Creating an efficient translation from PDDL to DEL

Bachelor's Project Thesis

Paul Silm, s3753816, p.silm@student.rug.nl
Supervisor: Dr B.R.M. Gattinger

Abstract: Automated planning programs are used to solve planning problems using search algorithms. Most planners accept the Planning Domain Definition Language (PDDL) as an input language. PDDL uses syntactic approach, which works quite well in most scenarios. However it is not well suited to deal with knowledge of agents, especially knowledge about knowledge. Meta-knowledge can be more intuitively represented with a semantic approach, such as in Dynamic Epistemic Logic (DEL), but the syntax is quite different from that in PDDL, making conversion from PDDL to DEL difficult. Here we present an automatic translation from the PDDL extension MAEPL (Multi-Agent Epistemic Planning Language) to DEL. The translation is written in Haskell, converting plain MAEPL descriptions to the existing data structures used by the Symbolic Model Checker SMCDEL. Additionally our tool checks input for semantic consistency and reports any errors to the user. Examples are used to demonstrate that the translation is able to convert any semantically valid MAEPL problem into DEL. While the translation results in an optimal branching factor for the planning problems' search trees, the search space contains many actions that are never taken. We discuss some methods to increase efficiency and other ideas for future work.

1 Introduction

A search for a plan in a finite and discrete state and action space can be automatized using planning programs. Generally a plan is defined by a list of possible actions, the initial state and a way to check whether a goal state has been reached. Take the apartment key example discussed by Engesser, Bolander, Mattmüller, and Nebel (2017), where Anne wants to leave the apartment and lock the door, but Bob who is soon to arrive does not have his own key. The possible actions would be leaving the key under the mat, announcing that the key is under the mat, and taking the key from under the mat. The initial state would be Anne having the key and the goal is that Bob has the key. There are various languages that such a planning problem can be defined in, suitable for various needs, usually using different syntax. This can make it difficult for the researchers working in different planning domains to understand and learn from each other's work. The Planning Domain Definition Language (PDDL) is intended to solve this problem by allowing extensions (Haslum, Lipovetzky, Magazzeni, and Muise, 2019). Even if a planning approach is incompatible with standard PDDL, a new extension can be created to satisfy the approach's specific requirements. The essential structure of a PDDL problem only consists of predicates, actions, the initial state, objects and the goal formula.

The nature of standard PDDL demands the use of a syntactic approach for representing states. A syntactic representation of a state only considers the truth values of defined predicates. This works great when all predicates can be easily predefined, however this is not always the case. An example of this is representing knowledge of multiple agents, as not only their knowledge about predefined predicates needs to be considered, but also what they know about the knowledge of themselves or others. For example, does Bob know whether Anne knows whether she has the key? Many different scenarios can come up that need to be accounted for, and syntactic representation is not equipped to deal with them efficiently. Therefore while syntactic approach is simple and useful for centrally coordinated planning, it is not well suited for finding plans with multiple agents with distributed knowledge.

The semantic representation is a better alternative, as by representing states and actions as models it allows the use of modal logic. The epistemic planning field is specifically concerned with dealing with distributed knowledge and capabilities of agents, thereby achieving more natural planning solutions. The framework of Dynamic Epistemic Logic (DEL) has been shown useful for expressive epistemic planning. Recent work has extended the traditional DEL-based epistemic planning framework with perspective shifts using Kripke models (Engesser et al., 2017). Kripke models are a natural way of providing the agents with the Theory of Mind allowing them to understand what other agents know, and plan their actions and communication accordingly. The initial state in our key example, where Bob does not know where the key is, is represented in DEL as shown in Figure 1.1.

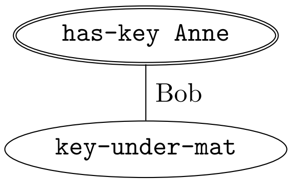


Figure 1.1: Initial state of the apartment key problem.

The predicates in each world show the list of all predicate instances true at the world. The double line around `has-key Anne` means that this is the actual world. The edge connecting the two worlds show that for Bob, these two worlds are indistinguishable. The model after Anne has put the key under the mat is shown in Figure 1.2.

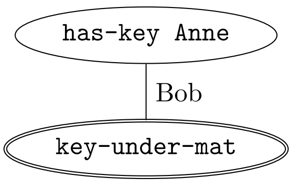


Figure 1.2: Model with the key under the mat.

We can imagine more possible worlds (e.g., one where both Anne and Bob have a key), but since neither Bob nor Anne considers them possible, they are not in the model in Figure 1.2. It is important to note that the model is common knowledge, meaning every agent knows that one of the worlds in the model is the actual world.

The use of this model allows each individual agent to plan for a common goal without the explicit commitment to any specific policy or need to negotiate with other agents prior to planning. Furthermore, agents can do strong planning, meaning they only take actions that are guaranteed to lead to the goal state (assuming all agents plan strongly) (Engesser et al., 2017). For example Anne has an option to leave the key under the mat, but it will be of no use to Bob if he does not know about its location. Therefore for Bob to be able to get the key Anne also needs to announce that she has left the key under the mat. The model after Anne's announcement is shown in Figure 1.3. Since neither agent considers the world where Anne still has the key possible, we can remove it. Now Bob can

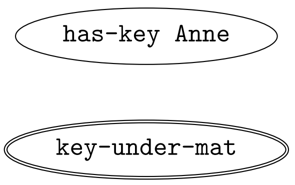


Figure 1.3: Model where Bob knows that the key is under the mat.

distinguish between the worlds. Having learnt that the key is under the mat he will retrieve it. Since Anne does not know about it, the new model is shown in Figure 1.4.

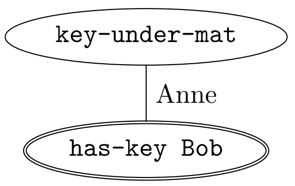


Figure 1.4: Goal state has been reached; Model showing that Bob has the key.

Since PDDL is ubiquitously used for planning applications, a translation from PDDL to DEL would help provide a better access to the newer multi-agent planning framework. The Multi-Agent Epistemic Planning Language (MAEPL) is an extension of the standard ADL subset of PDDL 1.2, and allows the definition of agents in addition to the predicates and actions in the domain description (Engesser, 2014). This is a version of PDDL that could be translated into DEL as it already requires the user to define a set of agents with their knowledge about the world and actions. However, PDDL has more features than DEL, and therefore much will necessarily be lost in translation between the languages.

Therefore the topic of this research project is to find out Which MAEPL features can be translated to an efficient representation of a DEL planning problem.

By efficiency we mean that any DEL problem translated from MAEPL can be solved as efficiently as the same problem written directly in DEL. This means there is no overhead caused by the translation. The goal is to find out what aspects of MAEPL cannot be carried over to DEL, and provide a Haskell module for translating from MAEPL to DEL for all those aspects that are needed and commonly in use. The source code is accessible at `https://github.com/paulsilm/PDDLtoDEL`.

The article is structured as follows: In Section 2 we cover the concepts required to understand and implement the translation, in Section 3 we cover the structure and the implementation choices for the program, in Section 4 we discuss the performance of the program and in Section 5 we discuss the results and some future work ideas.

2 Theoretical Background

Dynamic Epistemic Logic is a modal logic based on an epistemic language. It is a propositional logic with agents as separate entities. I is taken to denote the given finite set of *agents* in the model, and P is taken to denote the given finite set of atomic propositions.

