Exploiting Domain Knowledge to Improve Norm Synthesis

George Christelis*
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, UK
george.christelis@ed.ac.uk

Michael Rovatsos
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, UK
michael.rovatsos@ed.ac.uk

Ronald P. A. Petrick
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, UK
r.petrick@ed.ac.uk

ABSTRACT

Social norms enable coordination in multiagent systems by constraining agent behaviour in order to achieve a social objective. Automating the design of social norms has been shown to be NP-complete, requiring a complete state enumeration. A planning-based solution has been proposed previously to improve performance. This approach leads to verbose, problem-specific norms due to the propositional representation of the domain. We present a first-order extension of this work that benefits from state and operator abstractions to synthesise more expressive, generally applicable norms. We propose optimisations that can be used to reduce the search performed during synthesis, and formally prove the correctness of these optimisations. Finally, we empirically illustrate the benefits of these optimisations in an example domain.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*

General Terms

Algorithms, Theory, Design

Keywords

Normative systems, social norms, conflict resolution, automated planning

1. INTRODUCTION

Social norms have widely been accepted as a means of achieving coordination in multiagent systems by placing constraints on the behaviour of all agents in the system. These constraints reflect the global social objective held by the norm designer: should all agents adhere to the prescribed norms, the social objective will then be achieved. In such a system social norms discourage behaviour that violates the social objective.

Recent literature on normative systems often focuses on processes specified over existing, pre-specified social norms or normative systems [1, 2, 5]. In this paper we investigate the process of social norm synthesis: what feasible algorithmic process might an agent or system designer adopt to synthesise norms for their social objective? Furthermore, can this process provide guarantees regarding the achievability of agent goals in the constrained system?

In the general case, norm synthesis was shown to be NP-complete through a reduction from 3-SAT [9]. Subsequent synthesis approaches based on full state-space enumeration have been proposed, yet these are often intractable in the general case or not easily applicable in real world systems [4, 7, 10]. Furthermore, these approaches do not consider the effects the norms have on the achievability of agent goals.

The work presented in this paper extends an existing propositional planning-based approach [3], based on performing a localised search around specifications of undesirable states utilising the existing domain representation. Norms are synthesised over abstract representations of sets of states, resulting in concise, generally applicable norms. However, the resulting search is fundamentally limited in two ways: firstly, the propositional formalism limits norm expressiveness and synthesis performance, and secondly, the naive search needlessly considers solutions that are irrelevant, thereby limiting the scalability of the search in large real world domains.

In this paper we present a first-order extension to this propositional approach, allowing for more expressive social norms containing unbound variables. We describe the exact conditions that must hold for agent goals to be achievable under the synthesised norms and we show this more expressive approach to be sound based on a reduction to the existing propositional synthesis.

This increase in expressiveness results in additional complexity issues, related to the unground nature of the operators considered: instead of considering ground operator instances we synthesise over the unground variable-based operator schemata. We present a set of optimisations that reduce the resulting search space by ignoring operator sequences that are not possible in the underlying domain, or are irrelevant to the social objective. We provide proofs regarding the correctness of these optimisations, and empirically illustrate the potential benefits in an example domain.

We begin by presenting a running example in the following section. Section 3 details the first-order planning domain formalism and the resulting norm synthesis procedure, while Section 4 outlines optimisations to improve the synthesis performance. Section 5 presents empirical performance results in the example domain. We detail related work in Section 6 and conclude in Section 7.

*The author is a Commonwealth Scholar.

Cite as: Exploiting Domain Knowledge to Improve Norm Synthesis, G. Christelis, M. Rovatsos, R. Petrick, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. 831-838
Copyright © 2010, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

2. PARCEL DELIVERY DOMAIN

We use a series of running examples detailing an agent-based parcel delivery domain. Agents are positioned in a grid world of arbitrary topology and are able to move between adjacent grid nodes. Agents are tasked with fetching and delivering parcels from source to target locations. Consider these core action schemata:

Action	Preconditions	Effects
$\texttt{move}_i\texttt{(X,Y)}$	$\texttt{at}_i\texttt{(X)}$	$\neg\texttt{at}_i\texttt{(X)}$
	$\texttt{conn(X,Y)}$	$\texttt{at}_i\texttt{(Y)}$
$\texttt{pickup}_i\texttt{(X,P)}$	$\texttt{pAt(P,X)}$	$\neg\texttt{pAt(P,X)}$
	$\texttt{at}_i\texttt{(X)}$	$\texttt{hold}_i\texttt{(P)}$
$\texttt{drop}_i\texttt{(X,P)}$	$\texttt{hold}_i\texttt{(P)}$	$\neg\texttt{hold}_i\texttt{(P)}$
	$\texttt{at}_i\texttt{(X)}$	$\texttt{pAt(P,X)}$
$\texttt{destroy}_i\texttt{(P)}$	$\texttt{hold}_i\texttt{(P)}$	$\neg\texttt{hold}_i\texttt{(P)}$
$\texttt{idle}_i$	-	-

Agents are `at` locations connected via `conn` predicates. Parcel locations are represented via `pAt` predicates. An agent `hold`s a picked up parcel, and can either `drop` the parcel in a location or `destroy` the parcel. Agents can also choose to remain `idle`.

3. CONFLICT-ROOTED SYNTHESIS

The problem of norm synthesis can be stated as follows: given a specification of undesirable system states, what automated design process will create conditional action prohibitions that result in the undesirable states being avoided? Furthermore, can this process guarantee that under the prohibitions, agents are still able to achieve their goals?

Our approach is called *Conflict-Rooted Synthesis*, and is based upon performing localised searches around conflict state specifications. Directed searches are conducted locally around specifications of undesirable conflict states to identify what states are reachable from these states, and to investigate alternative means of achieving these states. If alternative conflict-free paths exist, then the designer can regiment the system in the knowledge that agents have alternative means of achieving their goals under prohibitions.

We next present the first-order planning formalism we use as the standardised representation of our domains, followed by an explanation of the synthesis process itself.

3.1 First-Order Planning Formalism

We adapt an existing predicate-based planning formalism [6] to support a restricted form of first-order logic, altering the traditional STRIPS formalism to allow for unground state specifications. This increase in expressiveness allows our synthesis process to produce more general, unground, domain independent social norms.

We begin by defining a language $\mathcal{L} = \mathcal{L}_c \cup \mathcal{L}_v \cup \mathcal{L}_p$ of finitely many predicate symbols ($\mathcal{L}_p$), constant terms ($\mathcal{L}_c$) and variable terms ($\mathcal{L}_v$). Constant terms are written in lowercase {`node1`, `parcel2`} while variable terms begin with uppercase letters {`Agent`, `Parcel`}. We adopt a $\overline{bar}$ notation to represent ground instances of set elements: a ground element is one that does not contain variable terms.

