

LTL_f/LDL_f synthesis algorithms to handle goal change
in autonomous agents – design, implementation and
evaluation

Student: Renzo Schram

Student number: 3454185

Program: master Artificial Intelligence, University Utrecht

External supervisor (Sapienza University): Prof. Fabio Patrizi

External co-supervisor (York University): Prof. Yves Lespérance

1st examiner/internal supervisor (Utrecht University): Dr. Gerard Vreeswijk

2nd examiner (Utrecht University): Dr. Brian Logan

July 1, 2024



**Utrecht
University**



SAPIENZA
UNIVERSITÀ DI ROMA

Abstract

In this research, we propose a formal definition for an intention management system (IMS) which an agent can use within a FOND domain for managing its intentions. These intentions can be expressed as any LTL_f expression, in which temporally extended goals can also be provided to the IMS. The IMS provides an agent with several query- and update operations, by which an agent is given as much freedom as possible for reaching and adjusting its intentions throughout a run, all while maintaining consistency among this changing set of intentions. The definition of this IMS is used for developing a proof of concept in the form of software, which has been evaluated on several problem scenarios within FOND domains, expressed in LTL_f . The IMS has proven to be effective for an agent in order to manage its dynamically changing intentions, complying to all requirements as described in the definition of the IMS. The definition of an IMS is meant as a proof of concept, proving its effectiveness in practice. This definition can be taken as a foundation for further research, where it could also be applied to LDL_f expressions, handling beliefs and desires, and several other possible extensions on our research.

Contents

Glossary	5
1 Introduction	6
2 Background	9
2.1 LTL	9
2.2 LTL_f	9
2.3 LDL_f	10
2.4 Automata	10
2.4.1 Deterministic Finite-state Automata (DFA)	11
2.4.2 Non-deterministic Finite-state Automata (NFA)	11
2.4.3 Alternating Finite-state Automata (AFA)	12
2.5 PDDL	12
2.6 FOND domains	13
2.7 Agent strategies	13
2.8 LTL_f synthesis algorithm	14
2.9 Maximally Permissive Strategy	16
2.10 (Reduced Ordered) Binary Decision Diagram	17
2.11 LTL/LTL_f Progression	18
3 Intention management system	20
3.1 Example scenario: Triangle-Tireworld	20
3.2 Operational environment for the IMS	22
3.3 Requirements for the IMS	22
3.4 Formal description of an IMS state	24
3.5 The query operations of the IMS	24
3.6 The update operations of the IMS	25
3.7 Progressing an LTL_f intention	26
3.7.1 Example for progressing an LTL_f formula	26
3.8 Usage of the IMS	26
3.8.1 Initialization of a run	26
3.8.2 Operations during a run	26
3.8.3 Completion of a run	28
4 Design of the IMS	29
4.1 Description for the design of the IMS	29
4.2 IMS state for the design	29
4.3 The query- and update operations for the design	30
4.4 Adding and dropping of an intention to list of intentions	32
4.4.1 Example for adding a new intention to the list of intentions	32
4.4.2 Revision of the intention list by the $I^+.isRealizable(\phi, k)$ query operation	34
4.5 Individual components for constructing an LTL_f formula	34
4.5.1 Description of the decision tree domain	35
4.5.2 Example of individual components for composing an LTL_f formula in the decision tree domain	36
4.6 LTL_f formula composition for extracting good agent moves	37
4.7 LTL_f formula composition for extracting good environment moves	39

4.8	Application of BDD's	40
5	Implementation of the IMS	42
5.1	Software used for the implementation	42
5.1.1	Lydia	42
5.1.2	SyftMax	42
5.2	Usage of the IMS by an agent	43
6	Evaluation of the IMS implementation	47
6.1	Translation of FOND domains from PDDL into LTL_f	47
6.1.1	Comparison of specifying a FOND domain in PDDL and LTL_f	47
6.2	Benchmark domains and problems	48
6.3	Hard- and software specification for developing and running the IMS	49
6.4	Design of benchmark problems	49
6.5	Development of benchmark domains and problems	50
6.6	Simulation of a run	50
6.7	Decision tree domain	50
6.7.1	Representation of the decision tree domain in LTL_f	50
6.7.2	Simulation of the decision tree domain	50
6.7.3	Evaluation of IMS in the decision tree domain	56
6.8	Slippery world domain	56
6.8.1	Description of the slippery world domain	56
6.8.2	Representation of the slippery world domain in LTL_f	57
6.8.3	Simulation of the slippery world domain (scenario 1)	58
6.8.4	Simulation of the slippery world domain (scenario 2)	68
6.8.5	Evaluation of IMS in the slippery world domain	73
6.9	Triangle-Tireworld domain	74
6.9.1	Description of the Triangle-Tireworld domain	75
6.9.2	Representation of the Triangle-Tireworld domain in PDDL	75
6.9.3	Representation of the Triangle-Tireworld domain in LTL_f	76
6.9.4	Simulation of the Triangle-Tireworld domain	77
6.9.5	Evaluation of IMS in the Triangle-Tireworld domain	78
6.10	General evaluation of the IMS implementation	78
7	Related work	79
8	Conclusion	82
9	Future work	83
A	Triangle-Tireworld domain simulation (handwritten log output)	88

Glossary

Acronyms

- **AFA:** alternating finite-state automata
- **AI:** artificial intelligence
- **BDD:** binary decision diagram
- **DAG:** directed acyclic graph
- **DFA:** deterministic finite-state automata
- **FOND:** Fully Observable and Non-Deterministic
- **IMS:** intention management system
- **LDL:** linear dynamic logic for infinite traces
- **LDL_f:** linear dynamic logic for finite traces
- **LTL:** linear temporal logic for infinite traces
- **LTL_f:** linear temporal logic for finite traces
- **NFA:** non-deterministic finite-state automata
- **PDDL:** planning domain definition language
- **POC:** proof of concept
- **PPLTL:** pure-past LTL
- **RE:** regular expressions
- **ROBDD:** reduced ordered binary decision diagram

Terms

- **actions:** what an agent does.
- **belief:** represents the informational state of an agent (belief set).
- **desire:** represents the motivational state of an agent (goals).
- **domain:** model of how the environment works.
- **fluents:** effects from the environment.
- **intention:** represents the deliberative state of an agent (plans).
- **maximally permissive strategy:** the entire set of strategies fulfilling a task.
- **motivational attitude:** what motivates an agent to act as it does.
- **non-deterministic environment:** where not everything can be determined from the start, arbitrariness has influence later on in traces (influence by fluents).
- **synthesis:** a combination of components to form a connected whole.
- **task:** goal of an agent, represented by its intentions.
- **temporal logic:** any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time (e.g. "I am always hungry", "I will eventually be hungry", or "I will be hungry until I eat something").
- **temporally extended goal:** a type of goal that spans a duration of time and involves a sequence of actions or subgoals to achieve. A temporally extended goal cannot be accomplished in a single, instantaneous action but requires a series of steps to be completed over time.
- **traces:** paths to take/sequence of actions.

1 Introduction

The field of artificial intelligence (AI) is experiencing an exponential growth in the last couple of years in terms of research and development, for which the field of planning within AI is no exception. Planning with the aid of AI is already being used by most people in their everyday life, whether this is consciously or unconsciously. One of the most well known examples for this is route planning software, for which a user can plan to go from one location to another. Several types of routes can be planned, allowing a user to change their route dynamically, even when this user is already on its way to the target location. As people tend to be unpredictable, so are their plans, motivational attitudes and intentions.

In the field of AI, people can be considered as autonomous agents, who often tend to take decisions which cannot be easily anticipated on. Desires and intentions can change sporadically, for which it would be not enough to plan only a singular strategy for fulfilling the intentions which this autonomous agents might have. Autonomous agents' motivational attitudes are dynamic and they progressively commit to desires and refine/revise their intentions over time. Early commitment to goals and plans may not be feasible or helpful if the world is very dynamic, the agent's desires may change, or if insufficient information makes planning difficult. But the agent should ensure that her intentions remain consistent.

In this research, it will be studied how to use LTL_f/LDL_f synthesis algorithms [12] to handle goal change in an agent, focusing on how to maintain consistency among a changing set of such temporally extended goals. We will do so by designing an intention management system, which is able to keep track of an agent's intentions, their realizability at any time and state, and compute all possible strategies to achieve these intentions (the maximally permissive strategy). For this, previous work on synthesis of maximally permissive strategies [40], strategy repair [19], as well as compositional techniques for strategy synthesis [8] will be examined. For further motivation on this approach, see [26].

An intention management system for managing goals and maintaining their consistency will be designed, a proof of concept in the form of software will be built which implements this design for the intention management system, a number of benchmark goal change problems will be selected or designed, and the implementation of these will be evaluated on the proof of concept.

In this research the focus will not be on probabilistic planning (e.g. Monte Carlo tree search), since this is not in the scope of this project, although this might be interesting to research in future work. Also, partially observable environments will not be covered, as this research is about fully observable non-deterministic (FOND) environments only, in which the domain is limited to a specified set of possible states and actions. Lastly, this research talks about LTL_f which only considers the future, as opposed to different forms of LTL_f focusing only on the past or both past and future. These forms are called pure-past LTL (PPLTL) and past LTL, respectively, and are not the focus of this research.

In FOND planning for temporally extended goals, it is useful to map LTL_f/LDL_f formulas into automata, since automata allow for reaching several subgoals during execution of a program, as opposed to reaching only a single goal, as would be in regular FOND planning. Automata also make it easier to determine the resulting states of a chosen action. The most popular algorithms and research on FOND planning for temporally extended goals are automata-based ([16], [12], [40], [17]), on top of research already performed for FOND planning using automata without temporally extended goals ([34], [10], [18], [8], [13], [11]), which we will exploit in this research. Automata operate by themselves without human intervention and according to a set of rules, which makes them predictable and work in the same manner as a computer program does. This makes it easier to apply the theoretical design of the automata into usable software, as we want to achieve in this research by

building a proof of concept. FOND planning can also be done without the use of automata, although this will not be covered in this research.

In [8], it has been studied how to obtain the deterministic finite-state automaton (DFA), given an LTL_f/LDL_f formula. The DFA specifies the result of each action in each state, which an agent can use to make a strategy to fulfill its intentions with a complexity of 2EXPTIME-complete, where the complexity could potentially introduce a bottleneck for processing the LTL_f/LDL_f formulas. In practice however, 2EXPTIME complexity would not be reached in most cases, as this is considered the worst-case complexity. Research accepts this complexity bottleneck, as in most cases the high expressiveness of LTL_f and LDL_f weigh up against the computational challenges, and also in exclusively theoretical research the exponential complexity is not a problem, as there would then be no practical applications in which computations have to be made.

How a strategy can be extracted from a DFA which fulfill a set of intentions is described in [12]. It describes a method for extracting a winning strategy given a DFA. For this research we not only want to extract a single winning strategy, but the entire set of winning strategies, which is called the maximally permissive strategy, described in more detail in Section 2.9. When the maximally permissive strategy is obtained, we know for each state, at any point in time, which actions will guarantee us to still reach our goal. This mapping of states containing at least one action which guarantees us to eventually reach our goal is called the 'winning region'. How to obtain the maximally permissive strategy given a DFA, is described in [40].

The intentions of an agent can be represented using LTL_f/LDL_f formulas. As the agent progresses, it can adopt new intentions and may drop intentions it already has in its list of intentions. If this would happen, the agent has to possibly change its strategy, and so the winning region could also change. To alter its strategies and recompute the winning region, the agent needs to know at any point in time which of the intentions it has are realizable, what actions in each state will guarantee the agent to stay in the winning region (maximally permissive strategy), and how the agent will eventually reach a final state. In this research it will be studied how LTL_f/LDL_f synthesis algorithms can be used to handle goal change in an agent in a non-deterministic environment using the maximally permissive strategy.

In [25], an approach is described for computing more than a singular plan for an agent, as is also done in [40] when a maximally permissive strategy is computed. Although [25] uses a somewhat related approach to that of [40] in terms of providing the agent with flexibility in choosing actions during a run, in [25] the possible environment reactions are not specified and the plans are not guaranteed to be winning. The research of [25] is focused on avoiding goal conflicts, where goal plan trees (GPT) are made in which several plans are expressed for achieving an agent's goal. This GPT represents the goals, plans, and actions for an agent, which is somewhat related to the winning region of [40], although in [25] the environment reactions are not included, and also in GPT's an agent plan is not guaranteed to be winning. In [25], intentions are revised based on the desires and beliefs of an agent, although in our research we only use the intentions of an agent. We only deal with FOND environments, in which everything within a domain is fully observable. Also important here is to note that in [25], the intentions of an agent are dynamically changed based on the desires of an agent which are consistent with its beliefs. In our approach, we do not focus on BDI agent architectures, but rather propose a tool for an agent, an intention management system, which an agent can use for keeping track of its intentions, their consistency, their progress, add and drop intentions, and retrieve all possible strategies for fulfilling these intentions. Although in this research we will not implement the approach of [25], it might be interesting to keep into account for future work. We will cover more on this later in Section 7.

As far as we know, no prior work proposes a general model of intention management which handles arbitrary LTL_f intentions in FOND domains, maintains their consistency/realizability, and

computes maximally permissive strategies through LTL_f synthesis. This is what we will investigate in this research. We will design an intention management system to keep track of an agent's changing intentions, their realizability, and how to achieve these intentions. It is important to note that in this research we assume the following: the action outcomes are always among those allowed by the FOND model, the agent only performs actions that are guaranteed to keep her in the winning region, and the agent never adopts unrealizable intentions. This theoretical design will be used to build a proof of concept in the form of software, which can then be evaluated using benchmark problems to test its performance and limitations.

After this research, we want to be able to answer the following research question:

What properties does an intention management system need in order to handle goal change in a FOND environment, such that it can maintain consistency among a changing set of temporally extended goals?

In order to answer this main research question, there is one main theoretical oriented objective (objective 1) and two smaller practical oriented objectives (objectives 2 and 3) to be achieved during this research:

1. Formally define the notion of an intention management system (theory/concept), which keeps track of a list of intentions over time, provides query and update operations to check the realizability of new intentions and add them if they are realizable, and compute all guaranteed strategies to achieve these intentions, if there are any. This system indicates what actions can be done to remain in the winning region, which realizable intentions are left, if adding a new intention is realizable, and if choosing a certain action will make progress towards a final state, where all the intentions have been achieved. As mentioned in the introduction, we will base this theoretical design on previous work on synthesis of maximally permissive strategies [40], strategy repair [19], as well as compositional techniques for strategy synthesis [8] for checking the realizability of an LTL_f/LDL_f formula and deriving the DFA. We extend this previous research by designing an intention management system which is able to maintain consistency among a changing set of temporally extended goals, keep track of a list of intentions, and check the realizability of new intentions, which has not yet been done in previous research. After this we use the theory of previous research for synthesizing the maximally permissive strategy.
2. Develop a proof of concept (software), based on the formally defined notion of an intention management system, which can take benchmark FOND domains and problems as input (specified in PDDL-format), computes the maximally permissive strategy, and updates the winning region based on what intentions are added and dropped to the list of intentions. At any time step, this proof of concept is able to tell what actions keep the agent in the winning region, and what steps will progress such that it can eventually reach a final state.
3. Evaluate the proof of concept on benchmark FOND domains and problems. These benchmark problems are provided in PDDL-format, which will be slightly altered such that the agent does not only start with an initial set of intentions, but will be able to add new intentions over time. These benchmark problems are scaled to test the performance and limits of the proof of concept.

2 Background

In this Section we will go over the preliminaries for this research, in which will be explained in what ways these are relevant. An intuitive idea is given of how the concepts work, for which some of these also include a more mathematical explanation, including some examples.

2.1 LTL

Linear Temporal Logic (LTL) is one of the most popular formalisms for expressing the temporal properties of reactive systems [30]. It can be used for expressing the valuations of a series of propositions over time, which is especially useful in the field of planning. Given a set of atomic propositions P the formulas of LTL are generated by the following grammar:

$$\phi ::= a \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \bigcirc\phi \mid \phi_1 \mathcal{U} \phi_2$$

where $a \in P$. $\phi_1 \mathcal{U} \phi_2$ expresses that ϕ_1 has to hold, until ϕ_2 is reached. $\bigcirc\phi$ expresses that in the next state, ϕ holds. We use common abbreviations such as *eventually* as $\diamond\phi \doteq \text{True} \mathcal{U} \phi$, *always* as $\square\phi \doteq \neg\diamond\neg\phi$, and as *release* as $\phi_1 \mathcal{R} \phi_2 \doteq \neg(\neg\phi_1 \mathcal{U} \neg\phi_2)$, well as the usual propositional logic abbreviations. $\diamond\phi$ expresses that ϕ has to hold *eventually*. $\square\phi$ expresses that ϕ *always* has to hold.

Formulas of LTL are interpreted over infinite sequences (called traces) of truth evaluations of variables in P , i.e. $\pi = \pi_0, \pi_1, \dots \in (2^P)^\infty$, where 2^P includes every possible combination of truth values for the propositions in P . This also includes the empty set, \emptyset , since 2^P includes every possible combination of truth values, which includes \emptyset , in which no propositions are true. Given a trace π , we define when an LTL formula ϕ holds at position i on π , written $\pi, i \models \phi$, inductively on the structure of ϕ , as follows:

- $\pi, i \models a$ iff $a \in \pi_i$ (for $a \in P$);
- $\pi, i \models \neg\phi$ iff $\pi, i \not\models \phi$;
- $\pi, i \models \phi_1 \vee \phi_2$ iff $\pi, i \models \phi_1$ or $\pi, i \models \phi_2$;
- $\pi, i \models \bigcirc\phi$ iff $\pi, i + 1 \models \phi$;
- $\pi, i \models \phi_1 \mathcal{U} \phi_2$ iff there exists $j \geq i$ such that $\pi, j \models \phi_2$, and for all $k, i \leq k < j$ we have that $\pi, k \models \phi_1$.

We say that π satisfies ϕ , written $\pi \models \phi$, if $\pi, 0 \models \phi$.

In this research, we will be using LTL_f , which is LTL interpreted over finite traces. More on LTL_f is described in Section 2.2.

2.2 LTL_f

In many applications, especially in AI, the goal/task must be achieved/completed within a finite horizon. Therefore, many recent works have looked at Linear Temporal Logic interpreted over *finite traces* (LTL_f) [12].

The syntax of LTL_f is the same as that of LTL, except for the following additional abbreviations: the *weak-next* (\bullet) defined as $\bullet\phi \doteq \neg\bigcirc\neg\phi$, and the *end of the trace (final)* defined as $\text{final} \doteq \bullet\text{false}$. In other words: $\bullet\phi$ expresses that ϕ holds, if there exists a next state.

Formulas of LTL_f are interpreted over finite sequences (called traces) of truth evaluations of variables in P , i.e. $\pi = \pi_0, \dots, \pi_n \in (2^P)^*$. We denote the length $n + 1$ of a trace π by $\text{length}(\pi)$. Given a finite trace π , we define when an LTL_f formula ϕ holds at position i on π where $0 \leq i < \text{length}(\pi)$, written $\pi, i \models \phi$, inductively on the structure of ϕ , as follows:

- $\pi, i \models a$ iff $a \in \pi_i$ (for $a \in P$);

- $\pi, i \models \neg\phi$ iff $\pi, i \not\models \phi$;
- $\pi, i \models \phi_1 \vee \phi_2$ iff $\pi, i \models \phi_1$ or $\pi, i \models \phi_2$;
- $\pi, i \models \bigcirc\phi$ iff $i < \text{length}(\pi) - 1$ and $\pi, i + 1 \models \phi$;
- $\pi, i \models \phi_1 \mathcal{U} \phi_2$ iff there exists $i \leq j < \text{length}(\pi)$ such that $\pi, j \models \phi_2$, and for all $k, i \leq k < j$ we have that $\pi, k \models \phi_1$.

As LTL_f is able to express a finite temporal specification, it is useful for applications in which it has to be specified at which point a trace should end. In this research, we can apply this in a scenario when all intentions of an agent have been fulfilled, and the program should terminate. A task for an agent can be declared in LTL_f , including the entire declaration of rules it has to follow and what actions are possible in order to fulfill this task. More on this is described in Section 2.8, where an LTL_f declaration is used for expressing a domain and goals for an agent, from which a winning strategy can be extracted.

2.3 LDL_f

Linear Dynamic Logic on finite traces (LDL_f) is an adaptation of LDL , introduced in [28], where regular LDL is interpreted over infinite traces [12]. LDL_f is obtained by merging LTL_f with regular expressions, but adopting a semantics based on finite traces [11]. LDL_f can be seen as an extension of LTL_f , in which we can also talk about actions, making LDL_f more expressive than LTL_f . The semantics for LDL_f can be found in [11], and the progression of LDL_f can be found in [12].

Although this research does talk about LDL_f to a certain extent, the focus is on LTL_f expressions. As the scope of this research would become too broad when LDL_f would be fully covered, it has been decided to have this research focused on LTL_f only. Section 3 describes an intention management system which is able to handle LTL_f , for which our automata-based approach could easily be adapted to handle LDL_f , which is described in the future work of Section 9.

2.4 Automata

Automata are abstract machines which are self-operating, designed for following or responding to a sequence of operations and instructions. An automaton can consist of several states and transitions, for which an input symbol can be provided within a state, and the transition function indicates what the resulting state will be after providing the input symbol [23]. An example for this is shown in Figure 2, where it is indicated for each state what the transition will be, given an input symbol.

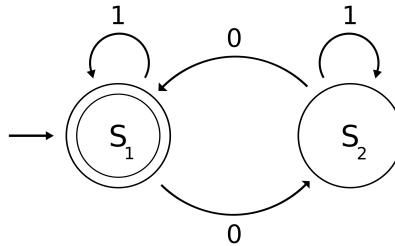


Figure 2: Example of an automaton

Many algorithms for LTL_f synthesis, as the one described in Section 2.8, are based on using automata, as well as some theoretical results. In this research, we also make use of several types

of automata, where the DFA is most important: a deterministic finite-state automata. From a DFA we can deterministically extract for each possible state which action will transition us into which next state, using a transition function. These automata and their applications in this research will be described in further detail in Section 2.4.1, 2.4.2, and 2.4.3. Using LTL_f synthesis, we obtain a DFA from which we can extract a winning strategy for an agent, as explained in Section 2.8.

2.4.1 Deterministic Finite-state Automata (DFA)

The DFA is a general tool to represent temporal specifications. For a DFA it applies that for an input symbol, there is only one corresponding resultant state (i.e. there is only one state transition possible given an input). In Figure 3 an example is shown of a DFA:

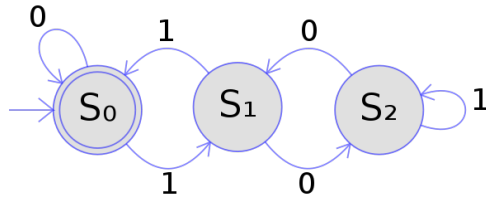


Figure 3: Example of DFA

As can be seen in this Figure, every input symbol (0 or 1) corresponds to exactly one resultant state, which makes it that this finite-state automata is deterministic. More formally,

Definition 2.1. A deterministic finite automaton (DFA) is a tuple

$$\mathcal{A} = (\Sigma, Q, q_0, \delta, F),$$

where: Σ is a finite input alphabet; Q is a finite set of states; $q_0 \in Q$ is the initial state; $\delta : Q \times \Sigma \rightarrow Q$ is the transition function; and $F \subseteq Q$ is the set of final states. The size of \mathcal{A} is $|Q|$. Given a finite trace $\alpha = \alpha_0 \alpha_1 \dots \alpha_n$ over Σ , we extend δ to be a function $\delta : Q \times \Sigma^* \rightarrow Q$ as follows: $\delta(q, \lambda) = q$, and, if $q_n = \delta(q, \alpha_0 \dots \alpha_{n-1})$, then $\delta(q, \alpha_0 \dots \alpha_n) = \delta(q_n, \alpha_n)$. A trace α is *accepted* if $\delta(q_0, \alpha) \in F$. The *language* of \mathcal{A} , written $\mathcal{L}(\mathcal{A})$, is the set of traces that the automaton accepts. Given the DFAs $\mathcal{A}_1, \dots, \mathcal{A}_n$, we can build a DFA $\mathcal{A} = \text{PRODUCT}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ such that $\mathcal{L}(\mathcal{A}) = \bigcap_{i \leq n} \mathcal{L}(\mathcal{A}_i)$ in polynomial time in the sizes of $\mathcal{A}_1, \dots, \mathcal{A}_n$.

In this research, we will be using DFAs for representing a FOND domain, described in Section 2.6, and all the intentions which an agent might have. The DFA is also used for extracting the maximally permissive strategy, which is explained in more detail in Section 2.9. More information on DFAs can be found in [34].

2.4.2 Non-deterministic Finite-state Automata (NFA)

As opposed to a DFA, in an NFA there is not always only one resultant state for every input symbol, which makes it non-deterministic. As can be seen in Figure 4, when we start in state q_0 and give 0 as an input symbol, we will transition either back into state q_0 or into state q_1 , which cannot be determined beforehand, making the finite-state automata non-deterministic.

For this research, NFAs are only used within the LTL_f synthesis algorithm described in Section 2.8, where a DFA can be determinized from an NFA. More information on NFAs can be found in [34].

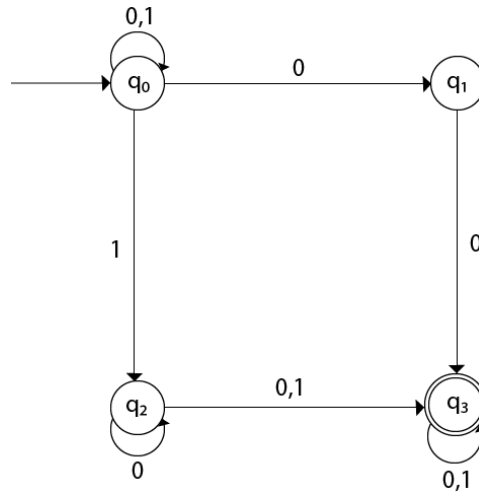


Figure 4: Example of NFA

2.4.3 Alternating Finite-state Automata (AFA)

While non-deterministic automata have the power of existential choice over transitions, alternating automata are computational models with the power of both existential and universal choice over transitions [4]. While a run for DFAs and NFAs is defined as a sequence of states, a run for AFAs is a tree (acyclic) [18]. In this research, we only use AFA's within the process of LTL_f synthesis, as described in Section 2.8. As no further knowledge is needed on this for this research, we refer to more information on AFAs which can be found in [4], [18], and [34].

2.5 PDDL

PDDL stands for 'Planning Domain Definition Language', which is considered the 'standard' language for representing classical planning tasks. This task consist of the following components: objects, predicates, initial state, goal state, action/operator. [22]

In PDDL, planning tasks are composed of two components:

1. A domain file, in which predicates and actions are defined
2. A problem file, in which objects, the initial state and the goal specification are defined

These files can be used as input for an agent to solve the intention problems in the PDDL-problem file. Important here is to note that PDDL is a relational language, but it is assumed that the object domain is finite. As this is the case, the predicates and operators can be grounded to obtain a propositional language. PDDL specifies the effects of an action, but does not give a specification of the frame axioms, i.e. what remains the same after an action is performed. A more detailed description of PDDL and its definitions can be found in [22].

In this research, we will use PDDL for expressing FOND domains, explained in Section 2.6, as this is one of the most commonly used manners for expressing FOND domains within the field of planning. An example of the well known FOND domain 'Triangle-Tireworld', expressed in PDDL, can be found in Section 6.9.2.