2.1 Dynamic Epistemic Logic

The *epistemic language* definition used in this paper is the definition provided by Engesser et al. (2017). A formula φ in *epistemic language* $\mathcal{L}$ is defined in Equation 2.1, where $p \in P$, $i \in I$, $K_i\varphi$ denotes that the agent i knows φ and $C\varphi$ denotes that it is common knowledge that φ.

$$\varphi \ ::= \ \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i\varphi \mid C\varphi \qquad (2.1)$$

This definition as well as the semantics that govern $\mathcal{L}$ are discussed by Engesser et al. (2017). The common knowledge $C\varphi$ operator denotes that φ is shared knowledge among all agents, in such a way that at no level would any agent $i \in I$ doubt whether some other agent does not have the same level of knowledge about φ (Van Ditmarsch, van Der Hoek, and Kooi, 2007, pp. 31). By level of knowledge in this specific case we mean knowledge about knowledge with respect to φ, or something that can be represented as $K_i \dots K_j\varphi$ where $i, j \in I$. Intuitively this means that common knowledge must be infinitely recursive.

2.1.1 Epistemic Model

An *epistemic model* is a tuple defined in Equation 2.2

$$\mathcal{M} = \langle W, V, (\sim_i)_{i\in I}, W_d\rangle \qquad (2.2)$$

where

- The *domain* W is a non-empty finite set of *worlds*.
- $V : P \to W \to Bool$ is an evaluation function that assigns a truth value to each atomic proposition at each world.
- $\sim_i \subseteq W \times W$ is an equivalence relation called the indistinguishability relation for agent i.
- $W_d \subseteq W$ is the list of designated worlds. We consider multipointed models meaning there can be multiple designated worlds.

In simpler terms, the indistinguishability relation $\sim_i \subseteq W \times W$ means that the agent i cannot tell the two worlds apart — they belong to the same equivalence class for the agent i. The propositions are considered true if they are evaluated as true in the designated world. For an agent i to know that a formula φ is true — $K_i\varphi$ — φ needs to be true in all worlds that the agent considers possible. The formal semantics are omitted to save space, see Van Ditmarsch et al. (2007).

A model $\mathcal{M}$ can be represented as a graph, where the nodes are worlds with their evaluations, and indistinguishability relations $w_a \sim_i w_b$ are represented as edges, with a label i, connecting worlds w_a and w_b. The models in Figures 1.1 to 1.4 are all epistemic models.

A good example of an epistemic problem comes from the Three Muddy Children problem: Three children are playing in the mud, and during playing some mud can end up on their foreheads. They cannot see whether they themselves are muddy, but they can see if others are. They also know that other children are in the same position in that only their own state is obscured. The children are assumed to be perfect reasoners, i.e. if something can be inferred from the situation (or the Kripke model in this case), then they will know it. Their father comes and tells them that at least one of them is muddy. He then asks for all children, who know whether they are muddy or not, to step forward. He will repeat the question until all children know whether or not they are muddy.

The initial state is shown in Figure 2.1. A1, A2 and A3 are the three children, and the letters in the names of the worlds describe for each children whether they are muddy or clean, e.g., only the first child is muddy in world *mcc*. The designated world is the one where all children are muddy.

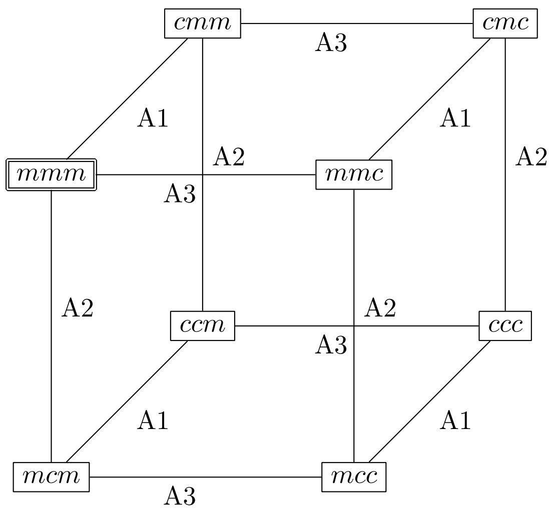


Figure 2.1: Three Muddy Children initial model.

In the model the front plane consists of nodes where the first child A1 is muddy, and back plane to where A1 is clean. A1 can distinguish between worlds that are within either plane, but not between the two planes, thus the edges connecting the corresponding nodes of front and back planes display the indistinguishability relationships of A1. The same applies to the other agents and dimensions, which is how we get a cube. For example if the children A2 and A3 were aware of their state of muddiness, the model would look like in Figure 2.2.

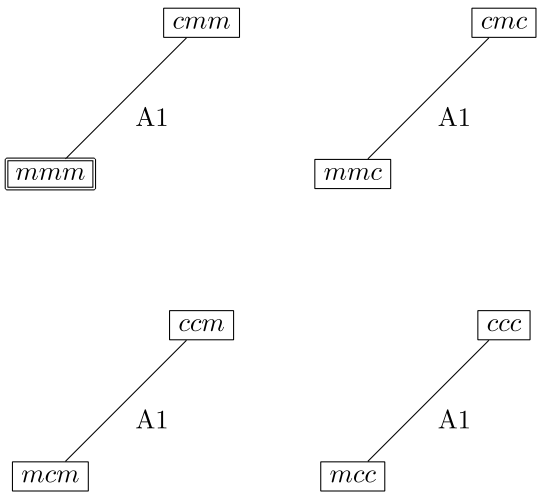


Figure 2.2: A Kripke model for 3 muddy children, where A2 and A3 can distinguish between all worlds but A1 cannot distinguish between worlds where the only difference is A1's state of muddiness.

2.1.2 Epistemic actions

What separates *dynamic epistemic logic* from *epistemic logic* is the update operation, with the information regarding the update being stored in the separate action model (Baltag, Moss, and Solecki, 2016).

An action model in DEL is defined similar to state model, with the addition of the precondition formula and the effect.

$$a = \langle\langle E, (\sim_i)_{i \in I}\rangle, E_d\rangle \tag{2.3}$$

The Equation 2.3 describes the action model where E is a list of events and $E_d \subseteq E$ is a list of designated events. Each event e in E is described as a tuple of the form $e = \langle \mathcal{P}, \mathcal{E}\rangle$ where $\mathcal{P}$ is a precondition formula, and $\mathcal{E}$ is the effect formula with a list of tuples $\langle p_k, f_k\rangle$ where p_k is an integer-labelled atomic proposition that is assigned the evaluation of the boolean formula f_k.

A simple action description is shown in Figure 2.3, where Anne is placing the key under the mat. This has two events, one designated and one not, distinguishable for Anne but not for Bob. The precondition of the actual event is `has-key Anne`, and the precondition of the trivial event is always true. The trivial event has no effect, but the actual event's effect is that `key-under-mat` is assigned $\top$ and `has-key Anne` is assigned $\bot$ — Anne's key is now under the mat.

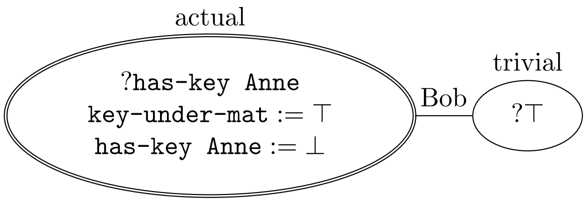


Figure 2.3: DEL action model for Anne placing the key under the mat.