The set of *atoms* $\mathcal{A}$ is composed of predicates with variable and constant terms, with the ground subset $\overline{\mathcal{A}} \subseteq \mathcal{A}$ composed of ground atoms only.

The set L contains the complete set of all literals over $\mathcal{A}$. If $L' \subseteq L$, we define $\neg L'$ to be element-wise negation such that $\neg L' = \{l | \neg l \in L'\} \cup \{\neg l | l \in L'\}$. We write $L_1 \backslash L_2$ to denote set difference.

A *state* $s \subseteq \overline{\mathcal{A}}$ is a subset of the ground atoms. States follow a closed world semantics. If atoms are not included in the state description, their negations are assumed to hold. We define a *state specification* S to be a truth assignment over $\mathcal{A}$, defined as a logical theory where $S \subseteq L$[1].

An operator o is a parameterised action schema triple of the form $\{pre(o)\} \xrightarrow{\texttt{name(o)}} \{post(o)\}$ where $name(o)$ is the parameterised operator definition, $pre(o) \subseteq L$ are the preconditions and $post(o) \subseteq L$ the effects. Let O represent the set of operator schemata in the domain.

We derive instances of an operator through variable substitution and unification, allowing unground operators for transitions between unground state specifications. We ground variables with constants where possible, and unify the remaining variables with those in the specification. A *substitution set* σ is defined as

$$\sigma \subseteq \{(v \leftarrow l) | v \in \mathcal{L}_v, l \in \mathcal{L}_c \cup \mathcal{L}_v\}.$$

Applying σ to a set of atoms $\mathcal{A}'$ is denoted $\sigma[\mathcal{A}']$. We write $s \models \overline{S}$ if the truth assignment of state s satisfies the ground $\overline{S}$. Similarly, $s \models S$ if there exists some grounding σ such that $s \models \sigma[S]$.

An operator o is *applicable* in S if $pre(o) \in S$. The result of applying o in S is defined as

$$App(S, o) = (S \backslash \neg post(o)) \cup post(o)$$

if $S \not\models \bot$, $S \models pre(o)$ and $post(o) \not\models \bot$. The result of applying a sequence of operators $\langle o_1, o_2 \ldots\rangle$ to a state specification S is a recursive function defined as follows:

$$\begin{aligned} Res(S, \langle\rangle) &= S, \\ Res(S, \langle o_1, o_2, ..., o_n\rangle) &= Res(App(S, o_1), \langle o_2, ..., o_n\rangle). \end{aligned}$$

For clarity, we have not detailed all substitution sets. Since specifications and operators are unground the notions of applicability and satisfiability must be extended. Satisfiability for an unground specification holds if $\exists\sigma.s \models \sigma[S]$. Similarly, set relations are defined over all possible substitution sets, so $S_1 \subseteq S_2$ if $\exists\sigma.S_1 \subseteq \sigma[S_2]$. As an extension, operator applicability also requires such a substitution: an operator o is applicable if $\exists\sigma.pre(o) \in \sigma[S]$.

We consider unground predicates as first class citizens of our formalism. For example, consider applying the operator $\texttt{move}_1\texttt{(X,Y)}$ in state $S = \{\texttt{at}_1\texttt{(X1)}, \texttt{conn(X1,Y)}\}$. Here, `move` is applicable in S, but only for the substitution $\sigma = \{(\texttt{X1} \leftarrow \texttt{X})\}$. It is this partial grounding and unification that we capture through the application of substitution sets.

3.2 Social Norm Representation

Prohibitionary norms in our system are conditional behavioural constraints that restrict the states in which operators are applicable. A set $P = \{p_1, p_2...\}$ of prohibitions contains tuples of the form $p_i = \langle\varphi, o\rangle$ where $\varphi \in 2^L$ and $o \in O$, denoting that if the current state satisfies the precondition φ, then the operator o is forbidden in this state. For example, consider the prohibition $\langle\{\texttt{at}_1\texttt{(node3)}\}, \texttt{pickup}_1\texttt{(P,node3)}\rangle$. The benefits of a first-order approach are clear here: this norm prohibits agent 1 from picking up *any* parcel in `node3`. The quantification over parcels in `node3` is not possible in the existing propositional approach.

[1]We extend the classical definition of a state specification to include truth assignments over unground and ground atoms.

3.3 Conflict Rooted Synthesis

Synthesis is defined as $\mathit{Synth}(S_C, O) \rightarrow P$. As input, S_C is the *conflict state specification* representing the undesirable system states, and O is the set of operator schemata. A set of prohibitions P is produced as output. Our approach searches for all operator sequences that traverse from conflict-free states, through conflict states modelled by S_C. We consider paths for *any* state modelled by the conflict state specification, exhaustively searching all hypothetical runs through states modelled by the conflict specification. Once all sequences are identified, we guarantee goal achievability in the normative system by showing that an alternative conflict-free path exists for each conflict sequence.

3.3.1 Definitions

Let $S \xrightarrow{o} S'$ denote a transition between state specifications S and S' through the application of operator o. A *run* is a sequence of the form $R = S_0 \xrightarrow{o_1} S_1 \ldots S_{n-1} \xrightarrow{o_n} S_n$. We use the notation $\mathit{first}(R)$ and $\mathit{last}(R)$ to refer to the first and last state specifications, and $|R|$ to denote the total number of state specifications in R.

A *conflict run* is a run where at least one of the state specifications is a conflict specification ($\exists i < |R|. S_C \subseteq S_i$). All other runs are *conflict-free*. We further differentiate classes of runs depending on whether they terminate with conflict specifications. A *complete* run R is a conflict run where $\mathit{first}(R)$ and $\mathit{last}(R)$ are conflict-free, and all other intermediate specifications are conflict specifications. An *incomplete* run R is a conflict run where $\mathit{first}(R)$ is conflict-free, and all other state specifications are conflict specifications.

3.3.2 Conflict Traversal

Conflict traversal is a search to identify all complete conflict runs, representing all that is achievable in conflict states. In order to identify these runs we apply an iterative process of run extension and refinement, searching the set of feasible runs in a breadth-first fashion.

Given a conflict specification S_C, the process returns a set $\mathcal{R}$ of complete runs. These runs are independent of the problem instance and goals of the agent, and are synthesised purely on the operator schemata. In the specification below we write $\mathcal{R}_i$ to represent all incomplete runs of length i. Two procedures are used to extend incomplete runs with some operator: *inference* constructs a new future state specification, and *refinement* alters the specifications of an existing run to ensure that the resulting extended run is consistent.

