Norm Identification through Plan Recognition

Nir Oren[1] and Felipe Meneguzzi[2]

[1] Department of Computing Science, University of Aberdeen
Aberdeen, AB24 3UE, Scotland
`n.oren@abdn.ac.uk`
[2] School of Computer Science, Potifical Catholic University of Rio Grande do Sul
Porto Alegre, 90619-900, Brazil
`felipe.meneguzzi@pucrs.br`

Abstract. Societal rules, as exemplified by norms, aim to provide a degree of behavioural stability to multi-agent societies. Norms regulate a society using the deontic concepts of permissions, obligations and prohibitions to specify what can, must and must not occur in a society. Many implementations of normative systems assume various combinations of the following assumptions: that the set of norms is static and defined at design time; that agents joining a society are instantly informed of the complete set of norms; that the set of agents within a society does not change; and that all agents are aware of the existing norms. When any one of these assumptions is dropped, agents need a mechanism to identify the set of norms currently present within a society, or risk unwittingly violating the norms. In this paper, we develop a norm identification mechanism that uses a combination of parsing-based plan recognition and Hierarchical Task Network (HTN) planning mechanisms, which operates by analysing the actions performed by other agents. While our basic mechanism cannot learn in situations where norm violations take place, we describe an extension which is able to operate in the presence of violations.

1 Introduction

Large scale multi-agent societies must be designed with resilience in mind, permitting agents to join or leave at any time. To ensure that the society functions as intended, constraints must be imposed on agent actions. Such constraints often take the form of norms — soft constraints which, when instantiated, oblige, prohibit or permit an agent to see to it that some state of affairs holds. Norm compliance by agents occurs for a variety of reasons, including rationality, fear of punishment, or benevolence. Now when a new agent enters the society, it must be made aware of the society's norms. When such norms are codified, obtaining them can be easy. However, even in such situations, factors such as limited bandwidth could make the transmission of the set of norms difficult. Additional difficulties — which are endemic to open multi-agent systems — such as the lack of a shared ontology, or norms being implicitly rather than explicitly specified, mean that a new agent must instead identify the norms with little or no assistance from its designer or other agents in the system.

One popular approach to norm identification[3] through learning [2, 14, 15], exemplified by [13], utilises what is referred to as an *observation-based* technique to track the behaviour of others in order to infer the norms currently in force. This technique is based on the detection of a *violation signal*, detecting when another agent has violated a norm, and by identifying the violating situation, learning the associated norm. While such an approach works well when norms are regularly violated and sanctions are explicitly applied, learning from violation signals when a system's agents (largely) comply with norms is difficult (if not impossible). In this paper, we propose an observational technique for norm learning that operates by observing compliant agents instead. We then extend this base technique to learn from both compliant, and violating agents.

At the heart of our technique lie two components — a planner and a plan recogniser. Using these, an agent entering the system observes the actions of other agents in the system and utilises the plan recogniser to identify their overarching goals. It then utilises the planner to generate alternative plans that achieve these goals. By comparing the plans other agents are actually executing to those generated by the planner, avoided (or repeatedly visited) actions and states can be identified. Repeatedly executing the plan recognition and planning steps over time enables the agent to conclusively identify those actions and states that are always avoided or visited in the execution of a plan. Since such states are analogous to prohibited or obliged situations, the agent can use them to identify norms. This basic approach cannot easily handle situations where agents *occasionally* violate norms, and we extend our technique to be able to handle norm violations by agents.

Our technique assumes that agents in the society have access to common *domain knowledge* in the form of a shared plan library that does not change over time, as well as a shared notion of the natural rewards within the system. Moreover, agents have no instant knowledge of the norms actually being enforced in the system. The reason for this assumption is that the plan library and the natural rewards in the environment are assumed to be based on some aspect of the *physical* structure of the environment. As an example, in the context of a road system, knowledge about routes to move from point A to point B might be known to all agents, as is knowledge that a certain road is full of potholes. However, the existence of a contextual prohibition on driving fast, exemplified by knowledge that the motorway police are strictly enforcing speed limits at a certain road might not be commonly known.

The remainder of this paper is structured as follows. The next section provides the background needed to understand the remainder of the paper, describing our planning formalism, outlining our plan recognition technique, and the nature of norms within the domain. Section 3 then describes our approach to norm identification. We examine related and future work in Section 4, before concluding in Section 5.

---

[3] In the literature, the task which we refer to as norm identification has also been called norm learning, norm recognition, and norm detection.

2 Background

In this section, we review the underlying techniques upon which we build our norm identification mechanism. We start by formally describing the planning model we will use to recognise alternate plans agents could execute, and follow this by detailing the plan recognition approach we use to analyse agent behaviour. Finally, we review related norm recognition approaches.