2.6 FOND domains

As already mentioned in the introduction, in our research we only make use of FOND domains. A FOND domain is a Fully Observable and Non-Deterministic domain, for which everything within a domain can be observed, yet not every effect of an action is deterministic. This limits our research to a certain degree, as we do not including anything related to probabilities or partially observable domains. We only deal with domains in which we can determine any possible outcome for an effect within a domain, although we cannot determine what this effect will be beforehand.

Following [14], we define a planning domain as follows:

Definition 2.2. A planning domain is a tuple

$$D = (2^{\mathcal{F}}, Act, React, \alpha, \beta, \delta),$$

where: \mathcal{F} is a finite set of fluents $2^{\mathcal{F}}$ is the state space; Act and $React$ are finite sets of agent actions and environment reactions, respectively; $\alpha : 2^{\mathcal{F}} \times Act \rightarrow 2^{\mathcal{F}}$ denotes agent action preconditions; β denotes environment reaction preconditions; and $\delta : 2^{\mathcal{F}} \times Act \times React \rightarrow 2^{\mathcal{F}}$ is the transition function such that $\delta(s, a, r)$ is defined if and only if $a \in \alpha(s)$ and $r \in \beta(s, a)$. We assume that planning domains satisfy the properties of *existence of agent action* (i.e., $\forall s \in 2^{\mathcal{F}}. \exists a \in \alpha(s)$), *existence of environment reaction* (i.e., $\forall s \in 2^{\mathcal{F}}, a \in \alpha(s). \exists r \in \beta(s, a)$), and *uniqueness of environment reaction* (i.e., $\forall s \in 2^{\mathcal{F}}, a \in \alpha(s). \delta(s, a, r_1) = \delta(s, a, r_2) \supset r_1 = r_2$).

With these properties, inspired by [9], we capture planning domains adopted in FOND [7, 20], say expressed in PDDL [22], though keeping explicit the reaction (corresponding to the `oneof` clauses in PDDL). Considering that the domain can be compactly represented, say in PDDL, we identify the size of the domain D with the size of its state space, which is exponential in the number of fluents $|\mathcal{F}|$.

Note that we have not included the initial state in the planning domain. Given an initial state $s_0 \in 2^{\mathcal{F}}$, a planning domain evolves as the agent and the environment move in turns. At each turn, the agent makes an action chosen among those that satisfy their preconditions, and the environment responds with some reaction, again chosen among those that satisfy their preconditions. State transitions are determined when both an agent- and environment move are performed.

2.7 Agent strategies

Within a domain, an agent can have strategies: plans which the agent wants to follow for fulfilling its goals. Any strategy which an agent might have is defined over the possible traces/paths within a domain. Following [10], we can define a domain trace as follows:

Definition 2.3. A *domain trace* starting from s_0 over D is a (finite or infinite) sequence $\tau = (a_0, s_0)(a_1, s_1) \dots$ where in the initial step (a_0, s_0) , a_0 is a dummy action, and (ii) for every $i > 0$ the step (a_i, s_i) is such that $a_i \in \alpha(s_{i-1})$ and there exists a reaction $r_i \in \beta(s_{i-1}, a_i)$ such that $\delta(s_{i-1}, a_i, r_i) = s_i$ (notice that for the uniqueness of the reactions such r_i is unique).

As the agent in each step starts by executing an action, and the environment reacts to this, the first move of the agent is considered a 'dummy' action, as mentioned in [12], where the agent action has no effect. This dummy action is necessary because steps are always offered pairwise for an agent action, and an environment reaction. Currently this tuple consists of a current action and a current environment reaction. There is another possible approach for this, where a current environment state and a next agent action are combined in this tuple. In this scenario, the agent does not have to perform a dummy move before the environment can initialize itself. However, as the agent action and

environment reaction are always offered pairwise, this would result in the agent having to perform a 'dummy' action in a final state for the environment, which has no effect on the environment as the environment has no next state. We do not choose the latter approach for our research, as this could cause issues in domains where a final state for the environment causes that the preconditions for agent actions are not fulfilled, and thus it cannot perform a final dummy action, breaking potential domain rules. For example, if we have a domain with states s_1 , s_2 and s_3 , where state s_3 is a final state, and as preconditions the agent can only execute an action if it is in state s_1 or s_2 , this will result in the agent not being able to execute any actions in state s_3 . If the domain specifies that the agent has to execute exactly one action at a time, the agent will break this domain rule as it is impossible to execute an action at this point, causing it to lose as no final dummy move can be executed. Would we let the agent start off with a dummy move, as we do in our research, this problem can be avoided.

For our approach as already mentioned, since the environment has not initialized itself yet in the first step of a trace, and an agent action has no effect whenever it is not located in an environment state yet, the first action of an agent will have no effect. Infinite traces are also called *plays* and finite traces are also called *histories*. Within these traces, an agent can apply a strategy.

We define an agent strategy as follows:

Definition 2.4. An agent strategy in s_0 is a function $\sigma : (2^{\mathcal{F}})^+ \rightarrow Act$ mapping state sequences (in traces) to agent actions such that, for every $\tau = (a_0, s_0) \dots (a_n, s_n)$, $\sigma(s_0 \dots s_n) \in \alpha(s_n)$.

Given an initial state s_0 and an agent strategy σ in s_0 , we can construct a play as follows: the initial step is (a_0, s_0) as usual, and for every $i > 0$ the step (a_i, s_i) is obtained by choosing environment reaction r_i such that $r_i \in \beta(s_{i-1}, a_i)$ and then having $s_i = \delta(s_{i-1}, a_i, r_i)$. We denote the set of play induced by a strategy σ at s_0 by $Play(\sigma, s_0)$.

A goal is an l_{tl}_f formula φ defined over the alphabet $\mathcal{F} \cup Act$ i.e. over fluents and actions. Every domain trace $\tau = (a_0, s_0)(a_1, s_1) \dots$ corresponds to a trace $(\{a_0\} \cup s_0)(\{a_1\} \cup s_1) \dots$. So we can evaluate l_{tl}_f formulas over finite domain traces.

When an agent strategy is guaranteed to let the agent fulfill its goal, we call this a winning strategy.

We can define a winning strategy as follows:

Definition 2.5. An agent strategy σ starting at s_0 is a *winning strategy* for φ if, for all plays $\pi \in Play(\sigma, s_0)$ there exists a finite prefix π^k that satisfies φ , i.e., $\pi^k \models \varphi$.

Given a domain \mathcal{D} and an initial state s_0 , l_{tl}_f synthesis is the problem of finding a winning strategy for φ , if one exists. l_{tl}_f synthesis in nondeterministic planning domains is 2EXPTIME-complete in the size of φ and EXPTIME-complete in the size \mathcal{D} (i.e. in $|\mathcal{F}|$), respectively, and can be solved via a reduction to solving games played over deterministic finite automata [10].

Notice that if σ is winning, then for every play π in $Play(\sigma, s_0)$ we have an index k corresponding to the length of the prefix that satisfies φ . We call $maxSteps(\sigma, s_0, \varphi)$ the maximum among all such indexes. Intuitively $maxSteps(\sigma, s_0, \varphi)$ tells us the maximum number of steps that are needed in order to fulfill the φ in spite of the adversarial reaction of the environment.

In Section 2.8, we will discuss how a winning agent strategy can be extracted using synthesis. One goal for this research is to extract the set of all possible winning strategies for an agent, both procrastinating and non-procrastinating, which is called the maximally permissive strategy. More on this is explained in Section 2.9.

2.8 LTL_f synthesis algorithm

LTL_f synthesis is the process of mapping an LTL_f formula into a winning strategy, i.e. a strategy which can guarantee an agent to reach its goal eventually. An LTL_f formula, as described in 2.2,

can be used to compute a winning strategy for a declarative specification by using an algorithm for LTL_f synthesis. It does so by taking an LTL_f formula as input for the algorithm, where this is then translated into a DFA, and this DFA can then be used for synthesizing a winning strategy. In the DFA it is specified what next state an agent transitions to, given an agent action and an environment reaction. The DFA represents the game arena in which strategies can be found for an agent to win. As from the DFA it can be extracted what the agent transitions into given an agent action and the possible environment reactions, the agent can extract with certainty how to reach its goal: a winning strategy. It does so by checking for a given DFA state and agent action, if for any possible environment reaction, there always is a strategy which guarantees the agent to reach a state in which all its intentions are fulfilled. It is possible that there are multiple winning strategies, which is explained in more detail in Section 2.9.

The algorithm for LTL_f synthesis for this research is defined in [12] and works as follows:

1. Given LTL_f formula φ
 - (a) Compute AFA for φ (linear)
 - (b) Compute corresponding NFA (exponential)
 - (c) Determinize NFA to DFA (exponential)
 - (d) Synthesize winning strategy for DFA game (linear)

As shown in the synthesis algorithm, an LTL_f formula is given as input. This is then transformed into an AFA. With this AFA, we can compute the NFA, which we can then use to determinize the DFA. The determinizing of an NFA to a DFA causes an exponential blow up, as transferring from non-deterministic into deterministic means that all possible outcomes are held into account. The exact manners of computing the AFA, NFA and DFA are described in [12]. The agent can now check the realizability of the DFA game, i.e. if there exists a winning strategy. If the DFA game is indeed realizable, the agent can synthesize a winning strategy which specifies what action to take in each state in order to reach a final state.

What is important to note is that step (d) of the algorithm is a recursive algorithm itself. It initializes the winning region to the final states (goal states). After this, it adds the states where some agent action ensures that it reaches the winning region in one step. It keeps doing so until no state can be added anymore to the winning region, such that the complete winning region in the DFA is mapped. When eventually the complete winning region is known, a winning strategy can be synthesized which explains, starting from an initial state, which actions to choose in order to reach a final state. Important here is to note that there only is a winning strategy whenever the initial state is also included in the winning region. When this is the case, the DFA game is realizable.

Going from LTL_f to a DFA is going from purely declarative to fully procedural. It relies on the possibility of obtaining a deterministic automaton, a DFA, which is a machine, and hence a process. This does not hold in the infinite trace settings. The basic idea of program synthesis is to have a mechanical translation of human-understandable task specifications into a program that is known to meet the specifications.

In this research, LTL_f synthesis is applied by providing an initial LTL_f formula which specifies a FOND domain and the intentions which an agent wants to achieve, for which a DFA is extracted. In this research, the LTL_f synthesis is performed by Lydia¹: a software for extracting a DFA and winning agent strategy [8]. From this DFA, the 'maximally permissive strategy' is extracted for the agent, which contains all possible strategies for eventually reaching a final state. More on this is explained in Section 2.9.

¹Source for code of Lydia: [35]

2.9 Maximally Permissive Strategy

A maximally permissive strategy (MPS) is the entire set of winning strategies fulfilling a task. The MPS consists of two type of strategies: the deferring strategies and the non-deferring strategies, also referred to as the procrastinating- and non-procrastinating strategies. The deferring strategies refer here to all moves which keep an agent in the winning region, but allow an agent to defer the winning moment. As opposed to the deferring strategies, the non-deferring strategies keep an agent both in the winning region, while also progressing towards a final state. This allows an agent to be able to choose among the strategies while in execution, without committing to any specific one beforehand. Synthesis of maximally permissive strategies for LTL_f specifications was first introduced in [40], aiming at giving as much freedom as possible to an agent for reaching a final state. It defines what the winning states are and what the actions are that can guarantee to keep an agent in the winning region. While the maximally permissive strategy leaves as much freedom to the agent as possible, it still always reaches the final state eventually. The maximally permissive strategy does not say when the agent has to switch to a non-deferring strategy, but he must eventually switch. When this happens is up to the agent.

The maximally permissive strategy is unique for each problem statement, as it is a set of strategies allowing for maximal permissiveness for the agent given a specific set of constraints and goals within the problem statement, containing only the strategies guaranteeing an agent to remain in the winning region. If for two different problem statements we would have the same maximally permissive strategy, this would imply that one of these two maximally permissive strategies is either not maximally permissive, or includes non-winning strategies. Once we find the maximally permissive strategy for a problem statement, the problem of reaching a final state can be solved. The maximally permissive strategy does not allow an agent to be lazy and stay in the same state indefinitely (non-procrastinating). Eventually it has to become efficient and makes progress towards the final state. The maximally permissive strategy gives multiple possible actions in each state which eventually reach the final state. How these actions are chosen by an agent is not important for this research. This could be a randomly selected action which is non-deferring. This could also be decided by heads-tails flipping of a coin to simply stay in the winning region or to progress towards the final state.

In [40], an algorithm for finding the maximally permissive strategy was designed, which will be used in this research. They have developed software which implements this algorithm, called Syftmax², and is described in more detail in Section 5.1.2. Syftmax builds upon other software, called 'Lydia'. Lydia is used to determinize the DFA of an LTL_f formula, where the DFA is then used for defining the maximally permissive strategy in Syftmax. More details on Lydia are described in Section 5.1.1.

One might think that computing the maximally permissive strategy would take significantly longer compared to computing only a single strategy of reaching a final state. However, this is not necessarily the case. In [40], it is proven that computing the maximally permissive strategy only brings minor overhead compared to computing a single strategy.

In this research, we will use the maximally permissive strategy to provide an agent with as much freedom as possible in choosing an action, while guaranteeing that the agent will eventually still reach a final state, in which all its intentions are fulfilled.

²Source for code of Syftmax: [39]

2.10 (Reduced Ordered) Binary Decision Diagram

A BDD (binary Decision Diagram) is a DAG-like (Directed Acyclic Graph) data structure which is used for representing Boolean functions, where each non-terminal node is labeled by a function variable. A BDD can be compressed into a more compact representation of sets and relations, which are called ROBDD's (Reduced Ordered BDD) [2, 3]. Operations can be applied directly on this compressed representation, making it more efficient compared to approaches which require decompression of the entire representation. Originally, BDD's were developed for symbolic model checking, for which these were later also applied to LTL/LTL_f.

As an example, we look at Figure 5. This BDD represents the logical function $(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_3)$. Each node has two outgoing edges, representing the valuation of 0, indicated with a dashed line, and 1, indicated with a solid line. As already shown in Figure 5, for a BDD a truth table can be extracted. By providing the valuations of variable x_1 , x_2 , and x_3 , the resulting value tells us if the provided function holds under these values.

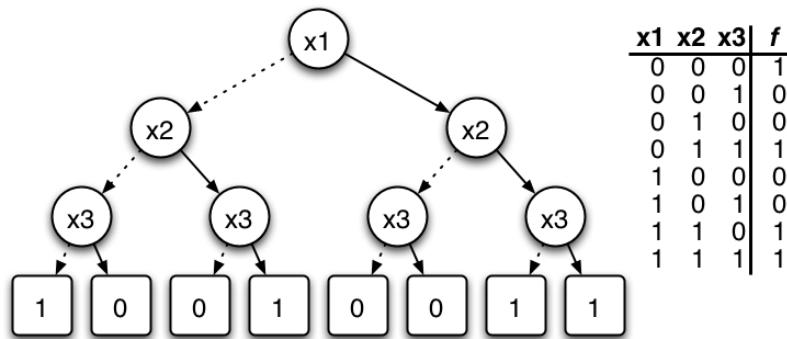


Figure 5: Example of a BDD

Using a BDD, we can extract the valuations of each type of function which can be expressed in propositional logic, including LTL_f. As LTL_f and BDD's are both based on Boolean functions, we can transform LTL_f formulas into BDD's, and extract from this whether the formula holds by providing the valuations for the Boolean propositions. However, the representation of a BDD as shown in Figure 5 gives us no advantage over using a regular truth table. In order to give us an advantage in terms of computation, this tree would need to be reduced in size, as now computations are executed which are redundant and we have repeating leaf nodes, 0 and 1. In Figure 6, a reduction of the BDD of Figure 5 is shown, called a ROBDD (reduced ordered binary decision diagram). It expresses the same function as in Figure 5, although this is represented more efficiently.

For example, as can be seen in Figure 6, when x_1 is *True*, x_3 is not relevant anymore for checking if the entire function holds. After x_1 is set to *True*, only x_2 decides if the function will hold. Here is the power of an ROBDD, as a graph can be represented more compactly, and unnecessary computations for checking if a function holds under a set of truth valuations for variables is executed more efficiently.

For the ROBDD in Figure 6, we can also write a truth table. When writing out the truth table for a ROBDD, we do not care about the variable valuations in which the entire function is not satisfied. We only care about the valuations of a graph which end up in leaf node '1'.

This would then result in the following table:

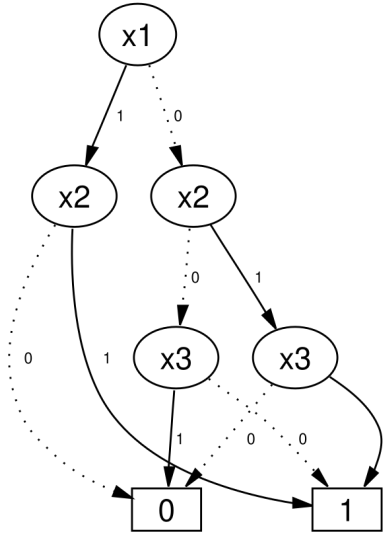


Figure 6: An equivalent Reduced Ordered BDD

x_1	x_2	x_3
0	0	0
0	1	1
1	1	2

Note here that we do not have to include truth values for function f in the table, as all the results in this table satisfy the entire function anyways. We see that we now only have three sets of variable valuations in which the function is satisfied: $[0, 0, 0]$, $[0, 1, 1]$, and $[1, 1, 2]$. Each of these sets of valuations are called a 'cube'. A cube is a set of variable valuations which satisfies the function represented by an ROBDD. In the third cube, we see that X_3 is valued as '2'. This implies that this variable is irrelevant for the outcome of the function, as this can take on either value 0 or 1.

As ROBDD's can be applied on LTL_f logic and provide a compressed representation, making it computationally efficient, these are used also in this research for extracting the truth valuations of LTL_f formulas. More information on (RO)BDD's can be found in [3].

2.11 LTL/ LTL_f Progression

As in this research we are dealing with temporally extended goals, i.e. a goal which spans a duration of time and involves a sequence of actions/subgoals to achieve, it is necessary to keep track of the progression of these goals. Since part of a temporally extended might already have been achieved, this does not have to be achieved again at a later point, for which only the part of the goal yet to be achieved needs to remain. This is achieved by progression.

We can define the *progression* of a LTL/LTL_f formula ϕ over a truth evaluation w of the variables

in P , $prog(\phi, w)$, as follows [1]:

$$\begin{aligned}
prog(\alpha, w) &\doteq \text{true if } w \models \alpha \text{ and false otherwise} \\
prog(\neg\phi, w) &\doteq \neg prog(\phi, w) \\
prog(\phi_1 \vee \phi_2, w) &\doteq prog(\phi_1, w) \vee prog(\phi_2, w) \\
prog(\bigcirc\phi, w) &\doteq \phi \\
prog(\phi_1 \mathcal{U} \phi_2, w) &\doteq prog(\phi_2, w) \vee (prog(\phi_1, w) \wedge \phi_1 \mathcal{U} \phi_2)
\end{aligned}$$

We can also add:

$$\begin{aligned}
prog(\phi_1 \wedge \phi_2, w) &\doteq prog(\phi_1, w) \wedge prog(\phi_2, w) \\
prog(\Diamond\phi, w) &\doteq (prog(\phi, w) \vee \Diamond\phi) \\
prog(\Box\phi, w) &\doteq (prog(\phi, w) \wedge \Box\phi) \\
prog(\phi_1 \mathcal{R} \phi_2, w) &\doteq prog(\phi_2, w) \wedge (prog(\phi_1, w) \vee \phi_1 \mathcal{R} \phi_2) \\
&(\equiv prog(\neg(\neg\phi_1 \mathcal{U} \neg\phi_2), w))
\end{aligned}$$

Intuitively, the progression of ϕ over a truth evaluation w is a formula ϕ' that represents what remains of ϕ after w has occurred, i.e., what remains to be satisfied over the rest of the trace after w . For example, consider the goal $\phi = \Diamond p$; if p holds in w , then the progression of ϕ over w is true, i.e., the goal had already been satisfied (in w); if on the other hand p does not hold in w , then the progression of ϕ over w is $\Diamond p$, i.e., we still need to eventually achieve p .

Note that after obtaining the progression, it is advisable to perform some Boolean simplifications of the resulting formula:

$$\begin{array}{ll}
\neg\text{false} \Rightarrow \text{true} & \neg\text{true} \Rightarrow \text{false} \\
\text{true} \wedge \phi \Rightarrow \phi & \phi \wedge \text{true} \Rightarrow \phi \\
\text{false} \wedge \phi \Rightarrow \text{false} & \phi \wedge \text{false} \Rightarrow \text{false} \\
\text{false} \vee \phi \Rightarrow \phi & \phi \vee \text{false} \Rightarrow \phi \\
\text{true} \vee \phi \Rightarrow \text{true} & \phi \vee \text{true} \Rightarrow \text{true}
\end{array}$$

3 Intention management system

As described in the introduction, the intention management system (IMS) should be able to handle goal change during a run, where intentions can be adopted and dropped. Important here is to note that the IMS only manages intentions of an agent, for which it holds that the IMS only manages the goals which an agent is dedicated to achieving. We are not dealing with desires or keeping track of a list of desires which can be dynamically changed by the IMS itself, based on their priorities and realizability. The IMS checks for realizability of all intentions which are provided by an agent, since the agent has to decide what intentions it intends on achieving. This means that the IMS is not responsible for what new intentions are added or dropped from the list of intentions. The IMS is only responsible for managing the intentions provided by the agent and checking their realizability.

As mentioned in Section 2.3, for this research LDL_f will not be covered, as this would increase the scope of this research too much. There would have to be an additional function in our approach for also progressing LDL_f intentions, as is described in [12]. Our automata-based approach could be easily adapted to handle LDL_f , although this has been left for future work, as described in Section 9. However, besides of progressing LDL_f intentions, our current design for the IMS should be able to work with LDL_f expressions also.

In terms of intentions, whenever an intention is added to the intention list, the IMS has to check whether this new intention is realizable in conjunction with the list of intentions the IMS already has. If this is not realizable, the new intention is not added to the intention list, where feedback is given to the agent on this, and which intentions should potentially be dropped from the intention list by the agent in order to add this new intention.

In the scenario where the new intention is realizable with the list of intentions the IMS already has, it is added to the intention list. Since the intention list has now changed, the winning region and maximally permissive strategy have to be recomputed by the IMS, since adding the new intention might have impact on the winning states and the maximally permissive strategy.

In the scenario where an intention is dropped by the agent from the intention list, the winning region and maximally permissive strategy also have to be recomputed. Also here it is true that whenever an intention is dropped, the winning states and maximally permissive strategy might change.

To give an idea of how the IMS would handle the adding and dropping of intentions during a run, we describe a theoretical scenario in Section 3.1 of how the IMS should operate in runtime, which takes place in the 'Triangle-Tireworld'-domain, as described in Section 6.9.1. After this example, we will give a more formal description for an IMS, which is based upon the given example.

3.1 Example scenario: Triangle-Tireworld

Consider an agent operating in the well known FOND domain called the Triangle-Tireworld (6.9.1) with a layout as shown in Figure 2.4 on p. 39 of [18]. The domain involves locations/nodes connected by roads. The agent can perform the action *move-car(from,to)* to go from location *from* to an adjacent location *to* provided that they are connected by an edge/road and she is at *from* and does not have a flat tire. After this action is performed, the agent will be at *to* and the agent may or may not have a flat tire, a nondeterministic effect which is decided by the environment. The agent can perform the action *changetire(location)* if she has a flat tire and there is a spare tire at *location*. After this action, she does not have a flat tire and there is no spare at *location*. See Example 4.4 on p. 18 of [17] for a precise specification of this FOND domain in PDDL. We assume that the domain layout is exactly as in Figure 2.4 on p. 39 of [18], except that there is a spare at location 13 and there is no spare at locations 22 and 33.

The scenario we want to model is as follows:

1. Initially, the agent is at location 11 and has the intention at priority 0 that

$$\diamond(at(32) \wedge final)$$

i.e., to eventually reach location 32 and stop³. She has no other intentions.

2. Then the agent does the action *move-car*(11, 12) and gets a flat tire.
3. Then the agent does the action *changetire*(12).
4. Then the agent acquires the intention at priority 1 that

$$\diamond(at(41) \wedge \bigcirc \diamond at(32))$$

i.e., to pass through location 41 on the way to location 32.

5. Then the agent does the action *move-car*(12, 21) and gets a flat tire.
6. Then the agent does the action *changetire*(21).
7. Then the agent acquires the intention at priority 2 that

$$\diamond(at(42) \wedge \bigcirc \diamond at(32))$$

i.e., to pass through location 42 on the way to location 32.

8. Then the agent does the action *move-car*(21, 31) and gets a flat tire.
9. Then the agent does the action *changetire*(31).
10. Then the agent does the action *move-car*(31, 41) and gets a flat tire.
11. Then the agent does the action *changetire*(41).
12. Then the agent drops the intention at priority 2 that

$$\diamond(at(42) \wedge \bigcirc \diamond at(32))$$

13. Then the agent does the action *move-car*(41, 32) and stops.

Observe that initially, the winning region includes all locations with a spare as well as the goal location 32. This is also the case after the agent goes to location 12. However after step 4 where the agent acquires the intention to pass through location 42 on the way to location 32, the winning region only includes the locations with spares on the left side of the Figure, as there are no spares at locations 22 and 33, so the agent cannot get from the right side to the left side of the Figure to go to location 42.

³Note that for FOND planning, we can actually drop *final* from the goal formula because there is no spare tire at location 32, so once the agent gets there she may not be able to leave.

3.2 Operational environment for the IMS

Using the example given in Section 3.1, we can extract the operational environment for the IMS, and the parties who are involved in this operational environment. As mentioned before, the IMS operates in a FOND domain, which means it is fully observable and non-deterministic. More details on FOND domains are explained in Section 2.6. In this operational environment we are dealing with an agent, the protagonist, and the environment, the antagonist. The information and queries sent by both parties towards the IMS are as follows:

The agent can send the following update operations to the IMS:

- *Add* intention to list of intentions during a run;
- *Drop* intention from list of intentions during a run;
- *Perform an action* from the maximally permissive strategy provided by the IMS. The possible actions only contain (non-)procrastinating actions in a current domain state, extracted from the maximally permissive strategy, which will keep the agent in the winning region. The agent can specify here if the provided actions should be either procrastinating or non-procrastinating;
- *Terminate* the system's operation.

The environment sends the following information to the IMS:

- A (non-)deterministic effect, as a *reaction* to an action chosen by the agent.

3.3 Requirements for the IMS

After knowing the operational environment for the IMS and which information is given to and asked by the IMS from each party involved, we can extract the responsibilities the requirements for the IMS and design a data structure which is able to handle the dynamic changing of intentions.