We illustrate each step using a simple example. Let $S_C = \{\mathtt{hold}_1\mathtt{(p)}\}$ represent the undesirable state where agent `1` holds parcel `p`.

Run Initialisation: Given S_C, we identify all *contributing* operators as those where:

$$\exists l \in \mathit{post}(o).(l \in S_C \wedge l \notin \mathit{pre}(o)) \tag{1}$$

$$\nexists l \in ((\mathit{pre}(o) \backslash \neg \mathit{post}(o)) \cup \mathit{post}(o)). \neg l \in S_C. \tag{2}$$

That is, at least one effect of o is in S_C, and the effected literal is not a prerequisite of the preceding state (by 1), and none of the effects or preconditions of o that are not removed by $\neg \mathit{post}(o)$ are inconsistent with S_C (by 2).

We construct a set of initial incomplete runs of the form $\mathcal{R}_2 = (S' \xrightarrow{o} S'_C)$ where:

$$S' = (S'_C \backslash \mathit{post}(o)) \cup \mathit{pre}(o)$$

$$S'_C = S_C \cup (\mathit{pre}(o) \backslash \neg \mathit{post}(o)) \cup \mathit{post}(o).$$

Here, S' is the *inferred* conflict-free precursor and S'_C is the *refined* conflict specification that includes the additional constraints imposed by the new contributing operator[2].

In our example, the operator $\mathtt{pickup}_1\mathtt{(p,X)}$ contributes to conflict since it brings about S_C in the run:

$$\mathcal{R}_2 = \left\{ \begin{array}{c} \mathtt{pAt(p,X)}, \mathtt{at}_1\mathtt{(X)} \\ \neg \mathtt{hold}_1\mathtt{(p)} \end{array} \right\} \xrightarrow{\mathtt{pickup}_1\mathtt{(p,X)}} \left\{ \begin{array}{c} \neg \mathtt{pAt(p,X)}, \mathtt{at}_1\mathtt{(X)} \\ \mathtt{hold}_1\mathtt{(p)} \end{array} \right\}.$$

Run Iteration: For each set of incomplete runs $\mathcal{R}_i$ we apply a process of inference and refinement to each run and generate a set of incomplete runs $\mathcal{R}_{i+1}$ and a set of complete runs that are appended to $\mathcal{R}$. For each run $R \in \mathcal{R}_i$ we identify all *viable* successor operators o under the condition:

$$\forall l \in \mathit{last}(R) : \nexists l' \in \mathit{pre}(o).(l = \neg l' \vee l' = \neg l).$$

An operator is viable if its preconditions are consistent with the final specification in the run. This selection of viable successor operators considers all alternatives (operators with all substitution sets). For o to be applicable a refinement of $\mathit{last}(R)$ is required. Let $L_+ = \mathit{pre}(o) \backslash \mathit{last}(R)$ be the literals added to $\mathit{last}(R)$ during refinement. Since literals in L_+ are neither added nor removed by any operators in R, they must be a precondition of $\mathit{first}(R)$ and every subsequent specification. The refined run is then:

$$R' = (S_0 \cup L_+) \xrightarrow{o_1} (S_1 \cup L_+) \xrightarrow{o_2} \ldots \xrightarrow{o_i} (S_i \cup L_+) \xrightarrow{o} S_{i+1}$$

where the inferred successor state specification S_{i+1} is

$$S_{i+1} = (S'_i \backslash \neg \mathit{post}(o)) \cup \mathit{post}(o).$$

In our example $\mathtt{move}_1\mathtt{(X,Y)}$ is viable. For consistency we introduce $L_+ = \{\mathtt{conn(X,Y)}\}$ into specifications of $\mathcal{R}_2$ so that the new operator is applicable. The resulting run, $\mathcal{R}_3$, is:

$$\left\{ \begin{array}{c} \mathtt{pAt(p,X)} \\ \mathtt{at}_1\mathtt{(X)} \\ \neg \mathtt{hold}_1\mathtt{(p)} \\ \mathtt{conn(X,Y)} \end{array} \right\} \xrightarrow{\mathtt{pickup}_1\mathtt{(p,X)}} \left\{ \begin{array}{c} \neg \mathtt{pAt(p,X)} \\ \mathtt{at}_1\mathtt{(X)} \\ \mathtt{hold}_1\mathtt{(p)} \\ \mathtt{conn(X,Y)} \end{array} \right\} \xrightarrow{\mathtt{move}_1\mathtt{(X,Y)}} \left\{ \begin{array}{c} \neg \mathtt{pAt(p,X)} \\ \mathtt{at}_1\mathtt{(Y)} \\ \neg \mathtt{at}_1\mathtt{(X)} \\ \mathtt{hold}_1\mathtt{(p)} \\ \mathtt{conn(X,Y)} \end{array} \right\}.$$

Termination: We terminate on two conditions, i) if the run is complete, or ii) if the inferred successor state has been previously considered in the run (we ignore loops). The resulting synthesised norms prohibit the initial operator in each complete run that leads to conflict.

3.3.3 Reachability Analysis

In order to guarantee reachability under the synthesised prohibitions we show that for all $R \in \mathcal{R}$ a conflict-free plan Δ exists such that $\mathit{Res}(\mathit{first}(R), \Delta) = \mathit{last}(R)$. If R contains specifications that are unground, it is not possible to find alternative conflict-free plans in the general case. Here the run is ground for each unique substitution set, after which we utilise classical planning to find an alternative plan. Finally, we accept a conflict-free plan Δ as an alternative to a conflict plan Δ_C if the effects of the plans are identical: for any S where Δ and Δ_C are applicable, $\mathit{Res}(S, \Delta) = \mathit{Res}(S, \Delta_C)$.

It is preferable to ground conflict-free pairs at the reachability analysis stage of norm synthesis. The traversal runs are unground and therefore common to all problem instances of the domain. Additionally, the refinement of a run during traversal is a process whereby constraints are placed on the possible variable groundings for that run, reducing the number of such groundings. Finally grounding prior to, or during the traversal produces many norms each conditional

[2]The set of contributing operators is minimal implying that agents have equal power in the normative system [2].

on a unique variable grounding, and does not take advantage of the expressiveness of the operator schemata.

3.4 Soundness

Our approach is sound: the social norms produced always avoid the conflict specification, and all traversals through conflict are analysed for reachability. This soundness property follows directly from the propositional approach [3]. During traversal all possible operators are considered as successors, thereby ensuring that all complete runs are checked for reachability. The first-order search is exhaustive, yet instead of considering all ground operators we consider instances of each unground operator, ensuring that operators exist for all variable bindings. Given a state specification S and operator o, we wish to identify all possible parameter bindings for o applicable in any states modelled by the specification S. We unify variables with existing literals in S, grounding constants where possible, thereby considering all possible variable assignments in a general fashion.