2.1 HTN Planning

We model the environment in which agents operate as a state transition system, consisting of individual states, each of which is a set of ground atoms of a first order language $\mathcal{L}$ containing finitely many predicate and constant symbols, as well as an infinite number of variable symbols[4] and no function symbols. This model is taken from the standard model used for planning, as described in [9].

An atom p holds in a state $s \in S$ iff $p \in s$. Given a set of literals g, s satisfies g (written $s \models g$) when there is some substitution σ such that every positive literal of $\sigma(g)$ is in s, and no negated literal of $\sigma(g)$ is in s.

The execution of an *action* by an agent causes the system to transition between states. In the planning literature, families of actions are specified using an action template (*operator*) containing variables referring to abstract elements in the domain. We specify such operators through a triple

$$o = (name(o), pre(o), post(o))$$

Here, $name(o)$ is a unique name for the operator, $pre(o)$ is a set of literals of $\mathcal{L}$, and $post(o)$ consists of two sets of literals, $post^+(o)$ and $post^-(o)$. $pre(o)$ identifies the preconditions that must hold in the state in which the operator is executed. $post(o)$ represents the effects, or post conditions, of executing the operator, in terms of literals added (in the case of $post^+(o)$) and removed (in the case of $post^-(o)$). The operator's name consists of an operator symbol, followed by a vector of distinct variables, such that all free variables in $pre(o), post(o)$ appear within this vector.

An action is a grounding substitution σ over all variables in the operator. An operator instance o is *applicable* to a state s, $s \models pre(o)$. An application of an action to a state s is either a new state s' such that $s' = (s \cup post^+(o) \backslash post^-(o))$ when o is applicable to s, or s otherwise.

In order to achieve a goal, an agent executes a plan — a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$. To identify this sequence, an agent can utilise classical planning techniques [9]. However, classical planning is computationally expensive, increasing the difficulty of creating agents which operate in (near) real-time. To overcome this, many agent frameworks have introduced the concept of a *plan library*. Such a plan library contains a set of plans generated offline, which can be composed to achieve high-level goals. In order to perform this composition,

---

[4] We adopt the prolog convention of representing constants with an initial lowercase letter, and variables with an initial uppercase letter.

higher level plans are made up of lower level plans, which in turn eventually reduce to primitive tasks that can be directly mapped to actions. These plans have a declarative component — multiple sub-plans might be feasibly invoked to achieve a single step in a high-level plan, and a selection between all these feasible sub-plans must be made. We can describe the problem of composing these sub-plans to achieve a high-level goal as a HTN (or rather STN) planning problem.

A hierarchical task network (HTN) planner [5] aims to decompose a set of high-level tasks, encoded as a *task network*, into a set of primitive tasks. The task network is a directed graph, consisting of the set of tasks, as well as temporal constraints between them, identifying what task must execute before what other tasks. We assume that all task networks are acyclic. Each task is an expression of the form

$$t(r_1, \ldots, r_n)$$

Here, t is a unique task symbol and $r_1, \ldots r_n$ are terms. Non-primitive tasks represent high-level tasks (e.g. `travel(S,D)` might represent the task of travelling from some source `S` to a destination `D`). Several *methods* can satisfy these tasks, for example to travel between two points one could fly or catch a train. We encode a method as a 4-tuple

$$m = (name(m), task(m), precond(m), network(m))$$

$name(m)$ is the name of the method, represented via a unique method name and a vector of terms. $task(m)$ encodes the task that the method can refine, while $precond(m)$ consists of a set of positive and negative precondition literals $precond^+(m)$, $precond^-(m)$, which must be satisfied by the state in order for the method to be *applicable*. Finally, the method identifies what tasks must be carried out in order to further refine $task(m)$ represented via a task network.

As an example, the method to fly from a source to a destination might consist of the task network containing tasks requiring one to buy a ticket; go to the airport; and fly to the destination. The first and last tasks could be primitive, while the middle task could be achieved via additional methods (e.g. going by bicycle, by car or by public transport). Such compound tasks can therefore be further decomposed by methods, into additional tasks.

Given an initial state, a task network identifying a set of high-level tasks that must be achieved and a set of operators and methods, an HTN planner searches for a plan by finding an appropriate decomposition of tasks (via methods). The precise manner in which this decomposition takes place, and its properties, are not critical for this paper, and we refer the reader to [9] for further details.

It is important to note that an HTN planning problem can be viewed as picking a set of leaf nodes from an AND/OR tree such that if a set of nodes has an AND parent, all nodes are picked, and if a set of nodes has an OR parent, only one node is picked. The exact sequence of the leaf nodes then depends on the temporal constraints imposed by the task network. Furthermore, each OR node represents a task, and given a plan, it is possible to identify both the set of

tasks, and specific task instances (i.e. ground tasks) that form the plan. Doing so lies at the heart of plan recognition.

2.2 Plan Recognition

A large body of work has examined the plan recognition problem [3, 8, 7, 16]. Due to its simplicity, we focus on a parser based plan recogniser in this paper. However, it is important to note that our plan recognition problem is an instance of keyhole plan recognition, and this approach can be trivially interchanged with more complex plan recognisers, such as [7].