When an action a is taken, the resulting model $\mathcal{M}'$ is calculated with a *product update* of $\mathcal{M}$ and a: $\mathcal{M} \otimes a = (\langle W', (\sim'_i)_{i\in I}, V'\rangle, W'_d)$ where

- $W' = \{(w, e) \in W \times E \mid \mathcal{M}, w \models \mathcal{P}_e\}$
- $\sim'_i = \{((w, e), (w', e')) \in W' \times W' \mid w \sim_i w' \wedge e \sim_i e'\}$
- $V'(p_k) = \{(w, e) \in W' \mid (p_k \in \mathcal{E}_e \wedge \mathcal{E}_{f_k} \models \top) \vee (p_k \notin \mathcal{E}_e \wedge \mathcal{M}, w \models p_k)$
- $W'_d = \{(w, e) \in W' \mid w \in W_d \text{ and } e \in E_d\}$

This means that the new set of worlds W' is a result of a cross product of all those worlds and events where the world satisfies the precondition of the event. The application of an event to a world — (w, e) — is in itself a new world. This intuitively means that all events are applied to all worlds that they can be applied to. The new indistinguishability relations $\sim'_i$ are then a result of a cross product of the new worlds, where the world (w, e) is indistinguishable for the agent i from the world (w', e') iff both the original worlds w and w' and the applied events e and e' were indistinguishable for i. This means that if the agent can tell the events apart, they will learn to distinguish the new worlds — they obtain knowledge about the model.

The evaluation of the propositions in each world is true iff it was set true in the effect of the event that resulted in the new world, or if it was true beforehand and it was not set to false in the event's effect. The list of designated worlds is then the cross product of the new worlds that were achieved as a result of a designated event being applied to a designated world. Note that applying an action with multiple designated events on a single designated world would result in multiple designated worlds. Similarly to multiple worlds being designated initially, these worlds correspond to different scenarios of the action, all of which need to be considered as s distinct true situation.

Considering the key example in Figure 1.1, when Anne takes the action of putting the key under the mat, both events are applied to both worlds.

Since the precondition of the actual event is not met for the world with `has-key Anne` then it is not applied to it. The result is three worlds, two with the key under the mat, and one where Anne still has it. Since Bob could not distinguish the worlds nor events, he is not able to distinguish the new worlds either. The two worlds with the key under the mat are therefore equivalent for Bob, and the resulting model is shown in Figure 2.4.

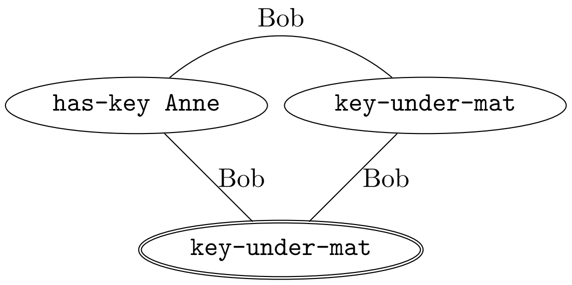


Figure 2.4: The model after Anne has put the key under the mat before simplification.

Since both the designated and non-designated worlds with `key-under-mat` are bisimilar — that is they have the same evaluation function V and the same relations — they can be merged into one designated world. After the two identical worlds have been merged the result is what we see in Figure 1.2. The third world where Anne still has the key comes from the trivial event being applied to the designated world where Anne has the key. This means that while Bob knows that the action of putting the key under the mat was applied, he did not observe whether the actual event took place.

Three Muddy Children can also be modified to a planning problem using DEL. In this case we want to find the sequence of actions leading to everyone knowing whether they are muddy. In the case displayed in Figure 2.1, all children are muddy, but nobody knows anything about their own muddiness. After father announces that at least one child is muddy, nobody still knows whether or not they themselves are muddy, because everybody can see at least one other child who is muddy. However, the fact that not all agents are clean became common knowledge. This removes the world *ccc* from the model, and we are left with the model in Figure 2.5.

Since nobody knows their state, after father makes the first request for children that know their state to step forward, nobody does. As a consequence it becomes common knowledge that there is at least two muddy children. If there had been one muddy child they would not have seen any other muddy child and could have concluded that they were the only muddy child. The model then becomes that in Figure 2.6.

Even though everybody knows that there are two muddy children, nobody can conclude anything

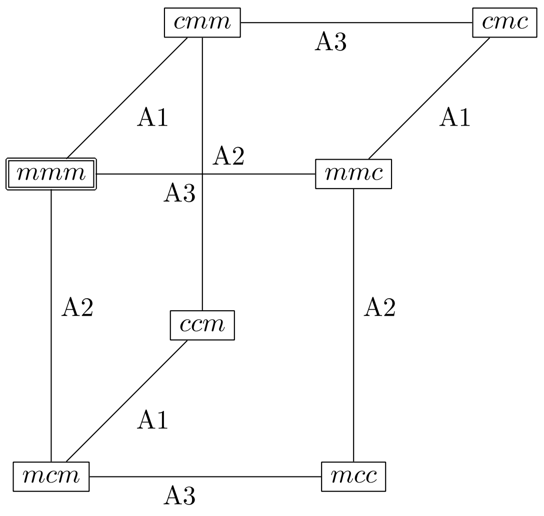


Figure 2.5: Three Muddy Children after the initial announcement of at least one child being muddy.

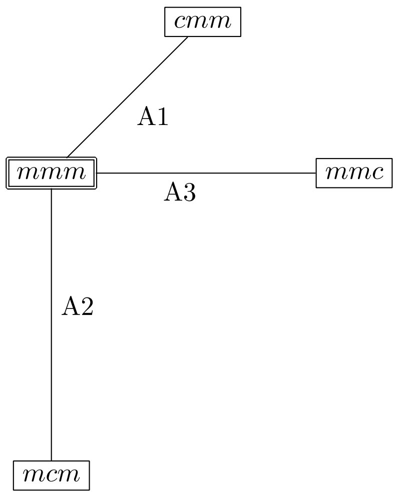


Figure 2.6: Three Muddy Children model after nobody stepping forward the first time.

about their own state since they have to consider the possibility that the other two children are the muddy ones. Therefore nobody still steps forward on the second request. Since now it is clear that there was nobody who could observe any clean child, the only remaining world in the model is *mmm*.

2.2 PDDL

PDDL is a universal planning language. A standard PDDL planning problem is split in two parts: The domain and the problem. The domain defines the circumstances, and can be used for many different problems. The domain consists of the following:

- An optional list of constant objects, present in any problems coupled to the domain.
- A list of first order logic predicate definitions.

- A list of actions, each with a precondition formula that needs to be true for the action to be taken, and an effect formula assigning values to predicates.

The PDDL problem defines the particular situation for which a plan is sought. It consists of the following elements:

- Objects present in the problem.
- Initially true predicates.
- A goal formula.

The planner is then used to find an ordered list of actions that can be taken in order to reach a state that satisfies the goal formula. Due to the universal purpose of PDDL the standard definition limits the problems that can be described, and therefore there are many extensions that allow the application creator and user to choose what features should be implemented. There are predefined extensions that can be listed as requirements for the planner in the domain file. The early detection of unsupported requirements saves the user from the trouble of fruitless debugging.

A comprehensive definition of the PDDL language with the predefined extensions is described at `https://planning.wiki/ref/pddl`.

