This writing is about an idea on how to solve the intention progression problem (IPP) with various and complex types of goals and norms. We assume that general notions and concepts in BDI agent and IPP are already known so that the following content can focus on our target problem and the corresponding approach.

Problem Description

The problem we want to solve is that when the agent is required to achieve complex of goals and comply complex norms, how should the agent progress (or adopt) its intentions. Here, the meaning of "complex" is related to time and order. Traditionally, a goal specify a certain world state the agent wants to bring about (or maintain if it is a maintenance goal), e.g., visiting a certain place, keeping the floor clean, etc. The goal conditions are usually represented as First-order Logic (FOL) formulas. Although this makes it convenient to implement agent programs, the expression power is limited. In practical applications there may be goals that cannot be captured by basic achievement or maintenance goals. For example, in a Mars rover scenario, the rover agent may have a (complex) goal to collect materials in a certain place and carefully transport them back to a depot. This task is not captured by an achievement or maintenance goal. Rather, it is better modelled by a combination of achievement and maintenance goals: the agent first tries to visit the specified place for collecting materials (achievement goal), then try to go back to depot (achievement goal); during the time when the agent is transporting, it must maintain its speed below a certain level in order to prevent the materials from being damaged (maintenance goal). In practical BDI agent programming languages and platforms, it is hard to design a plan for such complex goals. It is usually the case that complex goals are separated into multiple basic goals and their related plans is separately designed or generated.

Although it is possible to directly implement a complex goal using combination of basic goals, encoding the rules and order in these goals is hard, especially when the complex goal is really complex. Dastani[1] proposed a uniformed approach to represent complex goals using linear temporal logic (LTL) formulas (they call temporal goals) and described a mechanism for translating temporal goals to basic goals for practical usage considerations. However, their mechanism is hard-coded, only provides transformation rules for common forms of LTL formula; other forms of LTL formula require the user to manually handle. Besides, they only consider the translation from LTL formulas to basic goals, the problems about what plans to adopt and how to interleave multiple intentions are not considered. Although it is possible to separate the concern of LTL translation and the scheduling of the active intentions, it may cause the scheduler to lose the information about how the LTL formulas can be satisfied and consequently trapped in local optima.

The idea of using LTL to represent complex goals also apply to norms. We argue that the scheduler should take LTL formulas as input objects and decides what basic goals and norms should be adopted, what plans to execute and how to execute them to satisfy these LTL formulas (we assume that a plan library is already provided).

1 Representation and Method

Here we first define the environment and its state transitions. Based on which, we can give the formal representation of complex goals and norms using LTL.

Environment

Gutierrez et al.[2] described a model to represent environment and its state transitions. Here, based on their work, we represent an environment as follows.

We assume that the agent acts in an environment that can be in any state of a set $\mathcal{S}$ containing all possible environment states[1]. The environment starts from state s_0. The agent has a set of actions $\mathcal{A}$. The actions in $\mathcal{A}$ can be performed when their preconditions are satisfied. The postconditions of an action $a \in \mathcal{A}$ in a state $s \in \mathcal{S}$ is transforming s into another state s'. This is determined by a transformer function $\mathcal{T}(s, a) = s'$ (we assume that the transformation is deterministic). Here, we focus on the environment state and its transitions. The agent itself and its internal states are not included in the following representation, only the agent's actions that directly affect the environment states are considered.

Formally, an environment is defined by a structure $E = (\mathcal{S}, \mathcal{A}, \mathcal{T}, s_0)$ where:

- $\mathcal{S}$ is a non-empty finite set of states containing all possible environment states.
- $\mathcal{A}$ is the set of actions that the agent can perform.
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow 2^{\mathcal{S}}$ is the transformer function.
- $s_0 \in \mathcal{S}$ is the initial state of the environment.

An agent follows its strategy to execute actions until there is no action executable or all its goals are achieved. In this way, the action executions in an environment traces out an interleaved sequence of environment states and actions. We call this sequence a *run*:

$$\rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_n$$

Where ρ is a run, we use $si(\rho, u)$ to index the uth state and $ai(\rho, u)$ to index the uth action in ρ (u is a natural number). We assume a run ρ is finite, i.e., there is a final state s_n representing the end of ρ.

[1]We assume that each environment state is identified by a set of ground (first-order) atomic formulas that describe the environment.