The IMS should be able to do the following:

- Compute the *winning region* and *maximally permissive strategy*, given a domain and set of intentions. This should also be possible at a later point in a run;
- Check if either a single intention or a list of intentions given by the agent are *realizable* in the current LTL_f domain specification;
- Get the *current world state*, which consists of the valuations of the current agent- and environment variables;
- Check whether all intentions have been *fulfilled*. If so, provide this as feedback to the agent;
- Keep track of a *list of intentions*;
- Get the *length of the intention list*;
- *Retrieve an intention*, given a position in the intention list;
- Check the *truth valuation for a given LTL_f formula* in the current world state;

- In the current world state, check the *progression for each of the intentions* which are in the current list of intentions. If any intention have been fulfilled, provide this as feedback to the agent and remove this intention from the intention list, since it has already been achieved. If any progress has been made in a given intention, but has not yet been fulfilled completely, update the intention with what remains of it. The progression of intentions is explained in more detail in Section 3.7;
- Check if the current list of intentions is *realizable in conjunction with a newly provided intention* in the current domain state;
- *Add new and drop old intentions* from a list of intentions during the execution of a run, based on what is instructed by the agent;
 - If a newly added intention is realizable in conjunction with the intentions we already have, the IMS adds the new intention to the list of intentions it already had, and recomputes the winning region and maximally permissive strategy;
 - If a newly added intention is not realizable in conjunction with the intentions we already have, the IMS should forget about this new intention. However, the IMS will check, based on priority of the intentions (position in the list) which intentions would have to be dropped in order to add the new intention, and returns this to the agent. Intentions should only be added or dropped by the agent, so the IMS can only give suggestions to the agent of which intentions should be dropped;
 - If an intention from the intention list is dropped, recompute the winning region and maximally permissive strategy. Since the list of intentions was already realizable before dropping the given intention, the updated domain will also be realizable for the agent after recomputing the winning region and maximally permissive strategy, which is why we do not have to explicitly check again for realizability;
- Check if the *action chosen by an agent* in the current domain state is guaranteed to *keep the agent in the winning region*;
- Get the *set of all actions which keep the agent in the winning region* in the current domain state;
- Check if the *action chosen by an agent* in the current domain state is guaranteed to *keep both the agent in the winning region and make progress towards a final state* in which all intentions are satisfied;
- Get the *set of all actions which keep both the agent in the winning region* in the current domain state and *make progress towards a final state* in which all intentions are satisfied;
- *Terminate the system* if the agent gives instructions to do so.

These requirements are the foundation of the IMS. Section 4 provides a description how a proof of concept can be built in the form of software, which is based on these requirements.

Important here is to note that we only design the architecture of the IMS, since the workings of the agent and environment are not relevant in this research. We only want to know which outputs of the agent and environment are used as input for the IMS.

3.4 Formal description of an IMS state

Now that we have a specification of which queries the IMS should be able to handle, what its responsibilities are, and in which environments it should be able to handle, we can write out the information required for the IMS in each state.

An IMS state is a triple $I = \langle D, s, L \rangle$, where D is a FOND planning domain, s is a state of D , and L is a list of agent's intentions. At system initialization time, we set s to an initial state and provide such a prioritized list of intentions (possibly empty). The IMS state can be queried and updated through specific operations.

We can use these components of the IMS state in order to fulfill the requirements for the IMS, as described in Section 3.3. In Section 3.5 and 3.6, the query- and update operations are described which the IMS should be able to execute.

3.5 The query operations of the IMS

First we initialize the system with the FOND domain model M and (initial) world state s . This could be represented as a PDDL nondeterministic domain or in some other language (e.g., an NDBAT in the situation calculus).

The agent/system's intentions will be represented by a list $L = [\phi_0, \dots, \phi_n]$ of ltl_f formulas ordered in decreasing order of priority, i.e., ϕ_0 is the highest priority intention and ϕ_n the lowest. At any point in time, the system should ensure that this intention list L , i.e., its intention state, is *realizable*, i.e., that the agent has a strategy to ensure that all its intentions are satisfied no matter how the environment behaves. With a little abuse of notation, sometimes we will use L itself as an abbreviation for ltl_f formula $\bigwedge_{\phi_i \in L} \phi_i$. Note that the empty list $[\]$ corresponds simply to True.

At system initialization time, we also provide such a prioritized list of intentions.

The system will support query- and update operations. The query operations are:

- $I.getDomainState()$: returns the current domain state s ;
- $I.isInFinalState()$: returns whether the current IMS state is such that all intentions are satisfied and the agent may stop, i.e., is final;
- $I.getIntentions()$: returns the current list of intentions L ;
- $I.getIntentionsLength()$: returns $length(L)$;
- $I.getIntention(k)$: returns the intention at index k from L (requires that $k < length(L)$);
- $I.holdsInCurrentState(\psi)$: return the truth value of state formula ψ in the current world state s ;
- $I.isWinning(a)$: returns whether executing agent action a in the current IMS state I is such that there exists a winning strategy σ for L starting at the current state of the domain s such that $\sigma(s) = a$;
- $I.getWinningActions()$: returns the set of all actions a such that executing a in the current IMS state is guaranteed to remain in the winning region;

- *I.isProgressing(a)*: returns whether executing action a in the current IMS state is guaranteed to progress towards a fulfilment all the intentions, i.e., there exists a winning strategy σ for L starting at the current state of the domain s such that $\sigma(s) = a$ and for every other winning strategy σ' , such that $\sigma'(s) \neq a$ we have that $\maxSteps(\sigma, s, L) \leq \maxSteps(\sigma', s, L)$;
- *I.getProgressingActions()*: returns the set of all actions a such there exists a winning strategy σ for L starting at the current state of the domain s such that $\sigma(s) = a$ and for every other winning strategy σ' , such that $\sigma'(s) \neq a$ we have that $\maxSteps(\sigma, s, L) \leq \maxSteps(\sigma', s, L)$;
- *I.isRealizable(ϕ, k)*: checks whether inserting new intention ϕ at position k yields a realizable set of intentions; it returns `True` if $\phi \wedge \bigwedge_{\phi_i \in L} \phi_i$ is realizable; `False` if $\phi \wedge \bigwedge_{\phi_i \in [\phi_0, \dots, \phi_{k-1}]} \phi_i$ is not realizable; otherwise, it returns *I.isRealizable($\phi, k, length(L)$)*, where *I.isRealizable(ϕ, k, j)* is defined as follows: if $j = k-1$, then *I.isRealizable(ϕ, k, j)* = \emptyset ; else, *I.isRealizable(ϕ, k, j)* = *I.isRealizable($\phi, k, j-1$)* $\cup \{j\}$ if $\phi \wedge \bigwedge_{\phi_i \in [\phi_0, \dots, \phi_{k-1}]} \phi_i \wedge \bigwedge_{m \in I.isRealizable(\phi, k, j-1)} \phi_m \wedge \phi_j$ is realizable, and *I.isRealizable(ϕ, k, j)* = *I.isRealizable($\phi, k, j-1$)* otherwise; essentially, it returns the set intention indexes $\geq k$ that can/should be kept.

Intuitively, it may not always be obvious why each of these query operation might be needed. For example, one might wonder why it is useful to know whether a given action keeps the agent in the winning region by calling *I.isWinning(a)*, when this can already be extracted by retrieving the set of winning actions by calling *I.getWinningActions()*. As in this research the IMS is only meant to be a proof of concept, it is not tested on elaborate domains where the agent might have a large set of possible actions to perform. As the agent might retrieve a large set of actions, possibly hundreds of them, this might not be efficient for retrieving a winning action. To give the agent as much freedom as possible, the agent should also be able to provide an action itself, where it is simply returned if this action is either winning or progressing.

As how the queries are currently designed, it leaves the agent with as much freedom as possible, while the IMS is already prepared for potential new features in future research.

3.6 The update operations of the IMS

In this Section, we specify the update operations performed by an agent within the operational environment, as explained in Section 3.2.

The update operations are:

- *I.halt*: terminate the system's operation;
- *I.drop(k)*: drops the k -th intention from L (requires that $k < length(L)$). Updates $L = L.remove(k)$;
- *I.do(a)*: If *I.isWinning(a)* = `False` does nothing, otherwise the agent executes a , then the system observes the new world state due to the environment reaction s' and updates the IMS $I = \langle D, s, L \rangle$, to $I = \langle D, s', L' \rangle$, where $L' = [\phi'_0, \dots, \phi'_n]$ with each $\phi'_i = prog(\phi_i, (a, s'))$, i.e., obtained by progressing each $\phi_i \in L$ through (a, s') (note that here (a, s') stands for the interpretation $\{a\} \cup s'$);
- *I.adopt(ϕ, k)*: adopt the intention ϕ at priority k (requires that $k < length(L)$); Let $ris = I.isRealizable(\phi, k)$. If $ris = \text{True}$, then set L to *L.insert(ϕ, k)* and return `True`; else, do nothing and return `False`;

3.7 Progressing an LTL_f intention

For describing how LTL_f intentions are progressed, we use the theory described in Section 2.11. Intentions can be expressed in a complex manner, sometimes needing several specific steps in order to fulfill them. Every type of intention expressed in LTL_f can be progressed by the IMS. This can, for example, be reachability, sequences, or safety properties, among other properties [11]. Since in this research we are dealing with temporally extended goals, there is a need for progressing intentions, such that it can be measured to which extend a set of given intentions has been fulfilled. An example of how such an intention is progressed, is described in Section 3.7.1.

3.7.1 Example for progressing an LTL_f formula

We take an example intention from a pseudo domain: intention ' $\diamond(s1 \wedge \bigcirc \diamond(s2))$ '. Which domain this is exactly, is not relevant, since we only care about the progression of intentions, regardless of the exact domain. In this example, we consider variable $s1$ to be 'state 1', and $s2$ to be 'state 2'. In natural language, this intention would express the following: "eventually be in state 1, and next, eventually be in state 2".

In this setting, if the agent in the next step does not reach $s1$, the formula remains as it is before: $\diamond(s1 \wedge \bigcirc \diamond(s2))$. This is still considered progression, even when the intention is not altered, as is described in Section 2.11.

In another scenario where the agent at some point does reach $s1$, the formula is both progressed and changes from its previous form, since a part of the intention has now been fulfilled: $\diamond(s1)$. The intention now progresses from $\diamond(s1 \wedge \bigcirc \diamond(s2))$ to $\diamond(s2)$, where the remaining intention is to eventually reach $s2$: $\diamond(s2)$. Once we also reach $s2$, the intention progresses into *true*, as described in Section 2.11, since now the intention has been fulfilled, and afterwards will remain *true*.

3.8 Usage of the IMS

Now that we have a notion of the operational environment and of the query- and update operations which the IMS is supposed to handle, it can be specified how the IMS works during runtime. We split this up in three parts: the initialization of a run, the operations performed during a run, and the completion of a run.

3.8.1 Initialization of a run

Before a run is started, the IMS is provided with a FOND domain and an initial list of intentions. The entire intention list is assumed to be realizable, given the FOND domain. If this list is not realizable, the run is not started. Once a realizable list of intentions has been provided, the environment initializes with its initial variables. Note that no query- or update operations have been executed yet by the IMS up to this point, as realizability for the entire list of intentions and the initial DFA are not computed by the IMS itself during the initialization of a run.

3.8.2 Operations during a run

Firstly, the IMS calls the query $I.getDomainState()$, which returns the current domain state s . The agent can now perform an action, either procrastinating or non-procrastinating. The IMS provides a set of actions, chosen by the agent, either $I.getWinningActions()$, which contains both the procrastinating- and non-procrastinating actions, or $I.getProgressingActions()$, which contains

only the non-procrastinating actions. The agent picks an action, and the IMS calls update operation $I.do(a)$, where a is the action picked by the agent. This operation returns s , the environment reaction. All intentions of the intention list are progressed implicitly by update operation $I.do(a)$, and the domain transitions into a new state, given the agent- and environment action. For the IMS to implicitly check if an intention from the intention list has been fulfilled after progressing, it calls query operation $I.holdsInCurrentState(\psi)$, where ψ is the intention being checked.

In the new domain state, the agent is now able to call the update operations $I.adopt(\phi, k)$, $I.drop(k)$, or $I.halt$, which are for adding a new intention to the intention list, dropping an intention from the intention list, or terminating the system's operation, respectively.

- If the agent calls operation $I.adopt(\phi, k)$, the IMS checks for realizability of the new intention, ϕ , in conjunction with the current intention list, in the current domain state. For this, the IMS queries $I.getDomainState()$, $I.getIntentions()$, and $I.isRealizable(\phi, k)$ are used.
 - If $I.isRealizable(\phi, k)$ returns true, the new intention is added to the list of intention at position k using update operation $I.adopt(\phi, k)$. A new domain state and maximally permissive strategy are computed, since these have changed after altering the intention list.
 - If $I.isRealizable(\phi, k)$ returns false, the agent is informed that it is not possible to add the new intention to the list of intentions. $I.isRealizable(\phi, k)$ checks which subset of intentions with highest priority would be realizable in conjunction, where the IMS returns to the agent which intentions of the intention list would have to be dropped, based on priority, in order to add the new intention and make the intention list realizable, as specified in Section 3.6. The agent is informed about the intentions which would have to be dropped in order to add the new intention.
- If the agent calls operation $I.drop(k)$, the intention with index k is dropped from the intention list. A new domain state and maximally permissive strategy are computed, since these have changed after altering the intention list.
- If the agent calls operation $I.halt$, the run is finalized, which is explained in more detail in Section 3.8.3.

After calling $I.halt$, all query- and update operations which an agent might need during a run are covered, except for the following query operations:

- $I.getIntentionsLength()$;
- $I.getIntention(k)$;
- $I.isWinning(a)$;
- $I.isProgressing(a)$.

Note that these operations are not explicitly used by the agent in these scenarios. Query $I.getIntentionsLength()$ and $I.getIntention(k)$ can be used by the IMS implicitly when checking for the agent which subset of intentions from an intention list are realizable, for which these queries are also included for potential usage by the agent in future work. Similarly for the queries $I.isWinning(a)$ and $I.isProgressing(a)$, these are mainly included for future usage. Although the set of winning- and progressing actions can already be extracted by calling $I.getWinningActions()$ and $I.getProgressingActions()$, if the set of possible actions returned is extensive, it would be easier for the agent to provide an action by itself, for which it would be returned if this action is either winning or progressing.

3.8.3 Completion of a run

After each domain transition, the IMS calls the query operation *I.isInFinalState()*, which returns if all the intentions are satisfied. Feedback from the IMS is given to the agent if in the current domain state all intentions are satisfied, however the run is not automatically stopped, since only the agent can stop a run. If the agent wants to continue the run, it can. The agent can call the update operation *I.halt*, which makes the IMS terminate the system's operation.

4 Design of the IMS

We can evaluate the data structure of the intention management system described in Section 3 by making it into a proof of concept in the form of software, which we can then use to process benchmark FOND domains and problems. In Section 3, a description was given for an IMS which talks about domains as a general concept. However in this section, we will assume the domain and intentions are conjoined into a single DFA (2.4.1). In Section 4.1, a more detailed description is given for the design of the IMS.

4.1 Description for the design of the IMS

The IMS as a proof of concept is based on other software, called Lydia (5.1.1) and Syftmax (5.1.2).

Important here is to note that the benchmark FOND domains are specified in LTL_f . In our manner of implementation the domain specification is joined with the intention specification to obtain a single LTL_f formula. This is then transformed into a DFA by Lydia, representing the entire specification. Syftmax is ran on this afterwards, for obtaining the winning region and maximally permissive strategy. The DFA and maximally permissive strategy are only recomputed whenever the list of intentions from an agent changes. The composition of the LTL_f formula for running the synthesis algorithm on for extracting the DFA is explained in more detail in Section 4.6. When an agent executes an action, the IMS observes the environment reaction. After this, the DFA state is advanced into a next DFA state, and all intentions are progressed, given the world state.

The LTL_f specification for the domain and intentions remain the same throughout a run, except for when the DFA is recomputed. At this point, the initial environment initialization is set as the current environment variable valuations, and the goal is replaced by an updated list of intentions. As this is how we designed the IMS, for our implementation the LTL_f formula is split up in several components, such that dynamic adjustment of the LTL_f specification can be applied over the environment initialization and the goal during a run. The contents of these individual components are specified in more detail in Section 4.5.

As already mentioned, there are many possible manners for representing a FOND domain- and problem specification in LTL_f , where in this implementation we represent both the domain- and problem specification in a singular DFA. By doing so, we do not have to split the specification into several DFA's, as the singular DFA contains all necessary information for the IMS.

After a DFA for the domain- and problem specification is computed, it can be saved within the IMS state for the proof of concept, as shown in Section 4.2, which is then used for executing query- and update operations. This can be considered as one of the simplest manners for computing the winning region for an agent, as the IMS does not have to keep track of progression for several DFA's at the same time, and there is no interaction required with any other DFA's.

Another manner for computing the winning region is to compute a separate DFA for the domain, and compute separate DFA's for each intention of the agent, which in the end might be less computationally expensive. However, as this implementation for the IMS is only meant as a proof of concept, we do not focus on computing the winning region for the agent as efficiently as possible, and leave this as potential future work.

4.2 IMS state for the design

For describing the IMS state of the proof of concept, we start from the mathematical description for the IMS state as described in Section 3.4.

The state of an IMS for the proof of concept is written more elaborately, where we also express the winning region, (non-)procrastinating strategy, and current DFA state. Important here is to note that this is our manner of designing an IMS. However, there are many other ways of designing this. For distinguishing the IMS state for the mathematical description and the IMS state for our design, we will define the IMS state for our design as I^+ , as this contains more information than the mathematical definition for the IMS state. For our design, the IMS state is written as follows: $I^+ = (D, s, L, \text{DFA}, s_{\text{DFA}}, w, ps, nps)$, where

- D is the FOND domain represented as an LTL_f formula. This is used for constructing a DFA. An example for a FOND domain specified in LTL_f is provided in Section 4.5.2;
- s is the current state of the domain. In this state, the agent- and environment variables are captured, including their truth values, represented as a BDD. This is used for making transitions from one state to a next one in the DFA, and for progressing intentions;
- L is the current list of LTL_f intentions. At the initialization of a run, this list is provided by an agent. During a run, the agent can add and drop intentions from this list;
- DFA is the DFA, representing D , the FOND domain, and L , the current list of intentions. As already mentioned in the introduction, the DFA is computed using Lydia (5.1.1), where D and L are joined and provided as a singular LTL_f formula;
- s_{DFA} is the current state of the DFA, represented as a BDD. This is used for checking if the agent is in the winning region/a final state, and for extracting what moves an agent can execute in the current DFA state in order to stay in the winning region when combined with the maximally permissive strategy, consisting of ps and nps . Also, the current DFA state s_{DFA} , combined with s and the transition function of a DFA, returns the next s_{DFA} ;
- w is the winning region of the DFA, represented as a BDD;
- ps is the procrastinating strategy, extracted from the maximally permissive strategy, represented as a BDD;
- nps is the non-procrastinating strategy, extracted from the maximally permissive strategy, represented as a BDD;

As the IMS state for the design is more elaborate compared to the IMS state as provided in Section 3.4, the internal workings of the query- and update operations will also be different in our design. How the query- and update operations work in our implementation, is explained in more detail in Section 4.3.

4.3 The query- and update operations for the design

For the usage of query- and update operations, we take the specification provided in Section 3.5 and 3.6, and apply this to the IMS state for our design as described in Section 4.2. Here we describe how the variables for our design of the IMS state are used by the query- and update operations of the IMS internally as follows:

Query operations:

- $I^+.\text{getDomainState}()$: returns the current domain state s from the IMS state, containing the agent- and environment variables, including their truth valuations;

- $I^+.isInFinalState()$: returns whether the current IMS state is such that all intentions are satisfied and the agent may stop, i.e., is final. For this to be true, the current DFA state s_{DFA} has to be in w , and L has to be empty;
- $I^+.getIntentions()$: returns the current list of intentions L from the IMS state;
- $I^+.getIntentionsLength()$: returns $length(L)$, given L from the IMS state;
- $I^+.getIntention(k)$: returns the intention at index k from L (requires that $k < I^+.getIntentionsLength()$), where L is taken from the IMS state and k is provided externally;
- $I^+.holdsInCurrentState(\psi)$: return the truth value of state formula ψ in the current state s , where ψ is provided externally and s is taken from the IMS state;
- $I^+.isWinning(a)$: returns whether executing agent action a in s_{DFA} is part of the procrastinating strategy, keeping the agent in the winning region. Here, s_{DFA} and ps are taken from the IMS state, where a is provided externally;
- $I^+.getWinningActions()$: returns the set of all actions an agent can execute in s_{DFA} which remain in the winning region, where s_{DFA} and ps are taken from the IMS state;
- $I^+.isProgressing(a)$: returns whether executing agent action a in s_{DFA} is part of the non-procrastinating strategy, progressing the agent towards a final state. Here, s_{DFA} and nps are taken from the IMS state, where a is provided externally;
- $I^+.getProgressingActions()$: returns the set of all actions an agent can execute in s_{DFA} which progress towards a final state, where s_{DFA} and nps are taken from the IMS state;
- $I^+.isRealizable(\phi, k)$: A copy of L is made, ' L_{copy} ', where new intention ϕ is inserted at position k in L_{copy} . A new DFA is created using D , s , and L , for which realizability is checked. If L_{copy} is realizable, $I^+.isRealizable(\phi, k)$ returns *True*. If this returns *false*, $I^+.isRealizable(\phi, k)$ is iteratively called, where in each iteration a subset of L_{copy} is taken, starting with the highest priority intentions, and gradually decreasing which prioritized intentions are selected, as described in Section 4.4.1. Whenever a subset is found which is realizable, it is returned what intentions this realizable subset consists of. This maximally prioritized realizable subset of intentions is represented the same as described for the $I.isRealizable(\phi, k)$ -query in Section 3.5. If no realizable subset exists, return *False*. Variables ϕ and k are provided externally, while D , s , and L are taken from the IMS state;

We have now defined the complete set of query operations for the design of the IMS, as this set contains all the same query operations as specified in Section 3.5.

Update operations:

- $I^+.halt$: terminate the system's operation;
- $I^+.drop(k)$: drops the k -th intention from L (requires that $k < I^+.getIntentionsLength()$). As L has changed, DFA, s_{DFA} , w , ps , and nps are recomputed, as these have changed as a result of L . DFA is recomputed using D , s , and L , where s_{DFA} , w , ps , and nps are extracted from DFA. In the IMS state, DFA, s_{DFA} , w , ps , and nps are replaced. The only external variable is k , while the result of all used variables come from the IMS state;

- $I^+.do(a)$: If $I^+.isWinning(a) = False$, the IMS does nothing. Otherwise, the IMS executes a in s_{DFA} , where an environment reaction is returned. a and the environment reaction result in s , the current domain state, which is updated in the IMS state. Taking s_{DFA} and s , and applying these on the transition function extracted from DFA, the next s_{DFA} is returned. All intentions of $I^+.L$ are internally progressed, given $I^+.s$. The IMS state updates s_{DFA} with the resulting s_{DFA} . Variable a is provided externally, while DFA and s_{DFA} are taken from the IMS state;
- $I^+.adopt(\phi, k)$: try to adopt intention ϕ at priority k (requires that $k < I^+.getIntentionsLength()$). Firstly, $I^+.isRealizable(\phi, k)$ is called. If $I^+.isRealizable(\phi, k)$ returns *True*, adopt new intention ϕ at priority k . As L has changed when a new intention is adopted, DFA, s_{DFA} , w , ps , and nps are recomputed, as these have changed as a result of L . DFA is recomputed using D , s , and L , where s_{DFA} , w , ps , and nps are extracted from DFA. In the IMS state, DFA, s_{DFA} , w , ps , and nps are replaced. The only external variable is k , while the result of all used variables come from the IMS state; If $I^+.isRealizable(\phi, k)$ returns *False*, do nothing;

We now have all update operations for the design of the IMS, all of which have also been used in the definition for the IMS in Section 3.6. Together with the query operations, the design for all operations which the IMS can execute is now complete for usage by an agent. All operations of the design comply with the requirements for an IMS as described in Section 3.3, where these operations function as described in the introduction of Section 4. In Section 5.2, a description is given for how these query- and update operations can be used by an agent within a simulation of a FOND domain.

4.4 Adding and dropping of an intention to list of intentions

For our design of the IMS, the manner of adding new intentions and dropping old intentions is to check if a new intention is realizable in conjunction with the intentions we are currently already committed to. If the new intention is realizable, together with our current intentions, we can adopt it in our current list of intentions and recompute the maximally permissive strategy. If the new intention is not realizable in conjunction with our current intentions, the IMS does not add the new intention to the list of intentions and only stays committed to set of intentions we already had. However, the IMS can give a suggestion to the agent on which intentions would have to be dropped in order to add the new intention to the list of intentions. This suggestion is based on the priority of intentions, since the position of an intention in the intention list indicates its priority, for which a more detailed description is given in Section 4.4.2.

Note that checking realizability, adopting an intention, and dropping an intention are solely query- and update operations which an agent can call from the IMS. How these query- and update operations are used during a run is not part of the definition for the IMS, but is part of our implementation for the IMS, which we have used for evaluating the IMS in simulations. Section 3 discusses which query- and update operations can be called from the IMS, where our design and implementation for the IMS during a simulation is provided in Section 4 and 5. In Section 4.4.1, we provide an example of how an intention can be added and dropped from the list of intentions.

4.4.1 Example for adding a new intention to the list of intentions

Now that we have a general idea of how the IMS handles the adding of new intentions and checking their realizability, we will give a concrete example. We provide an example of how new intentions can be added to the intention list, as will be done during the implementation of the IMS in Section 5, and what happens in the scenario where it is not realizable to add a new intention to the intention list.

We will consider the following stepwise example scenario:

1. The agent provides the IMS a list of intentions at initialization, $[A, C]$. Where the left-most intention A has highest priority (priority 0), and the right-most intention C has lowest priority (priority 1). We set the intentions $[A, C]$ as ϕ_{goal} . Note: which specific domain is used here is irrelevant, since it is only used for explanation and is applicable to any domain.
2. As we will see in the implementation of the IMS in Section 5, the winning region is computed for the domain, and the IMS checks, using the winning region, if it is realizable for the agent to reach all given intentions. Note that in the formal definition for the IMS in Section 3, the manner for computing the winning region and maximally permissive strategy are not included, as this is part of the implementation, as will be discussed in more detail in Section 5. It is returned to the IMS that these intentions are indeed realizable for the agent. The IMS extracts the maximally permissive strategy for the agent using the winning region, and provides to the agent what actions can be executed in order to remain in the winning region. The agent picks an action, and the environment responds to this. The domain transitions into a new state, given the agent action, the environment response, and the initial domain state.
3. After this transition, the agent decides that it wants to add a new intention to the intention list: intention D , with priority 2. The IMS checks if intention D is realizable in conjunction with intentions $[A, C]$ in the current domain state by calling query operation $I^+.isRealizable(\phi, k)$, where D is represented by ϕ and priority 2 is represented by k . This appears to be realizable, so the agent adds intention D with priority 2 in the intention list by calling update operation $I^+.adopt(\phi, k)$, where ϕ represents D and priority 2 represents k , adding D in the right-most position of the intention list, resulting in list $[A, C, D]$. After adding the new intention, the winning region and maximally permissive strategy are recomputed by the IMS, as these have changed after adding a new intention.
4. After adding intention D to the intention list, the agent decides that it wants to add another intention: intention B with priority 1. Again, the IMS checks if intention B is realizable in conjunction with intentions $[A, C, D]$ by calling $I^+.isRealizable(B, 1)$. However, this is not realizable. The IMS returns to the agent that adding intention B is not realizable given the current intention list. In this scenario however, the IMS will check what intentions should be dropped if the agent would want to add intention B with priority 1. The exact process for how this is computed is described in Section 4.4.2.
5. The IMS starts by checking realizability by leaving out the intention with lowest priority: intention D (right-most intention). The IMS now checks if the intentions $[A, B, C]$ are realizable, which appears not to be the case. The IMS checks again for realizability, now by leaving out the intention with second-lowest priority: intention C . The IMS now checks realizability for intentions $[A, B, D]$, which appears to be realizable. The IMS returns to the agent that in order to add intention B to our intention list $[A, C, D]$, the agent would have to drop intention C by calling update operation $I^+.drop(k)$, where the priority of C is represented by k . Whether intention C is actually dropped in order to add intention B is decided by the agent. Note that the IMS does not drop any intentions from the intention list when this is not commanded by the agent. The IMS only gives feedback to the agent which intentions should be dropped in order to add a given intention, based on its priority.