We introduce constraints to explicitly define inequalities between variables. Consider $S = \{\texttt{parcel(p1)}, \texttt{at}_1\texttt{(x)}\}$ and operator $\texttt{pickup}_1\texttt{(P,X)}$. The table below details all variable bindings and constraints for this operator:

Operator Instances	Constraints
$\texttt{pickup}_1\texttt{(p1,x)}$	P=p1, X=x
$\texttt{pickup}_1\texttt{(p1,X)}$	P=p1, X≠x
$\texttt{pickup}_1\texttt{(P,x)}$	P≠p1, X=x
$\texttt{pickup}_1\texttt{(P,X)}$	P≠p1, X≠x

Since the resulting operator selection strategy is exhaustive, the resulting search is too, and the soundness properties from the propositional traversal follow.

4. TRAVERSAL OPTIMISATIONS

The conflict rooted synthesis search results in many complete traversal runs and corresponding reachability checks. For example, consider the delivery domain with actions as outlined in Section 2: a full traversal in this simple domain considers approximately 350000 complete runs of length 5 or less.

In this section we detail general optimisations to the traversal process, first providing an intuitive illustration of our techniques prior to delving into the formal theory. Our domain-independent optimisations exploit operator constraints to improve the norm synthesis process. Consider the following complete traversal run for $S_C = \{\texttt{at(y)}\}$:

$$\langle \texttt{move}, \texttt{destroy(p1)}, \texttt{pickup(p2, y)}, \texttt{idle}, \texttt{pickup(p3, y)}, \texttt{move}\rangle .$$

We detail each optimisation in the context of this example:

AP A Priori Filtering - operators that have no effect on the run can be ignored, greatly reducing the number of runs considered. In this example we can remove the `idle` action a priori, since it is not related to S_C.

TP Traversal Pruning - duplicate runs are not considered during traversal. If operators can be reordered to reduce the conflict run to a shorter, complete conflict run, then the longer instance can be ignored. Here, it is possible to `destroy` parcel `p1` prior to the `move` into conflict. The remaining operators form a shorter complete run.

RO Repetitive Operators - repetitive operators can share reachability plans. In the above example, if a conflict-free plan exists to pick up `p2`, then a similar plan exists to pick up `p3`, or any other parcel in `y`. We consider these repeated actions as a single instance, thereby reducing the number of runs considered.

DR Duplicate Runs - duplicate traversals are ignored. If a run is found that is a duplicate of an existing run, then the new run is not considered.

IR Incremental Reachability - reachability plans for shorter runs are modified for similar longer runs, thereby reducing the number of reachability checks required.

These optimisations never increase the search space size, make no assumptions regarding agent goals and preserve soundness by disregarding complete reachable runs. They are generally applicable in any domain that can be represented in our planning representation.

4.1 A Priori Operator Exclusion (AP)

During traversal, it seems reasonable to only consider operators applicable in conflict, or operators partially dependent on conflict states. However this is not the case: the following proposition shows that operators entirely independent of S_C must still be considered during traversal. We define $lit(o)$ to be the set of literals in o: $lit(o) = \{l | l \in pre(o) \cup post(o)\}$.

PROPOSITION 1. *Let O_D be a set of operators entirely independent of the conflict-state specification such that $\forall o \in O_D . lit(o) \cap S_C = \emptyset$. Operators in O_D cannot be ignored a priori.*

PROOF. By counterexample. Consider the operators:

$$\{z\} \xrightarrow{\texttt{A}} \{x, y, \neg z\} \qquad \{y\} \xrightarrow{\texttt{B}} \{q\} \qquad \{q\} \xrightarrow{\texttt{C}} \{\neg x\}$$

Let $S_C = \{x\}$. Operator `A` results in conflict, while `C` leaves conflict. Operator $\texttt{B} \in O_D$ since neither y nor q form part of S_C. Now consider the following complete conflict-run:

$$\{z\} \xrightarrow{\texttt{A}} \{x, y\} \xrightarrow{\texttt{B}} \{x, y, q\} \xrightarrow{\texttt{C}} \{y, q\}. \qquad (3)$$

This run achieves q through the application of `B`. Due to the side effect of `A`, even though `B` makes no reference to S_C, it is only applicable in a conflict-state. There is no conflict-free way to achieve q: the above run cannot be executed. If O_D is ignored, run (3) is not considered, and the resulting norms preserve reachability. A contradiction is reached. □

We introduce a stronger notion of operator independence, and prove how this notion can be used to exclude operators a priori. This simple approach is effective at reducing the number of operators considered, and hence the number of runs. We split the set of operators into two disjoint sets: O_C for operators dependent on each other or on S_C, and O_U for all other operators.

The following process can be used to create O_C. Initialise Λ to the set of literals in S_C and $O_C = \emptyset$. Let $O_1 = \{o | o \in O . lit(o) \cap \Lambda \neq \emptyset\}$. For each $o \in O_1$ let $\Lambda = \Lambda \cup lit(o)$ and $O_C = O_C \cup o$. We repeat the process for $O_2, O_3 \ldots$ until no new operators are added to O_C.

No plan exists where the effects of operators in O_C conflict with those in O_U. This represents universal independence: operator independence over all possible sequences of actions.

EXAMPLE 1. *A Priori Operator Filtering*
From the above example, it can be seen that neither A, B nor C are universally independent, and none can be excluded. Consider a further action $\{a, b\} \xrightarrow{\texttt{D}} \{c\}$. Here, $\Lambda = \{q, x, y, z\}$, and since no literal in Λ is referenced by `D` we

know $O_C = \{\texttt{A}, \texttt{B}, \texttt{C}\}$ and $O_U = \{\texttt{D}\}$. Therefore, operator D *can be ignored during traversal.*

PROPOSITION 2. *Let O_C be the operators dependent on, or effecting S_C. Let O_U be universally independent operators of O_C. If operators O_U are excluded from traversal, synthesis remains sound.*

PROOF. Consider a complete conflict plan Π. We separate the operators in Π into two universally independent sequences $\Pi_C = \{o | o \in O_C\}$ and $\Pi_U = \{o | o \in O_U\}$. Consider an arbitrary sequence $\{o_c, o_u\}$ where $o_c \in \Pi_C$ and $o_u \in \Pi_U$. Operator o_u can always be executed before o_c, since there are no common literals (by the definition of universal independence). We therefore rewrite the complete plan as $\Pi = \langle \Pi_U, \Pi_C \rangle$ by reordering all universal independent operators to the beginning without altering the dependencies, or effects of the plan. All operators in Π_U can be ignored when searching for a conflict-free alternative plan to Π since if Π_C can be replaced by Δ_C, then Π can be replaced by $\langle \Pi_U, \Delta_C \rangle$. $\square$

If the number of operators is $m_o = |O|$, then the complexity of a priori filtering is $O(m_o)$. This pre-processing step adds little to the computational requirements of the synthesis process, since it is independent of traversal.