Plan recognition aims to identify the actions the agent will follow (including future actions) in order to achieve its goals. [3] provides a detailed survey of such approaches, noting that in the general case, issues such as partial observability, the interleaving of plans and agent capabilities make the problem a difficult one to address. In this paper we ignore many of these complicating factors, and use an NLP-based approach (namely parsing algorithms c.f. [8]) to perform plan recognition. Such an approach utilises a technique similar to parts of speech tagging in order to identify a plan from within a plan library (or equivalently within a HTN planner). Performing plan recognition through parsing arises from the straightforward correspondence between HTN formal structures and those of context-free grammars (CFGs) [8]. Whereas plan generation for an HTN problem consists of successively refining a task network containing *non-primitive* tasks into one containing only *primitive tasks*, language generation using a CFG consists of transforming an initial string containing *non-terminal symbols* into one containing only *terminal symbols*. Refinements in an HTN planning process are made through *methods* which replace a non-primitive task into other, presumably less abstract, tasks that are either primitive or non-primitive. Analogously, language generation in CFGs consists of applying *production rules* to strings and replacing non-terminal symbols with other symbols which are either terminal or non-terminal. Based on these correspondences, techniques that can reconstruct which production rules were used to generate a certain string in a CFG can be, with some adaptation, be used to determine which methods were used to generate the intermediary non-primitive tasks that resulted in a certain plan from an HTN domain.

As an example, let us consider a grammar $\langle IS, NS, P, IS \rangle$ with a set of non-terminal symbols $NS = \{T_1, T_2, T_3, T_4, T_5\}$, a set of terminal symbols $S = \{a_1, a_2, a_3, a_4, a_5\}$, a starting symbol $IS = T_1$, and the following set P of five production rules.

$$T_1 \rightarrow T_2 T_3$$
$$T_1 \rightarrow T_2 T_4$$
$$T_2 \rightarrow a_1 a_2$$
$$T_3 \rightarrow a_3$$
$$T_4 \rightarrow a_4 a_5$$

This grammar is capable of generating two different strings '$a_1a_2a_3$' and '$a_1a_2a_4a_5$', each of which, when parsed, generates the corresponding parse trees in Figure 1a. The parse trees *explain* how each string is generated in terms of the production rules used in their derivation, all the way to the initial symbol. If we now consider the elements of the sets S and NS to correspond, respectively, to the elements of the sets of primitive and non-primitive tasks of an HTN domain, and convert the production rules of the form $\nu \rightarrow \omega$ to correspond to methods $m = (name(m), \nu, \top, \omega)$, then we would arrive at the and-or tree of the potential HTN expansions of task T_1 illustrated in Figure 1b. Since the reverse conversion is just as straightforward, we can see how a parsing-based technique can be used to determine what higher-level tasks explain the observation of a particular sequence of actions (a plan).
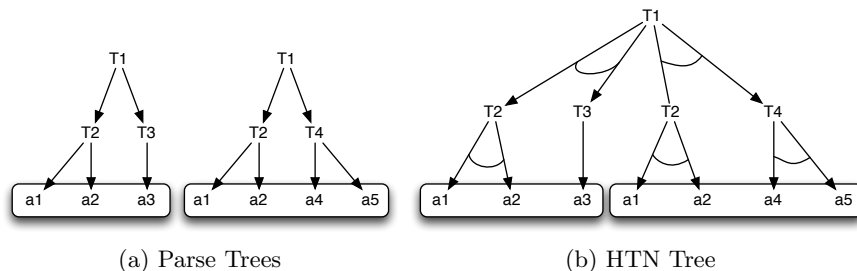

(a) Parse Trees   (b) HTN Tree

Fig. 1: HTN and Parse trees

2.3 Norms

The previous section described one portion of our model, namely the domain and the structure of agent plans. In this section, we formalise our notion of a norm, which comprises a simplification of the usual elements of a norm tuple [1, 10], namely: a *deontic modality*, a *context condition* representing the situations in which a norm should be enforced, and a *normative condition* that, together with the deontic modality, identifies expected behaviour. We note that whereas the conditions of a norm tuple are normally formalised in terms of a logic formula on the environment state, our norm language uses either fully specified states, or *task* symbols.

Syntactically, we write a norm as $\mathbf{X}_y z$, where $\mathbf{X} \in \{\mathbf{O}, \mathbf{F}\}$, y is a task, and z is a task or state. $\mathbf{X}$ is the modality of the norm, either an obligation ($\mathbf{O}$) or a prohibition ($\mathbf{F}$). y identifies the norm's context, specifying the situations under which the norm's condition, z must occur (in the case of an obligation), or must not occur (in the case of a prohibition). A task z occurs in context y iff z is a subtask of y, and under the appropriate grounding of variables, the

leaf tasks associated with its OR node are executed. In other words, a z occurs in the context of y if, in the process of task y's methods being executed, task z is executed. A state z occurs in context y if the state z is entered while the leaf tasks associated with y's OR node are executed. In other words, state z is entered while a method whose task is y, is being executed. If a norm's condition must occur and does not, or alternatively must not occur and does, then the norm is *violated*.

