Planning in a Normative System

Guilherme Krzisch** and Felipe Meneguzzi

School of Computer Science
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, Brazil

guilherme.krzisch@acad.pucrs.br,felipe.meneguzzi@pucrs.br

Abstract. A social norm describes a standard of behavior expected to be followed by agents in a given society, and failure to comply results in sanctioning and loss of utility for the violating agent. Agents need to take existing norms into account when generating plans to achieve their goals, to either forestall potential violations if an agent wants to be fully norm compliant, or understanding the effects on its utility if violations are acceptable. In this paper we model a normative system in terms of classical planning, and develop two norm formalizations: the first concerns actions in a given context; while the second concerns sequences of states and is based on Linear Temporal Logic. We use these norm formalizations to develop different planning approaches that take into account such norms, and empirically evaluate the algorithm's performance.

Keywords: normative system, classical planning, multi-agent system

1 Introduction

Multi-agent systems allow the design of complex behavior in terms of multiple autonomous agents that interact in a shared environment, e.g. e-commerce systems [17, Ch. 24]. As the agents are assumed to be self-interested, conflicts may arise, due to agents performing actions that do not consider the impact on other agents or on the system, which in turn can lead to undesirable system behavior [5]. In this context norms can be used to enforce desirable system behavior by encoding expected individual agent behavior, whose failure to comply leads to negative incentives, i.e. sanctions [14, Ch. 14]. Such norm systems aim to maximize individual agent autonomy, as norms can be violated, but require agents to reason about the consequences of their actions in terms of norm-compliance.

Much effort has been made towards developing norm-driven reasoning in the context of agents driven by a plan library [8, 9, 13, 1, 11, 10], however behavior generation using plan libraries substantially limits the flexibility of an agent's behavior to what is explicitly encoded in the library [7]. By contrast, generating behavior using first principle planners allows greater freedom to achieve goals by exploring the state-space of the environment rather than the space of

** This work is partially supported by grant from CNPq/Brazil (132339/2016-1).

plans described in the plan library [7]. Relatively less effort has been made to develop agents capable of performing first-principles planning while taking into account the impact of such plans relative to the norms in an environment [16, 15]. Behavior generation via first principles planning poses, in practice [4], a more computationally intensive problem. Consequently, computing the normative consequences of behaviors generated in such a way is a much harder problem.

In this work we develop a first-principles planning algorithm that takes into account the consequences of actions in terms of norm compliance and violation[1]. We model the system in terms of classical planning, and present two norm formalizations. Using these formalizations we describe two approaches for planning, one using the well-known Graphplan algorithm and the other based on a forward state-space search. We evaluate these approaches in a simple illustrative domain and in the *blocksworld* domain, and discuss the algorithms with related work.

2 Formalization

In this section we describe the formalization used to describe our problem. We first we give an overview of classical planning, in order to model the system; and then formalize two alternative types of norms.

2.1 Classical Planning

We use the classical planning formalism to represent the environment in which the agent reasons about norms. Specifically we use the Planning Domain Definition Language (PDDL) [6] formalism to describe the domain and the corresponding problems. A PDDL domain consist of the available predicates and actions, while a PDDL problem specifies the initial and goal states, as well as the objects in the environment. The planning process in this context consist of finding the sequence of actions that leads the agent from the initial to the goal state. Subsequent versions of the PDDL allows the attribution costs to actions; however, in this work, we will assume that all action costs are *one*.

2.2 Normative Model

Norms specify expected behavior of agents in a system. There are many different ways to formalize norms and most of them use deontic logic modalities to express a norm type, i.e. if a norm is an obligation, a permission or a prohibition. In our work we support two different types of norm formalization. The first one is adapted from [12], shown in Definition 1.

Definition 1 (Conditional Norm). *A conditional norm is a tuple* $n = \langle \mu, \chi, \rho, C \rangle$*, where:*

- $\mu \in \{obligation, prohibition\}$ *represents the norm's modality;*

[1] Source code and data set available at github.com/guilhermekrz/KPlanning.

