The Normative Supervisor: a Tool for Reinforcement Learning with Norms

Emery A. Neufeld

March 8, 2022

1 Architecture

The normative supervisor is an external module that can be used for both real time compliance checking (RTCC) and norm guided reinforcement learning (NGRL). It is composed primarily of (front-end and back-end) translator modules and a reasoner module.

The supervisor was originally employed for real time compliance checking in [Neufeld et al., 2021]; this entails, with each state transition, translating the agent's current state and the norms it is subject to into a theory of some logic for normative reasoning, and feeding it into a theorem prover, the output of which is parsed into a set of compliant actions (or minimally non-compliant actions, if no compliant actions exist), and sent to the agent. This effectively filters out non-compliant actions from the set of possible actions from which the agent can choose an optimal action. The process of RTCC is depicted in Figure 1.

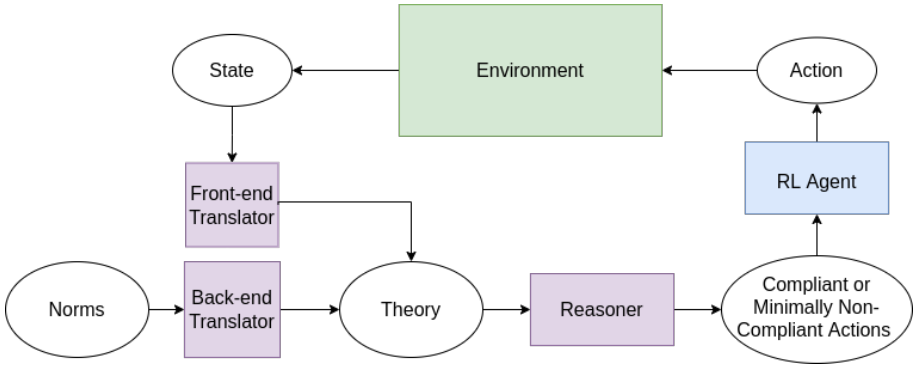

Figure 1: Purple boxes are the components of the normative supervisor.

In the below sections, we provide a sketch each of these modules.

1.1 Translation

The normative supervisor employs two translator modules, one front-facing one that interfaces with the agent, and one back-facing one that translates the normative system the agent is subject to. In the actual architecture of the supervisor, these two translators are implemented in the same class, a subclass of the abstract class `Translator` (which must all take an instance of the class `NormBase` in the constructor). For each new reasoner, a custom translator is needed because it serves to translate norms (represented as instances of one of the subclasses of `Norm`) and facts (represented

as attributes of the `Environment` class or one of its subclasses) into the language that the reasoner utilizes.

1.1.1 Front End

The front-end translator is in perpetual use, processing new data and proposed actions from the agent as the environment changes. It amounts to an algorithm that transforms input data from the agent into propositions which assert facts about the agent or the environment.

In the simplest case, when dealing with an environment modelled as a labelled Markov decision process, all the front-end translator must do is fetch the `label` attribute of the current `Environment` instant, and from each label construct a fact in the form of a literal in whatever language the reasoner employs. In more complex cases (like the Pac-Man game), the translator uses pre-defined predicates over attributes of the corresponding `Environment` subclass instant (like the locations of players represented as instances of the `GameObject` class) to construct the appropriate literals or rules.

1.1.2 Back End

The back-end translator translates the normative system the agent is subject to (in the form of a norm base that contains the individual norms that construct this system) into whatever kind of rules the reasoner works with. Together with the facts translated by the front-end translator, these compose the *GameState Theory* that will be input into the reasoner.

In the existing back-end translator components (for defeasible deontic logic), methods which generate representations of norms in the reasoner language from the elements of the translator class's `NormBase` attribute are all called in the `update(Environment, ArrayList<String>, Game)` method. Thus, they are called at every time step. This is because the number of rules in a defeasible deontic logic theory greatly impacts the efficiency with which the reasoner module can derive permissible actions from theory. Every `Norm` instance has a (possibly empty) list of contexts in which the norm applies. Thus, there are some generation methods which only construct DDL rules when this context is currently realized.