6. Now, the agent continues the run as before. Whenever the agent wants to add- or drop an intention from the list of intentions, the process as described before is repeated, until all intentions have been fulfilled.

4.4.2 Revision of the intention list by the $I^+.isRealizable(\phi, k)$ query operation

In Section 4.4.1, an example is given where a new intention ϕ at priority k is currently not realizable in conjunction with a current intention list. As already mentioned, in this scenario the IMS will not alter the intention list in order to add ϕ at position k in order to maximize utility, as in this research we only want an agent to add- and drop intentions from the intention list, where the IMS only serves as a tool for the agent in order to manage its intentions and for providing query- and update operations.

In Section 3.5 a formal description is already provided for how an intention list is revised for giving feedback to an agent for maximizing utility when an agent wants to add a new intention. For understanding this process of revision, we provide a more intuitive explanation.

The intention list could be seen as a binary encoding. When given an intention list for revision $[A, C, D]$, and an agent calls $I^+.isRealizable(B, 1)$, where realizability is checked for new intention B at priority 1, a copy is made of the intention list in which intention B is added, for which realizability can be checked, without actually adding it to the current intention list. This copied list with the new intention is represented as $[A, B, C, D]$.

Temporary intention list $[A, B, C, D]$ is transformed into a binary encoding $[0, 0, 0, 0]$, where 0 implies an intention is included in the current intention list checked for realizability, and 1 implies an intention is left out for the current check for realizability. If iteratively we would increase the binary encoding for the set of intentions, this would stepwise be processed as follows:

$[0, 0, 0, 0] = [A, B, C, D]$

$[0, 0, 0, 1] = [A, B, C]$

$[0, 0, 1, 0] = [A, B, D]$

$[0, 0, 1, 1] = [A, B]$

...

until $[1, 1, 1, 1] = \emptyset$.

If during this iterative process for checking realizability a realizable set of intentions is found which maximizes utility, this set is returned to the agent and the iterative search for other realizable sets is stopped. If at no point a set is found which is realizable, the emptyset is returned. This provides feedback for an agent on how a new intention can be added to an intention list, while maximizing utility.

4.5 Individual components for constructing an LTL_f formula

Within a domain, there are several components specifying the workings, restrictions, initialization, and goal for both the agent and the environment. For our research, we will distinguish two types of components: the domain specific components, and the problem specific components. The domain components specify the workings/rules of a domain, which includes the mutual exclusion of agent actions, the environment reactions to an agent action, the mutual exclusion of environment variables (if applicable), the frame (unchanging environment variables over time), and the preconditions for agent actions. The problem components includes the environment initialization and the initial goal.

The problem components can change depending on the proposed scenario, as this does not influence the workings/rules of the domain itself. Note here that even though the workings of the environment are included in the domain workings/rules, these could change whenever a domain is scaled up or down. For example, if we would take the slippery world domain described in Section

6.8.1, we see that there are four rows and four columns. If we would scale up the rows and columns to five, the workings/rules of the domain now also have to be applied to the fifth row and column, as these are now also part of the domain. Below, we describe the components in the domain- and problem specification:

Domain specification:

- $\phi_{ag\ uniq\ act}$: specifies the mutual exclusion of executing agent actions (agent executes exactly one action at a time)
- $\phi_{env\ trans}$: specifies all (non-)deterministic reactions of the environment, given an agent action. This also includes the frame, which describe environment variables which remain unchanged over time. If applicable, mutual exclusion of environment variables can also be specified here.
- $\phi_{ag\ act\ prec}$: specifies for the agent what the preconditions are in order to execute an action.

Problem specification:

- $\phi_{env\ init}$: specifies what the initial values are for the environment variables.
- ϕ_{goal} : specifies a conjunction of all intentions which an agent is dedicated to achieving.

As LTL_f is based on propositional logic, we also need to define the variables, i.e. atomic propositions, for the agent and the environment, which are used for writing out the domain- and problem specification.

Note that the manner of distinguishing components for a domain- and problem specification in this section is how we have designed it for our research. This does not originate from previous research, as we have designed this manner for distinguishing components as part of our implementation of the IMS. We use these components in Section 4.6 and 4.7 to compose LTL_f formulas for extracting good agent- and environment moves, which is explained in more detail in these sections.

4.5.1 Description of the decision tree domain

In the previous Section, we have described the individual components from which we compose an LTL_f formula to extract good agent moves. In order to explain how this would look like for an actual domain we use for evaluating our implementation of the IMS, we first give an example. This example is given using the 'decision tree domain'.

This domain is designed by ourselves, with the purpose of describing a FOND domain in one of its simplest forms. An advantage of this domain is that it is easy to explain, and simple to write out in LTL_f . An example with a visual representation of this domain is displayed in Figure 7. In this example, there are 6 environment states in total, although the amount of states can be easily scaled to preference. The agent has 2 action variables: left and right. The environment has 6 variables: s1, s2, s3, s4, s5, and s6, for which the environment can place the agent in only one state at a time. An example run in this domain is described as follows:

- The environment initializes in state s1. The goal for the agent is to eventually reach goal s5.
- In the initial state, s1, the agent can perform action left or right.
- The environment can respond to this by indicating in which state the agent will be next. In this example, the environment applies a non-deterministic effect in s1, where it does not matter if the agent executes left or right, since the environment will place the agent non-deterministically next in either state s2 or s3.

- In the next domain state, the agent will be located in either state s_2 or s_3 , which it cannot determine beforehand.
- In this next domain state, the agent has to reevaluate what action it needs to perform in order to eventually reach state s_5 . In s_2 this would be action right, and in s_3 this would be action left.

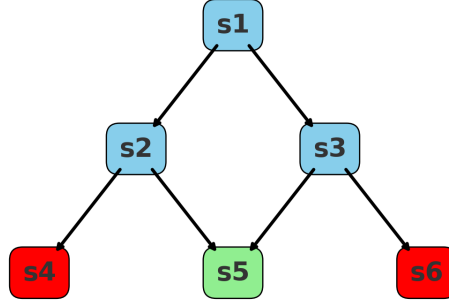


Figure 7: Decision tree domain

4.5.2 Example of individual components for composing an LTL_f formula in the decision tree domain

In this Section we describe an example domain- and initialization setting for the decision tree domain. The decision tree domain is described in more detail in Section 4.5.1. In this scenario, the domain is represented identical to the description given in Section 4.5.1, where the environment initializes in state s_1 , and the agent eventually has to reach state s_5 .

Before writing out the domain- and problem specification, we need to have the set of agent- and environment variables. These are as follows:

- agent variables: $\{l, r\}$
- environment variables: $\{s_1, s_2, s_3, s_4, s_5, s_6\}$

For which agent variable l is action 'left' and r is action 'right'.

Using the individual components of an LTL_f formula as described in Section 4.5 and the agent- and environment variables described above, we can write out the following components:

Domain specification:

- $\phi_{ag\ uniq\ act} : (l \vee r) \wedge \neg(l \wedge r)$
- $\phi_{env\ trans} :$
 - mutual exclusion of environment variables :

$$(s_1 \vee s_2 \vee s_3 \vee s_4 \vee s_5 \vee s_6) \wedge \neg(s_1 \wedge s_2) \wedge \neg(s_1 \wedge s_3) \wedge \neg(s_1 \wedge s_4) \wedge \neg(s_1 \wedge s_5) \wedge \neg(s_1 \wedge s_6) \wedge \neg(s_2 \wedge s_3) \wedge \neg(s_2 \wedge s_4) \wedge \neg(s_2 \wedge s_5) \wedge \neg(s_2 \wedge s_6) \wedge \neg(s_3 \wedge s_4) \wedge \neg(s_3 \wedge s_5) \wedge \neg(s_3 \wedge s_6) \wedge \neg(s_4 \wedge s_5) \wedge \neg(s_4 \wedge s_6) \wedge \neg(s_5 \wedge s_6)$$

- environment reactions : $(s1 \rightarrow \bigcirc(l \rightarrow (s2 \vee s3))) \wedge (s1 \rightarrow \bigcirc(r \rightarrow (s2 \vee s3))) \wedge (s2 \rightarrow \bigcirc(l \rightarrow s4)) \wedge (s2 \rightarrow \bigcirc(r \rightarrow s5)) \wedge (s3 \rightarrow \bigcirc(l \rightarrow s5)) \wedge (s3 \rightarrow \bigcirc(r \rightarrow s6)) \wedge (s4 \rightarrow \bigcirc((l \vee r) \rightarrow s4)) \wedge (s5 \rightarrow \bigcirc((l \vee r) \rightarrow s5)) \wedge (s6 \rightarrow \bigcirc((l \vee r) \rightarrow s6))$
- frame : -
- $\phi_{ag \text{ act } prec} : (l \vee r) \rightarrow (s1 \vee s2 \vee s3 \vee s4 \vee s5 \vee s6)$

Problem specification:

- $\phi_{env \text{ init}} : s1$
- $\phi_{goal} : \diamond(s5)$

As already mentioned before, $\phi_{env \text{ init}}$ and ϕ_{goal} will change depending on the proposed problem specification. During a run, ϕ_{goal} will change as new intentions get added and dropped.

4.6 LTL_f formula composition for extracting good agent moves

We want to extract strategies for the agent, given a domain and problem specification, for which we can make sure that the agent can reach its goal, as long as both the agent and the environment play according to the rules of a given domain specification. This means that whenever the agent breaks a rule of the domain, e.g. perform two actions at the same time when in the domain it is only allowed to perform one action at a time, the entire LTL_f specification does not hold anymore, and as a result the agent has no valid strategy anymore in which it can fulfill all its intentions. Likewise, the agent only cares about reaching its goal whenever the environment plays according to the rules. Otherwise, it might be that the environment breaks the domain specification/rules, which makes it impossible for the agent to reach its goal.

If for the agent there is a precondition where an action can only be executed whenever the agent is in an environment state, but the environment breaks the domain specification by saying that the agent is currently not in any environment state, the agent is blocked from performing any actions, and will be unable to eventually fulfill all its intentions. As this is the case, the agent should only care about following the domain rules whenever the environment also follows the domain rules. Otherwise, the agent is blocked from executing any actions, where we can say that the environment 'cheated' in the game arena, for which the agent wins by default.

In order to extract such strategies in which both the agent and environment have to follow the rules of the given domain, we compose the following formula, using the individual components described in Section 4.5:

$$(\Box (\phi_{ag \text{ uniq } act})) \wedge ((\phi_{env \text{ init}} \wedge \Box (\phi_{env \text{ trans}})) \rightarrow ((\Box (\bigcirc true \rightarrow \phi_{ag \text{ act } prec})) \wedge \phi_{goal}))$$

We use this formula for extracting a DFA, using LTL_f synthesis. How a DFA can be obtained using an LTL_f specification is described in Section 2.8. The resulting DFA represents the FOND domain and intentions of an agent, from which it can be extracted what agent- and environment actions will result in which next states. Using this, agent strategies can be extracted which guarantee the agent to eventually fulfill all its intentions. We use this DFA for extracting the maximally permissive strategy and for mapping the winning region. Important here is to note that this is just one way for extracting good agent moves. There are possibly different manners for extracting these moves more efficiently. However, since exploring all manners for extracting good agent moves is not the goal of this research, we do not explore this in more detail.

To show in which cases the above formula holds, we write out a truth table. Since for writing out a truth table we only care about the semantics used in propositional logic, we leave out the LTL_f -specific semantics, which leaves us with the following simplified formula:

$$\phi_{ag\ uniq\ act} \wedge (\phi_{env\ trans} \rightarrow (\phi_{ag\ act\ prec} \wedge \phi_{goal}))$$

Since $\phi_{env\ init}$ and $\phi_{env\ trans}$ are both controlled by the environment, we join these in the simplified formula.

For the simplified formula, we attain the truth table displayed in Table 1.

$\phi_{ag\ uniq\ act}$	ϕ_{env}	$\phi_{ag\ act\ prec}$	ϕ_{goal}	$\phi_{ag\ prec} \wedge \phi_{goal}$	$\phi_{env} \rightarrow (\phi_{ag\ prec} \wedge \phi_{goal})$	$\phi_{ag\ uniq\ act} \wedge (\phi_{env} \rightarrow (\phi_{ag\ prec} \wedge \phi_{goal}))$
1	1	1	1	1	1	1
1	1	1	0	0	0	0
1	1	0	1	0	0	0
1	1	0	0	0	0	0
1	0	1	1	1	1	1
1	0	1	0	0	1	1
1	0	0	1	0	1	1
1	0	0	0	0	1	1
0	1	1	1	1	1	0
0	1	1	0	0	0	0
0	1	0	1	0	0	0
0	1	0	0	0	0	0
0	0	1	1	1	1	0
0	0	1	0	0	1	0
0	0	0	1	0	1	0
0	0	0	0	0	1	0

Table 1: Truth table for LTL_f specification of a domain

With this truth-table, there are five scenarios in which the agent wins:

1. The agent and environment both play according to the rules, and the goal is achievable for the agent. Agent picks unique action and follows precondition. Environment sets init correctly and follows specified transitions. Goal is achievable
2. The agent plays according to the rules, but the environment does not, but the goal is still achievable
3. The agent plays according to the rules, but the environment does not, and the goal is not achievable. However, because the environment does not play fair, we do not have to reach the goal anymore. Note that the subformula $\phi_{env} \rightarrow (\phi_{ag\ prec} \wedge \phi_{goal})$ says the following in natural language: “if the environment plays according to the specification, then the agent plays according to the preconditions and the agent reaches the goal.”. Since the environment does not play according to the specification in this scenario, the agent does not care about the goal anymore, and the complete formula becomes true.
4. The agent picks a unique action, but the environment does not follow the rules, the agent does not follow the preconditions to execute a specific action, and the goal is achievable. Since the

agent only has to follow preconditions when the environment plays according to the rules, the formula becomes true.

5. The agent picks a unique action, but the environment does not follow the rules, the agent does not follow the preconditions to execute a specific action, and the goal is not achievable. In the formula, achieving the goal is only of relevance whenever the environment plays according to the rules. Since the environment does not play according to the rules in this scenario, achieving the goal becomes irrelevant. Since the agent does pick a unique action, this makes the formula true.

4.7 LTL_f formula composition for extracting good environment moves

With the formula we composed in Section 4.6, we are only able to extract in each state what actions the agent can do in order to stay in the winning region, where the rules of the domain are followed by the agent. However, we cannot use this to extract proper reactions for the environment which follow the rules of the domain, preventing the environment from 'cheating' in the game arena, and letting the agent fulfill the LTL_f specification instantly. For this, we have to compose a new formula for the environment which, similar to the agent formula specification, ensures that the environment only picks reactions which are according to the domain rules/specification. For this, we compose the following formula, using the individual components described in Section 4.5:

$$\phi_{env\ init} \wedge (\Box (\phi_{ag\ uniq\ act} \wedge \phi_{ag\ act\ prec}) \rightarrow \Box (\phi_{env\ trans}))$$

Similar to the description in Section 4.6 for extracting good agent moves, also for extracting good environment moves it is true that this is just one way of retrieving the good environment moves. Since this is only meant for evaluating the IMS as a proof of concept, we do not go into more detail for alternative formula forms for extracting good environment moves.

In natural language, the above formula says the following: "The environment initializes correctly, and if the agent always picks a unique action and follows the action preconditions, then always the environment follows the correct transitions." By rewriting this formula by only extracting the semantics of propositional logic, we can simplify the formula as follows:

$$\phi_{env\ init} \wedge ((\phi_{ag\ uniq\ act} \wedge \phi_{ag\ act\ prec}) \rightarrow \phi_{env\ trans})$$

With this simplified formula, we attain the following truth table displayed in Table 2.

In the truth table above, we do not care about if the agent plays fair, as long as the environment (the protagonist) plays according to the specification ONLY in the scenarios where the agent also does so. However, independent on if the agent plays according to the specification, the environment always has to initialize according to the specification.

Using the above specification, a DFA can be extracted which can be used for extracting good environment strategies, where the environment chooses actions which are in alignment with the effects of the domain specification, whenever the agent also plays according to the rules. The maximally permissive strategy can also be extracted for the environment moves, showing what moves the environment can execute in order to comply to the domain specification/rules. We can use these environment reactions as a response to the actions of the agent which are extracted from the other DFA (for the agent) as described in Section 4.6. Using these two DFA's, we extract moves for the agent and the environment, which both follow the respective formula specifications such that we can simulate a run in which both players try their best to win the game and do not break the domain specification/rules.

$\phi_{env\ init}$	$\phi_{ag\ uniq\ act}$	$\phi_{ag\ prec}$	$\phi_{env\ trans}$	$\phi_{ag\ uniq\ act} \wedge \phi_{ag\ prec}$	$(\phi_{ag\ uniq\ act} \wedge \phi_{ag\ prec}) \rightarrow \phi_{env\ trans}$	$\phi_{env\ init} \wedge ((\phi_{ag\ uniq\ act} \wedge \phi_{ag\ prec}) \rightarrow \phi_{env\ trans})$
1	1	1	1	1	1	1
1	1	1	0	1	0	0
1	1	0	1	0	1	1
1	1	0	0	0	1	1
1	0	1	1	0	1	1
1	0	1	0	0	1	1
1	0	0	1	0	1	1
1	0	0	0	0	1	1
0	1	1	1	1	1	0
0	1	1	0	1	0	0
0	1	0	1	0	1	0
0	1	0	0	0	1	0
0	0	1	1	0	1	0
0	0	1	0	0	1	0
0	0	0	1	0	1	0
0	0	0	0	0	1	0

Table 2: Truth table for LTL_f specification of the environment playing according to the rules of a domain

4.8 Application of BDD's

As already seen in Section 4.2, within the design for the IMS, the IMS state contains several components which are represented as a BDD, also referred to as an ROBDD. As described in Section 2.10, a BDD is used for compactly representing propositional functions. Two or more BDD's can be conjoined whenever they contain all the same variables (e.g. X_1 , X_2 , and X_3), where essentially the function corresponding to the individual BDD's are conjoined into a new function. From this conjoined function, a new ROBDD can be extracted, representing the two conjoined BDD functions. This is especially helpful when we want to extract the moves which keep an agent in the winning region in a current DFA state, as we will use will use in our design.

We provide an example usage as follows:

- We start off with four ordered variables used in a BDD: $[X_1, X_2, X_3, X_4]$, where X_1 represents an agent action 'left', X_2 represents agent action 'right', X_3 represents domain state 's1', and X_4 represents domain state 's2'.

- When we would extract procrastinating moves from the maximally permissive strategy, this will tell us in each domain state, which actions the agent can execute in order to remain in the winning region. This could be resembled with the usage of cubes, as discussed in Section 2.10, as follows:

[1, 0, 0, 1]

[0, 1, 1, 0]

As we know the ordering of each variable within a BDD, and the truth valuations is indicated above, we can extract that in order to remain in the winning region, the agent has to execute action 'left' (X_1) when in domain state 's2' (X_4), and has to execute action 'right' (X_2) when

in domain state 's1' (X_3). As a function, this could be represented as $((X_1 \wedge \neg X_2 \wedge \neg X_3 \wedge X_4) \vee (\neg X_1 \wedge X_2 \wedge X_3 \wedge \neg X_4))$.

- If we now want to extract which moves to execute when the agent is in DFA state 's1', we combine the function for the BDD of the procrastinating strategy as described above with the function for the BDD of DFA state 's1', for which the cubes look as follows:
[2, 2, 1, 0]
where '2' represents a 'do not care'-value, and can take on both value 0 and 1. Note here that the BDD for DFA state 's1' only contains a single cube satisfying the condition. The function for this BDD looks as follows: $(X_3 \wedge \neg X_4)$. Note that in this function, variables x_1 and x_2 are irrelevant, as they are valued as 2, and can take on either value 0 or 1.
- We conjoin the function of the BDD for the procrastinating moves with the function for the BDD for DFA state 's1', resulting in the following function: $((X_1 \wedge \neg X_2 \wedge \neg X_3 \wedge X_4) \vee (\neg X_1 \wedge X_2 \wedge X_3 \wedge \neg X_4)) \wedge (X_3 \wedge \neg X_4)$. This formula can be simplified to: $(\neg X_1 \wedge X_2 \wedge X_3 \wedge \neg X_4)$, as this produces an equivalent truth table.
- After conjoining the function of both BDD's and extracting a new function, we can obtain the cubes from the corresponding BDD graph. This results in the following cube: [0, 1, 1, 0], as this is the only variable valuation satisfying the function $(\neg X_1 \wedge X_2 \wedge X_3 \wedge \neg X_4)$.
- The resulting cubes tell the agent what actions can be performed in the current DFA state, 's1', in order to remain in the winning region. From the cube we can extract that the agent can execute action 'right' (x_2) and not 'left' (x_1) in order to remain in the winning region.
- We have now successfully extracted an agent move in the current DFA state.

As strategies, winning regions, DFA states, and the valuations of agent- and environment variables can be easily represented using (RO)BDD's, we apply this in the design for the IMS. This helps the IMS with efficiently querying good agent moves, query if the current DFA state is either a final state or is in the winning region, and all other possible queries as specified in Section 3.5.

5 Implementation of the IMS

In this Section we describe the actual implementation of the IMS in a proof of concept, which is based on the definition and design for the IMS, as described in Section 3 and 4. We take the description for the usage of the IMS and software, described in Section 3.8, 5.1.1, and 5.1.2, respectively, and translate this into a usable proof of concept in the form of software⁴.

As the IMS does not operate by itself during a run, it requires an agent in order to have the query- and update operations to be called. By itself, the IMS can be considered as a class that solely provides various query- and update operations, while saving all necessary information for the IMS state during a run. The agent can use the implementation of these query- and update operations of the IMS during an actual run of a FOND domain and problem, as described in Section 5.2.

5.1 Software used for the implementation

The IMS as a proof of concept is built upon other software, which are called Lydia and Syftmax. Lydia is used for extracting a DFA from a given LTL_f formula and for checking realizability, and Syftmax is used for extracting the maximally permissive strategy. More detailed descriptions on Lydia and Syftmax are given in Section 5.1.1 and 5.1.2.

5.1.1 Lydia

- **What can the software do:**
 - Translate LTL_f/LDL_f to a DFA.
 - Do synthesis over an LTL_f/LDL_f formula (retrieve single winning strategy).
 - Check for realizability of an LTL_f/LDL_f formula.
- **Input:**
 - Specification of an LTL_f/LDL_f formula.
 - Specification of the environment (= inputs) and the agent (= outputs) variables.
- **Output:**
 - A DFA.
 - A single winning strategy.
 - Realizability for an LTL_f/LDL_f formula (Boolean value).

Lydia⁵ [8] can translate LTL_f and LDL_f formulas into a DFA, and is also able to synthesize a winning strategy. The input for Lydia can be either an LTL_f or an LDL_f formula. This formula is translated into a DFA, which is represented as a BDD. To perform synthesis of an LTL_f formula (retrieve a winning strategy), it also has to be specified which variables are of the agent and environment.

5.1.2 SyftMax

- **What can the software do:**

⁴Source code for the IMS software, with installation instructions: [31]

⁵Source for code of Lydia: [35]

- Synthesize the maximally permissive strategy (the set of all winning strategies). This consists of both the set of procrastinating moves, and non-procrastinating moves.
- **Possible input:**
 - Specification of an LTL_f/SDL_f formula.
 - Specification of the environment (= inputs) and the agent (= outputs) variables.
- **Output:**
 - The maximally permissive strategy, split up as the procrastinating- and the non-procrastinating strategy.

SyftMax⁶ [40] builds upon Lydia, and is able to synthesize the maximally permissive strategy. Just like Lydia, the input for Syftmax can be either an LTL_f or an SDL_f formula, where it is also specified which variables are of the agent and which are of the environment.

5.2 Usage of the IMS by an agent

Now that we know the requirements for the IMS, what the operational environment is, what information goes in and out of the IMS during a run, and how the IMS can keep track of progression for intentions, we can describe how the IMS is used by an agent during runtime. This is implemented in an agent simulator which can be customized in various ways, which is displayed in Algorithm 1.

We will stepwise go through this algorithm to explain what happens in each step:

1. A FOND domain and partition file are provided externally, where the domain is provided as an LTL_f formula. Here, the partition file indicates what are the agent- and environment variables. The FOND domain is saved in variable ϕ_D , and the partition variables in p .
2. An initial set of intentions, expressed as LTL_f formulas, is provided by an agent, which is saved in variable L .
3. In method *'composeAgentFormula(ϕ_D, L)'*, ϕ_D and L (the FOND domain and initial list of intentions) are joined into a single formula *' ϕ_f '*, where the formula is composed as specified in Section 4.6.
4. The resulting formula ϕ_f and partition file p are given as input to Lydia (5.1.1), which returns a DFA and is saved in variable DFA.
5. Using Lydia, we check if DFA is realizable.
6. If DFA is indeed realizable, we initialize the IMS state I^+ , and save variables ϕ_D , L , and DFA in I^+ .
7. We can now extract the maximally permissive strategy for the agent using Syftmax (5.1.2), given ϕ_f and p . This returns both the set of procrastinating- and non-procrastinating strategies. The sets of procrastinating- and non-procrastinating strategies are then saved in the IMS state.
8. From the DFA, the initial DFA state is extracted. This is set as the current DFA state: $I^+.s_{DFA}$.
9. The domain is now initialized, and the main loop of the simulation starts. The loop does not stop, until the agent calls the update operation $I^+.halt$.