- χ *is a set of ground predicates that represents the* context *to which a norm applies, i.e. a norm is applicable in state* s *if* $s \models \chi$*;*
- ρ *is an action, representing the* object *of the norm's modality;*
- C *is the* penalty *(cost) incurred to an agent when this norm is violated.*

Example 1. The following conditional norm requires an agent to drive on the left side of the road if they are in England.

$$cn = \langle obligation, at(England), driveLeft(a, b), 20 \rangle$$

Definition 2 describes when a conditional norm is violated by an agent.

Definition 2 (Conditional Norm Violation). *A norm* $n = \langle \mu, \chi, \rho, C \rangle$ *is violated in state* s *by an agent* a *iff* $s \models \chi$ *and agent* a *either: executes action* ρ *in state* s *and* $\mu = prohibition$*; or does not execute action* ρ *in state* s *and* $\mu = obligation$*.*

The second type of norm uses a subset of Linear Temporal Logic formulae expressed using modal operators of PDDL3 described by Gerevini *et al.* [3], and formalized in Definition 3.

Definition 3 (LTL Norm).
An LTL norm is expressed using one of the following modal operators, all implicitly representing an obligation, where ϕ *and* ψ *are atomic formulae and* t *is a number*[2]*:*

- *(at end* ϕ*) -* ϕ *must be true in the final state*
- *(always* ϕ*) -* ϕ *must be true in all states in the plan*
- *(sometime* ϕ*) -* ϕ *must be true in at least one state in the plan*
- *(at-most-once* ϕ*) -* ϕ *must be true in at most one state in the plan*
- *(sometime-after* ϕ ψ*) - whenever* ϕ *is true in a state* s*, there must be a state* s' *equal to or after* s *where* ψ *is true*
- *(sometime-before* ϕ ψ*) - whenever* ϕ *is true in a state* s*, there must be a state* s' *before* s *where* ψ *is true*
- *(always-within* t ϕ ψ*) - whenever* ϕ *is true in a state* s*, there must be a state* s' *at most* t *steps after* s *where* ψ *is true*

Although modal operators can be nested, in the current work we simplify the expressivity of the language consider only flat operators. An LTL norm is violated if its interpretation is not true in a given plan.

These two norm formalizations have different complexities. While the first one can be checked in a single state, the second one needs to be checked along a path (i.e. a sequence of states, a finite trajectory). The next section describes different methods to find solutions given these norm formalization.

[2] For brevity, we use semi-formal descriptions for the modal operators adapted from PDDL3; for the full formal definitions we refer to [3]

3 Planning with Norms

A planner is responsible for finding a sequence of actions that leads from the initial to the goal state. When considering norms, this solution can be either norm-compliant or norm-violating[3]; the planner can also find solutions that minimize the cost of both actions and penalty costs relative to norm violations. In the next subsections we first present a scenario to motivate our problem, and then describe approaches based on the Graphplan algorithm and using forward state-space search to find such norm-driven plans.

3.1 Scenario

In this section we describe an example of a scenario with norms in order to provide a motivation for this work; we call this scenario *drinkdriving*, and use it to perform our experiments. In this scenario agents can move between locations; in some locations there are bars available, where agents can enter, exit, or become drunk; agents can sleep to become sober again. In order to avoid car crashes due to agents driving while drunk, there is a norm forbidding agents perform such behaviors, and an associated arbitrary sanction.

3.2 Graphplan

The Graphplan algorithm [2] uses a planning graph data structure to perform its planning process. The planning graph is a leveled graph, interleaving levels of predicates and levels of actions; the algorithm uses this graph in order to perform a backward search. It also encodes information about mutual exclusion relations between predicates and between actions, which provides information to prune partial solutions that do not lead to a valid solution.