2.3 MAEPL

Not all extensions are part of the language definitions, and MAEPL, created by Engesser (2014) to describe epistemic models in detail, is a further extension of the ADL (Action Description Language) extension of PDDL version 1.2. ADL is a predefined well supported super requirement, that includes the following requirements:

- Typing, which works similarly to object oriented programming classes, supporting subtypes. If the requirement is defined, each object must be declared with a type, e.g.,
  `L1 - letter`
  `A1 A2 - agent`.
- STRIPS, which allows actions to change the values of predicates via the use of the effect statement.
- Usage of disjunction in formulas.
- Usage of equality in formulas.
- Usage of exists and forall statements in formulas.
- Usage of implication in effects.

The MAEPL extension adds features required to describe an epistemic model. The formulas are extended further with the *knows* and *common-knowledge* statements, corresponding respectively to the $K_i\varphi$ and $C\varphi$ operations discussed in section 2.1.1. While the original extension by Engesser (2014) only added those to the goal formula, the restriction was relaxed here since it is intuitive to also use the statement in precondition formulas. The MAEPL files describing the apartment key and three muddy children problems are shown in Appendix A.

The actions are extended with an optional statement to declare the actor, and the action can be split into a list of events, some of which are designated — meaning they are executed when the action is taken and the preconditions are met. An event is defined by its precondition and effect statements. The non-designated events are never executed, but the agents can consider them possible. The indistinguishability relations of the events are then described in the observability statements that denote which events which agents can distinguish.

The problem file is similarly extended with a list of worlds, each of which have a list of initially true predicates and a keyword to tell whether it is designated. The worlds are followed by the observability statements describing the indistinguishability relations of the worlds.

Agent is defined as a regular type and the agents can be declared in the list of constants in the domain or the list of objects in the problem file, like any ordinary PDDL object.

The original PDDL language limits the contents of conjunction, disjunction and implication to literals, that is predicates or negations thereof. The restriction was relaxed to allow all formulas, since they are supported by DEL.

2.4 Planning

Planning itself is not the main topic here and is delegated to SMCDEL. A DEL action plan is an ordered list of actions that are applied to the initial state, leading to a state where the goal formula is true. When there are multiple designated worlds at any state, the plan needs to be such that for all designated world the goal state will be reached. A plan can also be implicitly coordinated, meaning all agents employ strong planning. If an agent does strong planning, he will not take any actions that are not guaranteed to lead to the goal. An implicitly coordinated plan is sometimes more intuitive, such as in the apartment key example, but is not always possible. For example if it were not possible to announce the key's location, Anne knows that even if she places the key under the doormat, Bob is not going to spend energy trying out actions (or searching for the key) if he is not sure that they lead to him getting in the apartment.

3 Translation

The program we present here has four components: The parser that parses the MAEPL description to a Haskell datatype, a semantic checker that checks if the input is a valid description of an (epistemic) planning problem, the mapping from the MAEPL datatype to a DEL datatype, and finally the adjusted planning functions from SMCDEL. The Haskell datatype for DEL was taken from the SMCDEL module, since it allows testing the validity of the output and provides auto-generated graphical representations of its models.

3.1 MAEPL parsing

The parsing functions are generated via the parser generator *Happy* (Gill and Marlow, 2021) and lexer *alex* (Dornan and Marlow, 2021). The PDDL datatype is used to store the MAEPL file input in a more structured form, providing an intuitive definition that matches well with the plain text format. The PDDL type comprises of Domain and Problem types. Domain type consists of the name, the requirements, the object type names, constant objects, predicates and actions, meaning it has one Haskell type for each statement MAEPL statement. The problem type consists of problem name, domain name, objects, initially true predicates, a list of worlds, list of observabilities and a goal formula. The Action datatype consists of a list of typed parameters, the name of the action taker, and list of events and observabilities. Each event type consists of a name, a boolean of whether it is designated or not, and the precondition and effect formulas. The effect is stored as a formula as opposed to a list of tuples as would be suitable for DEL, since it is defined as a formula in PDDL.

3.2 Semantic checking

The semantic checker ensures that the input description can be translated to a valid DEL planning problem. This was implemented mostly to simplify the debugging experience while writing a MAEPL file for a planning problem. It helps avoid undescriptive errors during the translation process, and allows more specific detection of the problem, since it can print the whole formula where, why and what problem occurs. The semantic checker checks that:

- The domain name is the same in both the problem and domain descriptions.
- There are no duplicate definitions (with the same name) of predicates, variables, objects, types, actions, events and worlds.
- All variables are defined and their defined types match the predicate parameter's type.
- The partitions in the observabilities only include existing event names for event observabilities, and world names for world observabilities.
- There is at least one designated world and at least one designated event in every action.
- The actors are agents.

Note that there is no check for clashing names between different datatypes, e.g., there might simultaneously be an object, a domain, a type and a predicate named ”first”. This does not break the translation nor the planner since the objects of different datatypes are never compared.

The semantic checker is also checking for certain flaws in the description that would be possible to avoid with the parser, for two reasons. Firstly, the parser errors are more complicated to design and in the current implementation are limited to only letting the user know the point where the parsing failed. With a semantic checker it is easier to describe the precise reason why the failure occurred, as the user might be unaware of the detailed definition of MAEPL. Secondly, it has the added benefit of limiting the parser's complexity, allowing incorrect definitions to pass the parser in exchange for simpler datatype definitions and translation code. The complementary checks ensure that:

- The effect formula is a conjunction consisting of elements that are either literals or implications of the following form:

  ```
  EffectElement := Literal
                 | Conj<Literal>
                 | Form -> EffectElement
  ```

  Note that the implication is only allowed once i.e. the `EffectElement` that is implied by a formula can only be a (conjunction of) literal(s).
- The predicates in the predicate list of the domain are defined with the types of the variables.
- The predicates inside formulas are specific instances of a predicate.
- The predicates that are assigned truth values either in the initial state description in the problem or as a result of an action are not equality predicates.

- The quantifiers introduce variables which do not yet exist, and there is at least one variable.

3.3 SMCDEL

The target of the translation in SMCDEL is a planning task, which is represented as a tuple with the initial Kripke model $\mathcal{M}$, the set of owned action models $\mathcal{A}$, and the goal formula g as follows:

$$\langle \mathcal{M}, \mathcal{A}, g \rangle$$

The initial model is a multi-pointed Kripke S5 model, which is constructed the same as in Equation 2.2. Actions are defined as multi-pointed S5 action models, which are constructed the same as in Equation 2.3. The action models in SMCDEL are given a label to make the final plan readable. The action model $A \in \mathcal{A}$ is labelled and owned:

$$A = \langle j, \langle l, a \rangle \rangle$$

where l is the action's label and $j \in \mathcal{I}$ is the actor.

3.4 Translation to DEL

The goal formula is mapped one-to-one from the PDDL goal formula, where forall and exists statements are translated respectively to a conjunction and disjunction. Each element in the conjunction/disjunction corresponds to the formula with a unique choice of objects matching the variable types. For example, if there are two agents `A1 A2 - agent` then `?a` in the formula

```
exists (?a - agent)
  (and (muddy ?a) (not (knows (muddy ?a)))
```

gets converted to `A1` and `A2`, resulting in the DEL formula in Equation 3.1 where p_1 is the SMCDEL proposition corresponding to the PDDL predicate instance of `muddy A1` and p_2 corresponds to `muddy A2`. Note that subtypes are currently not supported, and we hope to implement them in the future.

$$\bigvee \{(p_1 \wedge \neg K_{A1} p_1), (p_2 \wedge \neg K_{A2} p_2)\} \tag{3.1}$$

The Kripke model representing the starting epistemic model is translated from the problem definition, the worlds are translated directly, and the indistinguishability relations are derived from the observability statements. While the goal and the worlds with their observabilities are all found in the problem file, the domain file is needed for the predicates, constant objects, and types, therefore the problem file cannot be separately translated.