⁶Source for code of Syftmax: [39]

10. The set of winning moves in the current DFA state is requested using query $I^+.getWinningActions()$, where the current DFA state $I^+.s_{DFA}$ and the entire set of procrastinating strategies $I^+.ps$ are given as input. The BDD for the current DFA state and the BDD of procrastinating strategies are conjoined, as explained in Section 4.8, returning only the set of procrastinating strategies which are possible in the current DFA state.
11. The same is done for requesting the non-procrastinating moves, calling query $I^+.getProgressingActions()$, giving the current DFA state $I^+.s_{DFA}$ and the entire set of non-procrastinating strategies $I^+.nps$ as input, where the BDD's for these are conjoined, as explained in Section 4.8, and return the set of non-procrastinating strategies in the current DFA state.
12. The agent chooses an action, where in our design this can be selected from either the procrastinating- or non-procrastinating actions in the current DFA state.
13. Update operation $I^+.do(a)$ is called, where the chosen agent action a , the entire DFA $I^+.DFA$, and the current DFA state $I^+.s_{DFA}$ are used. The environment chooses a reaction, given the agent action. The chosen agent- and environment actions form a new domain state, $I^+.s$, which is internally used for transitioning the DFA into a new state. For this transition, the transition function of $I^+.DFA$ is applied to $I^+.s$ in the current DFA state $I^+.s_{DFA}$. This alters $I^+.s$ and $I^+.s_{DFA}$, which are updated in the IMS state. As already mentioned in the description for the update operation $I^+.do(a)$ of Section 3.6 and 4.3, the intentions from intention list $I^+.L$ are progressed and updated internally.
14. The simulator checks if the agent wants to perform an update operation. If so, we enter the switch-statement. If not, the switch-statement is skipped.
15. If the agent wants to perform an update operation, read which operation this is.
16. Enter case ' $I^+.adopt(\phi, k)$ ', which is for attempting to add a new intention, ϕ , to the intention list at position k .
17. $I^+.adopt(\phi, k)$ is executed. If this returns *True*, new intention ϕ is adopted at priority k and $\phi_{adopted}$ is set to *True*. As $I^+.L$ is updated when a new intention is adopted, $I^+.DFA$, $I^+.s_{DFA}$, $I^+.w$, $I^+.ps$, and $I^+.nps$ are recomputed and replaced in the IMS state, as these have changed as a result of change in $I^+.L$. Syftmax is called internally for recomputing $I^+.ps$ and $I^+.nps$.
18. If in the previous step $\phi_{adopted}$ was set to *False*, we enter an if-statement.
19. $I^+.isRealizable(\phi, k)$ is called, which will return a subset of realizable intentions. The set difference is taken of $I^+.L$ and the realizable subset returned by $I^+.isRealizable(\phi, k)$, resulting in the set of unrealizable intentions, i.e. the set of intentions which would have to be dropped in order to add new intention ϕ to $I^+.L$.
20. Feedback is given to the agent on which set of intentions needs to be dropped in order to add new intention ϕ to $I^+.L$.
21. Enter case ' $I^+.drop(k)$ ', where the agent wants to drop the intention at priority k from the intention list, $I^+.L$.
22. $I^+.drop(k)$ is executed, removing the intention at priority k from $I^+.L$. After dropping this intention, $I^+.L$, $I^+.DFA$, $I^+.s_{DFA}$, $I^+.w$, $I^+.ps$, and $I^+.nps$ are recomputed and updated in the

IMS state, as these have changed as a result of change in $I^+.L$. Syftmax is called internally for recomputing $I^+.ps$ and $I^+.nps$. Note that here we do not have to check for realizability of the DFA. Since $I^+.L$ was already realizable in the previous step, any subset of $I^+.L$ will also be realizable.

23. Enter case ' $I^+.halt$ '
24. Update operation ' $I^+.halt$ ' is called, and the system's operation is terminated.
25. We enter the case in which the agent does not give a correct update operation.
26. Continue the loop.
27. Query operation $I^+.isInFinalState()$ is called, which checks if the current DFA state $I^+.s_{DFA}$ is in $I^+.w$, and if $I^+.L$ is empty. If this is the case, the query returns *True*. Otherwise, returns *False*.
28. Feedback is given if the current state is indeed a final state.
29. If the agent is in a final state, the agent can decide to call update operation $I^+.halt$.
30. If agent calls update operation $I^+.halt$, the system's operation is terminated.
31. If the initial DFA in the beginning of the program is not realizable, the main loop for the program is not started. An else-statement is entered.
32. Feedback is provided to the agent, indicating that the initial DFA is not realizable.

As can be noticed, this proposed usage of the IMS by an agent does not include all query operations as specified in Section 3.5 and 4.3. This method of usage is to indicate in what manner our design for the IMS could be used by an agent, for which this happens to also be our manner for implementing the design for the IMS as software. There are several manners for implementing the IMS in software for an agent to use, although in this research we will only cover our way of implementing.

Now that we have a notion of how the IMS can be used by an agent in order to fulfill all the intentions an agent might have, we will evaluate the implementation of the IMS on some actual FOND domains and problems in Section 6.

Algorithm 1: Usage of the IMS by an agent simulator

```

1  $\phi_D, p \leftarrow$  FOND domain and partition file are provided externally, where the domain is
   expressed as an LTLf formula;
2  $L \leftarrow$  set the current intention list as the initial list of intentions given by an agent;
3  $\phi_f \leftarrow \text{composeAgentFormula}(\phi_D, L, p)$ ;
4 DFA  $\leftarrow$  Lydia.getDFA( $\phi_f, p$ );
5 if Lydia.getRealizability(DFA) == true then
6    $I^+ \leftarrow \text{initializeIMSState}(\phi_D, L, \text{DFA})$ ;
7    $I^+.ps, I^+.mps \leftarrow \text{Syftmax.getMaximallyPermissiveStrategy}(\phi_f, p)$ ;
8    $I^+.s_{\text{DFA}} \leftarrow I^+.\text{DFA.getInitialState}()$ ;
9   while true do
10     $ps_{\text{curr}} \leftarrow I^+.\text{getWinningActions}()$ ;
11     $nps_{\text{curr}} \leftarrow I^+.\text{getprogressingActions}()$ ;
12     $a \leftarrow \text{pickAgentMove}(ps_{\text{curr}}, nps_{\text{curr}})$ ;
13     $I^+.s, I^+.L, I^+.s_{\text{DFA}} \leftarrow I^+.\text{do}(a)$ ;
14    while readUpdateOperation() != empty do
15      switch readUpdateOperation() do
16        case  $I^+.\text{adopt}(\phi, k)$  do
17           $\phi_{\text{adopted}}, I^+.L, I^+.\text{DFA}, I^+.s_{\text{DFA}}, I^+.w, I^+.ps, I^+.nps \leftarrow I^+.\text{adopt}(\phi, k)$ ;
18          if  $\phi_{\text{adopted}} == \text{False}$  then
19             $L_{\text{unrealizable}} \leftarrow (I^+.L \setminus I^+.\text{isRealizable}(\phi, k))$ ;
20            print("drop these to make new intention realizable: ",  $L_{\text{unrealizable}}$ );
21          case  $I^+.\text{drop}(k)$  do
22             $I^+.L, I^+.\text{DFA}, I^+.s_{\text{DFA}}, I^+.w, I^+.ps, I^+.nps \leftarrow I^+.\text{drop}(k)$ ;
23          case  $I^+.\text{halt}$  do
24             $I^+.\text{halt}$ ;
25          otherwise do
26            continue;
27        if  $I^+.\text{isInFinalState}()$  then
28          print("final state reached. stop program?");
29          if readAgentUpdate() == halt then
30             $I^+.\text{halt}$ ;
31 else
32   print("not realizable");

```

6 Evaluation of the IMS implementation

In this Section, we evaluate the actual implementation of the IMS, described in Section 5, on a set of benchmark FOND domains and problems. For each domain, it is described if the IMS operates according to the IMS specification, and to which extend the IMS is able to handle the FOND domains when specified in LTL_f .

6.1 Translation of FOND domains from PDDL into LTL_f

Initially, a subgoal for this research was to directly translate a FOND domain, expressed in PDDL, into LTL_f . This has proven to be harder than expected, as PDDL is expressed in a different manner as opposed to LTL_f . In Section 2.5 it is already mentioned how in PDDL the frame axioms, i.e. what remains the same after an action is executed, are not specified. This deviates from expressions in LTL_f , as here both the effects and frame axioms have to be expressed. Besides of expressing frame axioms, PDDL also differs from LTL_f as to where PDDL is a relational language, but it is assumed that the object domain is finite, for which the predicates and operators can be grounded in order to obtain a propositional expression. On the contrary, LTL_f is already propositional by definition.

In [11] it is explained how LTL_f expressions can be translated into first-order logic. Although PDDL is not fully based upon first-order logic, PDDL and first-order logic are both relational languages, for which some of this knowledge can be applied when translating between LTL_f and PDDL. In [17], a more concrete method is provided for translating among LTL_f and PDDL.

Since the frame axioms are not included in PDDL, and the grounding of predicates and operators is not a direct translation from PDDL into LTL_f , translating PDDL directly into LTL_f appears to be a greater challenge than anticipated. Several approaches and softwares have been investigated in order to directly translate PDDL into LTL_f , however this has proven to be more challenging than expected. Considering the scope and duration of this research, a decision has been made to generate the individual LTL_f domain specifications using Python scripts⁷, which we developed ourselves. These scripts are made to be adjustable, such that domains can be easily adjusted in size. The scripts have been written for generating the Triangle-Tireworld and slippery world domain, as these are too labour intensive and repetitive to write out by hand. The decision tree domain is compact enough to write out by hand, so no script for generating this domain has been written.

6.1.1 Comparison of specifying a FOND domain in PDDL and LTL_f

As already mentioned in Section 2.5, PDDL is expressed as a domain file using relational language, and a problem file as an object domain, where expressions are not grounded yet, i.e. no rules/effects have been applied directly to any propositions yet. For example, if we take a relational expression, possibly expressed in PDDL, $move(x_1, x_2)$ where we have the object constants *Amsterdam* and *Rome*, we can ground formula $move(x_1, x_2)$ using the given object constants. This results in $move(Amsterdam, Rome)$, and $move(Rome, Amsterdam)$. This deviates from LTL_f , where LTL_f already applies all rules/effects to all propositions, resulting in a grounded expression from the start. Since PDDL is not written as a grounded expression, it can be represented significantly more compact as opposed to an equivalent LTL_f expression.

In the problem-file for PDDL, the agent- and environment variables, initial state, and goal state are defined which, as can be seen in the example in Section 6.9.2, is also rather compact compared to LTL_f expressions. Since for PDDL the domain always remains the same throughout runs, only the problem file needs to be altered in order to change a problem scenario.

⁷Source of Python scripts for domain generation: [31]

As LTL_f is based on propositional logic, the rules/effects of a domain are directly applied to all agent- and environment variables, i.e. atomic propositions. Because of this, whenever a domain is scaled up in terms of atomic propositions, the LTL_f specification becomes significantly larger compared to an equivalent FOND domain which is expressed in PDDL.

When comparing the PDDL and LTL_f specification written out in Section 6.9.2 and 6.9.3, it can be seen that the LTL_f expression is significantly larger. However, as Lydia and Syftmax are not able to process PDDL specifications, this research is limited to using LTL_f specifications for expressing domains and problem scenarios.

6.2 Benchmark domains and problems

As described in Section 6.1, converting a PDDL specification of a FOND domain into an LTL_f specification brings certain complications. As each agent- and environment variable has to be grounded when converting from PDDL into LTL_f , the LTL_f specification becomes significantly larger. Besides this, also the frame axiom is not included within a PDDL specification, as we have already described in Section 2.5.

Initially, we planned on testing the IMS on several benchmark FOND planning domains used in related research [38, 29] and intention progression competitions⁸, for which the domains are provided in PDDL. However as already mentioned, during our research we have discovered that translating PDDL into LTL_f comes with some complications. Our plan was to test the IMS on the Triangle-Tireworld domain [38, 29], the Miconic-N domain, and the Logistics domain, as are used in the International Intention Progression Competition. After the Triangle-Tireworld already caused difficulty for translating into LTL_f and using this within our software, we left the other FOND domains expressed in PDDL for usage in potential future work, and for now developed FOND domains which could be expressed in LTL_f more compactly.

For one of the most well-known FOND domain, the Triangle-Tireworld, only a handwritten log is given by the IMS, as the LTL_f specification for this became too long to be processed by Lydia and Syftmax. After translating the PDDL specification for the Triangle-Tireworld into LTL_f , it quickly became clear that this is not an efficient domain for testing in LTL_f , as grounding the PDDL specification to be expressed as propositional logic increases the size of the expression to a degree in which it is not computationally efficient anymore for evaluating the IMS as a proof of concept, as the proof of concept is not able to process the resulting LTL_f formula anymore, given our computational resources. In Section 6.3 a specification is given for which hard- and software has been used for developing and running the IMS as a proof of concept.

After encountering the complications for translating the Triangle-Tireworld domain into LTL_f and running this on our hard- and software, we have decided to not translate and use any other benchmark FOND domains from PDDL into LTL_f , as these would most likely result in similar complications as we have encountered with the Triangle-Tireworld domain. We have been developing two new types of FOND domains, called the decision tree domain (4.5.1) and the slippery world domain (6.8.1), which can be expressed rather compactly in LTL_f . These domains deal to some degree with compressing the length of an LTL_f specification.

In more detail regarding the slippery world domain, an agent is placed by the environment in a singular row and column, for which the domain consists out of several rows and columns. By applying mutual exclusion on the rows and columns, there is no need for a separate environment variable for each individual location. In the scenario where there would be four rows and four columns, normally this would be expressed by having 16 individual locations, as is done in the Triangle-Tireworld domain. However, if we use a combination of a single row and a single column, we get

⁸Website for the Intention Progression Competition (IPC): [5]

a grid which is expressed as a cartesian product, in which we can specify coordinates for the agent location. Instead of having 16 individual location variables, we can now express the locations by only having eight environment variables in total, which in turn compresses the LTL_f specification significantly. This has proven to work rather well for expressing a domain in LTL_f , for which it has been included as a domain for evaluating the IMS.

Secondly, the decision tree domain uses the most intuitive and basic characteristics possible within a FOND domain, for which it can be expressed rather compact as an LTL_f expression, as shown in Section 4.5.2. This also makes for an easily explainable domain for testing the IMS on, for which this can be considered as a baseline FOND domain for evaluating the IMS as a proof of concept.

Both the decision tree domain and the slippery world domain have been tested for usage within the IMS as a proof of concept, obtaining a DFA and maximally permissive strategy by Lydia and Syftmax, for which these domains were able to be used without any complications. For this reason, we have decided to include these FOND domains for evaluating the IMS, as it could then be demonstrated that the IMS operates as required in practice.

In summary, for evaluating the IMS as a proof of concept, the following FOND domains have been used: 1). decision tree domain (4.5.1), 2). slippery world domain (6.8.1), and 3). Triangle-Tireworld domain (6.9.1). Each domain provides a source with a detailed description. These domains have been chosen since they are either easy to comprehend, or are well known within the field of planning, as the goal for this research is to demonstrate how an IMS can maintain consistency among a changing set of temporally extended goals within a FOND domain.

6.3 Hard- and software specification for developing and running the IMS

The hard- and software specifications in which the IMS has been developed and ran is as follows:

- Operating system + type: Ubuntu 20.04.3 LTS, 64-bit (ran in Oracle VM VirtualBox)
- RAM: 10.5 GB
- Processor: AMD Ryzen 7 5800h with radeon graphics x 8
- Graphics: llvmpipe (LLVM 15.07, 128 bits)
- Disk capacity: 60.5 GB

6.4 Design of benchmark problems

To evaluate our proof of concept, the problems provided in PDDL-format would have to be altered. The original PDDL-problems only provide an initial set of intentions, which means it does not add new intentions to the agent's list of intentions as it progresses.

In our research, the agent should be able to adopt and drop intentions over time, which is why we cannot directly use the PDDL domain- and problem specifications as they are provided for usage in planning competitions. As mentioned in Section 6.2, the translation of PDDL specifications into LTL_f comes with several complications, which is why we have chosen to use a FOND domain which can be expressed more compactly in LTL_f . To demonstrate the extend to which the IMS can operate, several temporally extended goals are added during the problem scenario, demonstrating to which extend the IMS is able to handle the adding and dropping of intentions.

As input, the proof of concept system will be provided with an LTL_f specification of a FOND domain, for which intentions can be added and dropped by the agent during a run. The intentions

used in the scenarios are represented as LTL_f formulas. Progression of intentions is demonstrated in the simulations, where the intentions can be expressed as any type of LTL_f expression.

Lastly, to demonstrate the IMS functions exactly as intended, we give a simulation run for the Triangle-Tireworld domain in Section A, which corresponds exactly to the initial example problem described in Section 3.1 which the design for the IMS is based on.

6.5 Development of benchmark domains and problems

As already mentioned in the introduction of Section 6, Python scripts have been written in order to generate the FOND domains as LTL_f specifications, for which the domains can be altered to be scaled up or down. As the problem scenarios change during a run, not all which the IMS needs to be tested on can be specified in the LTL_f specification before a run starts. During a run, the agent can add and drop intentions, by which the initial LTL_f specification is altered. The initial LTL_f specification is only used for starting the run.

Since the problem scenarios which the IMS should be able to handle cannot be fully specified in LTL_f before the run starts, this will be handled after a run starts. The agent can add and drop intentions, for which the intention can be any LTL_f specification. In these problem scenarios, the IMS will be evaluated on all query- and update operations, as is specified in Section 3.5 and 3.6.

6.6 Simulation of a run

A simulation can be ran either automatically or manually. Whenever a simulation is ran automatically, actions and reactions are arbitrarily picked by the agent and environment. In a manual simulation, at each time step a set of actions and reactions is given, where a user can select what action and reaction should be played. In this way, specific scenarios can be executed in order to cover edge cases, which is what we have used for evaluating the IMS as a proof of concept.

6.7 Decision tree domain

We will evaluate the IMS using a simulation of the decision tree domain, for which an explanation has been provided earlier in Section 4.5.1. For simulating the decision tree domain, we compose a formula for extracting good agent moves, described in Section 4.6, and a formula for extracting good environment moves, as described in Section 4.7. These formulas are composed using the components described in Section 4.5.2. In this context, we will use a decision tree with three layers, which results in a total of six states, as visually displayed in Figure 7. The decision tree domain is meant as a simple example of a FOND domain, which can easily show the basic functionality of the IMS as a proof of concept.

6.7.1 Representation of the decision tree domain in LTL_f

In Section 4.5.2, a specification for the decision tree domain in LTL_f has already been provided. We will use this specification for simulating a scenario in the decision tree domain, which is described in more detail in Section 6.7.2.

6.7.2 Simulation of the decision tree domain

Here we see the log output when the IMS is operating in the decision tree domain. In this run the IMS demonstrates the queries for dropping and adopting, where also the scenario is shown in which

a new intention is not realizable in conjunction with the current intention list. The output below is directly copied from the IMS terminal output, where the blue text is added to indicate what is happening during each step in the program, and what is user input.

Log of the terminal output:

(an initial list of intentions is provided, for which realizabilty is checked)

```
initial intentions:
F(s5)
```

```
formula agent: (G((l | r) & !(l & r))) & (((s1) & G(((s1 | s2 |
s3 | s4 | s5 | s6) & !(s1 & s2) & !(s1 & s3) & !(s1 & s4) & !(s1 & s5)
& !(s1 & s6) & !(s2 & s3) & !(s2 & s4) & !(s2 & s5) & !(s2 & s6) & !(s3
& s4) & !(s3 & s5) & !(s3 & s6) & !(s4 & s5) & !(s4 & s6) & !(s5 & s6)
& (s1 -> X(l -> (s2 | s3))) & (s1 -> X(r -> (s2 | s3))) & (s2 -> X(l
-> s4)) & (s2 -> X(r -> s5)) & (s3 -> X(l -> s5)) & (s3 -> X(r -> s6))
& (s4 -> X((l | r) -> s4)) & (s5 -> X((l | r) -> s5)) & (s6 -> X((l
| r) -> s6)))) -> ((G((tt))) & (F(s5) )))
```

```
formula environment: ((s1) & ((G((l | r) & !(l & r))) -> (G(((s1
| s2 | s3 | s4 | s5 | s6) & !(s1 & s2) & !(s1 & s3) & !(s1 & s4) & !(s1
& s5) & !(s1 & s6) & !(s2 & s3) & !(s2 & s4) & !(s2 & s5) & !(s2 & s6)
& !(s3 & s4) & !(s3 & s5) & !(s3 & s6) & !(s4 & s5) & !(s4 & s6) & !(s5
& s6) & (s1 -> X(l -> (s2 | s3))) & (s1 -> X(r -> (s2 | s3))) & (s2
-> X(l -> s4)) & (s2 -> X(r -> s5)) & (s3 -> X(l -> s5)) & (s3 -> X(r
-> s6)) & (s4 -> X((l | r) -> s4)) & (s5 -> X((l | r) -> s5)) & (s6
-> X((l | r) -> s6))))))
```

The environment formula is realizable

The agent formula is realizable

----- NEW LOOP STARTS HERE! -----

(agent chooses if it wants to retrieve the procrastinating- or non-procrastinating moves for the current state)

```
get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 2
```

(this is a 'dummy' move for the agent which has no effect, as the agent starts, and the environment reacts. However, in the first move the environment needs to initialize, since otherwise there is no effect for the agent action)

```
possible agent moves in current state:
```

```
r
l
```

```
pick an action, using index (nr. from 1 to 2): 1
```

```
move selected: r
```

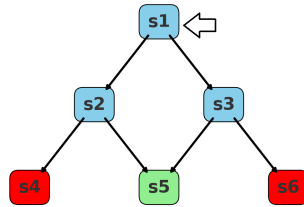
(only one option is given here for the environment: the initial environment value as written in the LTL_f specification)

possible environment moves in current state:
s1

random environment move picked: s1

current env vars set to true:
s1

(the agent is now located in environment state s1)



progressed intentions:
before: Fs5 -> after: Fs5

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index 2)? [1 or 2]: 2

(from here on, the first effective move is executed by the agent)

possible agent moves in current state:
r
l

pick an action, using index (nr. from 1 to 2): 2
move selected: l

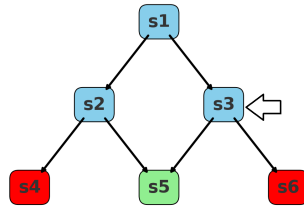
(the environment can pick either of two non-deterministic effects)

possible environment moves in current state:
s3
s2

random environment move picked: s3

current env vars set to true:
s3

(the agent is now located in environment state s3)



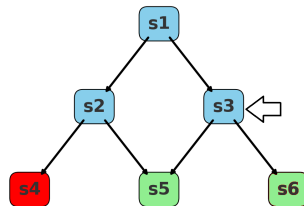
progressed intentions:
 before: Fs5 -> after: Fs5

call update operation? [y/n]: y

would you like to add an intention [1], drop [2] an intention, or
 stop the entire system [3]?: 1

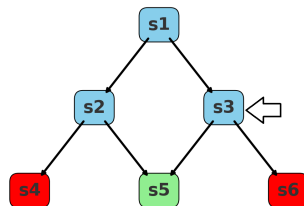
give new intention: F(s6)
 set priority using an index (position in the intention list. 0 is
 highest priority): 0
 trying to add new intention 'F(s6)' to intention list:

(the agent tries to add intention 'F(s6)' to the intention list. However, as the agent moves down
 the tree and cannot eventually reach both states, this is not realizable)



new goal: Fs5 & F(s6)
 adding new intention is not realizable.

(adding 'F(s6)' is currently not realizable, and the intention list remains as it was)



would you like to revise the current intentions in order to add the
 new intention? [y/n]: y

```
binary intention list dropout: 00
the following intentions are tested for realizability: F(s6) & Fs5
The formula is not realizable with these intentions.
```

```
binary intention list dropout: 01
the following intentions are tested for realizability: F(s6)
Formula is realizable if the following intentions are dropped:
Fs5
call update operation? [y/n]: y
```

```
would you like to add an intention [1], drop [2] an intention, or
stop the entire system [3]?: 2
```

```
which intention should be dropped from the list?: (index 1 until
1)
```

```
options:
```

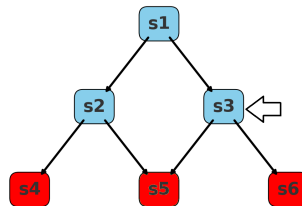
```
1 : Fs5
```

```
index chosen: 1
```

```
binary intention list dropout:
```

```
new formula agent: (G((l | r) & !(l & r))) & (((s3) & G(((s1 | s2
| s3 | s4 | s5 | s6) & !(s1 & s2) & !(s1 & s3) & !(s1 & s4) & !(s1 &
s5) & !(s1 & s6) & !(s2 & s3) & !(s2 & s4) & !(s2 & s5) & !(s2 & s6)
& !(s3 & s4) & !(s3 & s5) & !(s3 & s6) & !(s4 & s5) & !(s4 & s6) & !(s5
& s6) & (s1 -> X(l -> (s2 | s3))) & (s1 -> X(r -> (s2 | s3))) & (s2
-> X(l -> s4)) & (s2 -> X(r -> s5)) & (s3 -> X(l -> s5)) & (s3 -> X(r
-> s6)) & (s4 -> X((l | r) -> s4)) & (s5 -> X((l | r) -> s5)) & (s6
-> X((l | r) -> s6)))) -> ((G((tt))) & (tt)))
```

(intention 'F(s5)' is dropped, resulting in an empty intention list)



```
call update operation? [y/n]: y
```

```
would you like to add an intention [1], drop [2] an intention, or
stop the entire system [3]?: 1
```

```
give new intention: F(s6)
```

```
set priority using an index (position in the intention list. 0 is
highest priority): 0
```

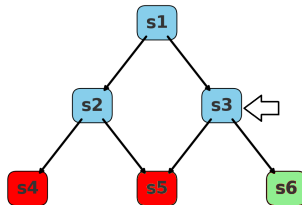
```
trying to add new intention 'F(s6)' to intention list:
```

```

new goal: F(s6)
adding new intention is realizable! new intention is adopted and
DFA is recomputed
new formula agent: (G((l | r) & !(l & r))) & (((s3) & G(((s1 | s2
| s3 | s4 | s5 | s6) & !(s1 & s2) & !(s1 & s3) & !(s1 & s4) & !(s1 &
s5) & !(s1 & s6) & !(s2 & s3) & !(s2 & s4) & !(s2 & s5) & !(s2 & s6)
& !(s3 & s4) & !(s3 & s5) & !(s3 & s6) & !(s4 & s5) & !(s4 & s6) & !(s5
& s6) & (s1 -> X(l -> (s2 | s3))) & (s1 -> X(r -> (s2 | s3))) & (s2
-> X(l -> s4)) & (s2 -> X(r -> s5)) & (s3 -> X(l -> s5)) & (s3 -> X(r
-> s6)) & (s4 -> X((l | r) -> s4)) & (s5 -> X((l | r) -> s5)) & (s6
-> X((l | r) -> s6)))) -> ((G((tt))) & (F(s6))))

```

(intention 'F(s6)' has been added to the intention list)



```

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 2

possible agent moves in current state:
r

pick an action, using index (nr. from 1 to 1): 1
move selected: r

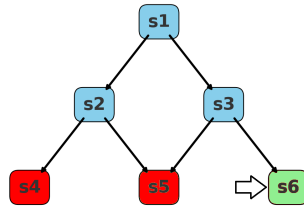
possible environment moves in current state:
s6

random environment move picked: s6

current env vars set to true:
s6

(the agent is now located in environment state s6: a final state of the DFA. All intentions of the
intention list have been fulfilled)
progressed intentions:
before: Fs6 -> after: 1

```



intention 'Fs6' has been fulfilled and is removed from the intention list

final state reached!

(agent calls update operation 'halt' to stop the program)

stop program? [y/n]: y

program stopped

Process finished with exit code 0

6.7.3 Evaluation of IMS in the decision tree domain

In Section 6.7.2 we display the log output for a run in the decision tree domain. Here, we test to which extent the IMS is able to maintain consistency in terms of realizability of intentions, where priority of the intentions and recalculation of the maximally permissive strategy is taken into account by the IMS. Important here is to note that it does not matter in this scenario if the agent chooses either the procrastinating- or the non-procrastinating actions, as in this domain these sets of actions are equivalent, as it is not possible to procrastinate. This evaluation is a basic demonstration of how the IMS operates during a run, using the algorithm described in Algorithm 1. In this domain, no complicated LTL_f intentions are provided yet. However, all query- and update operations as described in Section 3.5 and 3.6 have been used in this problem scenario.

6.8 Slippery world domain

In Section 6.8.1 a description is provided of the 'slippery world' FOND domain. As already mentioned in Section 6.2, this domain is designed by ourselves, which is used to express a FOND domain in LTL_f as compact as possible. The domain is easily scalable, without causing the LTL_f formula to increase significantly in size, as is the case when the Triangle-Tireworld is expressed in LTL_f . As opposed to the decision tree domain and the Triangle-Tireworld domain, in the slippery world domain an agent can always go back to a previous state, for which we can demonstrate the adding and dropping of temporally extended goals by the IMS in more detail. The individual components used for composing an LTL_f formula for the slippery world domain are shown in Section 6.8.2.

