The Normative Supervisor: a Tool for Reinforcement Learning with Norms

Emery A. Neufeld

February 22, 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 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, which's output 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.

In the below sections, we provide a high-level 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. For each new reasoner, a custom translator is needed because it serves to translate norms and facts 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

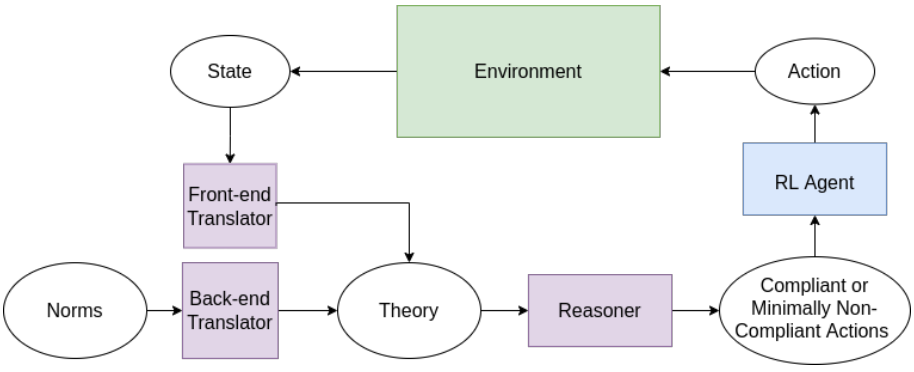

Figure 1: Purple boxes are the components of the normative supervisor.

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 construct facts from literals in whatever language the reasoner employs. In more complex cases (like the Pac-Man game), the translator uses pre-defined predicates over the game data (like the locations of the players) to construct the appropriate literals.

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.

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

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 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.

Summarily, there are several methods that every reasoner must have:

- `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 action)`: 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.

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.

2 Running Tests

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

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

Figure 2: Template for individual state representation.

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.

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 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.

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; see `Reasoner.java` for 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.