As an example, consider the task of travelling from Aberdeen to Paris, and assume that when travelling anywhere there is a prohibition imposed on transiting via London. The task instance could be represented as $\texttt{travel}(\texttt{aberdeen}, \texttt{paris})$, with the prohibition represented as $\mathbf{F}_{travel(X,Y)} at(london)$. Now one could have the method

$$(fly(X, Y), travel(X, Y), \{at(X), connect(X, Z), connect(Z, Y)\},$$
$$\{goto(X, Z) \prec goto(Z, Y)\})$$

Here, $\prec$ represents a temporal constraint, stating that $goto(X, Z)$ must occur before $goto(Z, Y)$. Finally, the *goto* primitive task is represented by the following operator.
$$(goto(X, Y), \{at(X)\}, \{\neg at(X), at(Y)\})$$

Given the preconditions $\texttt{connect}(\texttt{aberdeen}, \texttt{london}), \texttt{connect}(\texttt{london}, \texttt{paris})$, the instantiation of the plan will result in $\texttt{at}(\texttt{london})$ occurring in context $\texttt{travel}(\texttt{aberdeen}, \texttt{paris})$, violating the norm. Notice that, although using *tasks* in the context and norm conditions is syntactically different from the state formulas used in traditional norm representations, the net effect of our norm representation is equivalent in that these conditions represent subsets of the domain's state space. Given that the state space that *can be* traversed by an agent is completely determined by the tasks in its plan library, this representation is sufficient to encode all the norms that can be detected by our algorithm.

3 Norm Identification Approaches

We are now in a position to describe our main contribution — a plan recognition based approach to norm identification. The basic assumption of our approach is that an individual agent's behaviour is defined in terms of a *plan library* of procedural plans following the tradition of agent programming languages such as AgentSpeak(L) [11]. This kind of plan library is analogous to domain specifications for Hierarchical Task Network (HTN) planning. Like procedural goals in AgentSpeak(L), goals in HTNs are defined in terms of a partially ordered set of high-level abstract tasks (or *task network*), which can be recursively refined to more concrete tasks until a fully specified plan of executable actions is generated. It is straightforward to see that the set of plans an agent can execute is completely defined by such a library, much in the same way in which production rules define all valid strings that could be generated by a grammar. Consequently, by being aware of the plan library employed by agents, one can

Algorithm 1 The basic norm identification mechanism

Require: R, a set of runs
 1: **function** LEARNNORMS(R)
 2:     $potO \leftarrow$ all possible obligations
 3:     $potF \leftarrow \emptyset$, $notF \leftarrow \emptyset$
 4:     **for all** $r \in R$ **do**
 5:         $pO = \emptyset$
 6:         $pF = \emptyset$
 7:         $\pi \leftarrow$ PLANRECOGNISER(r)
 8:         **for all** tasks $t \in \pi$ **do**
 9:             **for all** subtasks $t' \in \pi$ of t **do**
10:                 $pO \leftarrow pO \cup \{\mathbf{O}_t t'\}$
11:                 $notF \leftarrow notF \cup \{\mathbf{F}_t t'\}$
12:             **for all** state s transitioned through as part of t **do**
13:                 $pO \leftarrow pO \cup \{\mathbf{O}_t s\}$
14:                 $notF \leftarrow notF \cup \{\mathbf{F}_t s\}$
15:             **for all** $\pi' \in$ all possible plans with the same start state as π and with a goal identical to t **do**
16:                 **for all** tasks $\tau \in \pi'$ **do**
17:                     **for all** subtasks $\tau' \in \pi'$ of τ, which are not subtasks of $\tau \in \pi$ **do**
18:                         $pF \leftarrow pF \cup \{\mathbf{F}_\tau \tau'\}$
19:                     **for all** states Σ visited as part of $\tau \in \pi'$ and not for $\tau \in \pi$ **do**
20:                         $pF \leftarrow pF \cup \{\mathbf{F}_\tau \Sigma\}$
21:         $potF \leftarrow (potF \cup pF)\backslash notF$
22:         $potO \leftarrow potO \cap_t pO$
23:     **return** $potO$, $potF$

---

use the sequences of actions executed by these agents to infer the higher-level goals that they pursue.

We begin this section by describing a novel plan recognition based approach to norm identification. This basic approach assumes agents will never violate norms. In order to overcome this assumption, we then describe an extension to the basic model.