6.8.1 Description of the slippery world domain

In Figure 8 we see a visual representation of the slippery world domain: a domain which is designed by ourselves. In this Figure we see a grid, consisting of several rows and columns, where in this example we have a total of four rows and four columns. The agent is restricted to performing the actions of going up, down, left, or right. The environment is restricted by having mutual exclusion

on rows and columns, meaning that the environment can only place the agent in one row and one column at a time. Whenever the agent performs an action, e.g. 'right', in environment state 'r1' and 'c1', the environment can react with a non-deterministic effect by either moving the agent one, or two positions in the direction indicated by the agent. This would result in the environment placing the agent in either environment state 'r1' and 'c2', or environment state 'r1' and 'c3'. This works similar for all other directions, where if the agent would perform action 'down' in environment state 'r1' and 'c1', the environment reacts by placing the agent in either 'r2' and 'c1', or in 'r3' and 'c1'.

	c1	c2	c3	c4
r1				
r2				
r3				
r4				

Figure 8: Slippery world domain

6.8.2 Representation of the slippery world domain in LTL_f

For describing the slippery world domain in LTL_f , we will break up the domain into components, as described in Section 4.5:

Agent- and environment variables:

- agent variables : $\{l, r, u, d\}$
- environment variables : $\{r1, r2, r3, r4, c1, c2, c3, c4\}$

Where agent variables l , r , u , and d imply action 'left', 'right', 'up', and 'down', respectively. For the environment variables, $r1$ until $r4$ specifies the row variables, while $c1$ until $c4$ specifies the column variables.

Domain specification:

- $\phi_{ag\ uniq\ act} : (l \vee r \vee u \vee d) \wedge \neg(l \wedge r) \wedge \neg(l \wedge u) \wedge \neg(l \wedge d) \wedge \neg(r \wedge u) \wedge \neg(r \wedge d) \wedge \neg(u \wedge d)$
- $\phi_{env\ trans} :$
 - mutual exclusion of environment variables :

$$((r1 \vee r2 \vee r3 \vee r4) \wedge \neg(r1 \wedge r2) \wedge \neg(r1 \wedge r3) \wedge \neg(r1 \wedge r4) \wedge \neg(r2 \wedge r3) \wedge \neg(r2 \wedge r4) \wedge \neg(r3 \wedge r4)) \wedge$$

$$((c1 \vee c2 \vee c3 \vee c4) \wedge \neg(c1 \wedge c2) \wedge \neg(c1 \wedge c3) \wedge \neg(c1 \wedge c4) \wedge \neg(c2 \wedge c3) \wedge \neg(c2 \wedge c4) \wedge \neg(c3 \wedge c4))$$

- environment reactions :
 $(c2 \rightarrow \bigcirc(l \rightarrow c1)) \wedge (c3 \rightarrow \bigcirc(l \rightarrow (c2 \vee c1))) \wedge (c4 \rightarrow \bigcirc(l \rightarrow (c3 \vee c2))) \wedge (c1 \rightarrow \bigcirc(r \rightarrow (c2 \vee c3))) \wedge (c2 \rightarrow \bigcirc(r \rightarrow (c3 \vee c4))) \wedge (c3 \rightarrow \bigcirc(r \rightarrow c4)) \wedge (r2 \rightarrow \bigcirc(u \rightarrow r1)) \wedge (r3 \rightarrow \bigcirc(u \rightarrow (r2 \vee r1))) \wedge (r4 \rightarrow \bigcirc(u \rightarrow (r3 \vee r2))) \wedge (r1 \rightarrow \bigcirc(d \rightarrow (r2 \vee r3))) \wedge (r2 \rightarrow \bigcirc(d \rightarrow (r3 \vee r4))) \wedge (r3 \rightarrow \bigcirc(d \rightarrow r4))$
- frame :
 $(c1 \rightarrow \bigcirc(u \rightarrow c1)) \wedge (c1 \rightarrow \bigcirc(d \rightarrow c1)) \wedge (c2 \rightarrow \bigcirc(u \rightarrow c2)) \wedge (c2 \rightarrow \bigcirc(d \rightarrow c2)) \wedge (c3 \rightarrow \bigcirc(u \rightarrow c3)) \wedge (c3 \rightarrow \bigcirc(d \rightarrow c3)) \wedge (c4 \rightarrow \bigcirc(u \rightarrow c4)) \wedge (c4 \rightarrow \bigcirc(d \rightarrow c4)) \wedge (r1 \rightarrow \bigcirc(l \rightarrow r1)) \wedge (r1 \rightarrow \bigcirc(r \rightarrow r1)) \wedge (r2 \rightarrow \bigcirc(l \rightarrow r2)) \wedge (r2 \rightarrow \bigcirc(r \rightarrow r2)) \wedge (r3 \rightarrow \bigcirc(l \rightarrow r3)) \wedge (r3 \rightarrow \bigcirc(r \rightarrow r3)) \wedge (r4 \rightarrow \bigcirc(l \rightarrow r4)) \wedge (r4 \rightarrow \bigcirc(r \rightarrow r4))$

- $\phi_{ag \text{ act } prec} : (c1 \rightarrow \bigcirc(\neg l)) \wedge (c4 \rightarrow \bigcirc(\neg r)) \wedge (r1 \rightarrow \bigcirc(\neg u)) \wedge (r4 \rightarrow \bigcirc(\neg d))$

Problem specification:

- $\phi_{env \text{ init}} : r1 \wedge c1$
- $\phi_{goal} : \diamond(r4 \wedge c4)$

6.8.3 Simulation of the slippery world domain (scenario 1)

For this simulation, the log outputs have been taken directly from the software for the IMS, where the blue text is added to indicate what is happening during each step in the program, and what is user input. In this simulation it is demonstrated how the IMS handles cases where a new intention can currently not be added, where the intention list is revised for checking which intentions should be dropped to add this new intention. Besides this, it is demonstrated how the IMS handles the dropping of intentions, and how intentions are progressed during a run.

Log of the terminal output:

(initial list of intentions is retrieved)

initial intentions:
 F(r4 & c4)

formula agent: (G((l | r | u | d) & !(l & r) & !(l & u) & !(l & d) & !(r & u) & !(r & d) & !(u & d)) & ((r1 & c1) & G((r1 | r2 | r3 | r4) & !(r1 & r2) & !(r1 & r3) & !(r1 & r4) & !(r2 & r3) & !(r2 & r4) & !(r3 & r4)) & ((c1 | c2 | c3 | c4) & !(c1 & c2) & !(c1 & c3) & !(c1 & c4) & !(c2 & c3) & !(c2 & c4) & !(c3 & c4)) & ((c2 -> X(l -> c1)) & (c3 -> X(l -> (c2 | c1))) & (c4 -> X(l -> (c3 | c2))) & (c1 -> X(r -> (c2 | c3))) & (c2 -> X(r -> (c3 | c4))) & (c3 -> X(r -> c4)) & (r2 -> X(u -> r1)) & (r3 -> X(u -> (r2 | r1))) & (r4 -> X(u -> (r3 | r2))) & (r1 -> X(d -> (r2 | r3))) & (r2 -> X(d -> (r3 | r4))) & (r3 -> X(d -> r4)) & (c1 -> X(u -> c1)) & (c1 -> X(d -> c1)) & (c2 -> X(u -> c2)) & (c2 -> X(d -> c2)) & (c3 -> X(u -> c3)) & (c3 -> X(d -> c3)) & (c4 -> X(u -> c4)) & (c4 -> X(d -> c4)) & (r1 -> X(l -> r1)) & (r1 -> X(r -> r1)) & (r2 -> X(l -> r2)) & (r2 -> X(r -> r2)) & (r3 -> X(l -> r3)) & (r3 -> X(r -> r3)) & (r4 -> X(l -> r4)) & (r4 -> X(r -> r4)))) -> ((G((X(c1 | !c1)) -> ((c1 -> X(!l)) & (c4 -> X(!r)) & (r1 -> X(!u)) & (r4 -> X(!d)))) & (F(r4 & c4))))

```

formula environment: ((r1 & c1) & ((G((l | r | u | d) & !(l & r)
& !(l & u) & !(l & d) & !(r & u) & !(r & d) & !(u & d))) -> (G((r1
| r2 | r3 | r4) & !(r1 & r2) & !(r1 & r3) & !(r1 & r4) & !(r2 & r3)
& !(r2 & r4) & !(r3 & r4)) & ((c1 | c2 | c3 | c4) & !(c1 & c2) & !(c1
& c3) & !(c1 & c4) & !(c2 & c3) & !(c2 & c4) & !(c3 & c4)) & ((c2 ->
X(l -> c1)) & (c3 -> X(l -> (c2 | c1))) & (c4 -> X(l -> (c3 | c2)))
& (c1 -> X(r -> (c2 | c3))) & (c2 -> X(r -> (c3 | c4))) & (c3 -> X(r
-> c4)) & (r2 -> X(u -> r1)) & (r3 -> X(u -> (r2 | r1))) & (r4 -> X(u
-> (r3 | r2))) & (r1 -> X(d -> (r2 | r3))) & (r2 -> X(d -> (r3 | r4)))
& (r3 -> X(d -> r4)) & (c1 -> X(u -> c1)) & (c1 -> X(d -> c1)) & (c2
-> X(u -> c2)) & (c2 -> X(d -> c2)) & (c3 -> X(u -> c3)) & (c3 -> X(d
-> c3)) & (c4 -> X(u -> c4)) & (c4 -> X(d -> c4)) & (r1 -> X(l -> r1))
& (r1 -> X(r -> r1)) & (r2 -> X(l -> r2)) & (r2 -> X(r -> r2)) & (r3
-> X(l -> r3)) & (r3 -> X(r -> r3)) & (r4 -> X(l -> r4)) & (r4 -> X(r
-> r4))))))

```

The environment formula is realizable

The agent formula is realizable

----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index 2)? [1 or 2]: 1

possible agent moves in current state:

d
u
r
l

(this is a dummy move for the agent, which has no effect, as the environment first needs to initialize)

pick an action, using index (nr. from 1 to 4): 3

move selected: r

(the environment has one option: initializing according to the LTL_f specification)

possible environment moves in current state:

r1 & c1

random environment move picked: r1 & c1

current env vars set to true:

r1
c1

(intentions remain unchanged)

progressed intentions:


before: $F(c4 \ \& \ r4) \rightarrow$ after: $F(c4 \ \& \ r4)$

call update operation? [y/n]: n

run continues without update operation

----- NEW LOOP STARTS HERE! -----

(agent is now located in world state r1, c1. The domain looks as follows:)

	c1	c2	c3	c4
r1	START			
r2				
r3				
r4				

get procrastinating (index 1) or non-procrastinating moves (index 2)? [1 or 2]: 1

possible agent moves in current state:

d

r

(first real move of the agent is picked)

pick an action, using index (nr. from 1 to 2): 2

move selected: r

(environment can pick any of the two non-deterministic effects)

possible environment moves in current state:

r1 & c3

r1 & c2

random environment move picked: r1 & c2

current env vars set to true:

r1

c2

progressed intentions:

before: F(c4 & r4) -> after: F(c4 & r4)

(agent calls for an update operation)

call update operation? [y/n]: y

would you like to add an intention [1], drop [2] an intention, or stop the entire system [3]?: 1

(agent attempts to add a new intention 'G(!c1)' at priority 0, the highest priority. the intention specifies that the agent should never enter column 1)

```
give new intention:  G(!c1)
set priority using an index (position in the intention list. 0 is
highest priority):  0
trying to add new intention 'G(!c1)' to intention list:
```

(new intention is realizable and is adopted into the intention list)

```
new goal:  F(c4 & r4) & G(!c1)
adding new intention is realizable!  new intention is adopted and
DFA is recomputed
new formula agent:  (not outputted here, as formula is too long)
call update operation? [y/n]:  n
```

run continues without update operation

----- NEW LOOP STARTS HERE! -----

(agent is located in world state r1, c2)

```
get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]:  1
```

possible agent moves in current state:

```
d
r
l
```

```
pick an action, using index (nr. from 1 to 3):  1
```

```
move selected:  d
```

possible environment moves in current state:

```
r3 & c2
r2 & c2
```

```
random environment move picked:  r3 & c2
```

current env vars set to true:

```
r3
c2
```

(intentions remain unchanged)

progressed intentions:

```
before:  G!c1 -> after:  G!c1
```

```
before:  F(c4 & r4) -> after:  F(c4 & r4)
```

```
call update operation? [y/n]:  y
```

(agent attempts to add a new intention)

would you like to add an intention [1], drop [2] an intention, or stop the entire system [3]?: 1

(agent tries to add intention 'F(r4 & c1 & XF(r1 & c1))' at the highest priority)

give new intention: F(r4 & c1 & XF(r1 & c1))

set priority using an index (position in the intention list. 0 is highest priority): 0

trying to add new intention 'F(r4 & c1 & XF(r1 & c1))' to intention list:

(adding the new intention is not realizable)

new goal: F(c4 & r4) & G!c1 & F(r4 & c1 & XF(r1 & c1))

adding new intention is not realizable.

(agent wants to revise the intentions, which will return which intentions from the intention list would have to be dropped to add the new intention)

would you like to revise the current intentions in order to add the new intention? [y/n]: y

binary intention list dropout: 000

the following intentions are tested for realizability: F(r4 & c1 & XF(r1 & c1)) & G!c1 & F(c4 & r4)

The formula is not realizable with these intentions.

binary intention list dropout: 001

the following intentions are tested for realizability: F(r4 & c1 & XF(r1 & c1)) & G!c1

The formula is not realizable with these intentions.

(IMS returns that intention 'G!c1' has to be dropped in order to add intention 'F(r4 & c1 & XF(r1 & c1))')

binary intention list dropout: 010

the following intentions are tested for realizability: F(r4 & c1 & XF(r1 & c1)) & F(c4 & r4)

Formula is realizable if the following intentions are dropped:

G!c1

call update operation? [y/n]: y

would you like to add an intention [1], drop [2] an intention, or stop the entire system [3]?: 2

(agent drops intention 'G!c1')

which intention should be dropped from the list?: (index 1 until 2)

options:

1 : G!c1

2 : F(c4 & r4)

```

index chosen: 1

new formula agent: (not outputted here, as formula is too long)
call update operation? [y/n]: y

would you like to add an intention [1], drop [2] an intention, or
stop the entire system [3]?: 1

(agent attempts to add new intention again)
give new intention: F(r4 & c1 & XF(r1 & c1))
set priority using an index (position in the intention list. 0 is
highest priority): 0
trying to add new intention 'F(r4 & c1 & XF(r1 & c1))' to intention
list:

(new intention is now realizable in conjunction with the intentions from the intention list)
new goal: F(c4 & r4) & F(r4 & c1 & XF(r1 & c1))
adding new intention is realizable! new intention is adopted and
DFA is recomputed
new formula agent: (not outputted here, as formula is too long)
call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1
possible agent moves in current state:
d
u
r
l

pick an action, using index (nr. from 1 to 4): 4
move selected: 1

possible environment moves in current state:
r3 & c1

random environment move picked: r3 & c1

current env vars set to true:
r3
c1

progressed intentions:
before: F(c1 & r4 & XF(c1 & r1)) -> after: F(c1 & r4 & XF(c1 &
r1))

```

```

before: F(c4 & r4) -> after: F(c4 & r4)

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1
possible agent moves in current state:
d
u
r

pick an action, using index (nr. from 1 to 3): 1
move selected: d

possible environment moves in current state:
r4 & c1

random environment move picked: r4 & c1

current env vars set to true:
r4
c1

(intention 'F(c1 & r4 & XF(c1 & r1))' is progressed, as the current world state is 'c1 & r4')
progressed intentions:
before: F(c1 & r4 & XF(c1 & r1)) -> after: F(c1 & r1) | F(c1 &
r4 & XF(c1 & r1))
before: F(c4 & r4) -> after: F(c4 & r4)

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1
possible agent moves in current state:
u
r

pick an action, using index (nr. from 1 to 2): 1
move selected: u

possible environment moves in current state:
r3 & c1

```



```

r2 & c1

random environment move picked:  r3 & c1

current env vars set to true:
r3
c1

progressed intentions:
before:  F(c1 & r1) | F(c1 & r4 & XF(c1 & r1)) -> after:  F(c1 &
r1) | F(c1 & r4 & XF(c1 & r1))
before:  F(c4 & r4) -> after:  F(c4 & r4)

call update operation?  [y/n]:  n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)?  [1 or 2]:  1
possible agent moves in current state:
d
u
r

pick an action, using index (nr.  from 1 to 3):  2
move selected:  u

possible environment moves in current state:
r2 & c1
r1 & c1

random environment move picked:  r1 & c1

current env vars set to true:
r1
c1

(intention 'F(c1 & r1)|F(c1 & r4 & XF(c1 & r1))' has now been achieved, and is removed from
the intention list)
progressed intentions:
before:  F(c1 & r1) | F(c1 & r4 & XF(c1 & r1)) -> after:  1
intention 'F(c1 & r1) | F(c1 & r4 & XF(c1 & r1))' has been fulfilled
and is removed from the intention list
before:  F(c4 & r4) -> after:  F(c4 & r4)

call update operation?  [y/n]:  n

```

```
run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1
possible agent moves in current state:
d
r

pick an action, using index (nr. from 1 to 2): 2
move selected: r

possible environment moves in current state:
r1 & c3
r1 & c2

random environment move picked: r1 & c2

current env vars set to true:
r1
c2

progressed intentions:
before: F(c4 & r4) -> after: F(c4 & r4)

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1
possible agent moves in current state:
d
r
l

pick an action, using index (nr. from 1 to 3): 2
move selected: r

possible environment moves in current state:
r1 & c4
r1 & c3

random environment move picked: r1 & c4

current env vars set to true:
r1
```

```
c4

progressed intentions:
before: F(c4 & r4) -> after: F(c4 & r4)

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1
possible agent moves in current state:
d
1

pick an action, using index (nr. from 1 to 2): 1
move selected: d

possible environment moves in current state:
r3 & c4
r2 & c4

random environment move picked: r3 & c4

current env vars set to true:
r3
c4

progressed intentions:
before: F(c4 & r4) -> after: F(c4 & r4)

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1
possible agent moves in current state:
d
u
1

pick an action, using index (nr. from 1 to 3): 1
move selected: d

possible environment moves in current state:
```

```

r4 & c4

random environment move picked:  r4 & c4

current env vars set to true:
r4
c4

(all intentions have been achieved)
progressed intentions:
before:  F(c4 & r4) -> after:  1
intention 'F(c4 & r4)' has been fulfilled and is removed from the
intention list

(IMS gives feedback to agent that a final state has been reached, and asks if the program should
be stopped. The agent calls update operation 'halt', and the program stops)
final state reached!
stop program? [y/n]:  y
program stopped

Process finished with exit code 0

```

6.8.4 Simulation of the slippery world domain (scenario 2)

For this simulation, the log outputs have been taken directly from the software for the IMS, where the blue text is added to indicate what is happening during each step in the program, and what is user input. In this simulation it is demonstrated how the IMS handles progressing intentions, while also handling an LTL_f intention which contains the 'until'-operator.

Log of the terminal output:

```

(initial list of intentions is retrieved)
initial intentions:
F(r1 & c1)

formula agent:  (G((l | r | u | d) & !(l & r) & !(l & u) & !(l &
d) & !(r & u) & !(r & d) & !(u & d))) & ((r2 & c4) & G((r1 | r2 |
r3 | r4) & !(r1 & r2) & !(r1 & r3) & !(r1 & r4) & !(r2 & r3) & !(r2
& r4) & !(r3 & r4)) & ((c1 | c2 | c3 | c4) & !(c1 & c2) & !(c1 & c3)
& !(c1 & c4) & !(c2 & c3) & !(c2 & c4) & !(c3 & c4)) & ((c2 -> X(l ->
c1)) & (c3 -> X(l -> (c2 | c1))) & (c4 -> X(l -> (c3 | c2))) & (c1 ->
X(r -> (c2 | c3))) & (c2 -> X(r -> (c3 | c4))) & (c3 -> X(r -> c4))
& (r2 -> X(u -> r1)) & (r3 -> X(u -> (r2 | r1))) & (r4 -> X(u -> (r3
| r2))) & (r1 -> X(d -> (r2 | r3))) & (r2 -> X(d -> (r3 | r4))) & (r3
-> X(d -> r4)) & (c1 -> X(u -> c1)) & (c1 -> X(d -> c1)) & (c2 -> X(u
-> c2)) & (c2 -> X(d -> c2)) & (c3 -> X(u -> c3)) & (c3 -> X(d -> c3))
& (c4 -> X(u -> c4)) & (c4 -> X(d -> c4)) & (r1 -> X(l -> r1)) & (r1

```

```

-> X(r -> r1)) & (r2 -> X(l -> r2)) & (r2 -> X(r -> r2)) & (r3 -> X(l
-> r3)) & (r3 -> X(r -> r3)) & (r4 -> X(l -> r4)) & (r4 -> X(r -> r4))))))
-> ((G((X(tt) -> ((c1 -> X(!l)) & (c4 -> X(!r)) & (r1 -> X(!u)) & (r4
-> X(!d)))))) & (F(r1 & c1) )))
  formula environment: ((r2 & c4) & ((G((l | r | u | d) & !(l & r)
& !(l & u) & !(l & d) & !(r & u) & !(r & d) & !(u & d))) -> (G((r1
| r2 | r3 | r4) & !(r1 & r2) & !(r1 & r3) & !(r1 & r4) & !(r2 & r3)
& !(r2 & r4) & !(r3 & r4)) & ((c1 | c2 | c3 | c4) & !(c1 & c2) & !(c1
& c3) & !(c1 & c4) & !(c2 & c3) & !(c2 & c4) & !(c3 & c4)) & ((c2 ->
X(l -> c1)) & (c3 -> X(l -> (c2 | c1))) & (c4 -> X(l -> (c3 | c2)))
& (c1 -> X(r -> (c2 | c3))) & (c2 -> X(r -> (c3 | c4))) & (c3 -> X(r
-> c4)) & (r2 -> X(u -> r1)) & (r3 -> X(u -> (r2 | r1))) & (r4 -> X(u
-> (r3 | r2))) & (r1 -> X(d -> (r2 | r3))) & (r2 -> X(d -> (r3 | r4)))
& (r3 -> X(d -> r4)) & (c1 -> X(u -> c1)) & (c1 -> X(d -> c1)) & (c2
-> X(u -> c2)) & (c2 -> X(d -> c2)) & (c3 -> X(u -> c3)) & (c3 -> X(d
-> c3)) & (c4 -> X(u -> c4)) & (c4 -> X(d -> c4)) & (r1 -> X(l -> r1))
& (r1 -> X(r -> r1)) & (r2 -> X(l -> r2)) & (r2 -> X(r -> r2)) & (r3
-> X(l -> r3)) & (r3 -> X(r -> r3)) & (r4 -> X(l -> r4)) & (r4 -> X(r
-> r4))))))

```

The environment formula is realizable

The agent formula is realizable

----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index 2)? [1 or 2]: 1

possible agent moves in current state:

d
u
r
l

(this is a dummy move for the agent, which has no effect, as the environment first needs to initialize)

pick an action, using index (nr. from 1 to 4): 4

move selected: 1

(the environment has one option: initializing according to the LTL_f specification)

possible environment moves in current state:

r2 & c4

random environment move picked: r2 & c4

current env vars set to true:

r2
c4

progressed intentions:

```


before: F(c1 & r1) -> after: F(c1 & r1)

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

```

(agent is now located in world state r2, c4. The domain looks as follows:)

	c1	c2	c3	c4
r1				
r2				START
r3				
r4				

```

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1

```

```

possible agent moves in current state:

```

```

d
u
l

```

```

pick an action, using index (nr. from 1 to 3): 2
move selected: u

```

```

possible environment moves in current state:
r1 & c4

```

```

random environment move picked: r1 & c4

```

```

current env vars set to true:

```

```

r1
c4

```

(intentions remain unchanged)

```

progressed intentions:

```

```

before: F(c1 & r1) -> after: F(c1 & r1)

```

```

call update operation? [y/n]: y

```

```

would you like to add an intention [1], drop [2] an intention, or
stop the entire system [3]?: 1

```

(agent attempts to add a new intention 'F(c1 U r4)' at priority 1)

```

give new intention: F(c1 U r4)
set priority using an index (position in the intention list. 0 is
highest priority): 1
trying to add new intention 'F(c1 U r4)' to intention list:

(new intention is realizable and is adopted into the intention list. DFA and maximally permissive
strategy are recomputed)
new goal: F(c1 U r4) & F(c1 & r1)
adding new intention is realizable! new intention is adopted and
DFA is recomputed
new formula agent: (not outputted here, as formula is too long)
call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1
possible agent moves in current state:
d
l

pick an action, using index (nr. from 1 to 2): 2
move selected: l

possible environment moves in current state:
r1 & c3
r1 & c2

random environment move picked: r1 & c2

current env vars set to true:
r1
c2

progressed intentions:
before: F(c1 & r1) -> after: F(c1 & r1)
before: F(c1 U r4) -> after: F(c1 U r4)

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1
possible agent moves in current state:
d

```

```

r
l

pick an action, using index (nr. from 1 to 3): 3

move selected: 1

possible environment moves in current state:
r1 & c1

random environment move picked: r1 & c1

current env vars set to true:
r1
c1

(intention 'F(c1 & r1)' is fulfilled and removed from the intention list. Also, since the agent is
now in column 1, intention 'F(c1 U r4)' is progressed into '(c1 U r4) | F(c1 U r4)')
progressed intentions:
before: F(c1 & r1) -> after: 1
intention 'F(c1 & r1)' has been fulfilled and is removed from the
intention list
before: F(c1 U r4) -> after: (c1 U r4) | F(c1 U r4)

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 2
possible agent moves in current state:
d

(as the agent asked for the non-procrastinating moves, only the moves are shown which make the
agent fulfill the remaining intentions in as little steps as possible)
pick an action, using index (nr. from 1 to 1): 1
move selected: d

possible environment moves in current state:
r3 & c1
r2 & c1

random environment move picked: r3 & c1

current env vars set to true:
r3

```



```

c1

progressed intentions:
before: (c1 U r4) | F(c1 U r4) -> after: (c1 U r4) | F(c1 U r4)

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 2
possible agent moves in current state:
d

pick an action, using index (nr. from 1 to 1): 1
move selected: d

possible environment moves in current state:
r4 & c1

random environment move picked: r4 & c1

current env vars set to true:
r4
c1

(the final intention of the intention list has now been fulfilled)
progressed intentions:
before: (c1 U r4) | F(c1 U r4) -> after: 1
intention '(c1 U r4) | F(c1 U r4)' has been fulfilled and is removed
from the intention list

(since a final state is reached, IMS asks if the agent want to stop the program. The agent calls
update operation 'halt', and the program is stopped)
final state reached!
stop program? [y/n]: y
program stopped

Process finished with exit code 0

```

6.8.5 Evaluation of IMS in the slippery world domain

After attempting translation of the Triangle-Tireworld from PDDL into LTL_f , the number of atomic propositions was kept as low as possible for expressing the slippery world domain as compactly as possible. This appeared to be a successful manner for expressing a FOND domain in LTL_f , as Lydia

and Syftmax had no issues computing the resulting DFA's and maximally permissive strategy for this.

