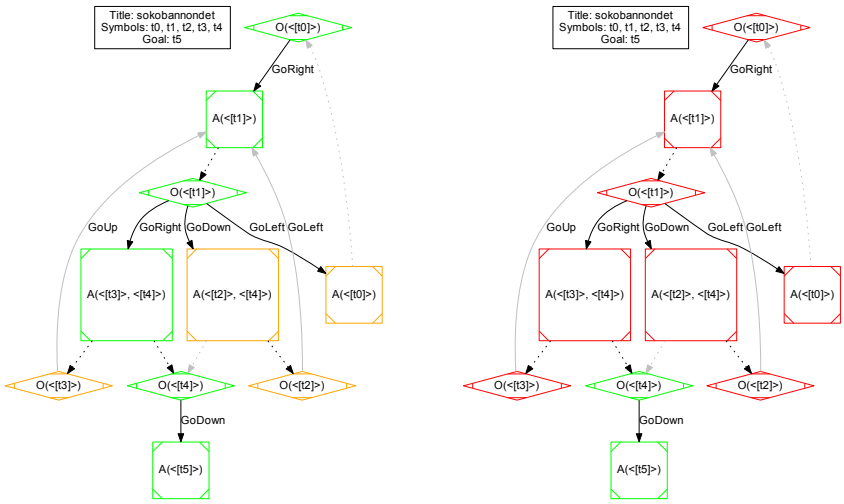
The interesting fragment of the input file (the event model *GoRight* is used as example) is shown in Figure 5.9. In this input fragment, we can again see the groupings of related events, namely the `_gr21` and `_gr22` events, which as explained in the theory has the same precondition t_1 and thus allows only one of them to be executed at runtime.

The rest of the input file follows the convention of the input files from the Simple and Partial domains.

```
1 EventModel _GoRight = [
2   _gr1 = t0 ; ~t0 & t1,
3   _gr21 = t1 ; ~t1 & t3,
4   _gr22 = t1 ; ~t1 & t4 ]
```

Figure 5.9: Fragment of the input text file for the domain NonDet.

The resultant graphs from the NonDet domain has this time been given in two versions. The first (leftmost) tree, is the resultant planning tree when the domain is solved *weakly*. Because of the non-determinism, it is not possible to solve the problem *strongly* as shown in the rightmost tree. The reason for this is that theoretically the agent could possibly never arrive at the teleport t_4 even though the stochastic process supposedly distributes an equal chance to each outcome. In the leftmost tree, though, is shown a *possible* plan to the problem that *may* be a solution.



(a) The planning tree if weakly solved.

(b) The planning tree if strongly solved.

Figure 5.10: The resultant planning trees for the NonDet domain.

The possible plan shown in Figure 5.10(a) can be read as *GoRight*;*GoRight*;*GoDown*, and we can then infer that the situation this plan is equipped to solve, is where the teleport at tile t_2 transports the agent to tile t_4 the first time the agent enters the teleport.

5.4 Domain: Complex

The last example is a little more complex in order to force the program to create a bigger planning tree. Looking at the domain in Figure 5.11, notice the addition of the *Possible Box Location*. This domain is an instance of the classical Sokoban game ([sok]) with the teleport added to provide the component of non-determinism. In the game, the objective is to push the box to the goal location.

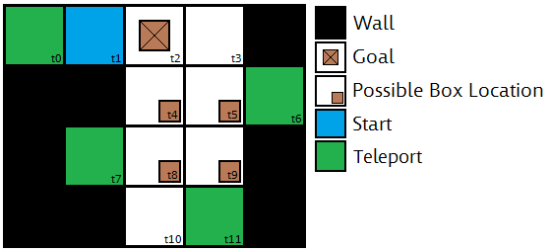


Figure 5.11: The domain Complex, which include the aspect of partial observability and non-determinism.

In order to push a box, the agent has to be located behind the box and no obstacles can be in front of the box. In this domain, the agent has to use the teleports in order to get behind the box to push it out onto the goal location. The domain contains both the aspects of non-determinism, by the teleports, and partial observability, as the agent does not know on which of the locations the box is located.

```
1  EventModel _GoLeft = [
2    ...
3    _gl11 = t1 & ~b0 & ~b6 & b4 ; t6 & ~t1,
4    _gl12 = t1 & ~b0 & ~b6 & b5 ; t6 & ~t1,
5    _gl13 = t1 & ~b0 & ~b6 & ~(b4 | b5) ; t6 & ~t1,
6
7    _gl21 = t1 & ~b0 & ~b7 & b8 ; t7 & ~t1,
8    _gl22 = t1 & ~b0 & ~b7 & b9 ; t7 & ~t1,
9    _gl23 = t1 & ~b0 & ~b7 & ~(b8 | b9) ; t7 & ~t1,
10
11   _gl31 = t1 & ~b0 & ~b11 & b5 ; t11 & ~t1,
12   _gl32 = t1 & ~b0 & ~b11 & b9 ; t11 & ~t1,
13   _gl34 = t1 & ~b0 & ~b11 & ~(b5 | b9) ; t11 & ~t1,
14   ...
15 ]
```

Figure 5.12: Part of the event model *GoLeft* from the input text file for the domain Complex.

The boxes are added to increase the number of actions available to the agent. Due to

these factors, both the input file and the resultant planning tree is very large and the final amount of nodes in the planning tree is close to 1260.

Part of the input file to the planning problem is given in Figure 5.12, where the most interesting aspect of the partial obervability and non-determinism combination is shown. A lot happens in this part of the *GoLeft* event model which takes the agent *left* from tile t_1 his start tile.

The reason there are so many lines to describe just this single move, is because the agent will enter a teleport tile at t_0 and then get teleported to one of the tiles t_6, t_7 or t_{11}, where he *may* discover a box on either of the locations t_4, t_5, t_8 or t_9 (a box located on a tile t_x is written b_x, i.e. a box on tile t_4 is indicated by b_4).

Since the move displayed is from tile t_1 all of the events in the partial event model listed requires that the agent be located at tile t_1. Furthermore all the events require that no boxes obstructs the agent's move (from t_1 to t_0) and that no box is placed on the destination tile (one of the three teleport tiles). If these first three precondition are satisfied, we see in lines 3-5 a construction similar to the one of the partial observability example in Section 5.2. The agent acquires the knowledge of whether or not the box is located at tile t_4 or t_4 or not at all. This construction is repeated in lines 7-9 and 11-13 just with the other teleport destinations.

As can be seen, this is just the event model input for the agent move from t_1 to t_0 (destination being t_6, t_7 or t_{11}). The entire event model for *GoLeft* is much bigger, not to mention the rest of the event models available to the agent.

5.5 Runtime of Examples

In the following, will be shown, the numbers for the four example runs in addition to the graphs visualizing these. Each of the examples has been run 4 times and the result furthest from the average discarded. This is because the runtime of each of the examples is very short and thus even the smallest spike in CPU usage will corrupt the numbers.