The way the back-end translator is constructed will depend largely on the language of the reasoner; for an example of constructing such a translator in the case of the Pac-Man game, see [Neufeld et al., 2021].

1.1.3 Default Rules: Non-Concurrence

In most cases, subclasses of `Translator` will have a method for generating *non-concurrence rules*, which will call `NormBase.generateNonConcurrence(ArrayList<String>)`. These methods generate rules which, for each possible action *action*, asserts that *action* counts as $\neg action'$ for each other action $action'$. This accounts for the fact that at every time step, the RL agent must select only a single action.

1.2 Reasoner

The reasoner is at the core of the normative supervisor, and its key ingredient is a theorem prover (or other reasoning engine). The reasoner's main job is to process the output this theorem prover generates when it is fed a GameState Theory. This output is used to construct a set of compliant actions (or, in case this set is empty, a set of minimally non-compliant actions). Therefore, the reasoner has methods both to parse out a set of compliant actions, and also to further utilize the theorem prover to determine which non-compliant actions will violate the normative system the least (when there are no compliant actions). In order to facilitate NGRL, the reasoner also needs a method for determining whether a given action is forbidden in a specific GameState Theory.

How these methods are implemented will depend entirely on the reasoning engine that is utilized. However, we use the two reasoners for defeasible deontic logic (`DDPLReasoner` and `DDPLReasoner2`) as a case study to demonstrate two possible approaches below.

1.2.1 First Reasoner Model

The original reasoner implemented (see the `DDPLReasoner` class) was designed with optimal performance in mind; that is, the goal of this implementation was to utilize the reasoning engine (SPINdle, in this case) as little as possible.

```
input : GameState, possible
output: legalActions
begin
    legalActions ← possible;
    reasoner ← SPINdle.Reasoner;
    conclusions ← reasoner.generateConclusions(GameState);
    if ¬conclusions.complete() then
        return ∅
    end
    for act ∈ possible do
        if conclusions.has(O(act)) then
            return {act}
        end
        else
            if conclusions.has(O(¬act)) then
                legalActions.remove(act);
            end
        end
    end
    return legalActions
end
```

Algorithm 1: `DDPLReasoner.findCompliantActions()`

In Algorithm 1, we describe in pseudocode how a set of compliant actions is derived from the GameState theory in the `DDPLReasoner` class. Simply put, we look directly for obligations or prohibitions of possible actions in the conclusions generated by the reasoning engine. When this algorithm returns an empty set, this triggers an algorithm which selects minimally non-compliant actions, which entails running the reasoning engine again, for each possible action (see Algorithm 2).

In this first framework for reasoning, we run the reasoning engine once when we look for compliant actions, and then $|possible|$ times when we look for minimally non-compliant actions (which also requires us to look at the inference log of the reasoner in order to determine degrees of non-compliance); thus, in most cases, we will only need to call the reasoning engine once per timestep. However, this efficienccy comes with a cost.

In order for this reasoning framework to be effective, we must be able to directly derive the explicit obligation or prohibition of the agent's possible actions from the GameState theory. In cases where we have a complex structure of constitutive norms, this can require some creative engineering of the norm base (see the formulation of the Pac-Man norm bases shown in [Neufeld et al., 2021] and [journal paper]).

Summarily, this reasoning model is computationally efficient (only requiring multiple runs of the reasoning engine when no compliant action is possible), but excludes some norm bases and

```
input : GameState, possible
output: bestActions
begin
    reasoner ← SPINdle.Reasoner;
    scores ← ∅;
    for act ∈ possible do
        GameState_act ← GameState;
        GameState_act.addFact(O(act));
        reasoner.generateConclusions(GameState_act);
        logger ← InferenceLogger(reasoner);
        applied ← logger.getRules(status = APPLIED);
        defeated ← logger.getRules(status = DEFEATED);
        score ← applied.size() − defeated.size();
        scores.add([act, score])
    end
    max ← max(scores);
    bestActions ← scores.getActions(score = max);
    return bestActions
end
```

Algorithm 2: `DDPLReasoner.findNCActions()`

necessitates significant preprocessing for others.