In Section 6.8.3 and 6.8.4, the logs of simulations are provided, which are extracted directly from the IMS log output. In these simulations, the adding and dropping of temporally extended goals is demonstrated, while the IMS also dynamically checks for the realizability of adding new intentions. In the simulation in Section 6.8.3, also a demonstration is given of how the IMS recomputes which intentions should be dropped, in case the agent wants to add an intention which is currently not realizable, but becomes realizable when another intention of the intention list is dropped.

In both simulations, the progression of intentions is also demonstrated. At various points throughout the simulations the agent requests the non-procrastinating moves from the IMS, returning to the agent which actions to perform in order to fulfill all remaining intentions in as little steps as possible. These simulations demonstrate how the IMS satisfies the description of an IMS as provided in Section 3, while implementing the design for the IMS as provided in Section 4, and operates according to the specification provided in Section 5.

6.9 Triangle-Tireworld domain

In Section 6.9.1, an explanation is provided for the Triangle-Tireworld FOND domain. As already mentioned in Section 6.2, the Triangle-Tireworld domain is a well known FOND domains within the area of planning related research and competitions, for which we have included this domain in our research.

The Triangle-Tireworld domain is most commonly expressed in PDDL, for which an example domain and problem description are provided in Section 6.9.2. As can be seen, this domain can be represented rather compact using PDDL, as the rules of the domain are in an abstract form, i.e. not grounded yet using the atomic propositions. However, for LTL_f , the rules of the domain are directly applied to all atomic propositions, which results in a significantly greater expression for writing out the domain specification. This expression is given in Section 6.9.3. The PDDL and LTL_f expression describe the same FOND domain, as where it can be seen that using PDDL is easier for scaling and expressing the domain more compactly.

Note that in the domain expressions provided in Section 6.9.2 and 6.9.3, there are only a total of six environment states, for which the domain would be represented as shown in Figure 9. Would we add even more states to this, the LTL_f expression would grow exponentially with each extra state. For example purposes, we only write down the version with six states in Section 6.9.2 and 6.9.3.

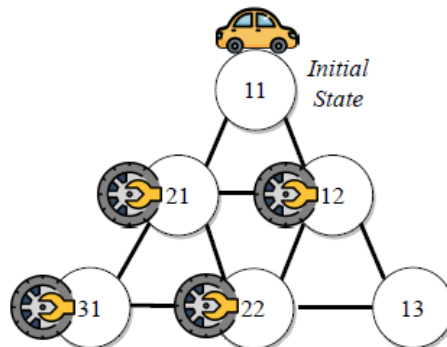


Figure 9: Triangle-Tireworld domain with six states

6.9.1 Description of the Triangle-Tireworld domain

One of the most well-known FOND domain models is the Triangle-Tireworld, where locations are connected by roads, and the agent can drive through them. The objective is to drive from one location to another. However, while driving between locations, a tire may go flat, and if there is a spare tire in the car's location, then the agent can use it to fix the flat tire. Figure 10 illustrates a FOND planning problem for the Triangle-Tireworld domain, where circles are locations, arrows represent roads, spare tires are depicted as tires, and the agent is depicted as a car [16]. In this example, the goal for the agent is to eventually reach state 15. The agent cannot go in a straight line here, as state 13 does not have a spare tire, and might put the agent in risk for not being able to change a tire in case it goes flat. For this, a more sophisticated strategy has to be planned out in order to reach state 15.

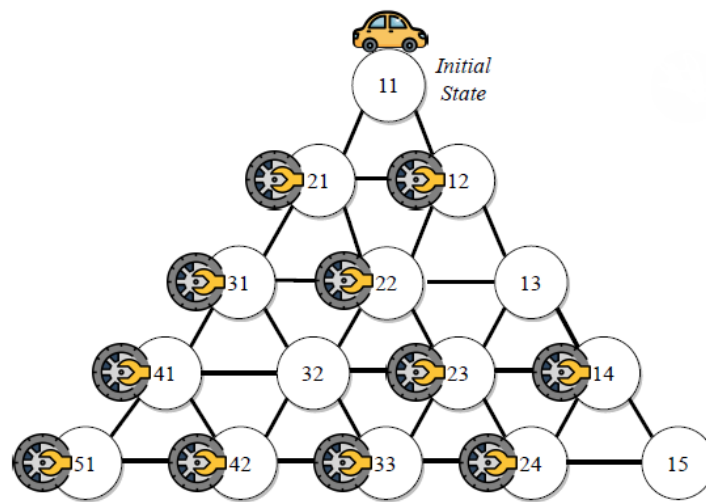


Figure 10: Triangle-Tireworld domain

6.9.2 Representation of the Triangle-Tireworld domain in PDDL

The Triangle-Tireworld domain is most commonly expressed in PDDL⁹. An explanation of how this is expressed, is described in Section 2.5. Below, we find how the Triangle-Tireworld is expressed in PDDL, both the domain- and an example problem.

domain file:

```
(define (domain triangle-tire)
  (:requirements :typing :strips :non-deterministic)
  (:types location)
  (:predicates (vehicle-at ?loc - location)
               (spare-in ?loc - location)
               (road ?from - location ?to - location)
               (not-flattire))

  (:action move-car
```

⁹Source for Triangle-Tireworld domain and problems in PDDL format: [21]

```

:parameters (?from - location ?to - location)
:precondition (and (vehicle-at ?from) (road ?from ?to) (not-flattire))
:effect (and
  (oneof (and (vehicle-at ?to) (not (vehicle-at ?from)))
    (and (vehicle-at ?to) (not (vehicle-at ?from)) (not (not-flattire))))))

(:action changetire
:parameters (?loc - location)
:precondition (and (spare-in ?loc) (vehicle-at ?loc))
:effect (and (not (spare-in ?loc)) (not-flattire)))

```

problem file:

```

(define (problem triangle-tire-1)
  (:domain triangle-tire)
  (:objects l-1-1 l-1-2 l-1-3 l-2-1 l-2-2 l-2-3 l-3-1 l-3-2 l-3-3 - location)
  (:init (vehicle-at l-1-1)(road l-1-1 l-1-2)(road l-1-2 l-1-3)(road l-1-1 l-2-1)(road l-1-2 l-2-2)
    (road l-2-1 l-1-2)(road l-2-2 l-1-3)(spare-in l-2-1)(spare-in l-2-2)(road l-2-1 l-3-1)
    (road l-3-1 l-2-2)(spare-in l-3-1)(spare-in l-3-1)(not-flattire))
  (:goal (vehicle-at l-1-3)))

```

For PDDL, only the problem file changes, depending on the scenario. The domain file remains the same throughout the different problem scenarios. In the problem file, all variables are defined, although not grounded yet.

6.9.3 Representation of the Triangle-Tireworld domain in LTL_f

For describing the Triangle-Tireworld domain in LTL_f , we will break up the domain into components, as described in Section 4.5:

Agent- and environment variables:

- agent variables : {movecar_11_21, movecar_11_12, movecar_12_21, movecar_12_11, movecar_12_22, movecar_12_13, movecar_21_12, movecar_21_11, movecar_21_31, movecar_21_22, movecar_13_22, movecar_13_12, movecar_22_31, movecar_22_13, movecar_22_21, movecar_22_12, movecar_31_22, movecar_31_21, changetire_11, changetire_12, changetire_21, changetire_13, changetire_22, changetire_31}
- environment variables : {vehicleat_11, vehicleat_12, vehicleat_21, vehicleat_13, vehicleat_22, vehicleat_31, sparein_11, sparein_12, sparein_21, sparein_13, sparein_22, sparein_31, road_11_21, road_11_12, road_12_21, road_12_11, road_12_22, road_12_13, road_21_12, road_21_11, road_21_31, road_21_22, road_13_22, road_13_12, road_22_31, road_22_13, road_22_21, road_22_12, road_31_22, road_31_21, flattire}

Domain specification:

- $\phi_{ag\ uniq\ act}$:
(mutual exlusion is applied over all agent variables which are specified above, for which the entire resulting formula is too long to be written out here fully)
- $\phi_{env\ trans}$:

- mutual exclusion of environment variables :
(**mutual exclusion is applied over all environment variables which are specified above, for which the entire resulting formula is too long to be written out here fully**)
- environment reactions :
 $(vehicleat_{11} \rightarrow \bigcirc(movecar_{11_21} \rightarrow (vehicleat_{21} \wedge (flattire \vee \neg flattire)))) \wedge$
 $(vehicleat_{11} \rightarrow \bigcirc(movecar_{11_12} \rightarrow (vehicleat_{12} \wedge (flattire \vee \neg flattire)))) \wedge$
(from here on similar expressions for all other states and possible transitions)
 $(sparein_{11} \wedge vehicleat_{11} \rightarrow \bigcirc(changetire_{11} \rightarrow (\neg sparein_{11} \wedge \neg flattire \wedge$
 $vehicleat_{11}))) \wedge (sparein_{12} \wedge vehicleat_{12} \rightarrow \bigcirc(changetire_{12} \rightarrow (\neg sparein_{12} \wedge$
 $\neg flattire \wedge vehicleat_{12})))$
(from here on similar expressions for all other states and possible transitions)
- frame :
 $(road_{11_21} \wedge road_{11_12} \wedge road_{12_21} \wedge road_{12_11} \wedge road_{12_22} \wedge road_{12_13} \wedge$
 $road_{21_12} \wedge road_{21_11} \wedge road_{21_31} \wedge road_{21_22} \wedge road_{13_22} \wedge road_{13_12} \wedge$
 $road_{22_31} \wedge road_{22_13} \wedge road_{22_21} \wedge road_{22_12} \wedge road_{31_22} \wedge road_{31_21} \wedge$
 $(\neg sparein_{11} \rightarrow \bigcirc(\neg sparein_{11})) \wedge ((sparein_{11} \wedge \neg changetire_{11}) \rightarrow \bigcirc(sparein_{11})) \wedge$
 $(\neg sparein_{12} \rightarrow \bigcirc(\neg sparein_{12})) \wedge ((sparein_{12} \wedge \neg changetire_{12}) \rightarrow \bigcirc(sparein_{12})) \wedge$
 $(\neg sparein_{21} \rightarrow \bigcirc(\neg sparein_{21})) \wedge ((sparein_{21} \wedge \neg changetire_{21}) \rightarrow \bigcirc(sparein_{21})) \wedge$
 $(\neg sparein_{13} \rightarrow \bigcirc(\neg sparein_{13})) \wedge ((sparein_{13} \wedge \neg changetire_{13}) \rightarrow \bigcirc(sparein_{13})) \wedge$
 $(\neg sparein_{22} \rightarrow \bigcirc(\neg sparein_{22})) \wedge ((sparein_{22} \wedge \neg changetire_{22}) \rightarrow \bigcirc(sparein_{22})) \wedge$
 $(\neg sparein_{31} \rightarrow \bigcirc(\neg sparein_{31})) \wedge ((sparein_{31} \wedge \neg changetire_{31}) \rightarrow \bigcirc(sparein_{31})))$
- $\phi_{ag\ act\ prec} : (\neg(vehicleat_{11} \wedge road_{11_21} \wedge \neg flattire) \rightarrow \bigcirc(\neg movecar_{11_21})) \wedge$
 $(\neg(vehicleat_{11} \wedge road_{11_12} \wedge \neg flattire) \rightarrow \bigcirc(\neg movecar_{11_12})) \wedge (\neg(vehicleat_{12} \wedge$
 $road_{12_21} \wedge \neg flattire) \rightarrow \bigcirc(\neg movecar_{12_21})) \wedge (\neg(vehicleat_{12} \wedge road_{12_11} \wedge \neg flattire)$
 $\rightarrow \bigcirc(\neg movecar_{12_11})) \wedge (\neg(vehicleat_{12} \wedge road_{12_22} \wedge \neg flattire) \rightarrow \bigcirc(\neg movecar_{12_22}))$
 $\wedge (\neg(vehicleat_{12} \wedge road_{12_13} \wedge \neg flattire) \rightarrow \bigcirc(\neg movecar_{12_13}))$
(from here on similar expressions for all other states and possible transitions)
 $\wedge (\neg(sparein_{11} \wedge vehicleat_{11} \wedge flattire) \rightarrow \bigcirc(\neg changetire_{11})) \wedge (\neg(sparein_{12} \wedge$
 $vehicleat_{12} \wedge flattire) \rightarrow \bigcirc(\neg changetire_{12})) \wedge (\neg(sparein_{21} \wedge vehicleat_{21} \wedge flattire) \rightarrow$
 $\bigcirc(\neg changetire_{21})) \wedge (\neg(sparein_{13} \wedge vehicleat_{13} \wedge flattire) \rightarrow \bigcirc(\neg changetire_{13})) \wedge$
 $(\neg(sparein_{22} \wedge vehicleat_{22} \wedge flattire) \rightarrow \bigcirc(\neg changetire_{22})) \wedge (\neg(sparein_{31} \wedge$
 $vehicleat_{31} \wedge flattire) \rightarrow \bigcirc(\neg changetire_{31}))$

Problem specification:

- $\phi_{env\ init} : (\neg flattire \wedge vehicleat_{11} \wedge sparein_{12} \wedge sparein_{11} \wedge sparein_{21} \wedge sparein_{13} \wedge$
 $sparein_{22} \wedge sparein_{31})$
- $\phi_{goal} : \diamond(vehicleat_{13})$

6.9.4 Simulation of the Triangle-Tireworld domain

During the evaluation of the Triangle-Tireworld domain, it became clear that translating PDDL into LTL_f is more complicated than anticipated. As a PDDL domain can be expressed in a relational language, i.e. without being grounded, it can be expressed rather compact. Whenever this expression is translated to LTL_f , the domain needs to be grounded using the atomic propositions, for which it became clear that this is not an efficient manner for translation. Would we want to express an equivalent domain into LTL_f , it might be best to express agent- and environment actions as, for

example, 'down-left', instead of talking about moves from one exact position to another, such as 'movecar_11_21', as this decreases the amount of agent- and environment actions significantly. For this, it is inefficient to translate PDDL directly to LTL_f , as it would be more efficient to lower the amount of atomic propositions available in LTL_f as much as possible.

As already mentioned in Section 6.2, translating the PDDL specification of the Triangle-Tireworld as described in the example of Section 3.1 into an LTL_f specification results in a specification which is too large to use in our software. The resulting LTL_f formula affected the software too much, for which Lydia and Syftmax were not able anymore to extract a DFA and maximally permissive strategy within a reasonable amount of time, where also too much computational power was required in order to simulate a run. The hard- and software specification on which the IMS software has been developed and run on, is described in Section 6.3.

As we could not run the resulting LTL_f formula on our hard- and software, we only provide a handwritten run for the Triangle-Tireworld, provided in Section A, in which it is described how the IMS would work for the Triangle-Tireworld if the computational resources could process this formula. In this handwritten run, we replicate the example scenario as described in Section 3.1, for this demonstrates if the IMS operates as intended. In Section 6.9.5, an evaluation is given for how well the IMS functions in the Triangle-Tireworld domain.

6.9.5 Evaluation of IMS in the Triangle-Tireworld domain

Despite the length of the resulting LTL_f formula, the IMS is still able to operate exactly as specified within a handwritten benchmark simulation. All query- and update operations which were required for operating could be used, resulting in a handwritten simulation which corresponds to the definition and design for the IMS as described in Section 3 and 4. The handwritten simulation described in Section A is identical to how the IMS operates in the initial example scenario described in Section 3.1, proving that the IMS is capable of doing exactly what it is supposed to do.

6.10 General evaluation of the IMS implementation

The simulations which the IMS has been evaluated on are described in Section 6.7.2, 6.8.3, 6.8.4, and appendix A. For each simulation, a short description is given of what functions of the IMS have been tested, and to which degree the IMS functions as expected.

In Section 6.7.3, 6.8.5, and 6.9.5 (handwritten log output), evaluations are already provided for the simulations which have been ran in each individual domain. The only complication which we have come across during the evaluation of the IMS was the processing of the LTL_f formula for the Triangle-Tireworld domain. However, this complication was not caused by the IMS, as Lydia was simply not able to computationally handle the resulting LTL_f formula when the PDDL specification for the Triangle-Tireworld was translated into LTL_f .

When considering all these evaluations, the implementation of the IMS covers all functionalities as how it was specified, where the IMS is able to handle all query- and update operations as described in Section 3.5 and 3.6, keep track of an intention list, progress LTL_f intentions during a run, compose an LTL_f formula for extracting moves which keep an agent in the winning region, recompose this formula also during runtime when intentions and the current domain state change, compose a formula for the environment for extracting good moves, compute a DFA and maximally permissive strategy, check for realizability, handle temporally extended goals, among all other functionalities of the IMS.

7 Related work

In the introduction of this research, a brief insight is given on what related work our research is based upon. A great amount of research has already been done related to intention revision, going back nearly 30 years. Although we will not cover all of this, we discuss some of the major approaches and how our approach relates to these.

As our research is about managing the intentions of agents, we need to understand what intentions are and how they relate to intelligent agents. In [36, p. 28–42], the mental state of intelligent agents is discussed, which is based upon the beliefs, desires, and intentions which an agent might have. Agents who are designed using this type of architecture are often referred to as BDI-agents (Beliefs, Desires, and Intentions). For the mental state of an agent, beliefs represent the informational state which an agent believes about the world it is in. Desires represent the motivational state of an agent: what does the agent desire to accomplish? Intentions represent the deliberative state of an agent: what has an agent already dedicated itself to achieving? Intentions can be considered as desires which an agent has committed to achieving. Our research only applies to the intentions which an agent might have, as the agent decides to which intentions it wants to commit to, and the IMS serves as a tool for the agent to manage these intentions and extract plans for achieving these intentions.

In [34], automata-theoretic approaches for LTL are already discussed. LTL specifications are translated into automata, which are then used for performing synthesis, i.e. verify that the automata meet the given LTL specifications. Also, these automata can be used for checking realizability, i.e. finding at least one strategy for an agent which eventually fulfills the entire specification. In our research, this means that an agent eventually has to fulfill all the intentions as provided in the intention list. This is the very essence what our research is based upon, as [34] covers the translation of LTL/LTL_f to automata, perform synthesis, and checking for realizability.

In [12] it is discussed how synthesis can be performed for LTL_f and LDL_f expressions, where an agent operates within a FOND environment. Although in this paper the term 'FOND' is not explicitly mentioned yet, it is discussed that the agent deals in an environment where it has no control over the environment variables, while it does have full observability. The extraction of a winning strategy is covered, which is based upon the synthesis problem. In this paper, it is mentioned that performing synthesis is actually equivalent to computing a winning strategy, as when synthesis is performed, it is analyzed if there exists at least one strategy which can guarantee for the agent to fulfill the entire specification, no matter how the environment reacts, i.e. the variables which the agent has no control over. In [10] the research of [12] is applied, where this describes the automata-theoretic foundations of FOND planning for both LTL_f and LDL_f temporally extended goals. It explains how a DFA can be computed using their proposed algorithm, given a domain and goal, specified as an LTL_f formula. From the resulting DFA, a winning strategy can then be extracted.

The idea for extracting a winning strategy is later exploited in [40]. They describe how to compute the entire set of winning strategies, instead of only a single strategy. This includes both the set of deferring- and non-deferring strategies, together forming the 'maximally permissive strategy'. Here, a deferring strategy refers to actions of an agent which guarantee to remain in the winning region, while deferring the winning moment. A non-deferring strategy refers to actions which guarantee an agent to both stay in the winning region, and also progress towards a goal state. In our research, we use the maximally permissive strategy for giving the agent as much freedom as possible in choosing an action. In the implementation of the maximally permissive strategy within our design of the IMS, we make use of Syftmax (5.1.2), where Syftmax was developed as an implementation tool in the paper of [40]. In [25], an approach is described for computing more than a singular plan for reaching a goal state, as is also done in [40] when a maximally permissive strategy is computed. Although [25] uses a somewhat related approach to that of [40] in terms of providing the agent with flexibility

in choosing actions during a run, in [25] the possible environment reactions are not specified and the plans are not guaranteed to be winning. The research of [25] is focused on avoiding goal conflicts, where goal plan trees (GPT) are made in which several plans are expressed for achieving an agent's goal. This GPT represents the goals, plans, and actions for an agent, which is somewhat related to the winning region of [40], although in [25] the environment reactions are not included, and also in GPT's an agent plan is not guaranteed to be winning. In [25], intentions are revised based on the desires and beliefs of an agent, although in our research we only use the intentions of an agent. The research of [25] presents a model of goal processing, named 'GROVE', which is used for providing choices over executing possible agent plans which are consistent with the agent's beliefs. In the introduction of this research, more on the research of [25] is discussed, and how this relates to our research.

In Section 3.5, we describe the query operation $I.isRealizable(\phi, k)$. Here, an intention is given with a certain priority, where the priority indicates the position within a list of intentions. Whenever an intention is given to the IMS which is not consistent with the intention list, the intention list is revised to check which intentions should be dropped from the intention list in order to add intention ϕ , maximizing utility. Important here is to note that the IMS does not drop- or add any intentions by itself, as only the agent is allowed to do so. This approach of dynamic goal revision is already proposed in [24]. Although in our research we do not implement this manner of goal revision directly, it is quite similar. In [24], no goals or intentions are ever dropped from the list they keep track of. This list is also prioritized, for which goals/intentions can only be set to active or inactive, but are not dropped. The active goals/intentions of this list are dynamically revised in order to maximize utility. In our approach, we only deal with intentions. This implies that an agent is already dedicated to achieving them, for which it should be that all intentions from the intention list should be realizable. If this is not the case, the agent should choose which intentions to drop or add, as we are not dealing with goals for which the goal is to maximize utility.

In [33] and [32] the problem of conflicting goals is also addressed, which discuss the representation and reasoning mechanisms for identifying these conflicts. Both papers discuss in what manner conflicting goals can be expressed. In [33], a logical language for expressing goals is introduced, where the semantics of these goals can be used in several manners in order to add expressivity, which can then be used to identify any conflicting goals. Another method for handling conflicting goals is proposed in [32], where a mechanism is provided for avoiding that the conditions of one goal undo the conditions for fulfilling another goal. This method schedules in which order goals should be executed in order to avoid such conflicts. Both these studies deal with the interaction of goals/intentions in a different manner as opposed to our research, as our manner for dealing with conflicting goals is based upon that of [24], where priorities are given to goals for indicating their importance.

Although in our research we do not let the list of intentions be dynamically adjusted by the IMS itself, as opposed to the approach proposed in [24], in our research and implementation for the IMS an agent is able to add and drop intentions from the intention list during a run. In our manner of implementating the IMS design, the DFA is recomputed, consisting of the FOND domain and the entire list of intentions. As in our research we are dealing with temporally extended goals, if we would recompute the DFA somewhere during a run, this could cause the progress in any partially fulfilled intentions to be lost, as some subgoals of the temporally extended goals could already have been fulfilled. For this is the case, there is a need for progressing intentions, such that the IMS will know to which degree each intention has already been fulfilled. A manner for progressing linear temporal logic expressions is proposed in [1], which we have used in our research for progressing temporally extended goals expressed in LTL_f . The paper discusses an approach for domain-dependent search control knowledge, which includes a declarative semantics for this search control knowledge. These semantics are used to check how an LTL expression progresses whenever a part of this expression

complies to a given state. For example, if in a next state formula f_1 is true, the LTL expression $O f_1$ progresses into f_1 .

As already mentioned in the begin of this Section, the revision- and progression of intentions is an active area of research for roughly 30 years already. During these years, the research in this field seems to have become rather inconsistent among each other in terms of terminology and how intentions are represented, among other inconsistencies. In order to address these inconsistencies and make research in this field more coherent, [27] has proposed a call for a competition related to intention progression. In [6], this proposal for an intention progression competition (IPC)¹⁰ is formalized, where the competition is mainly focused on how plans for an agent can be refined given a set of intentions, and which intentions can be advanced in order to successfully execute a plan. This problem of intention progression within BDI-agents is covered by [37], which focuses on estimating the likelihood of any conflicting intentions which an agent might have. Monte-Carlo Tree Search (MCTS) is applied during runtime simulations, which has shown promising results in relation to the intention progression problem. Similar to most research we discuss in this Section, [37] applies to BDI-agent architectures, for which the goal is to maximize utility for an agent. Would our research be expanded for application directly within the architecture of a BDI-agent, it might be interesting to look more into the research by [37].

In terms of extracting a DFA which represents a FOND domain and agent intentions, [8] proposes a method for inductively transforming each LTL_f/LDL_f subformula of an entire LTL_f/LDL_f specification into a DFA, and combining them through automata operators. This differs from our research, where we extract a singular DFA from the entire LTL_f specification. The method for computing several DFA's to represent the FOND domain and agent intentions would be an interesting approach to investigate for building further upon our design for the IMS, as this would be less computationally expensive in terms of recomputation when a change is made to the list of intentions. In our approach, the DFA for the entire specification has to be recomputed whenever a change is made to the list of intentions. However, if a DFA would be computed for the specification of the FOND domain and for each individual intention of the intention list, a new DFA would only have to be computed whenever a new intention is added to the intention list.

In [17], another approach is described for FOND planning, using both LTL_f and $PLTL_f$ (pure-past LTL_f). This is a master's thesis extract which make use of PDDL. An LTL_f specification can be given for a temporally extended goal, for which a DFA is computed using a tool called LTL_f2DFA ¹¹. The resulting DFA is then encoded into the FOND domain for which the goal is specified. Although this approach makes usage of domains which are expressed in PDDL, this might be interesting to implement if future work, where the IMS is able to handle PDDL specifications.

Lastly, our research is based upon that of [26], in which LTL_f synthesis techniques are investigated for maintaining consistency among intentions. The motivation for our research approach is discussed in [26], in which the issue we tackle related to intention management in our research is discussed more generally. Although in our research we do not deal with certain subjects discussed in [26], such as committing to intentions based on the desires an agent might have, it does provide more context and motivation for our research which is useful for future research.

¹⁰Website for the IPC: [5]

¹¹Source for LTL_f2DFA -tool: [15]

8 Conclusion

We have started off this research by asking ourselves what properties an intention management system needs in order to handle goal change in a FOND domain, while maintaining consistency among a changing set of temporally extended goal. To answer this question, we have started by writing out an example scenario, provided in Section 3.1, which such a system should be able to handle. Based on this example, we extract which parties interact with the IMS, and what information goes in- and out of this IMS. The information saved in the state of an IMS is described based on this information in Section 3.4, which is then used for executing query- and update operations as described in Sections 3.5 and 3.6. An example of how the IMS can be used by an agent is described in Section 3.8, giving an idea how the IMS could be used in practice.

Based on the formal description of an IMS, we have proposed a design for an IMS in Section 4, which we later turned into a proof of concept in the form of software. For doing so, we took the abstract description for an IMS state and the query- and update operations provided in Section 3, and gave a more formal description of what exact information is needed within the IMS state in order to execute the update- and query operations in practice, as described in Section 4.3. This design has been used for developing the actual proof of concept as software, for which a description is given how an agent would use this, and what changes within the state of an IMS during each step of these operations.

Lastly, the IMS as software has been evaluated on three FOND domains, expressed in LTL_f . An initial objective for this research was to translate benchmark FOND domains, expressed in PDDL, into LTL_f . However, no useable software appeared to be available for this. Several attempts have been made to make such software ourselves, however this proved to be more complex than anticipated. As an alternative option, one FOND domain expressed in PDDL, the Triangle-Tireworld domain, has been taken, and a Python script has been written to produce an equivalent FOND domain, expressed in LTL_f . Although the resulting LTL_f formula was usable in our software when the domain is scaled down significantly to only three environment states, this expression became exponentially larger when more environment states were added, to a point where this LTL_f could no longer be used for extracting a DFA and maximally permissive strategy. As our computational resources could not handle this, a handwritten log output has been provided, describing how the IMS operates if the software could handle the length of the given LTL_f expression.