LTL Representation

LTL is a modal temporal logic with modalities referring to time. In order to represent complex goals and norms by LTL (we call LTL object), in addition to conventional propositional logic operators we consider 3 tense operators: $\Box$ ("always..."), $\Diamond$ ("eventually...") and U ("...until...").

Each LTL object delivered to the agent corresponds to a basic goal or an organized combination of basic goals. We assume that the user gives information about what reward the agent can get if The LTL is satisfied. The following are some expamles on the relationship between LTL objects and basic goals or norms.

An LTL object $\Diamond s_g$ corresponds to an achievement goal for reaching a world state s_g. This means that the agent is required to bring about s_g eventually at some point in time. $\Box s_m$ corresponds to a maintenance goal for always maintaining s_m. $\Box(\tau \rightarrow (\Diamond done(a) \mathsf{U} \mu))$ corresponds to a prohibition norm where τ is the activation condition and μ is the deactivation condition, $done(a)$ denotes the fact that action a is performed. The triggering condition means that after τ is satisfied, performing action a before μ is satisfied triggers the norm and will receive value of Val. Val can be positive or negative indicating either an obligation norm (the agent is required to perform a) or a prohibition norm (the agent is prohibited to perform a). We assume that the norms are implemented as soft constraints, thus not limiting the agent's flexibility. The agent can choose obeying or violating the norm accordingly with respect to the value Val and its own strategy.

The complex goal example mentioned in the previous section can be represented by LTL as:

$$\Diamond c \wedge (\Diamond c \rightarrow \Diamond t) \wedge \Box(c \rightarrow (sUt))$$

where, c is collecting materials, t is transporting materials back to depot and s is slowing down the speed. This means that the agent is required to first adopt an achievement goal to collect materials (c); once it is achieved, the agent needs to adopt another achievement goal to transport the materials back to depot ($\Diamond c \rightarrow \Diamond t$); during the period of transporting, the agent needs to keep its speed slow ($\Box(c \rightarrow (sUt))$).

Our Approach

The state-of-the-art approaches to IPP is *SA* which is based on Monte-Carlo Tree Search (MCTS). However, *SA* only considers achievement goals; in the simulation phase of *SA* , the total value obtained for achieving goals is returned at the end of every simulation run. Because the goal condition is defined as a set of propositional literals, the checking process is rather strait straightforward: if the goal condition is satisfied, the goal is achieved. However, when LTL goals and norms (we call LTL objects) are included, we need extra algorithms to check whether a given LTL formula is satisfied.

LTL checker

In order to support intention scheduling with LTL objects, we first need a mechanism to check whether a run (or partial run) satisfies a given LTL formula. Here, we describe a basic framework to check whether an LTL is satisfied in a given run ρ (a sequence of state transitions).

The basic framework of the LTL checker are shown in figure 1. The checker

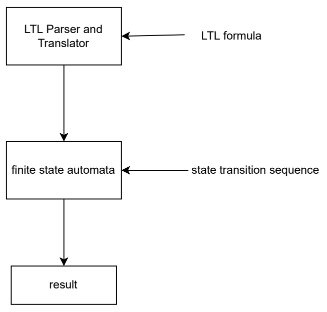

Figure 1: LTL checker

first uses LTL Parser and translator to translate an LTL formula to its equivalent automaton, then the state transition sequence is checked by the automaton.

The LTL parser and translator are already implemented in [3] and [4]. In [3], the automata are specifically for checking infinite state sequences while in [4], the automata are for checking finite state sequences.

This checker can be used to check any types of LTL objects. If multiple LTL objects need to be checked, two options are available. One is to process each LTL formula at a time and the other is to use conjunction to represent multiple formulas as one formula and then be processed by the checker.

The algorithm for checking a LTL object o in a run ρ is shown in algorithm 1. The checker takes 2 parameters as input: a run ρ and a LTL object. The checker first gets the LTL formula and transform it into an automaton (line 2-3). Then, each state in ρ is iteratively processed by the automaton (line 4-9): if the automaton is transitable in the current checking state, the transition is applied. Otherwise, the automaton doesn't accept current state. After l is processed, the checker then checks whether the automaton is in the final accept state, and return the corresponding result (line 10-13).