1.2.2 Second Reasoner Model

The second defeasible deontic logic reasoner (implemented as the class `DDPLReasoner2`) provides a less efficient, but more comprehensive approach to to discerning which possible actions are legal. Algorithm 3 illustrates in pseudocode the reasoning behind this approach.

In this second framework, for each possible action, we amend the GameState theory, adding as a literal a given action, and then run the reasoning engine, storing the result of the computation in a map *conclusions* with the given action as the key. We then check, for each literal *lit* in the GameState theory, whether we can prove $\mathbf{O}(lit)$ but not *lit*; if so, then the given action results in a violation, and we exclude it from our list of compliant actions. In essence, the intuition behind this approach is the definition of a violation, in which something that is obligatory but is not the case.

Algorithm 4 describes how we select minimally non-compliant actions in this framework; note that we need only input the *conclusions* map output by Algorithm 3, and we do not need to run the reasoner again. Degrees of non-compliance are determined by how many obligations are provable but not complied with (that is, we cannot prove the non-modal literal).

Summarily, this framework is in general more computationally demanding than the first, but is extremely versatile and does not require that we prove the prohibition or obligation of particular actions.

1.3 Integration: the Server

The normative supervisor interfaces with the agent through the server. Once the normative module server is running, the agent can query the normative supervisor for action recommendations (in the case of RTCC) or a compliance check (in the case of NGRL). When the query is sent, it must include the necessary data to update the GameState Theory through the translators.

```
input : GameState, possible
output: conclusions, legalActions
begin
    legalActions ← possible;
    reasoner ← SPINdle.Reasoner;
    conclusions ← ∅;
    for act ∈ possible do
        GameState_act ← GameState;
        GameState_act.addFact(act);
        concl ← reasoner.generateConclusions(GameState_act);
        conclusions.add([act, concl]);
        for lit ∈ GameState.literals do
            if concl.has(O(lit)) ∧ ¬concl.has(lit) then
                legalActions.remove(act)
            end
        end
    end
    return conclusions, legalActions
end
```

Algorithm 3: `DDPLReasoner2.findCompliantActions()`

```
input : conclusions
output: bestActions
begin
    reasoner ← SPINdle.Reasoner;
    scores ← ∅;
    for act ∈ conclusions.getActions() do
        concl ← conclusions.getConcl(action = act);
        score ← 0;
        for lit ∈ GameState.literals do
            if concl.has(+∂_O lit) ∧ ¬concl.has(+∂_C lit) then
                score + +;
            end
        end
    end
    min ← min(scores);
    bestActions ← scores.getActions(score = min);
    return bestActions
end
```

Algorithm 4: `DDPLReasoner2.findNCActions()`

2 Running Tests

Below are the instructions on how to use the normative supervisor as is (without adding any games or reasoning engines).

2.1 Prerequisites

The normative supervisor is written in Java (required: Java 8), and the games it interfaces with are written in Python (required: Python 2.7).

To run the normative supervisor, you only need two things:

- The runnable jars supplied: `ns_server.jar` and `ns_lab.jar`.
- Whatever game you want to run it on; for example, to use the Pac-Man game, run the `pacman.py` script in the `pacman` folder.

The only libraries you need in order to run the games in python are whatever modules are used in the code for the game (e.g., in the case of the Pac-Man game, `Tkinter`).

In order to compile the supervisor from its Java code, you will need `org.json` (which can be downloaded from `https://mvnrepository.com/artifact/org.json/json`) and SPINdle 2.2.4 (which can be downloaded from `http://spindle.data61.csiro.au/spindle/download.html`).

2.2 Normative Supervisor Laboratory

The normative supervisor laboratory (`ns_lab.jar`) allows you to input an individual state representation and view the corresponding output of the normative supervisor. The lab prints a representation of the state's GameState Theory and then the supervisor's output, which is a set of actions, and an indication of whether they are compliant or not. Generally, these individual state representations will take the form depicted in Figure 2.

```
{
 "name": <game>,
  "norms": <norm base>,
  "reasoner": <reasoner>,
  "labels": [ <labels> ],
  "id": 0,
  "request": "FILTER",
  "possible": [ <possible actions> ]
}
```

Figure 2: Template for individual state representation.