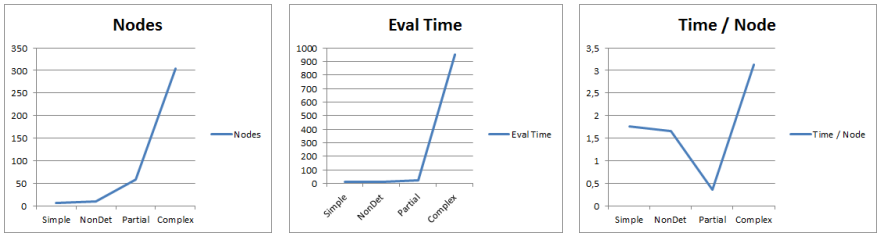
In Table 5.1 the numbers are shown. This table also displays the number of nodes in each of the final planning trees as well as the time-per-node ratio. This ratio is just a rough sketch of the runtime on bigger problems as the Complex example is the most computationally heavy problem given to the program. In Figure 5.13 the three excel graphs for the values given above are shown. The first graph shows the final number of nodes in the program.

As the planning trees had already revealed earlier, we get lot more nodes in the problems including partial observability.

	Simple	NonDet	Partial	Complex
Number of events in *GoRight*	2	3	7	33
Nodes in Planning Tree	7	10	59	305
Evaluation Time	11.2	13.65	23.4	951.32
	15.4	18.71	20.8	804.23
	10.5	17.36	17.56	1111.78
Avg. Eval. Time	12.37	16.57	20.59	955.78
Time / Node	1.77	1.65	0.35	3.13

Table 5.1: Runtime values for running the problems Simple, Partial, NonDet and Complex.

This is because for each proposition which is only partially observable, an extra world will exist in each equivalence class of the root node reflecting the unknown state between worlds in which the proposition holds and worlds where it does not. Combined with the notion of non-determinism, even though the domain Complex is only 1.5 times as big as the Partial domain in terms of propositional symbols available, the resulting planning tree contains 5 times as many nodes.



(a) Number of nodes in the final planning tree.

(b) Time to run problem in the program.

(c) Time per node ratio.

Figure 5.13: Three excel graphs showing the results from the CPU analysis of runnin the problems.

This is of course also due to the extra amount of events required in the individual event models. Even if the Complex domain is more extensive and thus has more places an agent can execute the action *GoRight*, the 33 events in the *GoRight* event model of the Complex domain is mostly a result of the combination between partial observability and non-determinism.

Chapter 6

Future Work

In this section will be described the possible extensions to the program. These extensions are suggestions for logical continuation of the program either because the theory behind has already been covered (see [TB12b]) or to serve the goal of getting the program closer to the well known standardized planners. The extensions has either been left out of the intial version of the program because the extra time required would be too costly, or because the extension itself was too big or required to big a change in the already existing code.

6.1 Cyclic Plans

In the planning of a domain with a non-deterministic component, as has been seen, it is not possible to provide a strong plans because as non-determinism suggests, the agent cannot make a plan to be *absolutely* sure that he will arrive in a state satisfying the goal formula. In the NonDet domain given earlier, the non-deterministic aspect of the domain is given by the teleports, and in this domain, it would help being able to formulate a plan along the lines of *while not* $K(t_4)$*, enter teleport on* t_3, i.e. the aspect of a sequence of repeatable plan-steps to execute as long as a given condition is not satisfied.

This sequence of plan-steps would serve to help in planning situations where not all the childnodes of an AND-node are solved, because some of the nodes are duplicates of earlier encountered nodes. In the planning tree of the strongly solved version of the NonDet domain given in Figure 5.10(b) it can be seen in the bottom of the tree, that node `A(<[t3]>,<[t4]>)` splits into `O(<[t3]>)` and `O(<[t4]>)` where only the latter is marked as solved. The former, however, *could* be solved too, by transforming the planning tree into a planning *graph* and then follow the reference up in the tree to arrive at `A(<[t1]>)` and start over.

6.2 NPDDL

The focus of this project has been to implement the theory of conditional epistemic planning and to make examples that shows how the theory works in practice. Therefore the notion of extending the program with a parser for inputting examples in other languages and thus make it possible for other users (not knowing the exact syntax used in this program) to utilize the program and its visual planning tree representations for their own examples would be a logical extension. A somewhat standardized language for planning domains, which is well known within the planning community is the Planning Domain Definition Language (PDDL) [pdd] which is built on the STRIPS and Action Description Language. In order to fully encompass the aspects of the program (partial observability and non-determinism) implemented in the program, the decision became to make a parser for *NPDDL* (Non-deterministic PDDL, [PB12]) that would translate NPDDL input to the syntax needed for this program in order to read NPDDL examples into the program and execute these.

The work with implemented grammar was well on the way, and was one of the extensions being actively worked on, in the duration of the project. The work on this, however, did not conclude due to lack of time and thus only the grammar part of the implementation can be seen as result (see Appendix A.2).

6.3 Plausibility Planning

Another logical extension of the dynamic epistemic logic framework is to augment the framework with planning based on plausibility models. Since strong plans, in a real world scenario where an agent may encounter many situations with an aspect of uncertainty, are not computationally viable, and there is no way to tell if one weak plan is better than another, the introduction of plausibility models would come in handy. The agent would then be able to pick the action he most believes in at the time of choice [pla].

6.4 Planning Improvements - Caching

A few of the things in the program, which can be changed in order to make the program work optimally, is the evaluation of models and formulas in the planning process. As mentioned in the appropriate sections, when a model is model checked if it satisfies a given formula, the formula tree is expanded in a top down approach using either BFS or DFS. A faster albeit implementationally more complex solution, is to have the formula tree being traversed from the leaf nodes and upwards, and in the process of doing this, *caching* sub-formulas in order to return their value later when needed. This will ensure that whenever a formula or sub-formula appears that has been evaluated

before (in that world), it will instantly return the value of the evaluated sub-formula, instead of having to redo the model check.

This method can also be applied in the string representation of the models. Each time a new model is checked, its string representation can be stored within the model object itself in order to avoid recalculating the string representation of that model. This could potentially save a lot of computing power, depending how many times a model is compared to other models for bisimilarity.

6.5 Input

Another aspect of the program which was left at the foundation initially built, was the process of the input. The grammar and parser which was generated works wonders, however writing the input in the specified syntax can quickly become an ardeous task. Thus a logical expansion for the program would be to create an input editor for specific domains (for example the sokoban domains) in order to quickly generate the input for far larger domains and thus be able to put the program to the test.