The simplest way to modify the Graphplan algorithm is to discard solutions if they violate (do not violate) the norms in the case of norm-compliant (norm-violation) planning. In order to do this we perform the solution extraction phase of the algorithm to find all possible solutions at the current level; we then iterate through each solution checking for norm compliance or violation.

The solution as outlined above does not take advantage of the fact that it is possible to prune partial solutions during the backward search phase, which improves the planning process efficiency. Algorithm 1 shows the solution extraction phase of the Graphplan algorithm, modified to backtrack when the partial solution cannot be a solution in relation to the set of norms.

In Line 8 of the algorithm we check if we can ascertain if the partial solution violates a norm; we are able to determine a norm violation if we can check the truth value of its context. As we only have partial information, the *newGoal* set of literals must include the literals of the norm context. For conditional norms, this is trivial; a norm violation can be checked at a single state, and we can

[3] Agents may want to violate norms, for example, to collect information of the sanctions of the existing norms when entering in a system.

propagate this information while doing the backward search. However, for LTL norms, a complex mechanism to keep track of the current norm status is required; we would need to keep track of the norm activation and deactivation conditions, propagating this information backwards and backtracking when necessary. The Graphplan structure and its backward search makes this tracking complex, and the resulting solution time and space consuming; therefore Algorithm 1 does not check LTL norms, and in the next section we present another planning approach based on a forward state-space search which is more suitable to plan with respect to this type of norm.

Having the information that a partial plan partially violates a norm, we can stop searching and begin to backtrack in two situations. The first, when we are searching for a *norm compliant* plan, is when it already occurred a violation (Line 9). The second, related to *norm violation* planning, is when we reached the first level and it has not occurred any violations; in this situation we could not have backtracked before because a partial plan that has no violations is still a candidate solution to *norm violation* planning, as this violation can occur at a later step (Line 3).

Algorithm 1 Solution Extraction Phase of the Normative Graphplan

```
1:  procedure SolutionExtraction(goal,level,isViolationPlan,type)
2:      if level=0 then
3:          if type = NormViolating and not isViolationPlan then backtrack
            return solution
4:      for each literal in goal do
5:          nondeterministically choose an action to achieve literal
6:      if set of chosen actions are mutex then backtrack
7:      newGoal ← preconditions of chosen actions
8:      newIsViolationPlan ← isViolation(chosen actions, newGoal)
9:      if type = NormCompliant and newIsViolationPlan then backtrack
10:     SolutionExtraction(newGoal, level − 1, newIsViolationPlan, type)
```

3.3 Forward state-space search

The second way in which we perform norm-aware planning is using forward state-space search. We use a Uniform-Cost Search (UCS) as a base algorithm, and modify it to take norms into consideration; this allows us, for example, to return solutions that minimize the combined cost of actions and penalties for norm violations. We choose UCS over A^* because it does not need any heuristics for its planning process; calculating heuristics in norm-aware planning is harder than in classical planning because it has to take into account norm penalties.

Algorithm 2 shows our modifications of the UCS algorithm. To guarantee norm-compliant or norm-violation solutions, we need to check if the partial plan violates a norm before testing if the *node* is a goal node (Lines 5 - 12) and before

Algorithm 2 Uniform-Cost Search with Norms

```
1: function UCS(problem,type)
2:     frontier ← add node with initial state
3:     while frontier.hasElements() do
4:         node ← frontier.pop()
5:         if isGoal(node) then
6:             if type = NormCompliant and node is not current violation then return solution(node)
7:             else if type = NormViolating and node is current violation then return solution(node)
8:             else if type = MinCost then
9:                 if node has already the correct cost then return solution(node)
10:                else
11:                    node.cost ← cost from actions and norm violation penalties
12:                    frontier.put(childNode)
13:        explored.add(node)
14:        for all applicable actions in node do
15:            childNode ← node.apply(action)
16:            if childNode is not in frontier and explored or childNode is in frontier with higher cost then
17:                if type = NormCompliant and childNode is not absolute violation then
18:                    frontier.put(childNode)
19:                else if type = NormViolating then
20:                    frontier.put(childNode)
21:                else if type = MinCost then
22:                    if childNode is absolute violation then
23:                        childNode.cost ← cost from actions and norm violation penalties
24:                    frontier.put(childNode)
```