4.2 Traversal Pruning (TP)

Consider the following incomplete run produced in traversal, $R = S_0 \xrightarrow{o_1} \dots \xrightarrow{o_n} S_n$: we need only consider successor actions that are conditional on the conflict state specification. We show that operators that are applicable in S_0 with effects that are not negated by any subsequent actions in the run need not be considered.

Operator o can be removed from consideration as a successor for R if the following *operator filtering* conditions hold:[3]

1. $pre(o) \cap S_n \subseteq S_0$: o is applicable in S_0,
2. $S_0 \cap \neg post(o) = \emptyset$: no operator is dependent on a literal that o affects,
3. $S_n \cap \neg post(o) = \emptyset$: no operator alters the effects of o.

PROPOSITION 3. *Only reachable runs are pruned under the operator filtering conditions.*

PROOF. We write runs simply as a sequence of operators for clarity. Consider an incomplete run:

$$R_k = \langle o_1 \dots o_{k-1}, o_k \rangle.$$

Assume R to be an extension of R_k that is not reachable:

$$R = \langle o_1 \dots o_{k-1}, o_k, o_{k+1} \dots o_n \rangle.$$

We assume that reachability holds for all other complete runs. Next, consider the complete run:

$$R' = \langle o_1 \dots o_{k-1}, o_{k+1} \dots o_n \rangle.$$

We reach a contradiction by showing that if R' is consistent and reachable, then so is R. If R' is consistent but not reachable then the traversal terminates prior to R being considered (since shorter runs are considered first). Let $\overline{R_k} = \langle o_k, o_1 \dots o_{k-1} \rangle$ be a reordered instance of R_k. We show that the conditions and effects of R_k are identical to $\overline{R_k}$ if the operator filtering conditions are met.

Let $S_0 = first(R_k)$ and $S_n = last(R_k)$. First, we show o_k is applicable in S_0. Let $L_+ = pre(o) \backslash S_n$ be the literals added during refinement, and the refined initial state be $S'_0 = S_o \cup L_+$. The remaining literals $pre(o) \cap S_n$ are already present in S_0. This is detailed in condition (1).

Next, we show $last(R_k) = last(\overline{R_k})$ by showing that the effects of o_k are preserved after reordering. Consider each effect literal $l \in post(o_k)$: the following table details the conditions where inconsistencies occur due to effects not being preserved, depending on whether l or $\neg l$ appear in S_0 and S_n. If neither l nor $\neg l$ are present, we write $\emptyset$.

S_0	S_n	Consistent	Reason
$\emptyset$	$\emptyset$	Yes	No conflicting effects
	l	Yes	l already present
	$\neg l$	No	$\neg l$ added by subsequent operator
$\neg l$	-	No	Another operator requires $\neg l$
l	l	Yes	No conflicting effects
	$\neg l$	No	Another operator adds $\neg l$

There are three conditions (highlighted above) under which an effect is not preserved, each of which is eliminated by the filtering conditions (2) and (3). Under these conditions, reordering o_k to the beginning of the sequence of actions does not alter the net effects of the run: $last(R_k) = last(\overline{R_k})$.

Finally, since o_k does not contribute to conflict the reachability of $\overline{R}$ (and therefore R) follows from the reachability of R'. Since we know the reachability of this shorter run has already been checked, then R is reachable, and a contradiction is reached. $\square$

EXAMPLE 2. *Traversal Pruning*
Let $S_C = \{\texttt{hold}_1\texttt{(P1)}, \texttt{hold}_2\texttt{(P2)}\}$ be a conflict specification prohibiting agents 1 *and* 2 *from holding parcels concurrently. Consider the partial run R_2:*

$$\left\{ \begin{matrix} \neg\texttt{hold}_1\texttt{(P1)}, \texttt{hold}_2\texttt{(P2)} \\ \texttt{at}_1\texttt{(N)}, \texttt{pAt(P1,N)} \end{matrix} \right\} \xrightarrow{\texttt{pickup}_1\texttt{(P1,N)}} \left\{ \begin{matrix} \texttt{hold}_1\texttt{(P1)}, \texttt{hold}_2\texttt{(P2)} \\ \texttt{at}_1\texttt{(N)}, \neg\texttt{pAt(P1,N)} \end{matrix} \right\}.$$

Let the successor operator o_m be $\texttt{move}_2\texttt{(N1,N2)}$*: agent* 2 *moves from some location* N1 *to* N2*. We are interested in the effects that* $\texttt{move}_2$ *has on our partial run. According to the operator filtering conditions, this action is ignored since:*

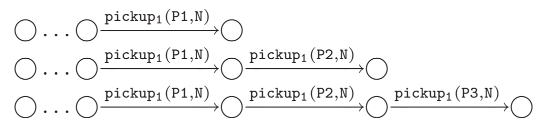
1. $pre(o_m) \cap last(R_2) = \emptyset \subseteq first(R_2)$*:* refinement would add all of $pre(o_m)$ to $first(R_2)$, so applicability holds.
2. $first(S_0) \cap \neg post(o_m) = \emptyset$*:* reordering o_m does not affect the applicability of the pickup action.
3. $last(S_n) \cap \neg post(o_m) = \emptyset$*:* pickup does not alter the effects of o_m.

Since the effects of o_m are preserved, the operator is ignored for this run, implying that $\texttt{move}_2$*'s contributions to the conflict run are achievable out of conflict.*

Let $m_l = |L|$ be the total number of literals. The computational complexity of traversal pruning is $O(m_l)$ derived from the set intersection operations in the operator filtering conditions.

4.3 Repetitive Operator Traversal (RO)

Consider the following incomplete runs created by repetitively applying pickup operators:



We aim to avoid repetition if possible, so as to minimise the resulting number of reachability checks. In this example, if

[3]We assume that o is not forbidden in S_n.

an agent is able to find an alternative plan to pick up `P1` in `N`, they could also pick up `P2`, `P3` ... if the resources required by these actions are introduced during the refinement of the run. We define conditions where operators can be considered to be repetitively applicable, and discuss the implications of this repetition on reachability checking.