Here, we describe the mechanism of how to adopt basic goals based on given LTL formulas. Assuming that an agent is initially allocated with a set of LTL objects; before its execution, the agent first transforms each of the LTL formula into an automaton (automata are encoded as a part of the agent's internal state). When the agent is in execution, each time the agent's belief base is updated, all automata are updated (state transition applied) according to the agent's current beliefs and basic goals or norms are adopted according to the current state of each automaton. For each automaton, there is a path from current state to the final accept state, the agent needs to decide which basic goal or norm should

Algorithm 1 LTL Checker

```
1: procedure CHECK($\rho, o$)
2:     $l \leftarrow getLTLformula(o)$
3:     $\alpha \leftarrow translate(l)$                ▷ translate the LTL to an automaton
4:     for $i \leftarrow 0, len(\rho) - 1$ do
5:         $s \leftarrow si(\rho, i)$
6:         if $\alpha.isTransitable(s)$ then
7:             $\alpha.transit(s)$
8:         else
9:             return $false$
10:    if $\alpha$ is in its final state then
11:        return $true$
12:    else
13:        return $false$
```

be adopted in order to reach the final state. If there is no transition rule to be applied for the current state in an automaton, the automaton doesn't change.

Figure 2 illustrate the process of the state transitions in an automaton and the resulting basic goal adoption (for the purpose of exemplification, we ignore the failure state). The automaton is translated from the previous complex goal example $\Diamond c \wedge (\Diamond c \rightarrow \Diamond t) \wedge \Box(c \rightarrow (sUt))$ where, c is collecting materials, t is transporting materials back to depot and s is slowing down the speed. This complex goal states that the agent first needs to visit the specified place for collecting materials, then try to transport the materials to the depot; during the time when the agent is transporting, it must maintain its speed below a certain level in order to prevent the materials from being damaged. Initially, the agent is in state q_1, in order to follow the path to final accept state, the agent first try to reach state q_2 by adopting an achievement goal to satisfy c (materials collected), after c is achieved, the current state is changed to be q_2. In q_2 there is an arrow pointing towards itself, the agent adopts a maintenance goal for maintaining s (keep the speed slow); the agent also adopts an achievement goal to satisfy t (materials transported) for reaching the final accept state. After t is satisfied, the automaton is in the final accept state, indicating that the complex goal is achieved.

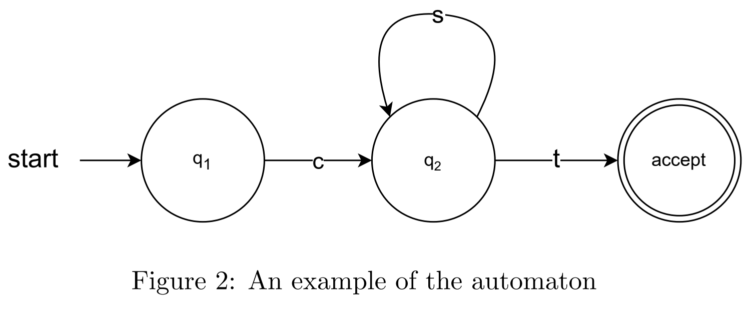

Figure 2: An example of the automaton

Algorithm

To support intention scheduling with LTL objects we modify the expansion and simulation phase of SA . While norms are not considered in the following algorithms, incorporating them is straightforward.

Algorithm 2 Expansion

```
1: input:$(n_s)$
2: $N \leftarrow \emptyset$
3: $s \leftarrow getWorldState(n_s)$
4: $AT \leftarrow getAutomata(n_s)$
5: $I \leftarrow getIntentions(n_s)$
6: for each $auto \in AT$ do
7:     $cas \leftarrow auto.currentState()$
8:     for each $nas \in auto.nextStates()$ do
9:         $ws \leftarrow targetWorldState(cas, nas)$
10:        if a basic goal for achieving or maintaining $ws$ is already adopted then
11:            $continue$
12:        if $nas == cas$ then
13:            adopt a maintenance goal for maintaining $ws$ and update $I$
14:        else
15:            adopt an achievement for achieving $ws$ and update $I$
16: for each $i \in I$ do
17:     $Act \leftarrow getExecutableActions(i)$
18:     for each $a \in Act$ do
19:         $i.progress(a)$
20:         $s_e \leftarrow s.apply(a)$
21:         for each $auto \in AT$ do
22:             if $auto.isTransitable(s_e)$ then
23:                 $auto.transit(s_e)$
24:         $n \leftarrow newNode(s_e, AT, I)$
25:         $N \leftarrow N \cup \{n\}$
26: $n_s.setChildren(N)$
27: for each $n_i \in N$ do
28:     $n_i.setParent(n_s)$
```