The observability statements are mapped to the indistinguishability relations. For example, the observability statement in Listing 3.1 can be read as "`A1` can tell the difference between four equivalence classes, each of which contain two worlds, one where he is muddy and one where he is not". The worlds in the same brackets belong to the same equivalence class, meaning the agent cannot distinguish them from each another. This would be represented in the Kripke model as shown in Figure 2.2. If we then would add the second and third agents' observabilities we would get the same graph as in Figure 2.2.

Listing 3.1: A1's initial observability in Three Muddy Children problem as defined in MAEPL

```
Obs_A1 =
  (partition (cmm mmm)
             (cmc mmc)
             (mcm ccm)
             (mcc ccc))
```

The action models are translated from the MAEPL actions. The events, the actor, the action label, and designated events are taken directly from the MAEPL description, and the indistinguishability relations are derived again from the observability statements. However, as discussed in 2.3, the PDDL standard uses first order logic, whereas DEL is based on propositional logic. This means that to translate PDDL (precondition, effect, and goal) formulas to DEL, it is first necessary to convert each first order logic predicate to an atomic proposition. This is achieved by first choosing the parameters of the predicates. Since there are no variables in the problem definition (except if a quantifier is present in the goal formula), the goal formula already has specific predicate instances and can be easily translated.

The predicate translation is achieved by creating all possible applications of the predicates by substituting the parameters with the constants and objects, and mapping them to SMCDEL's atomic propositions. That way each PDDL predicate instance has a corresponding atomic proposition that can be used in action translations.

The MAEPL actions have a similar issue since they are functions — they take parameters and return a result — thus each MAEPL action is translated into multiple DEL actions, each corresponding to a choice of objects as parameters. The scope of the parameters is only within the actions, yet the scope of regular objects is in the problem file. If it is important that a specific agent can take an action or some parameter must not be equal to some object, it is necessary to declare these objects as constant objects in the domain file.

Since unlike DEL the ADL subset of PDDL also allows using the equality predicate, the equality predicate is mapped onto $\top$ or $\bot$ instead of a formula or a proposition. While it might mean that an action with a precondition that cannot be met is listed among others, it will never be executed, therefore the branching factor in the search for a plan is not affected.

4 Evaluation

We now discuss the translation and planning performance of our program through the example problems discussed in the previous sections.

4.1 Translation

The translation, as discussed in Section 3, takes a MAEPL file and converts it into a planning problem in SMCDEL. An optional argument was implemented in the program to print the initial world model and all action models to tikz models in LaTeX. Since the printing takes a SMCDEL model, the output shows integer-labelled propositions in the world nodes.

4.1.1 Apartment Key

The apartment key problem from the introduction is used here with a small adjustment. To display the possibility of planning with multiple designated worlds, we adjust the action of placing the apartment key under the rug in a way that the action can also be taken when the key is already under the mat. Since the action can now be taken whether or not Anne has the key, the planner can find a solution for both when Anne starts out with having the key and when the key is under the mat. The MAEPL description of `put-key` with the rest of the problem and domain description is shown in Appendix A.2.

The mapping from PDDL predicates to DEL propositions p_k in the worlds for the apartment key problem is displayed in Listing 4.1.

The resulting initial world model can be seen in Figure 4.1. As expected, the model is the same as in Figure 1.1 except that both worlds are designated.

The `put-key` actions are shown in Figure 4.2. Note that the action in Figure 4.2a differs from that in 2.3 by allowing the actual event to take place even if the key is already under the mat.

The `announce` actions are shown in Figure 4.3. The action models are identical, but the agent taking the action must know that the action can be taken. Since both agents know that the action

Listing 4.1: Translations for propositions in the apartment key problem.

```
p1 = key-under-mat
p2 = has-key anne
p3 = has-key bob
```

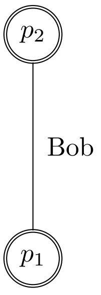
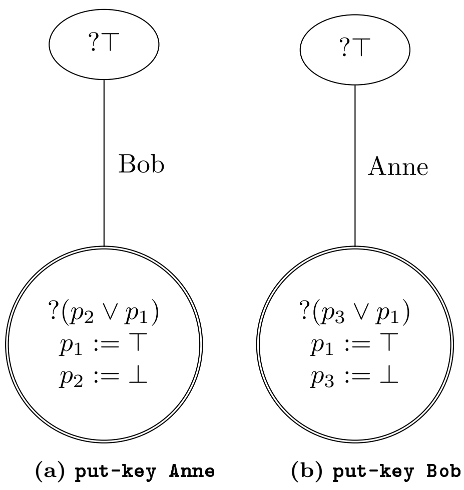


Figure 4.1: Translated initial model of the adjusted apartment key situation.



(a) `put-key Anne` **(b) `put-key Bob`**

Figure 4.2: Action models for the MAEPL action of `put-key ?a - agent`.

is only taken when the precondition is met, the action serves as the announcement that the key is under the mat.

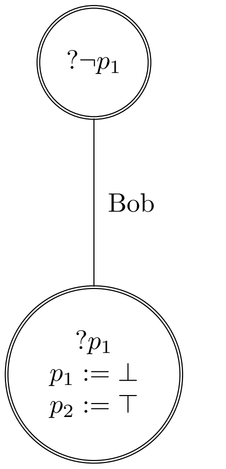
(a) `announce Anne` **(b) `announce Bob`**

Figure 4.3: Action models for the MAEPL action of `announce ?a - agent`.

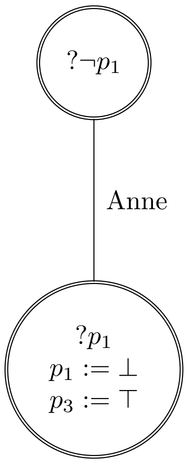
The `try-take` action models are shown in Figure 4.4. In either case, if the key was not under the mat, nothing changes, but if it was, the agent takes it. Naturally neither agent can observe what the other agent's action's outcome is.

4.1.2 Three Muddy Children

The Three Muddy Children discussed in Section 2 can also be described as a planning problem. Initially we created one action with two events — one where the actor steps forward and one where they do not. This however turned out to be an incorrect implementation since the first agent to let others know about his own state will give away



(a) `try-take Anne`



(b) `try-take Bob`

Figure 4.4: Action models for the MAEPL action of `try-take ?a - agent`.

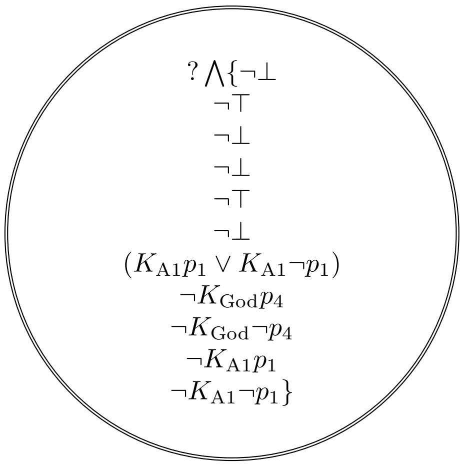


Figure 4.5: Action model corresponding to `knn A1 God A1`.

the crucial opportunity to learn about the other agents' original knowledge, thereby never learning his own state. Since simultaneous actions are not supported in SMCDEL planning functions, the correct implementation involves a new agent, called god (or father). God can take any of four actions, each corresponding to a state of different number of children knowing their states. The MAEPL description of the domain with these four actions is shown in Appendix A.1.1.