Chapter 7

Conclusion

This project has been about the implementation of the planning process of an agent wanting to solve a task without actually executing any action before the plan has been fully created. As stated in the introduction, the dynamic epistemic logic framework has been used in this regard to offer the notions of *knowledge* and *dynamism* beyond classical logic, namely that the agent can have knowledge regarding a proposition and that the agent can dynamically update his belief state to advance in the domain. Progressing from this, *conditionals* was introduced in the planning section of Dynamic Epistemic Logic to provide the agent with `if-then-else` constructs to manouver in situations where aspects of uncertainty was in place. Specifically it was shown in the example given in Figure 2.6 how an agent might go about making a plan that takes into account the fact that the goal might not be located at the first goal location, but maybe at the second or third, and the `if-then-else` construct was used in this example. Following this, *planning trees* was introduced as a way to organize the different epistemic models encountered in the domain, and their relationship in order to later extract a plan if a state satisfying the given goal formula was found. The introduction of *weak* and *strong* plans gave the notion of plans that *may* lead to a state satisfying the goal formula (weak plans) respectively and *always* lead to a state satisfying the goal formula (strong plans).

In the implementation section, the practical part of the project was described in detail. The method of inputting the information into the program was implemented using an left-to-right, leftmost derivation parser generator, `antlr`, to translate the input text file into objects in the program. The planning part of the program was implemented using object oriented programming, in which the epistemic models was implemented as sets of equivalence classes, which in turn contains sets of related worlds. This turned out to be an easy way to keep track of the equivalence relation when updating models via the product update, and thus the event model was implemented similarly. By iterating first the sets of worlds in an equivalence class and subsequently iterating the sets of events in an event model to be applied in the product update, the resulting worlds would be put into the same (new) equivalence class, them being equivalent in terms of the equivalence relation.

The planning tree was implemented with a node class each having a set of child nodes, as well as a reference to the parent node for later easy extraction of the plan. The unique string representation used with both the models and the nodes ensured that no duplicates of nodes would appear in the tree. If a duplicate model was found when product updating, a gray reference edge would be created pointing to that already existing model in the node present in the tree merely to indicate that the new node (and model) had been encountered before and thus would not be expanded. It was noticed that planning problems containing the aspect of partial observability lead to substantially larger planning trees than problems with only non-determinism or no uncertainty at all. The reason being that each time an agent is uncertain regarding a proposition in the same equivalence class, the number of worlds in that equivalence class increases, leading to a larger branching factor when the AND-nodes are split. In the extended example, combining both the notion of partial observability with non-determinism, the resultant planning tree became huge due to an highly increased amount of events in each model to account for all the different possibilities (a box being present or not and arriving at place a or place b).

The implementation of the planning process in the program was done using a recursively called function **Evaluate**$(\cdots)$ which in turn would call the necessary methods for model checking the model of the current node, and expanding if the formula was not satisfied. The expanding of a node was done in a derived class of the node, the OR-node or AND-node which would call either a function to product update the OR-node or to split the AND-node into child nodes. It was noticed that besides a few implementational quirks the runtime of the update functions ran in close to optimal time, the argument being that no matter which data structure chosen, *all* worlds and *all* events needed to be iterated over, in order to produce the new epistemic model, and since no singular world would ever be interesting out of the whole, the solution of sets of sets was a sound one.

In the Examples and Results section, was then shown four different domains with different aspects of the dynamic epistemic logic framework included. The SIMPLE domain was the example from the introduction showing how an agent could move around in a domain and plan to reach a known goal. The PARTIAL domain showed what happens in the case that the agent does not in advance know the location of the goal and how the event models has to be modified to allow the agent to discover this. The third example was the NONDET domain, which included the teleport component in order to add the notion of non-determinism to the domain. The idea being that no matter how many times an agent were to enter a teleport, the destination would remain the product of a randomized (stochastic) process (given more than one destination in the domain). The last example glanced upon the idea of the combination of partial observability as well as non-determinism and showed that the size of the resultant planning tree grew with the increase in propositional symbols and events in the event models. The resultant planning tree had 305 nodes in comparison to the 59 of the PARTIAL domain and the 10 of the NONDET example.

A major focus in the program has been to make it optimally running, not only in terms of the already implemented functionality, but also to give the option to include other

search methods with minimal work. As mentioned, both the model checking process as well as the process of expanding the planning tree utilizes the data structure of object oriented programming, polymorphism and derived classes to its full extend to allow for customized additions like additional search algorithms or tree-traversals.

It was shown that the program could handle the examples given without a hitch, however, that being said, with bigger and more complex domains the input given to the program will increasingly complicated to write in hand, and thus in order to fully put the program to the test a domain editor of sorts would be required. This was suggested as a future expansion of the program alongside the parser for NPDDL inputs, as there exists larger examples written in NPDDL syntax.

From beginning to end, the project has been very straigtforward. The main objective was to follow the "script", mainly the article [TB12b] and to understand and implement what was written. The product of the project, the program, implements the theory in a mostly optimzed fashion and allows for the user to specify own examples to input and run in the program. There is room for improvement in the program as, due to time restrictions, some of the aspects of the implementation was left as 'just' the foundation instead of improving upon these at a later time. For instance, this is true regarding the model checking component of the program, using a top-down approach to evaluate the formulas given, instead of starting with the atomic properties in a formula, which would have been more efficient.

Bibliography

[ant] ANTLR another tool for language recognition. `https://github.com/antlr/grammars-v4`. Accessed: 25/12/2012.

[Auc10] G Aucher. *An internal version of epistemic logic.* Studia Logica, 2010.

[cac] Caching caching in computer science. `http://en.wikipedia.org/wiki/Cache_(computing)`. Accessed: 13/03/2013.

[DH08] Kooi B. Ditmarsch H., Hoek W. *Dynamic Epistemic Logic.* Springer, 2008.

[ecs] Equivalence Classes equivalence classes. `http://en.wikipedia.org/wiki/Equivalence_class` pages 15 - 22. Accessed: 26/02/2013.

[gra] GraphViz graph visualization software. `http://www.graphviz.org/`. Accessed: 09/02/2013.

[MG04] Paolo Traverso Malik Ghallab, Dana Nau. *Automated Planning: Theory and Practice.* Morgan Kaufmann, 2004.

[non] Non-determinism non-determinism in computer science. `http://en.wikipedia.org/wiki/Nondeterministic_algorithm`. Accessed: 13/03/2013.

[oop] Object Oriented Programming object oriented programming. `http://en.wikipedia.org/wiki/Object-oriented_programming`. Accessed: 13/03/2013.

[PB12] Alessandro Cimatti Piergiorgio Bertoli. Extending pddl to nondeterminism, limited sensing and iterative conditional plans. 2012. `http://users.cecs.anu.edu.au/~thiebaux/workshops/ICAPS03/proceedings/PDDL-ICAPS03.pdf`.

[pdd] PDDL planning domain definition language. `http://en.wikipedia.org/wiki/Planning_Domain_Definition_Language`. Accessed: 09/03/2013.

[pla] Don't plan for the unexpected planning based on plausibility models. http://www2.imm.dtu.dk/~tobo/plausibility_planning.pdf. Accessed: 09/03/2013.