Modifications to the expansion phase In the expansion phase, a selected leaf node n_s is expanded according to the agent's current executable actions. Before checking executable actions in each intention, the agent needs to determine what new basic goals should be adopted according to current automaton (line 6-15): for each automaton, the agent first identifies current state of the automaton cas, then check each possible next state nas and (by calling the function $targetWorldState(...)$) get the target world state[2] ws, **which the agent needs to satisfy in order to transit from** cas **to** nas (line 7-9). Whether to adopt a new basic goal and what basic goal to adopt are determined by the relationship of cas and nas (line 10-15). Next, the agent checks each of its intention $i \in I$, progress i according to each of the executable actions $a \in Act$ in i. The postconditions of a are applied on s to get the resulting world state s_e, and s_e is used to update each automaton $auto \in AT$ (line 20-23). Then, each new node is generated according to the updated s_e, AT and I(line 24). Finally, all new nodes are set as the children of n_s (line 26-28).

[2]Automata state and world state are different, how an automaton transit its state is determined by the world state and not the other way around.

Algorithm 3 Simulation

```
1: input:($n_e$)
2: $I \leftarrow getIntentions(n_e)$
3: $s \leftarrow getWorldState(n_e)$
4: $AT \leftarrow getAutomata(n_e)$
5: **while** $true$ **do**
6:     **for** each $auto \in AT$ **do**
7:         $cas \leftarrow auto.currentState()$
8:         **for** each $nas \in auto.nextStates()$ **do**
9:             $ws \leftarrow targetWorldState(cas, nas)$
10:            **if** a basic goal for achieving or maintaining $ws$ is already adopted **then**
11:                $continue$
12:            **if** $nas == cas$ **then**
13:                adopt a maintenance goal for maintaining $ws$ and update $I$
14:            **else**
15:                adopt an achievement for achieving $ws$ and update $I$
16:    **if** $I$ is empty **then**
17:        $break$
18:    $i \leftarrow null$
19:    **while** $I$ is not empty **do**
20:        $i_r \leftarrow random(I)$
21:        $I \setminus \{i_r\}$
22:        **if** $i_r$ is executable **then**
23:            $i \leftarrow i_r$
24:            $break$
25:    **if** $i$ is $null$ **then**
26:        $break$
27:    $a \leftarrow$ randomly select an executable action in $i$
28:    $s \leftarrow s.apply(a)$
29:    **for** each $auto \in AT$ **do**
30:        $auto.transit(s)$
31: $n_t \leftarrow newNode(s, AT, I)$
       **return** $n_t$
```

Modifications to the simulation phase In the simulation phase, executable actions are randomly executed and automata are updated accordingly until a terminal state is reached. More specifically, for a randomly selected node n_e, the agent get all necessary components for simulation at start (line 2-4). At each deliberation cycle (inside the while loop), the agent first checks what new basic goals should be adopted according to current automata (line 6-15). Then, it randomly selects an executable intention i (if there is no executable intention, terminal state is reached(line 25-26)) and randomly select an executable action a for execution (line 27-28). The resultant state s is updated and s is used to update each automaton $auto \in AT$ (line 29-30). Finally, after the control flow jumps out from the deliberation loop cycle (reach a terminal state), a new node is generated and returned according to s, AT and I (line 31).

References

[1] Mehdi Dastani, M. Birna van Riemsdijk, and Michael Winikoff. Rich goal types in agent programming. In Liz Sonenberg, Peter Stone, Kagan Tumer, and Pinar Yolum, editors, *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan, May 2-6, 2011, Volume 1-3*, pages 405–412. IFAAMAS, 2011.

[2] Julian Gutierrez, Sarit Kraus, Giuseppe Perelli, and Michael J. Wooldridge. Giving instructions in linear temporal logic. In Alexander Artikis, Roberto Posenato, and Stefano Tonetta, editors, *29th International Symposium on Temporal Representation and Reasoning, TIME 2022, November 7-9, 2022, Virtual Conference*, volume 247 of *LIPIcs*, pages 15:1–15:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[3] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[4] Samuel Huang and Rance Cleaveland. A tableau construction for finite linear-time temporal logic. *J. Log. Algebraic Methods Program.*, 125:100743, 2022.