Learning and discovering norms in the context of multi-agent systems

Duangtida Athakravi

Supervisor: Dr. Alessandra Russo
Second Marker: Dr. Krysia Broda

June 21, 2011

Abstract

The normative framework, also known as institutions, is a way to formally model social rules and conventions of agent's organisation. These rules coordinate the overall behaviour of agents and describe their effects on the organisation. Just like any models, it can contain errors and consequently does not correctly reflects the system as intended by the designer.

The aim of this project is explore how such partial models may be improved using a revision method centred around use cases for capturing the model's behaviour. Past work have shown how Inductive Logic Programming (ILP) can be used to find revision suggestions from a partial model and use case. The complete model is then built up iteratively, by revising the partial model using the output of the learner, and then using the revised model in another revision cycle for learning revision suggestions, based on another use case capturing a different behaviour from the other previously one. However, as the learner can give many alternative ways for revising the model, the designer must choose between all the revision suggestions for the one that they think is most appropriate for the system.

This selection process can be confusing for the designer as it may not be obvious how the revised model will behave, judging by the revision suggestion alone. This project is aimed to find a method that the designer could employ to find differences between each revision suggestion, to provide them with a criteria to use for deciding between the suggestions.

Acknowledgements

Firstly, I would like to acknowledge the continuous support and feedback of my supervisor, Dr. Alessandra Russo, throughout the year, and for her innovative ideas that made this project possible. I would like to also thank my second supervisor Dr. Krysia Broda, for her help with the project.

I am also indebted to Domenico Corapi for his numerous advices on the project direction, his help and contribution on ASP implementation of TAL, and the time he gave to explanation TAL and inductive logic programming.

Lastly, I would like to thank to my family and friends for their support throughout my studies.

Contents

Chapter 1

Introduction

1.1 Motivation

Between the transition of a system specification to its implementation, models provide a high-level abstraction of the system without the complexity of the implementation details. It is a powerful tool that can be used for reasoning about the system, as it can be model checked for the satisfiability of given properties, and for estimation of the performance of the implemented system. Thus, for it to be of any use, it is important that the model correctly reflects the system specification.

We are interested in models for describing an organisation in a multi-agent system, the normative framework, also known as norms or institutions. In particular, how a partial normative framework can be revised to create a more complete one with respect to the system specifications. Previous work [1] have shown how normative frameworks declared in a formal declarative language can be revised using inductive learning through ASPAL, an Answer Set Programming (ASP) implementation of the inductive learning algorithm TAL.

In addition to its semantic and syntactic compatibility to the model of the normative framework and the learning framework, the non-monotonic nature of ASP's allows for the designer to avoid having to provide a fully specified description of the framework to the learner. Thus only the essential behaviour is needed to be captured in the use case, making it a more intuitive method for describing the system's behaviour.

The normative framework revision is carried out through an interactive process, where the designer supplies the use case, constructed from the specifications, and the partial framework to the learner. The learner then find ways to revise the framework so that it exhibit the behaviour from the use case. These revision suggestions are returned to the designer, who then decides which suggestion should be applied to the partial model. The revised model can then be used with another use case to gradually build up the complete model.

Seeing as the learning process is heavily dependent upon the use cases, we studied test generation, originally hoping to use it for generating use cases. However, this idea was not suitable for the the method we have devised for generating tests, as it would imply that the designer knew beforehand how the normative framework should be revised. Instead, we have found it suitable to use in an additional step following the learning, to address the problem faced by the designer in selecting the most suitable revision suggestion.

Reasoning about the difference in the revision suggestions to find the right one can be time consuming, as the suggestions would create a different revised model. As each version of the revised model is guaranteed to exhibit the behaviour of the use case, the designer must find a new criteria for differentiating each suggestion. Our test generation method can be employed by the designer to tackle this problem, as it is able to find the *relevant literals* that could be used to discriminate the revision suggestions. Thus they would provide the designer with the new criteria that would help them decide on a revision suggestion for the partial model.