3.1 Norm Identification via Plan Recognition

Our mechanism seeks to identify obligations and prohibitions within the domain, and conceptually operates in several steps. As input, we take in a set of *runs*, each representing the states transitioned through by the system in the process of achieving some top level goal. Note that these runs could originate from a single agent pursuing multiple goals over an extended period of time; multiple agents each pursuing goals (though we assume non-interference between agent actions); or a mixture of the two. Algorithm 1 formalises our mechanism.

The algorithm uses several variables: $potO$ stores all potential obligations in the system; $potF$ and $notF$ respectively store potential prohibitions as well as those states of affairs which are definitely not prohibited. When the algorithm

starts, $potO$ is initialised with all possible obligations in the system. Obligations are monotonically removed from this set as the algorithm runs. Since we assume a finite number of predicate and constant symbols, $potO$ will be finite, but potentially very large. Within an implementation, techniques such as masking, or storing only those elements not in the set can be used to mitigate the issues surrounding its size.

The algorithm operates over a set of runs, identifying norms for each such run. This is done by utilising the plan recogniser to identify the plan π being executed (Line 7). We assume that only one plan is ever returned by the plan recogniser. This plan π recursively identifies high-level tasks and the subtasks used to decompose them into lower level tasks, all the way down to the primitive task level. Collectively, these tasks, subtasks and primitive tasks are the tasks of the plan, and we refer to any descendent of a task as the task's subtask.

Broadly speaking, our algorithm operates in two phases. Lines 9–14 consider only the plan, while Lines 15–20 consider alternate plans which achieve the same goals. Examining the first phase in more detail, we iterate over all tasks t of the recognised plan, and over each task's subtasks. Any of these subtasks visited could, potentially, be one that must be visited according to the obligations in the system, and we therefore add it to a temporary variable pO (Line 10). Furthermore, since we assume that no violations can occur, the executed subtask cannot be prohibited in the context of the parent task, and we therefore add it to the list of prohibitions that definitely do not exist in Line 11. Norms can describe both tasks and states, and an identical procedure to the one followed above is carried out to identify potentially obligatory, and definitely not prohibited states (Lines 12–14).

Line 15 identifies all other possible plans that began with the same initial state as the actually executing plan, and finish by achieving the same goals as the current task. The aim here, and in the remainder of the second phase, is to identify tasks that could have been, but were not executed, as these represent potential prohibitions that were complied with. To do so, we iterate over all tasks τ within an alternate plan, and over all subtasks τ' of τ, where τ' is not a subtask of the t (the task within the original plan). Thus, each such subtask τ' was avoided in the context of task τ (equivalent to t) in the original plan, and one possible reason for it having been avoided, was that τ' was prohibited in the context of the task t. Lines 17–18 identify these potential prohibitions, and store them in the temporary variable pF. This process is repeated for those states visited as part of the task in Lines 19–20.

Finally, the set of potential prohibitions is updated by adding those newly discovered potential prohibitions of pF, and removing any prohibitions which definitely do not exist. The set of potential obligations is reduced to those states identified which are definitely obliged; the symbol $\cap_t$ is a *context sensitive intersection*. This operation preserves any elements in $potO$ which do not share the same context, and performs a normal intersection operation between any elements in $potO$ and pO which do share context.

Executing this algorithm over a large number of runs of the system will slowly remove from *potO* those contexts, tasks and states, which are not obliged but were often executed, and will non-monotonically alter the set of prohibitions *potF*.

Example 1. As a simple example, assume a run was seen wherein primitive tasks t_3, t_4, t_6, t_7 and t_8 occurred. The plan recogniser reconstructed the plan shown on the left of Figure 2 for this run, with the alternative plan shown on the right also being a potential plan to execute task t_1. For clarity, we ignore temporal constraints within these plans, and show only the index of the tasks.

Considering only the plans on the left, our algorithm would generate obligations of the form

$$\mathbf{O}_{t_1} t_2, \mathbf{O}_{t_1} t_3, \mathbf{O}_{t_1} t_4, \ldots$$
$$\mathbf{O}_{t_2} t_3, \mathbf{O}_{t_2} t_4, \mathbf{O}_{t_5} t_6, \ldots$$

When considering the second tree, some of these obligations would be pruned, and potential prohibitions would also be introduced, resulting in norms of the form

$$\mathbf{O}_{t_1} t_2, \mathbf{O}_{t_2} t_4, \mathbf{O}_{t_1} t_4,$$
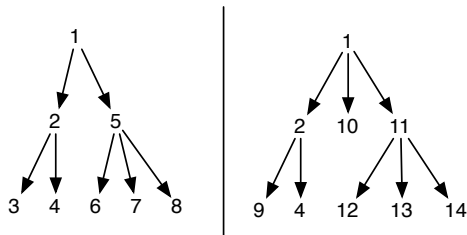$$\mathbf{F}_{t_1} t_9, \mathbf{F}_{t_1} t_{10}, \ldots$$

Fig. 2: An HTN representation of the tasks executed by a plan (left), and an alternative plan (right), to achieve the same goal.