[pol] Polymorphism polymorphism in computer science. http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming. Accessed: 20/02/2013.

[pom] Partial Obervability partial observability markov decision process. http://en.wikipedia.org/wiki/Partially_observable_Markov_decision_process. Accessed: 13/03/2013.

[sok] Sokoban Game sokoban game. http://en.wikipedia.org/wiki/Sokoban. Accessed: 26/02/2013.

[TB12a] Martin Holm Jensen Thomas Bolander, Mikkel Birkegaard Andersen. Conditional espistemic planning - long version. 2012.

[TB12b] Martin Holm Jensen Thomas Bolander, Mikkel Birkegaard Andersen. Conditional espistemic planning - short version. 2012.

[TB12c] Mikkel Birkegaard Andersen Thomas Bolander. Epistemic planning for single and multi-agent systems. 2012.

[THC01] Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms*. The MIT Press, second edition edition, 2001.

Appendix A

Input Grammar

A.1 Input Planning Problems

```
grammar CEPlistbranch;

options {
        language = CSharp3;
        backtrack = true;
        memoize = true;
}

tokens {
        TITLE = 'Title';
        PROPOSITIONALSYMBOLS = 'Symbols:';
        MODEL = 'Model';
        EVENTMODEL = 'EventModel';
        EQUALS = '=';
        LSQUARE = '[';
        RSQUARE = ']';
        OR = '|';
        AND = '&';
        START = '^';
        END = '$';
        NOT = '~';
        TRUE = 'T';
        FALSE = 'F';
        K = 'K';
```

```
25          COMMA = ',';
26          COLON = ':';
27          SEMI = ';';
28          DOUBLECOLON = '::';
29          LPAREN = '(';
30          RPAREN = ')';
31          TITLESTART = '_';
32  }
33
34  @parser::header {
35      using CEP.datastructure;
36      using System.Collections.Generic;
37  }
38  @lexer::header {
39      using CEP.datastructure;
40      using System.Collections.Generic;
41  }
42  @members {
43      private string modelTitle;
44      private string worldTitle;
45      private string formulaTitle;
46      private Formula tempFormula;
47      private List<PropositionalSymbol> symbols;
48  }
49
50  @parser::namespace { CEP.grammar }
51  @lexer::namespace { CEP.grammar }
52  // r | (f & Kp)
53
54  public cep :
55          'Title:' t = title { Cep.Title = t; }
56          PROPOSITIONALSYMBOLS p1 = propositionalsymbol { Model.
                Symbols.Add(p1); } (COMMA p2 = propositionalsymbol {
                Model.Symbols.Add(p2); })*
57          thisModel = model {  Cep.AddRoot(thisModel); }
58          (em = eventmodel { Cep.EventModels.Add(em); })*
59          (f = formula { Cep.Goal = f; })?
60          ;
61
62  public model returns [Model m] :
63          { m = new Model(); }
64          MODEL (t = title { m.Title = t; } EQUALS)? LSQUARE
65                  w = world { m.AddWorld(w); } ((COMMA w2 = world { m
                        .AddWorld(w2); })*)
66                  (DOUBLECOLON wr = worldrelation { m.
                        AddWorldRelation(wr.a, wr.b); } ((COMMA wr2 =
                        worldrelation { m.AddWorldRelation(wr2.a, wr2.b
                        ); })*))?
67          RSQUARE
68          ;
69
70  public eventmodel returns [EventModel em] :
71          { em = new EventModel(); }
72          EVENTMODEL t = title { em.Title = t; } EQUALS LSQUARE
```

```
73                  e = event { em.AddEvent(e); } ((COMMA e2 = event {
                        em.AddEvent(e2); })*)
74                  (DOUBLECOLON er = eventrelation { em.
                        AddEventRelation(er.a, er.b); } ((COMMA er2 =
                        eventrelation { em.AddEventRelation(er2.a, er2.
                        b); })*))?
75          RSQUARE
76          ;
77
78  world returns [World w]
79          : { w = new World(); }
80          t = title EQUALS (p1 = propositionalsymbol { w.AddSymbol(p1
                ); } (AND p2 = propositionalsymbol { w.AddSymbol(p2);
                })*)?
81          { w.Name = t; }
82          ;
83
84
85  worldrelation returns[WorldRelation wr]
86          : {wr = new WorldRelation(); }
87          t1 = title EQUALS t2 = title
88          { wr.a = new World(t1); wr.b = new World(t2); }
89          ;
90  event returns [Event e]
91          : { e = new Event(); }
92          (t = title EQUALS { e.Name = t; })? (f = formula {e.
                PreEvent = f; })? SEMI (p1 = propositionalsymbol { e.
                PostEvent.Add(p1); } (AND p2 = propositionalsymbol { e.
                PostEvent.Add(p2); })*)?
93          ;
94
95
96  eventrelation returns[EventRelation er]
97          : {er = new EventRelation(); }
98          t1 = title EQUALS t2 = title
99          { er.a = new Event(t1); er.b = new Event(t2); }
100         ;
101
102 public formula returns [Formula f] :
103         { formulaTitle = ""; }
104         (t = title { formulaTitle = t; } EQUALS )? e = expr { f = e
                ; f.Title = formulaTitle; }
105         ;
106
107 expr returns [Formula f] :
108          (p = propositionalsymbol { f = p; }
109         | NOT f2 = expr { f = new Not(f2); }
110         | K f3 = expr { f = new Knowledge(f3); }
111         | LPAREN f1 = expr { f = f1; } RPAREN)
112         ((AND f4 = expr { f = new And(f, f4); }) |
113         (OR f5 = expr { f = new Or(f, f5); }))*
114         ;
115
116 propositionalsymbol returns [PropositionalSymbol pros]
117         : TRUE { pros = new True(); }
```

```
118         | id1 = id { pros = new Atomic(id1); }
119         | NOT id2 = id { pros = new NotLiteral(new Atomic(id2)); }
120         ;
121
122 id returns [string s]
123         : { s = string.Empty; }
124         l1 = LOWERCASELETTER { s += l1.Text; } (l2 =
                LOWERCASELETTER { s += l2.Text; } | l3 = LETTER { s +=
                l3.Text; } | i = INTEGER { s += i.Text; })*
125         ;
126
127 title returns [string title]
128         : TITLESTART (l1 = LETTER { title += l1.Text; } | l2 =
                LOWERCASELETTER { title += l2.Text; } | i = INTEGER {
                title += i.Text; })+
129         ;
130
131 WS  :   ( ' '
132         | '\t'
133         | '\r'
134         | '\n'
135         ) { Skip(); }
136     ;
137
138 INTEGER : '0'..'9'
139         ;
140
141 LOWERCASELETTER
142         : 'a'..'z' ;
143
144 LETTER  : 'A'..'Z'
145         ;
```