A *repetitive operator* o can be applied as a successor for an incomplete run R if it does not affect a literal present in $last(R)$. A *repetitive literal* is any affected literal of a repetitive operator. We denote repetitively applied operators and repetitive literals with grammar-like $*$ suffixes, such as `move*(X,Y)`, `P*`, etc.

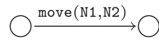
Let $\varphi = last(R) \cap pre(o)$ be the literals existing in $last(R)$ that o requires. The remaining literals are introduced during refinement. Any introduced literal can be affected, since repeated applications of the operator would simply introduce a literal for each consumed. Therefore, we are interested in the literals already present in R that are affected by o.

Operator o is repetitive if either of the following hold:

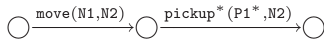
1. $\varphi \cap \neg post(o) = \emptyset$, or
2. $\forall l \in (\varphi \cap \neg post(o))$. l is repetitive.

That is, no literals are affected (1), or, every affected literal is introduced by a previous repetitive operator (2). Successor operators that consume repetitive literals must instantiate all sources of these repetitive literals into non-repetitive instances that can be affected.

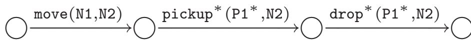
Example 3. *Repetitive Operators*
Consider the following incomplete run:



where an agent moves from `N1` *to* `N2`. *Let's consider the successor* `pickup(P1,N2)` *operator. Here,* $\varphi = \{$`at(N2)`$\}$ *and since* $\varphi \cap \neg post(o) = \emptyset$ *we consider* `pickup` *to be repetitive:*



Next, consider `drop(P1*,N)`. *Since* $\varphi = \{$`hold(P1*)`$\}$ *then condition (1) does not hold. However, since* `hold(P1*)` *is introduced through repetitive operator* `pickup*(P1*,N2)` *then* `drop(P1*,N)` *too is repetitive, and the resulting run is:*



Finally consider `destroy(P1*)` *that affects a single instance of* `P1*`. *Since this action is not repetitive, one instance* `P3` *of* `P1*` *is created by instantiating the contributing* `pickup*`*:*

We write $rep(o)$ if o is repetitive. The traversal and reachability alterations are defined for an incomplete run R:

- If $rep(o)$ then $R' = \langle R \to o \rangle$, else
- If $\neg rep(o)$ then $\forall l \in (\varphi \cap \neg post(o)).rep(l)$, let o_i affect l. Instantiate o_i to provide a non-repetitive instance of l. If o_i in turn contains repetitive literals, then repeat the instantiation for o_i.

The use of repetitive operators allows us to utilise the traversal pruning optimisations to further reduce the number of operators considered. For some successor operator o_n, if a precursor operator o_m exists that is repetitive and identical to o_n, and the effects of o_m are preserved until the end of the existing run, then o_n can be ignored.

The complexity of checking if an operator is repetitive is $O(m_l)$, where $m_l = |L|$ is the total number of literals, and is therefore independent of the length of the run.

4.4 Duplicate Runs (DR)

For completeness, we extend an optimisation from previous work: during traversal, duplicate runs can be ignored [3]. In the first-order case, two runs R_1 and R_2 are duplicates if $\exists\sigma$. $first(R) = \sigma[first(R')]$, $last(R) = \sigma[last(R')]$ and if the contributing operators (the initial operators leading to conflict in each run) are equal. This optimisation is more complex to evaluate, due to the search for variable substitution sets. We argue that this duplicate run optimisation should be invoked after all other optimisations, once the set of runs to operate on is as small as possible.

4.5 Incremental Reachability (IR)

We check the reachability of a run through an incremental modification of an existing conflict run and conflict-free reachable plan.

4.5.1 Operator Insertion

We begin by defining under what conditions an operator can be inserted into a plan without affecting the applicability of the other operators.

Example 4. *Operator Inserting*
Consider the following plan in the parcel delivery domain

$$\Pi = \langle \texttt{move(N1,N2)}, \texttt{drop(P1,N2)} \rangle$$

where an agent moves into `N2` *and drops* `P1`. *Let* o^* *be either* `drop(P1,N1)` *or* `drop(P2,N1)`, *in the resulting plan* $\Pi' = \langle o^*, \texttt{move}, \texttt{drop} \rangle$. *Action* `drop(P1,N1)` *results in an inconsistency in the run, since a future action* `drop(P1,N2)` *requires the agent to hold* `P1`, *whereas* `drop(P2,N1)` *is admissible, since the agent is dropping* `P2` *which does not affect its ability to drop* `P1` *in* `N2` *(assuming* `P1` $\neq$ `P2`*).*

Consider a sequence of operators $\Pi = \langle o_1 \dots o_k, o_{k+1} \dots o_n \rangle$. A modified sequence $\Pi' = \langle o_1 \dots o_k, o^*, o_{k+1} \dots o_n \rangle$ is the original sequence Π with the new operator o^* inserted arbitrarily. For simplicity, we write $\Pi = \langle \Pi_1; \Pi_2 \rangle$ and $\Pi' = \langle \Pi_1; o^*; \Pi_2 \rangle$. Assuming Π is consistent, we now define conditions under which Π' is consistent. Let

$$\mathit{eff}(\Pi) = (pre(\Pi) \backslash \neg post(\Pi)) \cup post(\Pi)$$

be the net effects of the sequence of operators in Π.

Proposition 4. *Operator* o^* *can be inserted into* Π *to form* Π' *without conflicting with existing operators iff:*

1. $\mathit{eff}(\Pi_1) \cap \neg pre(o^*) = \emptyset$, *and*
2. $pre(\Pi_2) \cap \neg post(o^*) = \emptyset$.

Proof. Π' is inconsistent if the effect of some precursor operator in Π_1 results in o^* being inapplicable, or the effects of o^* result in a subsequent operator being inapplicable.

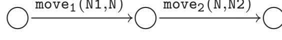
Firstly, assume o^* is not applicable. For this to be the case a literal $l \in pre(o^*)$ exists such that $\neg l$ is an effect of Π_1, or is a necessary precondition of Π_1. However, from (1), since $\mathit{eff}(\Pi_1)$ incorporates all effects, as well as preconditions not affected, no such l exists, and o^* must be applicable.