1.2 Objectives

The main aim of this project is to improve the procedure used to revise normative frameworks. We have found a way to achieve this by integrating test generation as a post-learning process to help the designer select the most suitable revision suggestion for their model. This can be broken down into steps as follows:

- Identifying the characteristics that define relevant literals. This characterisation is needed so that we have the means for differentiating between literals that could effect the hypothesis space from ones that are irrelevant.
- Find a method for generating literals that satisfy the characterisation of relevant literals to produce a set of literals that we could that we know are potential criteria in discriminating the revision suggestions.
- Define a criteria for ranking the relevant literals in order to find one most critical for discriminating the hypotheses. This to identify the critical criteria so that the designer is be able to quickly dismiss the unwanted revision suggestions.
- Integrate the our study on relevant literals to apply it to use cases and normative framework revision.

1.3 Contributions

The main contribution of this project is a reasoning method for differentiating different model revisions. The method that we have described allows for direct comparison between each possible revision, without having to implement the different revised models. Furthermore, it is compatible for implementation in ASP, making it highly integrable with the existing procedure for revising normative frameworks.

The following list outlines the contributions made in the development of this project:

- We have identified the characteristics of relevant literals and apply it to use case for normative framework revision.
- We have describe how to formulate ASP program to abductively generate answer sets containing relevant literals.
- We have defined an algorithm for extracting relevant literals from answer sets, and described a scoring method for ranking the extracted literals according to the number of revision suggestions they reject, thus identifying the most critical literal for discarding hypotheses.
- We have shown through our case study how to integrate test generation into normative framework revision, as means for finding the criteria to differentiate the revision suggestions from each other.

In addition the original objectives, we also made further contributions on the ASP implementation of TAL. Although implementing TAL in ASP has great benefits in terms of compatibility with the normative framework model and our test generation procedure, the ASP implementation of TAL used in [1] used extremely high computation time to generate answers, and in some cases was unable find solutions due to the memory overflow caused by the instance explosion when grounding a program. This is a common problem for any ASP implementations of ILP problems.

We have explored alternative implementations of TAL in ASP through different translations of TAL's top theory, and identified an implementation that can quickly solve the problem of learning revision suggestions.

1.4 Report Structure

The following is the brief outline of the report:

- In chapter 2 we provide the necessary technical background knowledge for our work. These include an explanation on the normative framework, and a formalisation that is compatible to answer set programming. We summarise the relevant topics on abductive test generation and test characterisation, followed by explanations on inductive logic programming and answer set programming, as well as some descriptions of tools that were used for the project and their features.
- In chapter 3 we discuss the current procedure for normative framework revision and how we will add to it. We explain each addition we have proposed, relevant literals generation and extraction from answer sets,

and scoring method for the relevant literal, and demonstrate how each of these work using simple test programs.

- In chapter 4 we provide the alternative formalisations for inductive learning's mode declarations in ASP. For each alternative we explain the reason why we took such approach, give provide example of the implementation, and discuss their performance.
- In chapter 5 we show the result of our work through the use of a case study, stepping through each stage of the normative framework revision, and evaluate the performance of our approach when applied to normative framework.
- In chapter 6 we summarise our work in terms of what have been achieved and its limitations. We then discuss the further work and possible extensions to our study.

Chapter 2

Background

2.1 Preliminary Notations and Terminologies

The following is the list of notations and terminology that we will be using throughout this chapter:

- **Literal:** A statement that do not have any logical connectives, apart from negation, in it. For example, l and $\neg l$ are literals of the propositional symbol l.
- **Horne Clause:** A clause containing disjunction of literals with at most one positive literal.
- **Logic Programs:** We will be assuming the conventional logic program notation. A program consists of Horn clauses expressed as rules of the form $H \leftarrow L_1, \ldots, L_n$. Where the head of the rule H is an atom and the body $L_1, \ldots, L_n$ represents a conjunction of literals L_1 to L_n, with n ≥ 0.

  It is also assumed that each rule is universally quantified, thus the rule $H(X) \leftarrow L_1(X_1), \ldots, L_n(X_n)$ is assumed to be equivalent to $\forall X, X_1, X_n(H(X) \leftarrow L_1(X_1), \ldots, L_n(X_n))$
- **Integrity Constraints:** A set closed formulae that must be satisfied by a a logic program. Programs P with a set integrity constraints IC is satisfiable if all integrity constraints are derivable by the program, $\forall c_i \in IC : P \vDash c$
- **Negations:** We will be using *not* for indicating negation as failure, while $\neg$ represents logical negation.

2.2 Normative Framework

A multi-agent system is made up of rational agents interacting with each other. The term *rational agent* is based on the description given in [7], as an agent that can observe and learn from its environment. The agent can then use what it has learnt from its observation, together with its background knowledge, to reason about the best course of actions to take in order to achieve its goal.