A.2 Input NPDDL

```
1  grammar npddl;
2
3  options {
4          language = CSharp3;
5          backtrack = true;
6          memoize = true;
7  }
8
9  tokens {
10         DEFINE = 'define';
11         DOMAINDEF = 'domain';
12         DOMAIN = ':domain';
13         PROBLEM = 'problem';
14         TYPES = ':types';
15         PREDICATES = ':predicates';
16         FUNCTIONS = ':functions';
```

```
17          ACTION = ':action';
18          PRECONDITION = ':precondition';
19          EFFECT = ':effect';
20          OBSERVABLE = ':observable';
21          OBSERVATION = ':observation';
22          TYPEDEF = ':typedef';
23          INIT = ':init';
24          OBSERVABILITY = ':observability';
25          STRONGGOAL = ':stronggoal';
26          WEAKGOAL = ':weakgoal';
27          STRONGCYCLICGOAL = ':strongcyclicgoal';
28          PARTIAL = ':partial';
29          FULL = ':full';
30          ONEOF = 'oneof';
31          UNKNOWN = 'unknown';
32          RANGE = 'range';
33          PLUS = '+';
34          MINUS = '-';
35          EQUALS = '=';
36          LT = '<';
37          GT = '>';
38          LTE = '<=';
39          GTE = '>=';
40          NOT = 'not';
41          ASSIGN = 'assign';
42          IFF = 'iff';
43          AND = 'and';
44          OR = 'or';
45          IMPLY = 'imply';
46  }
47
48  @parser::header {
49      using NPDDL.datastructure;
50      using System;
51      using Action = NPDDL.datastructure.Action;
52      using Type = NPDDL.datastructure.Type;
53
54  }
55  @lexer::header {
56      using System;
57      using NPDDL.datastructure;
58
59  }
60
61  @parser::namespace { CEP.grammar }
62  @lexer::namespace { CEP.grammar }
63
64
65  public npddl returns [Npddl n]
66          : { n = new Npddl(); Console.WriteLine("Construct NPDDL");
                }
67          domain { n.Domain = $domain.d; }
68          (observable { n.Observables.Add($observable.o); })
69          observation*
70          problem { n.Problem = $problem.p; }
```

```
71          ;
72
73  domain returns [Domain d]
74          : { Console.WriteLine("Construct Domain"); }
75          '(' DEFINE '(' DOMAINDEF title { d = Domain.Get($title.s);
                } ')'
76                  types { d.Types = $types.ts; }
77                  predicates { d.Predicates = $predicates.ps; }
78                  (functions { d.Functions = $functions.fs; })?
79                  (action { d.Actions.Add($action.a); })*
80          ')'
81          ;
82
83  observable returns [Observable o]
84          : { o = new Observable(); Console.WriteLine("Construct
                Observable"); }
85          '(' OBSERVABLE ID { o.Y = $type.t; } '-' formula { o.
                Definition = $formula.f; }  ')'
86          ;
87
88  observation returns [Observation o]
89          : { Console.WriteLine("Construct Observation");  }
90          '(' OBSERVATION { o = new Observation(); } ')'
91          ;
92
93  problem returns [Problem p]
94          : { p = new Problem(); Console.WriteLine("Construct Problem
                ");  }
95          '(' DEFINE '(' PROBLEM t1 = title ')'
96                  '(' DOMAIN t2 = title ')' { p.Domain = Domain.Get(
                        t2); }
97                  (typedefs { p.Typedefs = $typedefs.ts; })?
98                  inits { p.Inits = $inits.ins; }
99                  (observability { p.Observability = $observability.o
                        ; })?
100                 goal { p.Goal = $goal.g; }
101         ')'  { p.Title = t1; }
102         ;
103
104 types returns [List<Type> ts]
105         : { ts = new List<Type>();  Console.WriteLine("Construct
                Types"); }
106         '(' TYPES (type { ts.Add($type.t); })+ ')' { int h = 0; }
107         ;
108
109 type returns [Type t]
110         : { t = null; Console.WriteLine("Construct Type"); }
111         '(' title { t = Type.Get($title.s); } ')'
112         ;
113
114 typedefs returns [List<Typedef> ts]
115         : { ts = new List<Typedef>();  Console.WriteLine("Construct
                Typedefs");  }
116         '(' TYPEDEF (typedef { ts.Add($typedef.t); })* ')'
117         ;
```

```
118
119 typedef returns [Typedef t]
120         : { t = new Typedef();  Console.WriteLine("Construct
                Typedef");  }
121         type { t.Type = $type.t; } '-' formula { t.Definition =
                $formula.f; }
122         ;
123
124 observability returns [Observability o]
125         : { Console.WriteLine("Construct Observability"); }
126         '(' OBSERVABILITY (
127           PARTIAL { o = Observability.Partial; }
128         | FULL { o = Observability.Full; }
129         ) ')';
130
131 inits returns [List<Init> ins]
132         : { ins = new List<Init>();  Console.WriteLine("Construct
                Inits"); }
133         '(' INIT (init { ins.Add($init.i); })* ')'
134         ;
135
136 init returns [Init i]
137         : { Console.WriteLine("Construct Init"); }
138         formula { i = new Init($formula.f); }
139         ;
140
141 goal returns [Goal g]
142         : { g = new Goal(); Console.WriteLine("Construct Goal");  }
143         '('
144         ( WEAKGOAL { g.Type = GoalType.Weak; }
145         | STRONGGOAL { g.Type = GoalType.Strong; }
146         | STRONGCYCLICGOAL { g.Type = GoalType.StrongCyclic; })
147         formula { g.Formula = $formula.f; }
148         ')';
149
150 predicates returns [List<Predicate> ps]
151         : { ps = new List<Predicate>();  Console.WriteLine("
                Construct Predicates"); }
152         '(' PREDICATES (predicate { ps.Add($predicate.p); })+ ')'
153         ;
154
155 predicate returns [Predicate p]
156         : {  Console.WriteLine("Construct Predicate"); }
157         '(' ID { p = Predicate.Get($ID.Text); } ')'
158         ;
159
160 functions returns [List<Function> fs]
161         : { fs = new List<Function>();  Console.WriteLine("
                Construct Functions"); }
162         '(' FUNCTIONS
163         ((function { fs.Add($function.f); })+)
164         ')'
165         ;
166
167 function returns [Function f]
```

```
168         : {  Console.WriteLine("Construct Function"); }
169         '(' ID ')' '-' type { f = Function.Get($ID.Text, $type.t);
                }
170         ;
171
172 action  returns [Action a]
173         : { a = new Action(); Console.WriteLine("Construct Action")
                ; }
174         '(' ACTION title
175         PRECONDITION formula
176         EFFECT formula
177         ')'
178         ;
179
180 formula returns [Formula f]
181         :  {  Console.WriteLine("Construct Formua"); }
182         (function { f = $function.f; }
183         | predicate { f = $predicate.p; }
184         | type { f = $type.t; }
185         | assign { f = $assign.a; }
186         | range { f = $range.r; }
187         | equals { f = $equals.e; }
188         | unknown { f = $unknown.u; }
189         | iff { f = $iff.i; }
190         | when { f = $when.w; }
191         | not { f = $not.n; }
192         | plus { f = $plus.p; }
193         | and { f = $and.a; }
194         | or { f = $or.o; }
195         | lt { f = $lt.lt; }
196         | gt { f = $gt.gt; }
197         | lte { f = $lte.lte; }
198         | gte { f = $gte.gte; }
199         | imply { f = $imply.i; })
200         ;
201
202 assign  returns [Assign a]
203         : { a = new Assign(); Console.WriteLine("Construct Assign")
                ;  }
204         '(' ASSIGN function formula { a.Function = $function.f; a.
                Formula = $formula.f; } ')'
205         ;
206
207 equals  returns [Equals e]
208         : { e = new Equals(); Console.WriteLine("Construct Equals")
                ;  }
209         '(' EQUALS f1 = formula f2 = formula { e.A = f1; e.B = f2;
                } ')'
210         ;
211
212 unknown returns [Unknown u]
213         : { u = new Unknown();  Console.WriteLine("Construct
                Unknown"); }
214         '(' UNKNOWN formula { u.Formula = $formula.f; } ')'
215         ;
```

```
216
217 range returns [Range r]
218         : { r = new Range();  Console.WriteLine("Construct Range");
                }
219         '(' RANGE '(' i1 = INT { r.From = int.Parse(i1.Text); } ')'
                  '(' i2 = INT { r.To = int.Parse(i2.Text); } ')' ')'
220         ;
221
222 not     returns [Not n]
223         : { n = new Not();  Console.WriteLine("Construct Not"); }
224         '(' NOT formula { n.Formula = $formula.f; } ')'
225         ;
226
227 iff     returns [IFF i]
228         : { i = new IFF(); Console.WriteLine("Construct IFF");  }
229         '(' IFF  f1 = formula f2 = formula { i.A = f1; i.B = f2; }
                ')'
230         ;
231
232 when    returns [When w] : '(' PLUS formula ; // not done
233
234 plus    returns [Plus p]
235         : { p = new Plus();  Console.WriteLine("Construct Plus"); }
236         '(' PLUS f1 = formula f2 = formula { p.A = f1; p.B = f2; }
                ')'
237         ;
238
239 minus   returns [Minus m]
240         : { m = new Minus();  Console.WriteLine("Construct Minus");
                }
241         '(' MINUS f1 = formula f2 = formula { m.A = f1; m.B = f2; }
                 ')'
242         ;
243
244 and     returns [And a]
245         : { a = new And();  Console.WriteLine("Construct And"); }
246         '(' AND (f = formula { a.Add(f); })+ ')'
247         ;
248
249 or      returns [Or o]
250         : { o = new Or(); Console.WriteLine("Construct Or");  }
251         '(' OR (f = formula { o.Add(f); })+ ')'
252         ;
253
254 lt      returns [LT lt]
255         : { lt = new LT();  Console.WriteLine("Construct LT"); }
256         '(' LT f1 = formula f2 = formula { lt.A = f1; lt.B = f2; }
                ')'
257         ;
258
259 gt      returns [GT gt]
260         : { gt = new GT();  Console.WriteLine("Construct GT"); }
261         '(' GT f1 = formula f2 = formula { gt.A = f1; gt.B = f2; }
                ')'
262         ;
```

```
263
264 lte     returns [LTE lte]
265         : { lte = new LTE();  Console.WriteLine("Construct LTE"); }
266         '(' LTE f1 = formula f2 = formula { lte.A = f1; lte.B = f2;
                } ')'
267         ;
268
269 gte     returns [GTE gte]
270         : { gte = new GTE();  Console.WriteLine("Construct GTE"); }
271         '(' GTE f1 = formula f2 = formula { gte.A = f1; gte.B = f2;
                } ')'
272         ;
273
274 imply   returns [Imply i]
275         : { i = new Imply(); Console.WriteLine("Construct Imply");
                }
276         '(' IMPLY f1 = formula f2 = formula { i.A = f1; i.B = f2; }
                ')'
277         ;
278
279 title returns [string s]
280         : {Console.WriteLine("Construct Title");}
281         ID { s = $ID.Text; }
282         ;
283
284 ID      : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'
    -')*  { Console.WriteLine("Construct ID [" + Text + "]"); }
285         ;
286
287 INT     : { Console.WriteLine("Construct INT"); }
288         '0'..'9'+
289         ;
290
291 COMMENT
292     :   '//' ~('\n'|'\r')* '\r'? '\n' { Skip();}
293     |   '/*' ( options {greedy=false;} : . )* '*/' { Skip(); }
294     ;
295
296 WS  :   ( ' '
297         | '\t'
298         | '\r'
299         | '\n'
300         ) { Skip();}
301     ;
302
303 STRING
304     :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
305     ;
306
307 fragment
308 HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;
309
310 fragment
311 ESC_SEQ
312     :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
```

```
313     |   UNICODE_ESC
314     |   OCTAL_ESC
315     ;
316
317 fragment
318 OCTAL_ESC
319     :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
320     |   '\\' ('0'..'7') ('0'..'7')
321     |   '\\' ('0'..'7')
322     ;
323
324 fragment
325 UNICODE_ESC
326     :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
327     ;
```