Next, assume the effects of o^* result in a subsequent operator o' no longer being applicable. Since o' was applicable in Π, this conflict must be caused by an effect of o^* (since all intermediate operators are the same). Let $l \in pre(o')$ be a literal where $\neg l \in post(o^*)$. It follows that $l \in pre(\Pi_2)$ since a subsequent operator prior to o' adds l. However a contradiction is reached, since by (2) no such literal can exist. $\square$

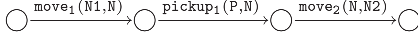
4.5.2 Checking Incremental Reachability

Conflict traversal is a breadth-first search of complete conflict runs where reachability is checked for runs of increasing length. We now present an alternative way of checking reachability that incrementally builds on the reachability of shorter runs. We begin with a simple example.

EXAMPLE 5. *Incremental Reachability*
Consider the shortest possible complete conflict plan in the parcel delivery domain for $S_C = \{\texttt{at}_1\texttt{(N)},\texttt{at}_2\texttt{(N)}\}$ where agents collide at N:



Suppose that we show a conflict-free alternative plan Δ exists for this conflict run. As our traversal continues, we begin to consider conflict-runs composed of three actions, such as:



where agent 1 *performs the same* move *actions, but also picks up a parcel* P *in* N*. A conflict-free plan exists for this run: the* pickup *action can be performed directly after Δ to achieve the correct effects. We adopt this incremental approach in order to reduce the number of conflict-runs considered.*

Let $\Pi=\langle\Pi_1;\Pi_2\rangle$ and $\Pi'=\langle\Pi_1;o^*;\Pi_2\rangle$. Assume that a conflict-free plan Δ exists for Π. Given that Π' is an incremental extension of Π, we are interested in showing whether o^* can be inserted in Δ to form an alternative plan Δ' for Π'.

We begin by characterising the effects of o^* that are preserved or overwritten. Let $\mathit{eff}(\Pi)$ and $\mathit{eff}(\Pi')$ be the net effects of Π and Π' respectively. Let $E = \{\mathit{post}(o^*)\backslash\neg\mathit{post}(\Pi_2)\}$ be the effects preserved and $\overline{E} = \{\mathit{post}(o^*)\backslash E\}$ be the effects overwritten. Since there are no conditional effects in our formalism all differences between the runs can be attributed to the operator o^*: $E \subseteq \mathit{post}(o^*)$.

PROPOSITION 5. *Consider $\Pi = \langle\Pi_1;\Pi_2\rangle$ with alternative plan $\Delta = \langle\Delta_1;\Delta_2\rangle$. Let $\Pi' = \langle\Pi_1;o^*;\Pi_2\rangle$ represent Π with operator o^* inserted. Π' is reachable with alternative plan $\Delta' = \langle\Delta_1;o^*;\Delta_2\rangle$ if:*

1. *Δ' is consistent (by Proposition 4),*
2. *$\big(\mathit{pre}(o^*)\backslash\mathit{eff}(\Pi_1)\big) = \big(\mathit{pre}(o^*)\backslash\mathit{eff}(\Delta_1)\big)$: the refined preconditions match,*
3. *$E \subseteq \mathit{post}(\Delta_2)$: all effects in E are preserved,*
4. *$(\neg\mathit{post}(\Delta_2) \cap \overline{E}) = \emptyset$: all effects in $\overline{E}$ are overwritten,*
5. *the result is conflict-free.*

PROOF. In order to show that Π' is reachable by Δ', we must show the following:

i $\mathit{pre}(\Pi') = \mathit{pre}(\Delta')$,
ii $\mathit{post}(\Pi') = \mathit{post}(\Delta')$,
iii Δ' is conflict-free and consistent.

Since Δ is a conflict-free alternative to Π, $\mathit{pre}(\Pi) = \mathit{pre}(\Delta)$ and $\mathit{post}(\Pi) = \mathit{post}(\Delta)$ hold.

Consider (i): let $L_+ = \mathit{pre}(o^*)\backslash\mathit{eff}(\Pi_1)$ be the literals that are added to the initial conditions during refinement of Π', and L'_+ be those added during the refinement of Δ'. By (2) $L'_+ = L_+$, and since $\mathit{pre}(\Pi) = \mathit{pre}(\Delta)$ by definition if follows that $\mathit{pre}(\Pi') = \mathit{pre}(\Delta')$.

Consider (ii): the difference in effects between Π and Π' are all attributable to o^*. Condition (3) ensures that all literals added to the effects of Π' by o^* will also be added to Δ', while (4) ensures that any effect of o^* removed by a subsequent operator in Π' will be removed in Δ' too. Therefore, since $\mathit{post}(\Pi) = \mathit{post}(\Delta)$, the effects of Δ' and Π' are identical, and $\mathit{post}(\Pi') = \mathit{post}(\Delta')$.

Finally, (iii) follows directly from (1) and (3). □

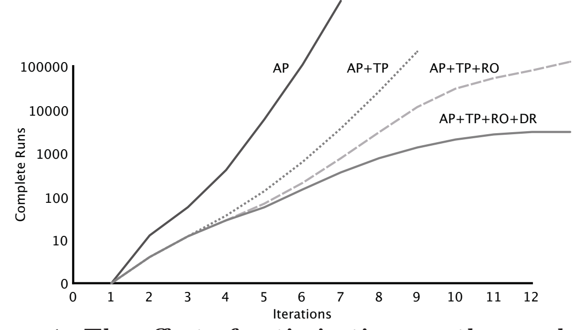


Figure 1: The effect of optimisations on the number of runs generated during traversal.

This result is significant in a first-order domain formalism. The ability to adapt existing reachability plans for new runs results in fewer reachability checks and fewer variable groundings. As will be demonstrated in the next section, the number of reachability checks can be significantly reduced.

5. EVALUATION AND DISCUSSION

We now quantify the traversal improvements in the parcel domain. The ordering of optimisations affects their relative effectiveness, with earlier optimisations operating on more runs than subsequent ones. We present a simple screen test where each optimisation is enabled in turn. The traversal runs for 5 iterations (the outcomes are deterministic) with $S_C = \{\texttt{at}_1\texttt{(X)},\texttt{at}_2\texttt{(X)}\}$. Total number of complete and incomplete runs is recorded, along with the percentage improvement over a full search.

Optimisation	Incomplete	% Diff.	Complete	% Diff.
Full Search	13631844	-	349288	-
AP: A Priori Filtering	4749104	65.16%	155008	55.62%
TP: Traversal Pruning	244	99.99%	132	99.99%
RO: Repetitive Operators	1075449	92.11%	77820	77.72%
DR: Duplicate Runs	43157	99.68%	8052	97.69%

We utilise all optimisations concurrently with an ordering based on each optimisation's computational complexity. First we apply a priori filtering prior to traversal. The traversal pruning follows due to its computational efficiency and effectiveness. The repetitive operator optimisation follows. Finally, due to its complexity, we invoke the duplicate run filtering. Figure 1 depicts the number of runs generated on a logarithmic scale against the number of iterations, detailing the performance gains as each optimisation is added: The improvement after each optimisation is clear. With all optimisations enabled, no new runs are generated after 11 iterations. The solved problem has 3052 complete runs, compared to 13631844 after just 5 iterations of full search.