This is actually an example of how centralized coordination can be embedded as an implicit coordination problem in MAEPL, as one agent knows all the states and takes all the actions. While SMCDEL allows for global actions — actions without an actor — that would serve the same purpose, global actions are not supported by the translation because SMCDEL does not support planning with a mix of global and owned actions.

The initial model, as defined by the MAEPL file in Appendix A.1.2, is too large to fit in the text and is therefore shown in Appendix C in Figure C.1. The model matches perfectly with the desired model shown in Figure 2.1.

The translation includes a total of 257 action models, whereas only 25 have satisfiable precondition. For example, the MAEPL action `knn` takes three agents as parameters, which means the list `[A1, God, A1]` is also accepted as parameters even though the preconditions check that no parameter is equal to God and no two parameters are the same. The example action model, corresponding to the action `knn A1 God A1` is shown in Figure 4.5. The first six elements in the conjunction are checking whether the parameters are valid, which in this case they are not, and the K_ip predicates check that only the first child knows their state of muddiness.

4.2 Planning

Using the `--debug` argument we can see the intermediate model states throughout the planning. The intermediate states are useful to see how the model changes throughout the plan. An example debugging output using the apartment key problem is shown in Appendix B. The debug output first prints whether the parsing was successful, then prints the Haskell datatype of PDDL, and then the initial multi-pointed Kripke model. For the explicit search the BFS search tree is printed, for the implicit search the Iterative Deepening DFS tree is printed. The first two actions in Appendix B show the BFS search tree at depth one. The indented actions are taken at depth two. Each action label is followed by the Kripke model on which it is applied. To understand the Kripke model datatype, see `https://github.com/jrclogic/SMCDEL`.

4.2.1 Apartment Key

As discussed in Section 4.1.1, the initial model has two designated worlds for which the planner needs to find a solution. If we run the program on the apartment key problem with out any arguments we get the plan shown in Listing 4.2. The plan shows that to reach the goal state, first Anne should take the action `put-key` with the parameter list `["anne"]`, and then Bob should take the action `try-take` with the parameter list `["bob"]`.

Listing 4.2: Explicitly coordinated plan to solve the apartment key problem.

```
anne: put-key ["anne"];
bob: try-take ["bob"].;
```

To see what plan the agents would come up with if they were to plan strongly, we use the implicit coordination flag `-ic`. Running the program with the flag we get the plan shown in Listing 4.3. The sequence of actions is the same as what was discussed in Section 1. The initial model is shown in Figure 4.1. The model after Anne has taken the action of putting the key under the mat is shown in Figure 4.6. Note that this is semantically equivalent to the model in Figure 2.3, since the three worlds where `key-under-mat` is true are bisimilar.

Listing 4.3: Implicitly coordinated plan to solve the apartment key problem.

```
anne: put-key ["anne"];
anne: announce ["anne"];
bob: try-take ["bob"].;
```

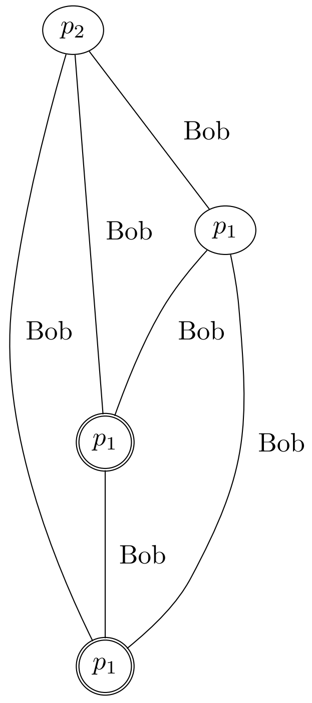


Figure 4.6: Model after Anne places the key under the mat.

After Anne announces that the key is under the mat, the new state becomes as shown in Figure 4.7. Note that this is equivalent to the model in Figure 1.3 — the missing non-designated world in Figure 4.7 is irrelevant since it is not connected to any designated world. Furthermore, the two worlds where the key is under the mat are bisimilar, and can be simplified into one.

The `--debug` option does not show the final output, but it is clear that since all worlds have `key-under-mat` as true, then the action model in Figure 4.4b will succeed, making `has-key Bob` true in all worlds. Since all designated worlds have the goal formula as true, we have reached the goal state.

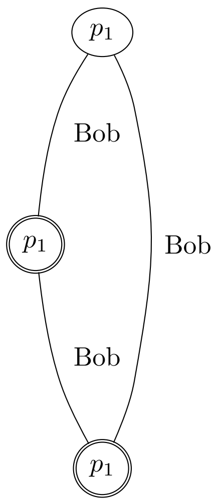


Figure 4.7: Model after Anne announces that the key is under the mat.

5 Discussion

Based on the examples discussed in Section 4, the translation results in the correct output. However, as seen in the case of Three Muddy Children, there can be many unnecessary actions resulting in inefficient planning.

5.1 Correctness of translation

The two problems translated in Section 4 cover most of the features MAEPL has to offer. The indistinguishability relations, formulas, events, worlds and predicates were all tested to give the correct output. The `common-knowledge` operation as well as implications and quantification formulas with multiple variables have not been tested, but since they constitute only a small part of the formula definition, the reader can verify for themselves that the SMCDEL formula indeed corresponds to the MAEPL formulas involving these cases. Despite having tested a broad scope of features with the two examples, it might be possible that there are certain edge cases which have not been accounted for. Since it is not obvious what situations could constitute test cases, an automatic test suite would be a viable solution to increase the probability that the translation works correctly. Currently subtyping is not supported despite being required by MAEPL. This limits the expressivity of the accepted input files, but the translation correctness is not affected.

5.2 Efficiency of translation

The initial state model with the indistinguishability relations is mapped one to one from MAEPL to SMCDEL for the worlds, true predicate instances, and observabilities for all present agents. The actions however could not be mapped one to one, since DEL actions do not have parameters unlike in PDDL. It is currently possible to write a fully efficient planning problem in MAEPL, if no parameters are used and every action is taken by an agent defined as a constant. The usage of subtypes can solve the problem in a similar manner, where each agent could have its own separate subtype of agent. This would allow the use of parameter variables simplifying the problem description in cases where not all actions require unique parameters. While the translation is therefore efficient in some respect, it is not easier to write a problem description in MAEPL than in DEL. A better solution, complementary to subtyping, would be to check the satisfiability of each action's precondition formula, thereby overcoming the shortcoming of subtyping where unique objects are required as parameters. Further optimizations like recursively converting propositions that can never become true to $\perp$ in all formulas, and then filtering out actions with unsatisfiable preconditions would filter out all actions that can never be taken (Engesser, 2014).

5.3 Future Work

Since subtyping is required by MAEPL but is not yet supported, it is high up on the priority list for improvements. Implementing a test suite and improving the efficiency are the other obvious improvements to the project. Another simple improvement would be adjusting the SMCDEL planning functions to allow a mix of global and owned actions. Simple convenience features like SMCDEL model to LaTeXprinting with automatic translation to PDDL predicates or a web interface would reduce the entry barriers and support adoption.