Appendix B

Project Code

The project code has not been included in the report as appendix, as this would bring the total number of pages up to a staggering amount. Rather here, is a short guide to the code, as included in the zip-file with the project. The program has been written in .NET C# and thus in order to achieve the best overview of the written code, the solution should be opened in Visual Studio 2010. If this is not available, any texteditor can open the `*.cs` files, however the dependancy links between the classes would be unavailable.

B.1 Structure

The program has four sub-projects in the solution: **CEP**, **NPDDL**, **Agent** and **Environment**. **CEP** stands for Conditional Epistemic Planner and is responsible for the planning process. The **Agent** project is responsible for specifying which input to load in, as well as instantiate and call the **CEP**. The **Environment** will be called whenever a simulation is needed. The last sub-project in the solution is the **NPDDL** project, which was never finished, but nevertheless has some of the code written.

Furthermore, the **CEP** project contains a number of sub-folders. These, as they are named, contains the data structure of the program `ModelTree`, `Model`, `EquivalenceClass`, `World`, `EventModel`, `Event`, `Formula`, `PropositionalSymbol` and so on. The *planning*

folder contains the classes and logic for the planning process and lastly the *grammar* folder contains the created code from the parser program `antlr`.

It should be noted, that even with the full solution file, the program cannot run without the specified example input files in the correct location. Cf. **Agent** in order to load in the correct input.

Appendix C

Examples

This chapter will list the four inputs shown in the report, namely Simple, Partial, NonDet and Complex.