adding the *childnode* to the frontier (Lines 16 - 24). For this, we introduce two concepts: *absolute violation* and *current violation*, which refer to partial plans. The first one indicates that there is no possibility that the current partial plan will be norm-complaint again, and the second one means that it is currently violating some norm, but there is a possibility that, in the future, it will no longer violate the norm. An example for the first concept is a partial plan where an atomic-formula ϕ from a LTL norm *(at-most-once ϕ)* is true in two intermediate states; an example for the second is a partial plan where ϕ from a LTL norm *(sometime ϕ)* is not true in any intermediate states so far.

If we are interested in norm-compliant plans, we discard nodes whose partial plans are *absolute violations* (Line 17); furthermore, *current violation* nodes are added to the frontier (Line 17), but they cannot be tested for goal condition (Line 6). For norm-violation plans, all nodes are added to the frontier (Line 19), but only *current violation* nodes are tested for goal condition (Line 7).

Finally, if we want plans that minimize cost (from actions and norm violations), we add *absolute violation* nodes to the frontier with the cost from actions

plus the cost of the penalty of the violated norms (Line 23). Note that the cost of *current violation* norms are not counted when adding the respective node to the frontier, because then the returned solution would not necessarily be optimal; as these norms can be fulfilled at a later time, we cannot assume their penalties yet. The cost of *current violation* norms are indeed counted if they are a goal node; in this case, this node is added back to the frontier with its cost updated with the norm penalties (Lines 11 - 12). For example, if there is a goal node g in the frontier with pending *current violations*, its cost is updated to take into account the norm penalties, and this node is added back to the frontier. If another goal node h, with no pending violations, has less cost than node g, then node h is the final solution with minimal cost; otherwise, node g is removed again from the frontier, this time without pending violations, and is the final solution. In this way, the algorithm guarantees the return of the solution with minimal cost.

In this paper we propose two methods to check if a node is *absolute violation* or *current violation*. The first method builds the current partial plan from the chosen actions from each node; with this partial plan we can check for norm-violation. The second method keeps track of norm-violation information in each node; when a new node is created, we update it with the current action. While the first approach does not keep information, and always has to rebuild the partial plan for each node, being more CPU intensive, the second approach keeps information, and thus it is more memory intensive.

4 Experiments and Results

We performed experiments in the *blocksworld* domain, from the International Planning Competition, and in the *drinkdriving* domain, described previously. We ran each planner on problems of increasing complexity and number of norms. In this section we refer to the first version of the planer using Graphplan in a normative context described in Section 3.2 as *Naive Graphplan*, and *Graphplan* for the second solution described in Algorithm 1; for the forward-state space planners, described in Algorithm 2, we refer as *Forward 1* and *Forward 2* the planners using the first and second method to check node violation.

In Figure 1 we show the results for the four implemented planners, in the *blocksworld* domain, considering conditional norms and trying to find norm-compliant solutions[4]. Around problem number 50, all but the *Graphplan* planner failed to return solutions, either due to timeout or memory constraints. The *Graphplan* planner has a relatively low running time compared to the other approaches, only increasing its time for the largest problems.