5.1 Incremental Reachability

Finally, we illustrate the benefit of incremental reachability checking. We generate 3052 unique complete runs that either prohibit $\texttt{move}_1\texttt{(N1,X)}$ or $\texttt{move}_2\texttt{(N1,X)}$ (moving to X), and leave conflict through $\texttt{move}_1\texttt{(X,N2)}$ or $\texttt{move}_2\texttt{(X,N2)}$ (leaving X). Incremental reachability allows us to produce alternative plans for all these sequences based on the reachability of simpler sequences of move operators alone, resulting in grounding only 4 times.

No operators, besides move, alter the location of the agent, even though some are conditional on the location. It is al-

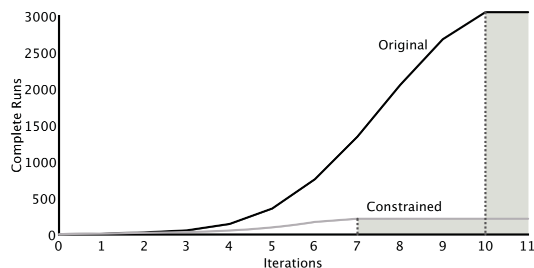


Figure 2: Simplifying traversal using domain-specific constraints.

ways possible to execute non-`move` actions in a conflict-free way if the required portion of the conflict state is accessible in the alternative plan, with the remainder avoided to ensure no conflict. In the above example, if the problem instance allows both agent `1` and `2` to access location `X` independently, then conflict-free plans exist for any combination of actions performed in `X`. As a result, reachability need only be shown for the sequence $\langle \texttt{move}_i\texttt{(N1,X)}, \texttt{move}_i\texttt{(X,N2)}\rangle$ (with $i = \{1, 2\}$). All other conflict-free plans can be derived from a conflict-free solution to the above sequence, so long as agent `1` and `2` can access `X` independently.

5.2 Domain Constraints

Traversal with no knowledge of the initial system state results in general, problem independent results: in the parcel domain norms govern situations where an agent occupies multiple locations concurrently, thereby covering problems where agents start in multiple locations. Our search can be improved by introducing domain constraints that introduce these assumptions. Consider, for example:

$$\{\texttt{at}_1\texttt{(A,X)}, \texttt{at}_2\texttt{(A,Y)}, \texttt{X=Y}\}.$$

Incorporating this knowledge into the search process is simple. For every run, if a substitution exists that violates the constraint, then the run can be discarded. The impact on traversal performance is shown in Figure 2. The additional knowledge reduces the total number of complete runs from 3052 to 216, and iterations required for a solution from 10 to 7. The relative performance benefits are clear: while the optimisations provide a significant improvement in general, domain specific knowledge can further reduce the number of runs considerably.

6. RELATED WORK

Shoham and Tennenoltz [8, 9] provided the first formal notion of behavioural constraints in a state-based system, and showed the problem of norm synthesis to be NP-complete. No generally applicable algorithm accompanied the complexity result. This work was extended by Onn and Tenneholtz [7], where prohibitions are synthesised in a robot mobilisation domain. An efficient process is detailed, however the solution is domain-specific operating only on a fixed graph topology.

Fitoussi and Tennenholtz [4] described the synthesis of *minimal* and *simple* prohibitions. This work is concerned with prohibition refinement rather than synthesis as a prohibition is assumed provided by the designer. In [10] van der Hoek, Roberts, and Wooldridge analyse properties of social norms in domains specified in Alternating-time Temporal Logic, and reduce synthesis to a model checking problem. This work assumes a complete action-based alternative transition system representation which is often not feasible to develop and maintain in practical systems.

7. CONCLUSIONS

In this paper we detail a first-order approach to norm synthesis. The ability to synthesise norms containing variables provides greater expressiveness and subsequently more general social norms. However, the computational complexity of this unground search is significant, since first-order operator schemata contain less problem-specific knowledge. A set of optimisations is detailed that improves the performance of first-order synthesis without jeopardising the soundness of the underlying process. We prove these optimisations correct, and empirically detail their impact in a simple grid-world domain. The optimisations greatly reduced the number of runs considered, allowing for the sample domain to be solved completely.

As future work, we plan to formalise the inclusion of domain constraints in the traversal process. These constraints enable the process to be applied to larger, more complex systems. Furthermore, an extended empirical evaluation of the optimisations across varying domains will allow for more general performance statements. An extension of our synthesis process to facilitate the synthesis of sanctions would automate prohibition norm synthesis for enforcement based normative systems.

8. REFERENCES

[1] T. Ågotnes, W. van der Hoek, J. Rodriguez Aguilar, J. Sierra, and M. Wooldridge. The simple normative systems language. In *Agent Organizations: Models and Simulations, IJCAI 07 Workshop (AOMS 2007)*, January 2007.

[2] T. Ågotnes, W. van der Hoek, M. Tennenholtz, and M. Wooldridge. Power in normative systems. In *Proc. of the 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 145–152, Budapest, Hungary, May 2009.

[3] G. Christelis and M. Rovatsos. Automated norm synthesis in agent-based planning environment. In *Proc. of the 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 161–168, Budapest, Hungary, May 2009.

[4] D. Fitoussi and M. Tennenholtz. Choosing social laws for multi-agent systems: Minimality and simplicity. *Artificial Intelligence*, 119:61–101, 2000.

[5] F. López y López, M. Luck, and M. Dinverno. A normative framework for agent-based systems. *Computational and Mathematical Organization Theory*, 12(2-3):227–250, Oct. 2006.

[6] B. Nebel. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence*, 12:271–315, May 2000.

[7] S. Onn and M. Tennenholtz. Determination of social laws for multi-agent mobilization. *Artificial Intelligence*, 95:155–167, Jun 1997.

[8] Y. Shoham and M. Tennenholtz. On the synthesis of useful social laws for artificial agent societies. In *Proc. of the 10th National Conference on Artificial Intelligence*, pages 276–281, 1992.

[9] Y. Shoham and M. Tennenholtz. On social laws for artificial agent societies: Off-line design. *Journal of Artificial Intelligence*, 73(1-2):231–252, Feb. 1995.

[10] W. van der Hoek, M. Roberts, and M. Wooldridge. Social laws in alternating time: Effectiveness, feasibility, and synthesis. *Synthese*, 156(1), May 2007.