C.1 Simple

```
Title: _sokobansimple
Symbols: t0, t1, t2, t3

Model _init = [ _w1 = t0 ]

EventModel _GoRight = [
  _gr1 = t0 ; ~t0 & t1,
  _gr2 = t1 ; ~t1 & t3]
EventModel _GoLeft = [
  _gl1 = t1 ; ~t1 & t0,
  _gl2 = t3 ; ~t3 & t1]
EventModel _GoUp = [ _gu = t3 ; ~t3 & t2 ]
EventModel _GoDown = [ _gd = t2 ; ~t2 & t3 ]

_goal = t2
```

C.2 Partial

```
Title: _sokobanpartial
Symbols: g1, g2, g3, t0, t1, t2, t3, t4, t5, t6, t7

Model _init = [
  _w1 = t0 & g1,
  _w2 = t0 & g2,
  _w3 = t0 & g3
  :: w1 = w2, w2 = w3 ]
EventModel _GoRight = [
  _gr0 = t0 ; ~t0 & t1,
  _gr11 = t1 & g1 ; ~t1 & t3,
  _gr12 = t1 & ~g1 ; ~t1 & t3,
  _gr21 = t3 & g2 ; ~t3 & t4,
  _gr22 = t3 & ~g2 ; ~t3 & t4,
  _gr31 = t4 & g3 ; ~t4 & t7,
  _gr32 = t4 & ~g3 ; ~t4 & t7 ]

EventModel _GoLeft = [
  _gl0 = t1 ; ~t1 & t0,
  _gl1 = t3 ; ~t3 & t1,
  _gl2 = t4 ; ~t4 & t3,
  _gl3 = t7 ; ~t7 & t4 ]

EventModel _GoUp = [

  _gu0 = t3 ; ~t3 & t2,
  _gu1 = t5 ; ~t5 & t4,
  _gu2 = t7 ; ~t7 & t6 ]

EventModel _GoDown = [
  _gd0 = t2 ; ~t2 & t3,
  _gd1 = t4 ; ~t4 & t5,
  _gd2 = t6 ; ~t6 & t7 ]

 _goal2 = (t2 & g1) | (t5 & g2) | (t6 & g3)
```

C.3 NonDet

```
Title: _sokobannondet
Symbols: t0, t1, t2, t3, t4

Model _init = [
  _w1 = t0
]

EventModel _GoRight = [
  _gr1 = t0 ; ~t0 & t1,
  _gr21 = t1 ; ~t1 & t3,
```

```
  _gr22 = t1 ; ~t1 & t4 ]

EventModel _GoLeft = [
  _gl1 = t1 ; ~t1 & t0,
  _gl2 = t2 ; ~t2 & t1 ]


EventModel _GoDown = [
  _gd11 = t1 ; ~t1 & t2,
  _gd12 = t1 ; ~t1 & t4,
  _gd2 = t4 ; ~t4 & t5 ]

EventModel _GoUp = [
  _gu1 = t3 ; ~t3 & t1,
  _gu21 = t5 ; ~t5 & t2,
  _gu22 = t5 ; ~t5 & t3 ]

_goal = t5
```