In the *drinkdriving* domain, shown in Figure 2, we obtained different results. The forward planners[5] remained with a lower running time than both Graphplan planners. There are two possible explanations for this difference: the *blocksworld*

[4] The figures show smoothed results, using a sample of 100 data points interpolated using splines

[5] *Forward 2* has a similar time performance than *Forward 1*, and was omitted for clarity

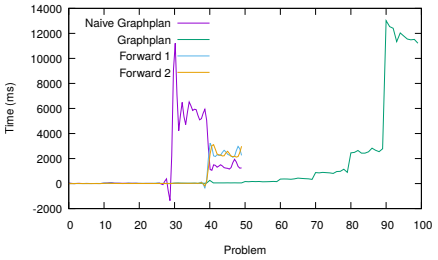

Fig. 1. Time performance for norm-compliant planners considering conditional norms, in the *blocksworld* domain

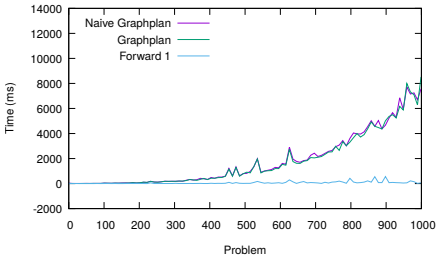

Fig. 2. Time performance for norm-compliant planners considering conditional norms, in the *drinkdriving* domain

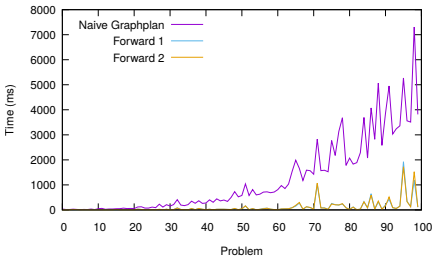

Fig. 3. Time performance for norm-violation planners considering LTL norms, in the *drinkdriving* domain

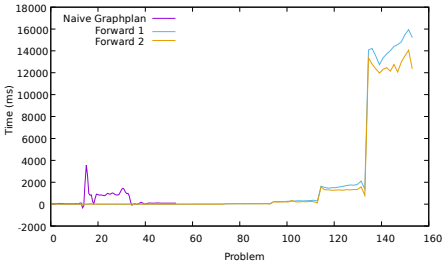

Fig. 4. Time performance for norm-compliant planners considering LTL norms, in the *blocksworld* domain

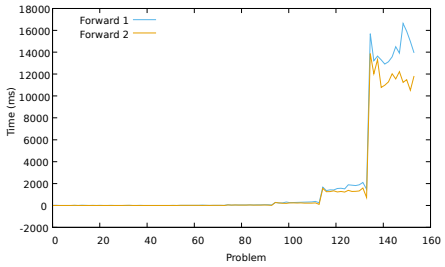

Fig. 5. Time performance for minimum cost planners considering LTL norms, in the *blocksworld* domain

domain lends itself more to parallelization than the *drinkdriving* domain or the problems used in our experiments for the *drinkdriving* domain are less complex than the problems from the *blocksworld* domain.

Figures 3, 4 and 5 show the results of experiments using LTL norms. *Naive Graphplan* failed to return solutions for all but the small problems in the *blocksworld* domain. The *Forward 1* and *Forward 2* planners achieved a similar time performance result, with an advantage to the *Forward 2* planner on large problems. As our hypothesis is that the second forward approach uses more memory than the first one (in order to have a better time performance), we measured the total memory allocated by each planner and obtained results showing that the second approach is more memory intensive. However, these results are not conclusive, as this measure is only approximate, and the data obtained exhibited high variance.

5 Related Work

Previous work have considered the problem of planning in a normative environment. In Panagiotidi et al. [16] they formalized norms with activation, deactivation, maintenance and repair conditions; in order to find a plan they introduced a special action responsible to check norm compliance at each intermediate state. In this way, they take advantage of the well-established PDDL language and can use existing planners to find norm-compliant plans. The main drawback of this work is the use of the *forall* command in PDDL to iterate through each possible combination of predicates in order to check norm compliance; this makes the number of possible combination of predicates scales exponentially for larger problems, and thus make the proposed approach feasible only for small problems.

In Panagiotidi et al. [15] norms are specified in Linear Temporal Logic (LTL), and they use TLPlan as their base planner. As in the above work, the resulting planner is only able to return norm-compliant plans and lacks the ability to minimize violations in case no norm-compliant plan is possible.