As translating the PDDL specification of the Triangle-Tireworld domain into LTL_f proved to be inefficient, we have developed two other FOND domains by ourselves which could be more compact in LTL_f : the decision tree domain, and the slippery world domain. The IMS has been tested on several simulations within these domains, where the IMS has proven in practice to operate exactly as initially specified in the original goal statement.

The IMS has demonstrated to be able to handle temporally extended goals, apply intention progression, dynamically alter the list of intentions and the accompanying strategies for fulfilling these intentions, and to perform all query- and update operations as intended. By doing so, we show that our description of an IMS gives answer to our initial research question.

9 Future work

One of the more remarkable findings in this research has been that the direct translation from PDDL into LTL_f appears to be more difficult than anticipated. As already mentioned, previous work has been done on this, but no concrete software is, as far as we could find, available for performing this translation. In future research, it would be beneficial to find a more efficient manner for directly translating PDDL into LTL_f , without the resulting expression becoming unnecessarily large. As the frame of a domain is not captured in PDDL, this is something which needs to be added when generating the LTL_f expression. Also, a more compact representation should be found for expressing actions and environment variables. As we saw in Section 6.9.3, when PDDL was translated directly into LTL_f , the resulting expression grew exponentially with each new variable added, as there were no universal actions, such as 'left' or 'right'.

Since the translation from PDDL into LTL_f is too computationally expensive using our approach, it might be interesting to explore manners for directly extracting a DFA from a PDDL expression. As this would avoid the expensive computation of the DFA from an LTL_f expression, the IMS could potentially handle more extensive FOND domains.

Besides of the translation from PDDL into LTL_f , the computation of the domain and intentions into a DFA could also be improved. In our approach, the domain, initialization, and intentions were conjoined into a singular LTL_f formula, from which a singular DFA was extracted. This might be one of the simplest way for extracting a DFA which represents the winning region for an agent, but it is more computationally expensive doing so compared to computing the DFA separately for the domain and each individual intention, as these will be smaller compared to the conjoined product. Also, when using a singular DFA, each time a new intention is added or dropped, the entire DFA is recomputed, which makes it even more computationally expensive. If we would add or drop a new intention when we have separate DFA's for each component, we would only have to compute a new DFA for the new intention specification, or drop a DFA of an intention which is dropped.

As for the design of the IMS, this research has mainly only covered the managing of LTL_f intentions. However, as how the IMS is set up currently, it would only require minor adjustments in order to also be able to handle LDL_f expressions, in which we could also talk about agent actions. To do so, adjustments would have to be made in how the intentions are currently progressed, as this now is only able to handle LTL/LTL_f intentions.

Lastly, the IMS could also be altered to not only deal with intentions, but to also deal with goals or desires. The agent would not have to drop intentions by itself, but the IMS would be able to manage intentions dynamically by keeping track of a list of goals, sorted on priority, for which it is checked which intentions with highest priority are realizable in conjunction. As how the IMS is set up currently, to achieve this the design only needs minor adjustments in the 'Ladopt' update query, and how the intentions are saved and dropped.

Bibliography

- [1] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artif. Intell.*, 116(1-2):123–191, 2000. doi: 10.1016/S0004-3702(99)00071-5. URL [https://doi.org/10.1016/S0004-3702\(99\)00071-5](https://doi.org/10.1016/S0004-3702(99)00071-5).
- [2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*, volume 26202649. 01 2008. ISBN 978-0-262-02649-9.
- [3] Randal E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In Richard L. Rudell, editor, *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1995, San Jose, California, USA, November 5-9, 1995*, pages 236–243. IEEE Computer Society / ACM, 1995. doi: 10.1109/ICCAD.1995.480018. URL <https://doi.org/10.1109/ICCAD.1995.480018>.
- [4] Janusz A. Brzozowski and Ernst L. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theor. Comput. Sci.*, 10:19–35, 1980. doi: 10.1016/0304-3975(80)90069-9. URL [https://doi.org/10.1016/0304-3975\(80\)90069-9](https://doi.org/10.1016/0304-3975(80)90069-9).
- [5] Rafael C. Cardoso, Simon Castle-Green, Alexander Dewfall, and Brian Logan. The Intention Progression Competition website. URL <https://www.intentionprogression.org/>.
- [6] Simon Castle-Green, Alexi Dewfall, and Brian Logan. The intention progression competition. In Cristina Baroglio, Jomi Fred Hübner, and Michael Winikoff, editors, *Engineering Multi-Agent Systems - 8th International Workshop, EMAS 2020, Auckland, New Zealand, May 8-9, 2020, Revised Selected Papers*, volume 12589 of *Lecture Notes in Computer Science*, pages 144–151. Springer, 2020. doi: 10.1007/978-3-030-66534-0_10. URL https://doi.org/10.1007/978-3-030-66534-0_10.
- [7] Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Strong planning in non-deterministic domains via model checking. In Reid G. Simmons, Manuela M. Veloso, and Stephen F. Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems, Pittsburgh, Pennsylvania, USA, 1998*, pages 36–43. AAAI, 1998. URL <http://www.aaai.org/Library/AIPS/1998/aips98-005.php>.
- [8] Giuseppe De Giacomo and Marco Favorito. Compositional Approach to Translate LTL_f/LDL_f into Deterministic Finite Automata. In Susanne Biundo, Minh Do, Robert Goldman, Michael Katz, Qiang Yang, and Hankz Hankui Zhuo, editors, *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, pages 122–130. AAAI Press, 2021. URL <https://ojs.aaai.org/index.php/ICAPS/article/view/15954>.
- [9] Giuseppe De Giacomo and Yves Lespérance. The nondeterministic situation calculus. In Meghyn Bienvenu, Gerhard Lakemeyer, and Esra Erdem, editors, *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, pages 216–226, 2021. doi: 10.24963/KR.2021/21. URL <https://doi.org/10.24963/kr.2021/21>.
- [10] Giuseppe De Giacomo and Sasha Rubin. Automata-Theoretic Foundations of FOND Planning for LTL_f and LDL_f Goals. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh In-*

- ternational Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 4729–4735. ijcai.org, 2018. doi: 10.24963/IJCAI.2018/657. URL <https://doi.org/10.24963/ijcai.2018/657>.
- [11] Giuseppe De Giacomo and Moshe Y. Vardi. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 854–860. IJCAI/AAAI, 2013. URL <http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997>.
- [12] Giuseppe De Giacomo and Moshe Y. Vardi. Synthesis for LTL and LDL on Finite Traces. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1558–1564. AAAI Press, 2015. URL <http://ijcai.org/Abstract/15/223>.
- [13] Giuseppe De Giacomo, Marco Favorito, Jianwen Li, Moshe Y. Vardi, Shengping Xiao, and Shufang Zhu. LTL_f synthesis as AND-OR graph search: Knowledge compilation at work. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 2591–2598. ijcai.org, 2022. doi: 10.24963/IJCAI.2022/359. URL <https://doi.org/10.24963/ijcai.2022/359>.
- [14] Giuseppe De Giacomo, Gianmarco Parretti, and Shufang Zhu. LTL_f Best-Effort Synthesis in Nondeterministic Planning Domains. In *ECAI*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, pages 533–540. IOS Press, 2023.
- [15] Sapienza Università di Roma. LTL_f 2DFA. URL <http://ltrlf2dfa.diag.uniroma1.it/>.
- [16] Ramon Fraga Pereira, Francesco Fuggitti, Felipe Meneguzzi, and Giuseppe De Giacomo. Temporally Extended Goal Recognition in Fully Observable Non-Deterministic Domain Models. *CoRR*, abs/2306.08680, 2023. doi: 10.48550/ARXIV.2306.08680. URL <https://doi.org/10.48550/arXiv.2306.08680>.
- [17] Francesco Fuggitti. FOND planning for LTL_f and $PLTL_f$ goals. *CoRR*, abs/2004.07027, 2020. URL <https://arxiv.org/abs/2004.07027>.
- [18] Francesco Fuggitti. *Efficient Techniques for Automated Planning for Goals in Linear Temporal Logics on Finite Traces*. PhD thesis, Dept. of Electrical Engineering and Computer Science, York University, Toronto, ON, Canada, October 2023. URL <https://yorkspace.library.yorku.ca/items/1ad05d81-533a-48ae-a8c0-c6f2ef7ca315>.
- [19] Pierre Gaillard, Fabio Patrizi, and Giuseppe Perelli. Strategy repair in reachability games. In Kobi Gal, Ann Nowé, Grzegorz J. Nalepa, Roy Fairstein, and Roxana Radulescu, editors, *ECAI 2023 - 26th European Conference on Artificial Intelligence, September 30 - October 4, 2023, Kraków, Poland - Including 12th Conference on Prestigious Applications of Intelligent Systems (PAIS 2023)*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, pages 780–787. IOS Press, 2023. doi: 10.3233/FAIA230344. URL <https://doi.org/10.3233/FAIA230344>.

- [20] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013. ISBN 9781608459698. doi: 10.2200/S00513ED1V01Y201306AIM022. URL <https://doi.org/10.2200/S00513ED1V01Y201306AIM022>.
- [21] Tomas Geffner. FOND domains, triangle-tireworld in PDDL. URL <https://github.com/tomsons22/FOND-SAT/tree/master/F-domains/triangle-tireworld>.
- [22] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun, and Daniel Weld. PDDL - The Planning Domain Definition Language. *ICAPS*, 08 1998.
- [23] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley series in computer science. Addison-Wesley, Reading, Mass, 1979. ISBN 978-0-201-02988-8.
- [24] Shakil M. Khan and Yves Lespérance. A logical framework for prioritized goal change. In Wiebe van der Hoek, Gal A. Kaminka, Yves Lespérance, Michael Luck, and Sandip Sen, editors, *9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada, May 10-14, 2010, Volume 1-3*, pages 283–290. IFAAMAS, 2010. URL <https://dl.acm.org/citation.cfm?id=1838246>.
- [25] Sam Leask, Natasha Alechina, and Brian Logan. A Computationally Grounded Model for Goal Processing in BDI Agents. In *Proceedings of the Proceedings of the 6th Workshop on Goal Reasoning (GR'2018), Stockholm, Sweden*, volume 13. EMAS, 2018.
- [26] Yves Lespérance. On LTL_f synthesis and goal formation/revision. Short paper for the AAAI 2023 Spring Symposium On the Effectiveness of Temporal Logics on Finite Traces in AI, San Francisco, CA, 2023.
- [27] Brian Logan, John Thangarajah, and Neil Yorke-Smith. Progressing intention progression: A call for a goal-plan tree contest. In Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee, editors, *Proceedings of the 16th Conference on Autonomous Agents and Multi-Agent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, pages 768–772. ACM, 2017. URL <http://dl.acm.org/citation.cfm?id=3091234>.
- [28] Moshe Y. Vardi. The rise and fall of linear time logic. *GandALF 2011, EPTCS 54*, 2011. doi: 10.4204/EPTCS.54.0.2. URL <https://www.cs.rice.edu/~vardi/papers/gandalfl1-myv.pdf>.
- [29] Christian Muise, Sheila McIlraith, and Christopher Beck. Improved Non-Deterministic Planning by Exploiting State Relevance. *Proceedings of the International Conference on Automated Planning and Scheduling*, 22:172–180, May 2012. ISSN 2334-0843, 2334-0835. doi: 10.1609/icaps.v22i1.13520. URL <https://ojs.aaai.org/index.php/ICAPS/article/view/13520>.
- [30] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. doi: 10.1109/SFCS.1977.32. URL <https://doi.org/10.1109/SFCS.1977.32>.

- [31] Renzo Schram. IMS software and Python scripts, including instructions, June 2024. URL https://github.com/renzoq/IMS_thesis.
- [32] John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting & avoiding interference between goals in intelligent agents. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 721–726. Morgan Kaufmann, 2003. URL <http://ijcai.org/Proceedings/03/Papers/105.pdf>.
- [33] M. Birna van Riemsdijk, Mehdi Dastani, and John-Jules Ch. Meyer. Goals in conflict: semantic foundations of goals in agent programming. *Auton. Agents Multi Agent Syst.*, 18(3): 471–500, 2009. doi: 10.1007/S10458-008-9067-4. URL <https://doi.org/10.1007/s10458-008-9067-4>.
- [34] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham M. Birtwistle, editors, *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings)*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1995. doi: 10.1007/3-540-60915-6_6. URL https://doi.org/10.1007/3-540-60915-6_6.
- [35] WhiteMech. Lydia software, May 2024. URL <https://github.com/whitemech/lydia>.
- [36] Michael J. Wooldridge. *An Introduction to MultiAgent Systems, Second Edition*. Wiley, 2009. ISBN 978-0-470-51946-2.
- [37] Yuan Yao, Natasha Alechina, Brian Logan, and John Thangarajah. Intention progression using quantitative summary information. In Frank Dignum, Alessio Lomuscio, Ulle Endriss, and Ann Nowé, editors, *AAMAS '21: 20th International Conference on Autonomous Agents and Multi-agent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, pages 1416–1424. ACM, 2021. doi: 10.5555/3463952.3464115. URL <https://www.ifaamas.org/Proceedings/aamas2021/pdfs/p1416.pdf>.
- [38] H.L.S. Younes, M. L. Littman, D. Weissman, and J. Asmuth. The First Probabilistic Track of the International Planning Competition. *Journal of Artificial Intelligence Research*, 24:851–887, December 2005. ISSN 1076-9757. doi: 10.1613/jair.1880. URL <https://jair.org/index.php/jair/article/view/10433>.
- [39] Shufang Zhu. SyftMax software, December 2023. URL <https://github.com/Shufang-Zhu/SyftMax>.
- [40] Shufang Zhu and Giuseppe De Giacomo. Synthesis of Maximally Permissive Strategies for LTL_f Specifications. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 2783–2789. ijcai.org, 2022. doi: 10.24963/IJCAI.2022/386. URL <https://doi.org/10.24963/ijcai.2022/386>.

A Triangle-Tireworld domain simulation (handwritten log output)

Here we see the handwritten log output when the IMS is operating in the Triangle-Tireworld domain, where we replicate the exact example scenario described in Section 3.1. The output below is written down as to how the IMS software theoretically operates, since the resulting LTL_f formula was too elaborate for Lydia and Syftmax to process. In this handwritten simulation, the bold text indicates user input.

As already mentioned in Section 3.1, we assume that the domain layout is exactly as in Figure 11, except that there is a spare at location 13 and there is no spare at locations 22 and 33.

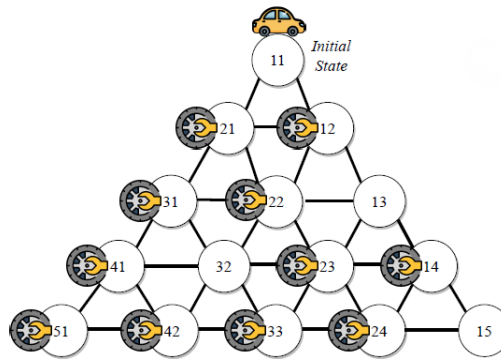


Figure 11: Visual representation of the Triangle-Tireworld domain

Log of the terminal output:

(an initial list of intentions is provided, for which realizabilty is checked. The intention says in natural language: "eventually be in location 32, and next there cannot be another state", which means that the agent needs to have state 32 as a final state)

```
initial intentions:
F(vehicleat_32 & X(ff))
```

```
formula agent: (not outputted here, as formula is too long)
formula environment: (not outputted here, as formula is too long)
```

```
The environment formula is realizable
The agent formula is realizable
----- NEW LOOP STARTS HERE! -----
```

(agent chooses if it wants to retrieve the procrastinating- or non-procrastinating moves for the current state)

```
get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1
```

(this is a 'dummy' move for the agent which has no effect, as the agent starts, and the environment reacts. However, in the first move the environment needs to initialize, since otherwise there is no

effect for the agent action)

possible agent moves in current state:

movecar_11_21

movecar_11_12

pick an action, using index (nr. from 1 to 2): 1

move selected: movecar_11_21

(only one option is given here for the environment: the initial environment values as written in the LTL_f specification)

possible environment moves in current state:

(!flattire & vehicleat_11 & sparein_21 & sparein_12 & sparein_31
& sparein_13 & sparein_41 & sparein_23 & sparein_14 & sparein_51 & sparein_42 &
sparein_24)

random environment move picked: (!flattire & vehicleat_11 & sparein_21
& sparein_12 & sparein_31 & sparein_13 & sparein_41 & sparein_23 & sparein_14
& sparein_51 & sparein_42 & sparein_24)

current env vars set to true:

vehicleat_11

sparein_21

sparein_12

sparein_31

sparein_13

sparein_41

sparein_23

sparein_14

sparein_51

sparein_42

sparein_24

(the agent is now located in environment state vehicleat_11)

progressed intentions:

before: F(vehicleat_32 & X(ff)) -> after: F(vehicleat_32 & X(ff))

call update operation? [y/n]: n

run continues without update operation

----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1

possible agent moves in current state:

movecar_11_21

movecar_11_12

```
pick an action, using index (nr. from 1 to 2): 2
move selected: movecar_11_12
```

(the environment can pick either of two non-deterministic effects. for displaying purposes, we will not show all environment variables which have remained unchanged)

```
possible environment moves in current state:
(!flattire & vehicleat_12)
(flattire & vehicleat_12)
```

```
random environment move picked: (flattire & vehicleat_12)
```

```
current env vars set to true:
```

```
flattire
vehicleat_12
sparein_21
sparein_12
sparein_31
sparein_13
sparein_41
sparein_23
sparein_14
sparein_51
sparein_42
sparein_24
```

(the agent is now located in environment state vehicleat_12 with a flat tire)

```
progressed intentions:
```

```
before: F(vehicleat_32 & X(ff)) -> after: F(vehicleat_32 & X(ff))
```

```
call update operation? [y/n]: n
```

```
run continues without update operation
```

```
----- NEW LOOP STARTS HERE! -----
```

```
get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1
```

```
possible agent moves in current state:
```

```
changetire_12
```

```
pick an action, using index (nr. from 1 to 1): 1
```

```
move selected: changetire_12
```

```
possible environment moves in current state:
```

```
(!flattire & vehicleat_12 & !sparein_12)
```

```
random environment move picked: (!flattire & vehicleat_12 & !sparein_12)
```

current env vars set to true:

vehicleat_12
sparein_21
sparein_31
sparein_13
sparein_41
sparein_23
sparein_14
sparein_51
sparein_42
sparein_24

(the spare tire in environment state 12 is now removed)

progressed intentions:

before: $F(\text{vehicleat}_{32} \ \& \ X(\text{ff})) \rightarrow$ after: $F(\text{vehicleat}_{32} \ \& \ X(\text{ff}))$

call update operation? [y/n]: y

would you like to add an intention [1], drop [2] an intention, or
stop the entire system [3]?: 1

give new intention: $F(\text{vehicleat}_{41} \ \& \ XF(\text{vehicleat}_{32}))$

set priority using an index (position in the intention list. 0 is
highest priority): 1

trying to add new intention 'F(vehicleat_41 & XF(vehicleat_32))'
to intention list:

(adding the new intention is realizable, so the new goal becomes conjunction of all intentions)

new goal: $F(\text{vehicleat}_{32} \ \& \ X(\text{ff})) \ \& \ F(\text{vehicleat}_{41} \ \& \ XF(\text{vehicleat}_{32}))$

adding new intention is realizable! new intention is adopted and
DFA is recomputed

new formula agent: (not outputted here, as formula is too long)

call update operation? [y/n]: n

run continues without update operation

----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1

possible agent moves in current state:

movecar_12_21

pick an action, using index (nr. from 1 to 1): 1

move selected: movecar_11_21

(the environment can pick either of two non-deterministic effects. for displaying purposes, we will not show all environment variables which have remained unchanged)

possible environment moves in current state:

(!flattire & vehicleat_21)

(flattire & vehicleat_21)

random environment move picked: (flattire & vehicleat_21)

current env vars set to true:

flattire

vehicleat_21

sparein_21

sparein_31

sparein_13

sparein_41

sparein_23

sparein_14

sparein_51

sparein_42

sparein_24

(the agent is now located in environment state vehicleat_21 with a flat tire)

progressed intentions:

before: F(vehicleat_32 & X(ff)) -> after: F(vehicleat_32 & X(ff))

before: F(vehicleat_41 & XF(vehicleat_32)) -> after: F(vehicleat_41 & XF(vehicleat_32))

call update operation? [y/n]: n

run continues without update operation

----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index 2)? [1 or 2]: 1

possible agent moves in current state:

changetire_21

pick an action, using index (nr. from 1 to 1): 1

move selected: changetire_21

(the environment can pick either of two non-deterministic effects. for displaying purposes, we will not show all environment variables which have remained unchanged)

possible environment moves in current state:

(!flattire & vehicleat_21 & !sparein_21)

random environment move picked: (!flattire & vehicleat_21 & !sparein_21)

```
current env vars set to true:
flattire
vehicleat_21
sparein_31
sparein_13
sparein_41
sparein_23
sparein_14
sparein_51
sparein_42
sparein_24

(the spare tire in environment state 21 is now removed)
progressed intentions:
before: F(vehicleat_32 & X(ff)) -> after: F(vehicleat_32 & X(ff))
before: F(vehicleat_41 & XF(vehicleat_32)) -> after: F(vehicleat_41
& XF(vehicleat_32))

call update operation? [y/n]: y

would you like to add an intention [1], drop [2] an intention, or
stop the entire system [3]?: 1

give new intention: F(vehicleat_42 & XF(vehicleat_32))
set priority using an index (position in the intention list. 0 is
highest priority): 2
trying to add new intention 'F(vehicleat_42 & XF(vehicleat_32))'
to intention list:

new goal: F(vehicleat_32 & X(ff)) & F(vehicleat_41 & XF(vehicleat_32))
& F(vehicleat_42 & XF(vehicleat_32))
adding new intention is realizable! new intention is adopted and
DFA is recomputed
new formula agent: (not outputted here, as formula is too long)

(intention 'F(vehicleat_42 & XF(vehicleat_32))' has been added to the intention list)
call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1

possible agent moves in current state:
movecar_21_31

pick an action, using index (nr. from 1 to 1): 1
```

move selected: movecar_21_31

(the environment can pick either of two non-deterministic effects. for displaying purposes, we will not show all environment variables which have remained unchanged)

possible environment moves in current state:

(!flattire & vehicleat_31)

(flattire & vehicleat_31)

random environment move picked: (flattire & vehicleat_31)

current env vars set to true:

flattire

vehicleat_31

sparein_31

sparein_13

sparein_41

sparein_23

sparein_14

sparein_51

sparein_42

sparein_24

(the agent is now located in environment state vehicleat_31 with a flat tire)

progressed intentions:

before: F(vehicleat_32 & X(ff)) -> after: F(vehicleat_32 & X(ff))

before: F(vehicleat_41 & XF(vehicleat_32)) -> after: F(vehicleat_41 & XF(vehicleat_32))

before: F(vehicleat_42 & XF(vehicleat_32)) -> after: F(vehicleat_42 & XF(vehicleat_32))

call update operation? [y/n]: n

run continues without update operation

----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index 2)? [1 or 2]: 1

possible agent moves in current state:

changetire_31

pick an action, using index (nr. from 1 to 1): 1

move selected: changetire_31

possible environment moves in current state:

(!flattire & vehicleat_31 & !sparein_31)

```
random environment move picked: (!flattire & vehicleat_31 & !sparein_31)

current env vars set to true:
vehicleat_31
sparein_13
sparein_41
sparein_23
sparein_14
sparein_51
sparein_42
sparein_24

(the spare tire in environment state 31 is now removed)
progressed intentions:
before: F(vehicleat_32 & X(ff)) -> after: F(vehicleat_32 & X(ff))
before: F(vehicleat_41 & XF(vehicleat_32)) -> after: F(vehicleat_41
& XF(vehicleat_32))
before: F(vehicleat_42 & XF(vehicleat_32)) -> after: F(vehicleat_42
& XF(vehicleat_32))

call update operation? [y/n]: n

run continues without update operation
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1

possible agent moves in current state:
movecar_31_41

pick an action, using index (nr. from 1 to 1): 1
move selected: movecar_31_41

possible environment moves in current state:
(!flattire & vehicleat_41)
(flattire & vehicleat_41)

random environment move picked: (flattire & vehicleat_41)

current env vars set to true:
flattire
vehicleat_41
sparein_13
sparein_41
sparein_23
sparein_14
sparein_51
```

```

sparein_42
sparein_24

(the agent is now located in environment state vehicleat_41 with a flat tire)
progressed intentions:
before: F(vehicleat_32 & X(ff)) -> after: F(vehicleat_32 & X(ff))
before: F(vehicleat_41 & XF(vehicleat_32)) -> after: F(vehicleat_32)
before: F(vehicleat_42 & XF(vehicleat_32)) -> after: F(vehicleat_42
& XF(vehicleat_32))

call update operation? [y/n]: n
----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index
2)? [1 or 2]: 1

possible agent moves in current state:
changetire_41

pick an action, using index (nr. from 1 to 1): 1
move selected: changetire_41

possible environment moves in current state:
(!flattire & vehicleat_41 & !sparein_41)

random environment move picked: (!flattire & vehicleat_41 & !sparein_41)

current env vars set to true:
vehicleat_41
sparein_13
sparein_23
sparein_14
sparein_51
sparein_42
sparein_24

(the spare tire in environment state 41 is now removed)
progressed intentions:
before: F(vehicleat_32 & X(ff)) -> after: F(vehicleat_32 & X(ff))
before: F(vehicleat_32) -> after: F(vehicleat_32)
before: F(vehicleat_42 & XF(vehicleat_32)) -> after: F(vehicleat_42
& XF(vehicleat_32))

call update operation? [y/n]: y

would you like to add an intention [1], drop [2] an intention, or
stop the entire system [3]?: 2

```


which intention should be dropped from the list?: (index 1 until 3)

options:

1 : F(vehicleat_32 & X(ff))

2 : F(vehicleat_32)

3 : F(vehicleat_42 & XF(vehicleat_32))

index chosen: 3

new formula agent: (not outputted here, as formula is too long)

(intention 'F(vehicleat_42 & XF(vehicleat_32))' is dropped, and a new DFA and maximally permissive strategy are computed in the backend of the IMS)

call update operation? [y/n]: n

----- NEW LOOP STARTS HERE! -----

get procrastinating (index 1) or non-procrastinating moves (index 2)? [1 or 2]: 1

possible agent moves in current state:

movecar_41_32

movecar_41_42

pick an action, using index (nr. from 1 to 2): 1

move selected: movecar_41_32

possible environment moves in current state:

(!flattire & vehicleat_32)

(flattire & vehicleat_32)

random environment move picked: (flattire & vehicleat_32)

current env vars set to true:

flattire

vehicleat_32

sparein_13

sparein_23

sparein_14

sparein_51

sparein_42

sparein_24

(the agent is now located in environment state vehicleat_32 with a flat tire)

progressed intentions:

before: F(vehicleat_32 & X(ff)) -> after: X(ff)

before: F(vehicleat_32) -> after: 1

(all intentions are now almost fulfilled. 'X(ff)' implies that there cannot be a next state, so in order to fulfill this also, the agent has to halt the program)

call update operation? [y/n]: y

```
would you like to add an intention [1], drop [2] an intention, or  
stop the entire system [3]?: 3
```

```
program stopped
```

```
Process finished with exit code 0
```