A larger project would be a translation from DEL to MAEPL. This could be either straight from SMCDEL description or from a DEL planning problem file. The major aspects which are lost in translation from MAEPL to DEL — names of object, constants, worlds, actions or events, and most notably the predicates — cannot be recovered. While undescriptive names are not necessarily a problem — since a DEL description does not have descriptive names in the first place — the first order logic predicates and original action definitions are difficult to recover. A naive approach to convert each DEL action to one MAEPL action would not be difficult to implement, and in case of an optimized DEL problem (using the techniques covered in 5.2) the naive approach might work the best. Unoptimized DEL problems could use clever tricks to return the original structure of MAEPL predicates and actions. Perhaps a more tangible enhancement would be to convert the MAEPL descriptions to the symbolic structures of SMCDEL. The planning aspect of SMCDEL is currently only implemented using explicit representations of states and actions, as covered in 3.3. However the symbolic representation might provide for more efficient planning, and while SMCDEL has a translation from explicit to symbolic structures, it might be more efficient to translate directly from MAEPL.

References

A. Baltag, L. S. Moss, and S. Solecki. The logic of public announcements, common knowledge, and private suspicions. In *Readings in Formal Epistemology*, pages 773–812. Springer, 2016. doi:10.1007/978-3-319-20451-2_38.

C. Dornan and S. Marlow. Alex: A lexical analyser generator for Haskell (version 3.2.6), 2021. URL `https://www.haskell.org/alex/`.

T. Engesser. Cooperative epistemic planning. Master's thesis, University of Freiburg, 2014.

T. Engesser, T. Bolander, R. Mattmüller, and B. Nebel. Cooperative epistemic multi-agent planning for implicit coordination. In S. Ghosh and R. Ramanujam, editors, *Proceedings of the Ninth Workshop on Methods for Modalities*, Electronic Proceedings in Theoretical Computer Science, pages 75–90. Open Publishing Association, 2017. doi:10.4204/EPTCS.243.6.

A. Gill and S. Marlow. Happy: the parser generator for Haskell (version 1.20.0). 2021. URL `https://www.haskell.org/happy/`.

P. Haslum, N. Lipovetzky, D. Magazzeni, and C. Muise. An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187, 2019. doi:10.2200/S00900ED2V01Y201902AIM042.

Hans Van Ditmarsch, Wiebe van Der Hoek, and Barteld Kooi. *Dynamic epistemic logic*, volume 337. Springer Science & Business Media, 2007.

A Appendix: Example MAEPL files

A.1 Three Muddy Children

A.1.1 Three Muddy Children Domain

```
(define (domain muddy)
    (:requirements :strips :typing)
    (:types agent)
    (:constants A1 A2 A3 God - agent)
    (:predicates
        (muddy ?a - agent)
    )

    (:action announce-at-least-one
        :parameters ()
        :byagent God
        (:event-nondesignated trivial
            :precondition (and)
            :effect (and)
        )
        (:event-designated actual
            :precondition
                (exists (?b - agent) (muddy ?b))
            :effect
                (and)
        )
    )

    (:action round-kkk
        :parameters (?a1 ?a2 ?a3 - agent)
        :byagent God
        :precondition
            (and
                (not (= ?a1 ?a2))
                (not (= ?a1 ?a3))
                (not (= ?a2 ?a3))
                (not (= God ?a1))
                (not (= God ?a2))
                (not (= God ?a3))
                (or (knows ?a1 (muddy ?a1))
                    (knows ?a1 (not (muddy ?a1))))
                (or (knows ?a2 (muddy ?a2))
                    (knows ?a2 (not (muddy ?a2))))
                (or (knows ?a3 (muddy ?a3))
                    (knows ?a3 (not (muddy ?a3))))
            )
        :effect (and)
    )

    (:action round-kkn
        :parameters (?a1 ?a2 ?a3 - agent)
        :byagent God
        :precondition
            (and
                (not (= ?a1 ?a2))
                (not (= ?a1 ?a3))
                (not (= ?a2 ?a3))
                (not (= God ?a1))
                (not (= God ?a2))
                (not (= God ?a3))
                (or (knows ?a1 (muddy ?a1))
                    (knows ?a1 (not (muddy ?a1))))
                (or (knows ?a2 (muddy ?a2))
                    (knows ?a2 (not (muddy ?a2))))
                (not (knows ?a3 (muddy ?a3)))
                (not (knows ?a3 (not (muddy ?a3))))
            )
        :effect (and)
    )

    (:action round-knn
        :parameters (?a1 ?a2 ?a3 - agent)
        :byagent God
        :precondition
            (and
                (not (= ?a1 ?a2))
                (not (= ?a1 ?a3))
                (not (= ?a2 ?a3))
                (not (= God ?a1))
                (not (= God ?a2))
                (not (= God ?a3))
                (or (knows ?a1 (muddy ?a1))
                    (knows ?a1 (not (muddy ?a1))))
                (not (knows ?a2 (muddy ?a2)))
                (not (knows ?a2 (not (muddy ?a2))))
                (not (knows ?a3 (muddy ?a3)))
                (not (knows ?a3 (not (muddy ?a3))))
            )
        :effect (and)
    )

    (:action round-nnn
        :parameters (?a1 ?a2 ?a3 - agent)
        :byagent God
        :precondition
            (and
                (not (= ?a1 ?a2))
                (not (= ?a1 ?a3))
                (not (= ?a2 ?a3))
                (not (= God ?a1))
                (not (= God ?a2))
                (not (= God ?a3))
                (not (knows ?a1 (muddy ?a1)))
                (not (knows ?a1 (not (muddy ?a1))))
                (not (knows ?a2 (muddy ?a2)))
                (not (knows ?a2 (not (muddy ?a2))))
                (not (knows ?a3 (muddy ?a3)))
                (not (knows ?a3 (not (muddy ?a3))))
            )
        :effect (and)
    )
)
```

A.1.2 Three Muddy Children Problem

```
(define (problem muddy-3)
    (:domain muddy)
    ; All agents know that other agents know whether they're muddy or not
        (:world-nondesignated ccc)
        (:world-nondesignated ccm
            (muddy A3))
        (:world-nondesignated cmc
            (muddy A2))
        (:world-nondesignated cmm
            (muddy A2)
            (muddy A3))
        (:world-nondesignated mcc
            (muddy A1))
        (:world-nondesignated mcm
            (muddy A1)
            (muddy A3))
        (:world-nondesignated mmc
            (muddy A1)
            (muddy A2))
        (:world-designated mmm
            (muddy A1)
            (muddy A2)
            (muddy A3))
    (:observability (partition (cmm mmm) (cmc mmc) (mcm ccm) (mcc ccc)) A1)
    (:observability (partition (cmm ccm) (cmc ccc) (mcm mmm) (mmc mcc)) A2)
    (:observability (partition (ccm ccc) (cmc cmm) (mmm mmc) (mcm mcc)) A3)
    (:goal
        (forall (?a - agent)
            (or
                (= ?a God)
                (knows ?a (muddy ?a))
                (knows ?a (not (muddy ?a)))
            )
        )
    )
)
```

A.2 Apartment Key

A.2.1 Apartment Key Domain

```
(define (domain doormat)
  (:types agent)
  (:predicates
    (key-under-mat)
    (has-key ?a - agent)
  )

  (:action put-key
    :parameters (?a - agent)
    :byagent ?a
    (:event-nondesignated trivial
      :precondition
        (and)
      :effect
        (and)
    )
    (:event-designated actual
      :precondition (or (has-key ?a) (key-under-mat))
      :effect
        (and
          (key-under-mat)
          (not (has-key ?a)))
    )
    :observability full ?a
    :observability none
  )

  (:action announce
    :parameters (?a - agent)
    :byagent ?a
    :precondition (key-under-mat)
    :effect
      (and)
  )

  (:action try-take
    :parameters (?a - agent)
    :byagent ?a
    (:event-designated e1
      :precondition (not (key-under-mat))
      :effect
        (and)
    )
    (:event-designated e2
      :precondition (key-under-mat)
      :effect
        (and
          (not (key-under-mat))
          (has-key ?a))
    )
    :observability none
    :observability full ?a
  )
)
```