3.2 Allowing Norm Violation

Our basic approach is, in some sense, the dual of violation signal based approaches such as [15] — it can only obtain an accurate model of the norms in the system when no violations take place. Consider for example obligations. Since the set *potO* shrinks monotonically, a violation of an obligation would permanently remove it from the set of potential obligations. In this section, we describe an extension of our basic approach which aims to overcome this limitation.

Algorithm 2 outlines this extension. In turn, it uses another function, UP-DATECOUNTER(r, OC, FC), which is described in Algorithm 3. The latter algorithm takes a single run, and a pair of counters (described next), and updates these counters.

Algorithm 2 Threshold-based filtering heuristic

Require: R, a set of runs
Require: OT, FT, thresholds in $\mathcal{R}^+$
 1: **function** TLEARNNORMS(R)
 2:     initialise OC to $(0,0)$ for every possible obligation.
 3:     initialise FC to $(0,0)$ for every possible prohibition.
 4:     **for all** $r \in R$ **do**
 5:         $(OC, FC) \leftarrow$ UPDATECOUNTER(r, OC, FC)
 6:     **for all** $(\mathbf{O}_y z, oy, on) \in OC$ **do**
 7:         **if** ($on = 0$ and $oy > 0$) or ($on \neq 0$ and $oy/on > OT$) **then**
 8:             $potO = potO \cup \{\mathbf{O}_y z\}$
 9:     **for all** $(\mathbf{F}_y z, fy, fn) \in FC$ **do**
10:         **if** $fn = 0$ or $fy/fn > FT$ **then**
11:             $potF = potF \cup \{\mathbf{F}_y z\}$
12:     **for all** y, z such that both $\mathbf{O}_y z$ and $\mathbf{F}_y z$ **do**
13:         remove $\mathbf{O}_y z$ from $potO$, $\mathbf{F}_y z$ from $potF$
14:     **return** $potO, potF$

Counters OC and FC respectively store a pair of values for every possible obligation and every possible prohibition that could exist in the system (requiring the same compression techniques discussed in Section 3.1 when implemented). The first element of this pair is the number of times the obligation (or prohibition) appears to indeed be an obligation (or prohibition). The second element is associated with the number of times the obligation (or prohibition) appears *not to be* a valid obligation or prohibition.

Lines 6–8 compute the set of possible obligations by checking whether the ratio between an obligation existing, and not existing, exceeds some threshold [15][5]. In doing so, we ensure that there must be at least one situation in which the obligation is a potential obligation.

Lines 9–11 perform a similar operation for prohibitions. It should however be noted that no positive examples of a prohibition are needed if no negative examples exist in order for the prohibition to be considered a potential prohibition (Line 10). Finally, we perform a sanity check, removing any prohibitions and obligations with the same context and normative condition.

We now turn our attention to the UPDATECOUNTER(r, OC, FC) function, described in Algorithm 3. This algorithm bears similarities to Algorithm 1. We begin by utilising the plan recogniser to identify the plan executed in the current run. Again, our algorithm operates in two phases, first considering the current plan, and then considering alternative plans which achieve the same goal. Also, as was previously done, we consider every task found within the plan, and all its subtasks. A task t and a subtask t' executed as part of the plan may have been executed due to an obligation, and their presence thus increases the likelihood

---

[5] While we utilise a ratio, an additive approach, or some other technique are equally applicable.

Algorithm 3 Updating potential norm occurrences.

Require: r, a single run
Require: FC a relation of counter tuples, of the form $\mathbf{F}_y z \times \mathcal{N} \times \mathcal{N}$
Require: OC a relation of counter tuples, of the form $\mathbf{O}_y z \times \mathcal{N} \times \mathcal{N}$
 1: **function** UPDATECOUNTER(r)
 2:     $\pi \leftarrow$ PLANRECOGNISER(r)
 3:     **for all** tasks $t \in \pi$ do **do**
 4:         **for all** subtasks $t' \in \pi$ of t **do**
 5:             $(oy, on) \leftarrow OC[\mathbf{O}_t t']$
 6:             $OC[\mathbf{O}_t t'] \leftarrow (oy + 1, on)$
 7:             $(fy, fn) \leftarrow FC[\mathbf{F}_t t']$
 8:             $FC[\mathbf{F}_t t'] \leftarrow (fy, fn + 1)$
 9:         **for all** states s transitioned through as part of t **do**
10:             $(oy, on) \leftarrow OC[\mathbf{O}_t s]$
11:             $OC[\mathbf{O}_t s] \leftarrow (oy + 1, on)$
12:             $(fy, fn) \leftarrow FC[\mathbf{F}_t s]$
13:             $FC[\mathbf{F}_t s] \leftarrow (fy, fn + 1)$
14:         **for all** $\pi' \leftarrow$ all possible plans with the same start state as π and with a goal identical to t **do**
15:             **for all** tasks $\tau \in \pi'$ **do**
16:                 **for all** subtasks $\tau' \in \pi'$ of τ, which are not subtasks of $\tau \in \pi$ **do**
17:                     $(fy, fn) \leftarrow FC[\mathbf{F}_\tau \tau']$
18:                     $FC[\mathbf{F}_\tau \tau'] \leftarrow (fy + 1, fn)$
19:                 **for all** states Σ visited as part of $\tau \in \pi'$ and not for $\tau \in \pi$ **do**
20:                     $(fy, fn) \leftarrow FC[\mathbf{F}_\tau \Sigma]$
21:                     $FC[\mathbf{F}_\Sigma \tau'] \leftarrow (fy + 1, fn)$
22:     **return** OC, FC