In order to test the output for a single state, simply run the following:

```
java -jar ns_lab.jar <path-to-state-file>
```

The laboratory is useful primarily for debugging purposes, but it can be useful (e.g., as in [journal paper]) for comparing the outputs for different norm bases or reasoners.

2.3 Normative Supervisor Server

In order to run a more extensive battery of tests, you will need to use the normative supervisor server. For ease of use, a bash script (`run.sh`) has been supplied. To use it, open the file and edit the fields in the portion explicitly marked for configuration. Here you can choose the game you want to use the supervisor with, the norm base and reasoner you want to use, whether you want

to perform real-time compliance checking (RTCC) or norm-guided reinforcement learning (NGRL) or both, the name of the agent you want to play the game, the feature extractor used by the agent (if it is using function approximation), the weight of the ethical policy (if using NGRL with linear scalarization), the number of games you want to train the agent on, the number of games you want to test the agent with, the name of the file you want the results to be written to, the layout/game map you want the agent to play the game in, whether or not you want to fix the random seed in the game, and whether or not you want the game graphics displayed (in the case of the Pac-Man game).

For an example, see Figure 3.

```
#------------------ CONFIGURE ------------------
game='pacman'
normbase='vegan'
reasoner='DDPL'
agent='ApproximateWeightedAgent'
approximated='yes'
extractor='HungryExtractor'
weight='10'
num_train='200'
num_games='100'
record='test'
RTCC='yes'
NGRL='yes'
fixed_seed='yes'
graphics='yes'
layout=''
```

Figure 3: ApproximateWeightedAgent (weight on ethical policy is 10, feature extractor used is the HungryExtractor) playing Pac-Man for 100 games on the default layout (results written to `test.csv`) after 200 training games. NGRL and RTCC are performed with the DDPL reasoner for the vegan norm base, and the random seed is fixed; the test games will be displayed.

3 Extending the Framework

3.1 Adding New Norm Bases

Norm bases are the collections of norms that represent a normative system. Different kinds of norms are constructed from one or more Term objects. In the normative supervisor, they contain five sets (stored as Java ArrayLists) of norms:

- Regulative norms: regulative norms are norms that involve some deontic modality. In this case, however, we only store prescriptive norms (prohibitions and obligations) in this set (strong permissions are given their own set). These norms – like all extensions of the Norm class – contain a (possible empty) ArrayList of Terms *conds* representing the conditions under which the norm is triggered. There must also be a prescription, which is a Term p that has a deontic modality assigned to it. So all regulative norms take the form $*(p|conds)$ where $* \in \{\mathbf{O}, \mathbf{F}\}$ (where $\mathbf{O}(p)$ is the obligation of p and $\mathbf{F}(p)$ is the prohibition of p).

- Exception norms: these are where we store strong permissions (because often, during translation, they will have to be implemented differently from prohibitions and obligations). These look exactly like other regulative norms, except $* \in \{\mathbf{P}\}$

- State constitutive norms: there is only one ConstitutiveNorms class, but again, whether these counts-as relations are over state properties or actions will often affect how the translator should handle them, so they are kept in separate sets. These norms also have *conds* attribute, and have a *lower term* (a more concrete property of a state) and a *higher term* (a more abstract property of a state), where the lower term *counts as* the higher term.

- Action constitutive norms: in form these are identical to state constitutive norms, except the lower and higher terms are both representing an action.
- Priority norm: this is not a subclass of the Norm class; these could be viewed as 'meta norms'. A priority norm is just the name of two norms (as String), and indicates that one is of higher priority of the other.

In order to add a new norm base, there are a few things you will have to do:

1. If the norm base is for an existing game, you must edit the file <Game>NormBase.java. If it is for a new game, you must create this file for a subclass of NormBase, in supervisor/games/<game>.
2. You should add a new method with the name generate<normbase>() in which the required terms are constructed, and from them, the norms that define the normative system.
3. You need to modify the defaultNormBase() method in ProjectUtils.java to call the method you added to <Game>NormBase.java when the normbase name is input.