A.2.2 Apartment Key Problem

```
(define (problem doormat-1)
  (:domain doormat)
  (:objects
     anne bob - agent)
  (:world-designated w1
    (has-key anne))
  (:world-designated w2
    (key-under-mat))
  (:observability full anne)
  (:observability none bob)
  (:goal
    (has-key bob)
  )
)
```

B Appendix: Example debug output

```
$ stack exec PDDL-DEL-exe -- examples/key.pddl --debug
Successful parsing
CheckPDDL (Domain {name = "doormat", reqs = [], types = ["agent"], constants = [],
    predDefs = [PredAtom "key-under-mat",PredDef "has-key" [VTL ["?a"] "agent"]],
    actions = [Action {aname = "put-key", params = [VTL ["?a"] "agent"], actor = "?a",
     events = [Event False "trivial" (And []) (And []),Event True "actual" (Or [Atom (
    PredSpec "has-key" ["?a"]),Atom (PredAtom "key-under-mat")]) (And [Atom (PredAtom "
    key-under-mat"),Not (Atom (PredSpec "has-key" ["?a"]))])], evobss = [ObsSpec Full
    ["?a"],ObsDef None]},Action {aname = "announce", params = [VTL ["?a"] "agent"],
    actor = "?a", events = [Event True "" (Atom (PredAtom "key-under-mat")) (And [])],
     evobss = []},Action {aname = "try-take", params = [VTL ["?a"] "agent"], actor = "?
    a", events = [Event True "e1" (Not (Atom (PredAtom "key-under-mat"))) (And []),
    Event True "e2" (Atom (PredAtom "key-under-mat")) (And [Not (Atom (PredAtom "key-
    under-mat")),Atom (PredSpec "has-key" ["?a"])])], evobss = [ObsDef None,ObsSpec
    Full ["?a"]]}]}) (Problem {pname = "doormat-1", dname = "doormat", objects = [TO ["
    anne","bob"] "agent"], init = [], worlds = [World True "w1" [PredSpec "has-key" ["
    anne"]],World True "w2" [PredAtom "key-under-mat"]], wobss = [ObsSpec Full ["anne
    "],ObsSpec None ["bob"]], goal = Atom (PredSpec "has-key" ["bob"])})
(KrMS5 [0,1] [("anne",[[0],[1]]),("bob",[[0,1]])] [(0,[(P 1,False),(P 2,True),(P 3,
    False)]),(1,[(P 1,True),(P 2,False),(P 3,False)])],[0,1])
anne: put-key ["anne"]  (KrMS5 [0,1] [("anne",[[0],[1]]),("bob",[[0,1]])] [(0,[(P 1,
    False),(P 2,True),(P 3,False)]),(1,[(P 1,True),(P 2,False),(P 3,False)])],[0,1])
anne: try-take ["anne"] (KrMS5 [0,1] [("anne",[[0],[1]]),("bob",[[0,1]])] [(0,[(P 1,
    False),(P 2,True),(P 3,False)]),(1,[(P 1,True),(P 2,False),(P 3,False)])],[0,1])
bob: try-take ["bob"]   (KrMS5 [0,1] [("anne",[[0],[1]]),("bob",[[0,1]])] [(0,[(P 1,
    False),(P 2,True),(P 3,False)]),(1,[(P 1,True),(P 2,False),(P 3,False)])],[0,1])
        anne: put-key ["anne"]  (KrMS5 [0,2,1,3] [("anne",[[0],[2],[1],[3]]),("bob
    ",[[0,2,1,3]])] [(0,[(P 1,False),(P 2,True),(P 3,False)]),(2,[(P 1,True),(P 2,
    False),(P 3,False)]),(1,[(P 1,True),(P 2,False),(P 3,False)]),(3,[(P 1,True),(P 2,
    False),(P 3,False)])],[2,3])
        anne: announce ["anne"] (KrMS5 [0,2,1,3] [("anne",[[0],[2],[1],[3]]),("bob
    ",[[0,2,1,3]])] [(0,[(P 1,False),(P 2,True),(P 3,False)]),(2,[(P 1,True),(P 2,
    False),(P 3,False)]),(1,[(P 1,True),(P 2,False),(P 3,False)]),(3,[(P 1,True),(P 2,
    False),(P 3,False)])],[2,3])
        anne: try-take ["anne"] (KrMS5 [0,2,1,3] [("anne",[[0],[2],[1],[3]]),("bob
    ",[[0,2,1,3]])] [(0,[(P 1,False),(P 2,True),(P 3,False)]),(2,[(P 1,True),(P 2,
    False),(P 3,False)]),(1,[(P 1,True),(P 2,False),(P 3,False)]),(3,[(P 1,True),(P 2,
    False),(P 3,False)])],[2,3])
        bob: try-take ["bob"]   (KrMS5 [0,2,1,3] [("anne",[[0],[2],[1],[3]]),("bob
    ",[[0,2,1,3]])] [(0,[(P 1,False),(P 2,True),(P 3,False)]),(2,[(P 1,True),(P 2,
    False),(P 3,False)]),(1,[(P 1,True),(P 2,False),(P 3,False)]),(3,[(P 1,True),(P 2,
    False),(P 3,False)])],[2,3])
anne: put-key ["anne"];
bob: try-take ["bob"].;
```

C Appendix: Translated models

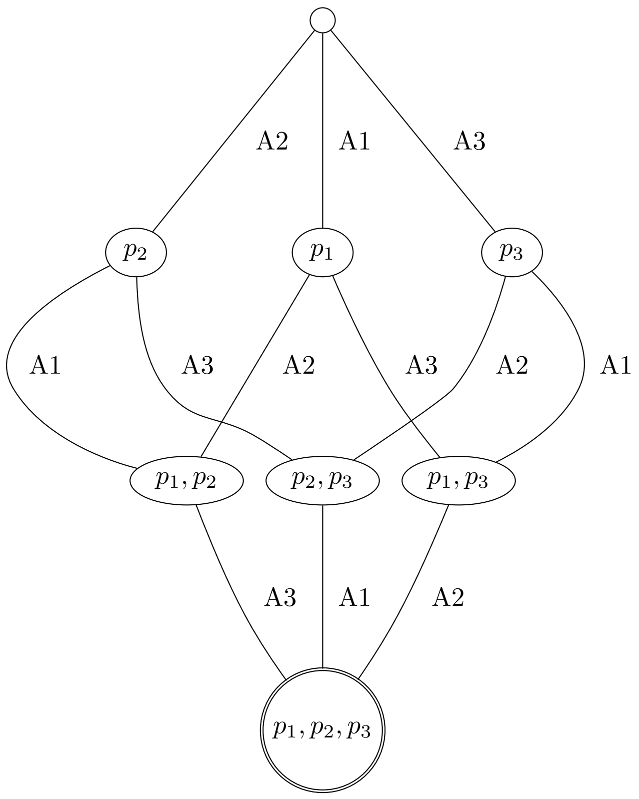


Figure C.1: Translated Three Muddy Children initial model. Here p_k represents the PDDL predicate `muddy Ak`.