---

that $\mathbf{O}_t t'$ is an obligation. We therefore increment the obligation counter to reflect this fact (Lines 5,6). Similarly, the execution of this task and subtask means that a prohibition on this action does not exist, reducing its likelihood (Lines 7,8). Lines 9–13 repeat this check for tasks and their constituent states.

The second phase (Lines 14–21) again considers all alternative plans, as generated by the planner. Any tasks and subtasks not executed in the plan are potential prohibitions, and their counter is appropriately incremented. This operation is repeated for the visited states in the alternative plan. Finally, the algorithm returns the updated counters.

4   Discussion and Related Work

Savarimuthu and Cranefield [12] provide a detailed survey regarding the state of the art in norm learning. Existing approaches can be broadly classified into three categories: experiential, observational and communication-based. Experiential techniques rely on an agent's perception of sanctions and rewards being applied to its own actions and deriving a model of the norms within the environment

from these experiences. Observational techniques rely on an agent tracking the behaviour of others and trying to identify norms from the sanctions and rewards applied to others, before acting in the environment. Finally, communication-based techniques assume that an agent either asks other agents for the set of norms in force within a society, or is told these norms. While our approach is firmly associated with observational techniques within this classification, we drop the assumption that an agent can observe sanctions and rewards being applied to other agents (c.f. the violation signals of [15]).

In unpublished work, we modelled a domain consisting of a road network in which agents drive from place to place, but must adhere to certain norms, such as driving on the correct side of the road, avoiding (or passing through) certain areas of the network, and so on. We introduced a new agent into this environment and left it to learn the norms by observing the behaviour of pre-existing agents using an implementation of our approach. This domain consisted of a hypergraph (to represent multi-lane roads), with agents planning to travel from one node to another using an HTN domain containing a hierarchy of tasks to move the agent about. Norms prohibiting or obliging agents to enter specific nodes while travelling were created, as were norms prohibiting certain types of behaviour (such as making left turns). Executing our algorithm allowed us to identify the set of prohibited and obliged states and tasks when all agents were fully norm compliant. As expected, when only few runs were input to the algorithm, many "false positive" obligations were obtained. The norm specification, and associated norm identification algorithms in this evaluation operated over entire plans, eschewing the extra complexity of context, and therefore limiting its applicability to the current work. Indeed, a detailed evaluation of our current approach forms one of our intended avenues of future work.

Since one key component of our approach is a plan recognition algorithm, the quality of the norms identified is strongly dependant on the accuracy of the underlying plan recogniser. While it is desirable to utilise a powerful keyhole plan recogniser such as the one described in [4], we utilised a simpler technique both in order to simplify the description of our approach, and due to the difficulties inherent in extending this plan recogniser from the propositional to the first order domain. In the future, we aim to employ new algorithms based on recent advances in probabilistic parsing-based plan recognition [6, 7].

We note that given our algorithm's assumption that every transited state is a potential obligation, as the amount of data (plan runs) supplied to the algorithm increases, the number of potential obligations will increase with the total number of steps of all plans processed. This large number of potential obligations can be interpreted in two ways. First, they might represent false positives with regards to the formally enforced norms in a society. On the other hand these potential obligations might represent informal conventions that agents tend to follow within a society, either because they have no other choice (given their plan library), or because these choices have a higher utility. In future work, we intend to evaluate filters on potential obligations due to agents having no alternative plans to fulfil a goal, as well as to look further into plan utilities.

It should also be noted that our algorithm can also generate false positives in certain situations. Apart from the false positive obligations generated when few runs exist, false positive prohibitions can be generated when real prohibitions make it impossible to reach some states (e.g. if a prohibition exists on visiting states $(0,1)$ and $(1,1)$, and these are the only paths to a state $(0,0)$, then our algorithm will identify $(0,0)$ as prohibited). However, since norm compliant agents cannot reach such states, such false positives are not a problem in practice.