3.2 Adding New Games

Adding new games to the normative supervisor framework will in most cases be very simple. Below, we review the steps involved in this simplest case and in more complex instances.

1. Start by creating a new folder in supervisor/games with the name of the new game.
2. In the simplest case (where the game environment is a simple labelled MDP), we will only need to add one file to the new folder: <Game>NormBase.java. The process for adding new norm bases is detailed in the previous section.
3. In more complex cases (e.g., the Pac-Man game), additional files may need to be added to supervisor/games/<game>. Namely, more complex games that cannot be modelled as a mere labelled MDP may require subclasses of the Game, GameObject, or Environment classes that are specific to the new game. You may also need to extend the NormativeSupervisor class and translators. In the former case, you will need to modify run(String) in NormativeModuleServer.java to instantiate the game-specific NormativeSupervisor subclass.
4. In util/ProjectUtils.java, modify defaultState(String, String, String) by adding a new *if* clause which defines a default string for the initial state of the game, formatted as one of the JSON strings that are passed from the agent to the supervisor.

4 Adding new Reasoners

There are several steps that must be implemented in order to add a new reasoner to the normative supervisor:

1. A new subclass of the abstract class Reasoner must be created in the folder normative_supervisor/supervisor/reasoner. The methods necessary for the use of the new reasoner are declared in the abstract class:
   - update(): a method to update the GameState Theory input into the reasoning engine.
   - reason(): a method to run the reasoning engine with the GameState Theory as input.
   - findCompliantActions(): a method to parse the output generated in reason() into a set of compliant actions.

- `checkActionCompliance(String)`: a method to check whether a specific action is compliant.
- `findNCActions()`: a method to be called when `findCompliantActions()` returns an empty set. Will utilize some kind of metric to determine which actions are more or less compliant.
- `printTheory()`: to print the translated GameState Theory.

See `Reasoner.java` for more detailed descriptions of what these methods must do.

2. A new subclass of the abstract `Translator` class must be created in `normative_supervisor/supervisor/normsys`. It must be able to translate each type of norm into whatever formal expressions are used to compose the GameState theory for the reasoning engine in use. Thus it should have generation and getter methods for regulative norms (prescriptive norms), exception norms (permission norms), state constitutive norms, action constitutive norms, and priority norms. It must include an `init(Environment, ArrayList<String>)` method (which doesn't need to do anything in the case of a non-game-specific translator), and an `update()` method that calls all the generation methods.

   - Some games (like Pac-Man) require custom translators because they are fed environmental data from the agent instead of labels. In these cases the norm base will contain norms containing terms that are predicates, which must be translated into facts or rules in the language of the reasoning engine.

     As an example, in addition to the methods described above, Pac-Man's translators must also contain the following:

     (a) During the `init(Environment, ArrayList<String>)` method, 2-dimensional arrays of objects (in the DDPL translator, these are literals) representing the location of game entities should be constructed.

     (b) A method for translating the data in a `PacmanEnvironment` object to whatever construct is used to describe facts in the reasoning engine's language should exist and be called in `update(PacmanEnvironment, ArrayList<String>, Game)`.

     (c) Methods translating terms that are predicates to facts or rules should exist.

3. You must modify the class `Game` in `normative_supervisor/supervisor/games/Game.java`. In particular, modify the `init()` method by adding a new *if* clause for whatever name you have assigned the reasoner (e.g., "DDPL") in which the new reasoner and translator are assigned to the `reasoner` and `translator` variables respectively, and the translator is initialized. If the reasoner is supposed to work for the Pac-Man game, for example, `PacmanGame.java` must be similarly modified.

References

[Neufeld et al., 2021] Neufeld, E., Bartocci, E., Ciabattoni, A., and Governatori, G. (2021). A normative supervisor for reinforcement learning agents. In *Proceedings of CADE 28 - 28th International Conference on Automated Deductions*, pages 565–576.