6 Conclusion

In this work we developed different approaches to plan in a normative system; more specifically, two based on the Graphplan algorithm and two based on forward state-space search. The normative system consists of classical planner and either conditional norms or norms based on Linear Temporal Logic. We performed experiments using two domains, with different problems and norms, and analyzed the results. While existing work performs norm-compliant planning, our approach is also able to perform norm-violating and minimizing cost planning.

For future work we intend to perform experiments with more complex problems, to get a better understanding of the advantages and disadvantages of our approaches. We want to support more expressive norms, e.g. LTL norms with nested operators. We also intend to investigate whether a Graphplan-based approach can support the full spectrum of LTL-based norms, and to propose modifications on different search algorithms, e.g. Iterative deepening depth-first search or A^*. Finally, we aim to develop a translation schema to compare our approach to the related work of Panagiotidi et al. [16, 15].

References

1. Alechina, N., Dastani, M., Logan, B.: Programming norm-aware agents. In: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems, International Foundation for Autonomous Agents and Multiagent Systems (2012) 1057–1064
2. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. Artificial intelligence **90**(1) (1997) 281–300
3. Gerevini, A., Long, D.: Plan constraints and preferences in PDDL3. The Language of the Fifth International Planning Competition. Tech. Rep. Technical Report, Department of Electronics for Automation, University of Brescia, Italy **75** (2005)
4. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: theory and practice. Elsevier (2004)
5. Jennings, N.R.: Commitments and conventions: The foundation of coordination in multi-agent systems. The knowledge engineering review **8**(3) (1993) 223–250
6. McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: Pddl-the planning domain definition language. (1998)
7. Meneguzzi, F., De Silva, L.: Planning in bdi agents: a survey of the integration of planning algorithms and agent reasoning. The Knowledge Engineering Review **30** (1 2015) 1–44
8. Meneguzzi, F., Luck, M.: Norm-based behaviour modification in BDI agents. In: Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems. (2009) 177–184
9. Meneguzzi, F., Oren, N., Vasconcelos, W.: Using constraints for norm-aware BDI agents. In: The Fourth Annual Conference of the International Technology Alliance, London, UK (2010)
10. Meneguzzi, F., Rodrigues, O., Oren, N., Vasconcelos, W.W., Luck, M.: BDI reasoning with normative considerations. Engineering Applications of Artificial Intelligence **43**(0) (2015) 127 – 146
11. Meneguzzi, F., Vasconcelos, W., Oren, N., Luck, M.: Nu-bdi: Norm-aware bdi agents. In: European Workshop on Multiagent Systems. (2012)
12. Oren, N., Panagiotidi, S., Vázquez-Salceda, J., Modgil, S., Luck, M., Miles, S.: Towards a formalisation of electronic contracting environments. In: Coordination, organizations, institutions and norms in agent systems IV. Springer (2009) 156–171
13. Oren, N., Vasconcelos, W., Meneguzzi, F., Luck, M.: Acting on norm constrained plans. In: Computational Logic in Multi-Agent Systems, 11th International Workshop. Number 6814 in LNCS (2011) 347–363
14. Ossowski, S.: Agreement technologies. Volume 8. Springer Science & Business Media (2012)
15. Panagiotidi, S., Alvarez-Napagao, S., Vázquez-Salceda, J.: Towards the norm-aware agent: Bridging the gap between deontic specifications and practical mechanisms for norm monitoring and norm-aware planning. In: International Workshop on Coordination, Organizations, Institutions, and Norms in Agent Systems, Springer (2013) 346–363
16. Panagiotidi, S., Vázquez-Salceda, J.: Norm-aware planning: Semantics and implementation. In: Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology-Volume 03, IEEE Computer Society (2011) 33–36
17. Van der Hoek, W., Wooldridge, M.: Multi-agent systems. Foundations of Artificial Intelligence **3** (2008) 887–928