Finally, while Section 3.2 described how normative violations can be dealt with, we have not considered an important class of norms that emerges in such situations, namely contrary-to-duty obligations. Such obligations aim to cause corrective actions to occur within a system, such as obliging an agent to pay reparations for their violations. Contrary-to-duty obligations are context dependent, and are not captured by our structure of norms. Indeed, extending our norm representation to capture rich context conditions (as opposed to just tasks as in the current approach) while still performing plan based norm identification forms one direction of future work which we are pursuing.

We are currently investigating several additional avenues of future work. First, we currently assume no interference between agent actions. Such interference can cause an agent to prefer one plan over another even in the absence of norms, and we intend to investigate how such situations can be detected and handled by the mechanism. Second, different actions can have different costs for an agent, again creating false positives in our current mechanism. Complicating this is the idea that norm violation can also be associated with a cost — if the cost of the latter is smaller than the benefit of a violating plan, a rational agent can choose to violate a plan. We must thus extend our framework to take such costs into account. We also intend to investigate the effectiveness of our approach when only part of a run is visible to the plan recogniser, and to combine our plan recognition based approach with violation signal based techniques.

5 Conclusions

In this paper we developed an algorithm for norm identification based on a plan recognition based algorithm for norm identification. Unlike existing approaches, our mechanism can operate in situations where all agents within the system comply with their norms. However, doing so requires agents to always act in a norm compliant manner. Since this requirement is unrealistic, we extended our plan recognition approach to cater for violations through counting the number of times obligations and prohibitions potentially do, or do not, exist. This, coupled with threshold functions (in the spirit of [15]) allow our approach to operate in domains where violations can occur.

The use of a plan recognition component to drive norm identification appears promising, and we have highlighted several avenues for future work which we intend to actively pursue.

References

1. H. Aldewereld, F. Dignum, A. García-Camino, P. Noriega, J. A. Rodríguez-Aguilar, and C. Sierra. Operationalisation of norms for usage in electronic institutions. In *Proc. of the Fifth Int'l Joint Conf. on Autonomous Agents and Multiagent Systems*, pages 223–225, 2006.

2. G. Andrighetto, R. Conte, P. Turrini, and M. Paolucci. Emergence in the loop: Simulating the two way dynamics of norm innovation. In G. Boella, L. W. N. van der Torre, and H. Verhagen, editors, *Normative Multi-agent Systems*, volume 07122 of *Dagstuhl Seminar Proceedings*, 2007.

3. M. G. Armentano and A. Amandi. Plan recognition for interface agents. *Artif. Intell. Rev.*, 28(2):131–162, 2007.

4. D. Avrahami-Zilberbrand and G. A. Kaminka. Incorporating observer biases in keyhole plan recognition (efficiently!). In *Proc. AAAI*, 2007.

5. K. Erol, J. Hendler, and D. S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 2, pages 1123–1128, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

6. C. Geib. Considering state in plan recognition with lexicalized grammars. 2012.

7. C. W. Geib and R. P. Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artif. Intell.*, 173(11):1101–1132, July 2009.

8. C. W. Geib and M. Steedman. On natural language processing and plan recognition. In *Proc. 20th Int'l Joint Conf. on Artif. Intell.*, pages 1612–1617, 2007.

9. M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004.

10. N. Oren, S. Panagiotidi, J. Vazquez-Salceda, S. Modgil, M. Luck, and S. Miles. Towards a formalisation of electronic contracting environments. In *Proc. of Coordination, Organization, Institutions and Norms in Agent Systems, the Int'l Workshop at AAAI 2008*, pages 61–68, Chicago, Illinois, USA, 2008.

11. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. W. Perram, editors, *Proc. of the 7th MAAMAW*, volume 1038 of *LNCS*, pages 42–55. Springer-Verlag, 1996.

12. B. T. R. Savarimuthu and S. Cranefield. Norm creation, spreading and emergence: A survey of simulation models of norms in multi-agent systems. *Multiagent and Grid Systems*, 7(1):21–54, 2011.

13. B. T. R. Savarimuthu, S. Cranefield, M. Purvis, and M. K. Purvis. Identifying conditional norms in multi-agent societies. In M. D. Vos, N. Fornara, J. V. Pitt, and G. A. Vouros, editors, *COIN@AAMAS&MALLOW*, volume 6541 of *Lecture Notes in Computer Science*, pages 285–302. Springer, 2010.

14. B. T. R. Savarimuthu, S. Cranefield, M. A. Purvis, and M. K. Purvis. Internal agent architecture for norm identification. In *Proc. of the 5th Int'l Workshop on Coordination, organizations, institutions, and norms in agent systems*, COIN'09, pages 241–256, Berlin, Heidelberg, 2010. Springer-Verlag.

15. B. T. R. Savarimuthu, S. Cranefield, M. A. Purvis, and M. K. Purvis. Obligation norm identification in agent societies. *Journal of Artificial Societies and Social Simulation*, 13(4), 2010.

16. G. Sukthankar. *Activity Recognition for Agent Teams*. PhD thesis, Carnegie Mellon University, 2007.