C.4 Complex

```
Title: _sokobancomplex
Symbols: s11, s21, s31, s41, s32, s42, s52, s23, s33, s43, s34, s44
    , b11, b21, b23, b31, b41, b32, b42, b52, b33, b43, b34, b44

Model _init = [
  _w1 = s21 & b32,
  _w2 = s21 & b42,
  _w3 = s21 & b52,
  _w4 = s21 & b33,
  _w5 = s21 & b43
  :: w1 = w2, w2 = w3, w3 = w4, w4 = w5
]

EventModel _goleft = [
  _e11 = s21 & ~b11 & ~b52 & b32 ; s52 & ~s21,
  _e12 = s21 & ~b11 & ~b52 & b42 ; s52 & ~s21,
  _e13 = s21 & ~b11 & ~b52 & ~(b32 | b42) ; s52 & ~s21,
  _e21 = s21 & ~b11 & ~b23 & b33 ; s23 & ~s21,
  _e22 = s21 & ~b11 & ~b23 & b43 ; s23 & ~s21,
  _e23 = s21 & ~b11 & ~b23 & ~(b33 | b43) ; s23 & ~s21,
  _e31 = s21 & ~b11 & ~b44 & b42 ; s44 & ~s21,
  _e32 = s21 & ~b11 & ~b44 & b43 ; s44 & ~s21,
  _e33 = s21 & ~b11 & ~b44 & b34 ; s44 & ~s21,
  _e34 = s21 & ~b11 & ~b44 & ~(b42 | b43 | b34) ; s44 & ~s21,
  s31 & ~b21 ; s21 & ~s31,
  _e71 = s41 & ~b31 & b32 ; s31 & ~s41,
  _e72 = s41 & ~b31 & b33 ; s31 & ~s41,
  _e73 = s41 & ~b31 & b34 ; s31 & ~s41,
  _e74 = s41 & ~b31 & ~(b32 | b33 | b34) ; s31 & ~s41,
  _e81 = s42 & ~b32 & b33 ; s32 & ~s42,
```

```
30    _e82 = s42 & ~b32 & b34 ; s32 & ~s42,
31    _e83 = s42 & ~b32 & ~(b33 | b34) ; s32 & ~s42,
32    _e91 = s52 & ~b42 & b43 ; s42 & ~s52,
33    _e92 = s52 & ~b42 & b44 ; s42 & ~s52,
34    _e93 = s52 & ~b42 & ~(b43 | b44) ; s42 & ~s52,
35    _e41 = s33 & ~b23 & ~b11 ; s11 & ~s33,
36    _e51 = s33 & ~b23 & ~b52 & b32 ; s52 & ~s33,
37    _e52 = s33 & ~b23 & ~b52 & b42 ; s52 & ~s33,
38    _e53 = s33 & ~b23 & ~b52 & ~(b32 | b42) ; s52 & ~s33,
39    _e61 = s33 & ~b23 & ~b44 & b42 ; s44 & ~s33,
40    _e62 = s33 & ~b23 & ~b44 & b43 ; s44 & ~s33,
41    _e63 = s33 & ~b23 & ~b44 & b34 ; s44 & ~s33,
42    _e64 = s33 & ~b23 & ~b44 & ~(b42 | b43 | b34) ; s44 & ~s33,
43    _e101 = s43 & ~b33 & b32 ; s33 & ~s43,
44    _e102 = s43 & ~b33 & b34 ; s33 & ~s43,
45    _e103 = s43 & ~b33 & ~(b32 | b34) ; s33 & ~s43,
46    _e111 = s44 & ~b34 & b32 ; s34 & ~s44,
47    _e112 = s44 & ~b34 & b33 ; s34 & ~s44,
48    _e113 = s44 & ~b34 & ~(b32 | b33) ; s34 & ~s44
49
50  ]
51
52  EventModel _goright = [
53    s11 & ~b21 ; s21 & ~s11,
54    _e71 = s21 & ~b31 & b32 ; s31 & ~s21,
55    _e72 = s21 & ~b31 & b33 ; s31 & ~s21,
56    _e73 = s21 & ~b31 & b34 ; s31 & ~s21,
57    _e74 = s21 & ~b31 & ~(b32 | b33 | b43) ; s31 & ~s21,
58    _e81 = s31 & ~b41 & b42 ; s41 & ~s31,
59    _e82 = s31 & ~b41 & b43 ; s41 & ~s31,
60    _e83 = s31 & ~b41 & b44 ; s41 & ~s31,
61    _e84 = s31 & ~b41 & ~(b42 | b43 | b44) ; s41 & ~s31,
62    _e91 = s32 & ~b42 & b43 ; s42 & ~s32,
63    _e92 = s32 & ~b42 & b44 ; s42 & ~s32,
64    _e93 = s32 & ~b42 & ~(b43 | b44) ; s42 & ~s32,
65    _e11 = s42 & ~b52 & ~b11 ; s11 & ~s42,
66    _e21 = s42 & ~b52 & ~b23 & b33 ; s23 & ~s42,
67    _e22 = s42 & ~b52 & ~b23 & b34 ; s23 & ~s42,
68    _e23 = s42 & ~b52 & ~b23 & ~(b33 | b34) ; s23 & ~s42,
69    _e31 = s42 & ~b52 & ~b44 & b42 ; s44 & ~s42,
70    _e32 = s42 & ~b52 & ~b44 & b43 ; s44 & ~s42,
71    _e33 = s42 & ~b52 & ~b44 & b34 ; s44 & ~s42,
72    _e34 = s42 & ~b52 & ~b44 & ~(b42 | b43 | b34) ; s44 & ~s42,
73    _e101 = s23 & ~b33 & b32 ; s33 & ~s23,
74    _e102 = s23 & ~b33 & b34 ; s33 & ~s23,
75    _e103 = s23 & ~b33 & ~(b32 | b34) ; s33 & ~s23,
76    _e111 = s33 & ~b43 & b42 ; s43 & ~s33,
77    _e112 = s33 & ~b43 & b44 ; s43 & ~s33,
78    _e113 = s33 & ~b43 & ~(b42 | b44) ; s43 & ~s33,
79    _e41 = s34 & ~b44 & ~b11 ; s11 & ~s34,
80    _e51 = s34 & ~b44 & ~b52 & b32 ; s52 & ~s34,
81    _e52 = s34 & ~b44 & ~b52 & b42 ; s52 & ~s34,
82    _e53 = s34 & ~b44 & ~b52 & ~(b32 | b42) ; s52 & ~s34,
83    _e61 = s34 & ~b44 & ~b23 & b33 ; s23 & ~s34,
84    _e62 = s34 & ~b44 & ~b23 & b43 ; s23 & ~s34,
```

```
  _e63 = s34 & ~b44 & ~b23 & ~(b33 | b43) ; s23 & ~s34


]


EventModel _goup = [
  s32 & ~b31 ; s31 & ~s32,
  s42 & ~b41 ; s41 & ~s42,
  _e11 = s33 & ~b32 & b42 ; s32 & ~s33,
  _e12 = s33 & ~b32 & ~b42 ; s32 & ~s33,
  _e21 = s43 & ~b42 & b32 ; s42 & ~s43,
  _e22 = s43 & ~b42 & ~b32 ; s42 & ~s43,
  _e31 = s34 & ~b33 & b43 ; s33 & ~s34,
  _e32 = s34 & ~b33 & ~b43 ; s33 & ~s34,
  _e41 = s44 & ~b43 & b33 ; s43 & ~s44,
  _e42 = s44 & ~b43 & ~b33 ; s43 & ~s44
  :: e11 = e12, e21 = e22, e31 = e32, e41 = e42
]

EventModel _godown = [
  _e11 = s31 & ~b32 & b42 ; s32 & ~s31,
  _e12 = s31 & ~b32 & ~b42 ; s32 & ~s31,
  _e21 = s41 & ~b42 & b32 ; s42 & ~s41,
  _e22 = s41 & ~b42 & ~b32 ; s42 & ~s41,
  _e31 = s32 & ~b33 & b43 ; s33 & ~s32,
  _e32 = s32 & ~b33 & ~b43 ; s33 & ~s32,
  _e41 = s42 & ~b43 & b33 ; s43 & ~s42,
  _e42 = s42 & ~b43 & ~b33 ; s43 & ~s42,
  _e51 = s33 & ~b34 & b44 ; s34 & ~s33,
  _e52 = s33 & ~b34 & ~b44 ; s34 & ~s33,
  _e611 = s43 & ~b44 & ~b11 ; s11 & ~s43,
  _e621 = s43 & ~b44 & ~b52 & b32 ; s52 & ~s43,
  _e622 = s43 & ~b44 & ~b52 & b42 ; s52 & ~s43,
  _e623 = s43 & ~b44 & ~b52 & ~(b32 | b42) ; s52 & ~s43,
  _e631 = s43 & ~b44 & ~b23 & b33 ; s23 & ~s43,
  _e632 = s43 & ~b44 & ~b23 & b43 ; s23 & ~s43,
  _e633 = s43 & ~b44 & ~b23 & ~(b33 | b43) ; s23 & ~s43

]

EventModel _pushleft = [
  s31 & b21 ; s21 & ~s31 & b11 & ~b21,
  s41 & b31 ; s31 & ~s41 & b21 & ~b31,
  s52 & b42 ; s42 & ~s52 & b32 & ~b42,
  s43 & b33 ; s33 & ~s43 & b23 & ~b33
]

EventModel _pushright = [
  s11 & b21 ; s21 & ~s11 & b31 & ~b21,
  s21 & b31 ; s31 & ~s21 & b41 & ~b31,
  s32 & b42 ; s42 & ~s32 & b52 & ~b42,
  s23 & b33 ; s33 & ~s23 & b43 & ~b33
]

```

```
140 EventModel _pushup = [
141   s33 & b32 ; s32 & ~s33 & b31 & ~b32,
142   s43 & b42 ; s42 & ~s43 & b41 & ~b42,
143   s34 & b33 ; s33 & ~s34 & b32 & ~b33,
144   s44 & b43 ; s43 & ~s44 & b42 & ~b43
145 ]
146
147 EventModel _pushdown = [
148   s31 & b32 ; s32 & ~s31 & b33 & ~b32,
149   s41 & b42 ; s42 & ~s41 & b43 & ~b42,
150   s32 & b33 ; s33 & ~s32 & b34 & ~b33,
151   s42 & b43 ; s43 & ~s42 & b44 & ~b43
152 ]
153
154 _goal2 = b21
155 _goal = (Kb32) | (Kb42) | (Kb52) | (Kb33) | (Kb43)
```