The normative framework is used to describe the social rules of the organisation. Agent's actions are seen as events that occurred within the organisation. Although agents are autonomous and thus can initiate any event at any time, the normative framework acts as a constraint on "unsociable" events by administering sanctions upon the agent that evoked them. In this way, the normative framework can help to direct the behaviour of agents within the organisation.

Another concept captured by the normative framework is *conventional generation* [2], where the organisation creates *institutional events*, events with certain implication in the context of the organisation, as a result of *observable events*, those that represent the agent's actions. For example, shooting somebody and murder.

The effects of events on the organisation is represented by *fluents*, these are properties of the organisation, brought about or terminated at some instants by events. In the case of shooting someone of a neighbouring country, in time of peace could result in dispute and unrest between the two countries.

As fluents represent certain properties of the organisation, their presence can be used to identify unique states of the organisation.

2.2.1 Social Constraints

The social constraints *permission*, *physical capabilities* and *institutionalised power* are concepts that the normative framework aims to express. The constraints are explained further in the following bullet points, as adapted from [6]:

- Events may have *institutional effect*, depending on the state of the system and the agent performing it. Thus, if the agent has the institutional power, then they are *empowered* to change the state of affairs in the organisation by performing certain events. This may also depend on the state of the organisation. For example, a prime minister and cabinet taking a country to war.

- *Permitted* or *prohibited* events, and *obligation* are specific to the domain of the system and will allow the characterisation of an agent's behaviour as "acceptable" or "unacceptable". Hence, an agent performing a prohibited event or not complying with its obligation will result in its behaviour being considered "unacceptable". Consequently, this allows for the system to also be characterised as "acceptable" or "unacceptable", depending on the majority of its agents' behaviour. It is also important that the specification for permission and obligation is not inconsistent, as an agent should be able to perform the necessary events for completing its obligation.

- *Sanctions* and *enforcement* deals with "unacceptable" behaviour of agents, defining when is an agent sanctioned and what penalty must it endure.

Penalties could be handed out when agents perform forbidden actions or if they do not carry out their obligations. Different state of the system could create different kinds of sanction for the same event. For example, a soldier shooting an opposing soldier at war will have different consequence to shooting them when at peace.

2.2.2 Modelling the Normative Framework

We will be following the model described in [2] for representing normative frameworks. This formalisation is designed to be easily translated into answer set program, which is useful for bottom-up specification verification and compatible to the learning tool ASPAL.

The framework is modelled as a quintuple $I := < E, F, C, G, S_0 >$ of:

- Institutional events $E = E_{obs} \cup E_{inst}$
  The types of events that may occur in the framework are observable events E_{obs} and institutional events $E_{inst} = E_{instact} \cup E_{viol}$, the latter containing institutional actions $E_{instact}$ and violations E_{viol}.
- Fluents $F = D \cup W \cup M \cup O$
  Properties of the institution that may hold at certain time of execution, the conjunction of these is used for describing the state of the institution.
  - Domain fluents D
    Describes the domain in which the organisation operates
  - Institutional powers W
    Each $pow(e)$, where $e \in E_{instact}$, denotes the capabilities of some event to be generated
  - Event permissions M
    Each permission $perm(e)$, where $e \in E_{instact} \cup E_{obs}$, denotes the permission for the event to be brought about.
  - Obligations O
    Each obligation $obl(e, d, v)$, where $e \in E, d \in E, v \in E_{inst}$, denotes that the event e should be brought about before the occurrence of the event d, else be subjected to the violation v.
- Consequence relation $C : X \times E \rightarrow 2^F \times 2^F$
  Consisting of descriptions of how fluents are initiated C_{int} or terminated C_{term}, depending on certain state ϕ of the organisation and the the event e occurring in it.
- Event generation relation $G : X \times E \rightarrow 2_{E_{inst}}$
  Describes how the occurrence of one event under certain conditions (state) can generate other events inside the organisation.
- Initial state S_0
  The set of fluents that holds when the organisation is initialised.

2.3 Test

This following section on test is based on the paper [11]. A test has the form of a pair (A, o), where A is a conjunction of achievable literals, the initial condition specified by the tester, o is an observable, the outcome (o or $\neg o$) to be decided by the tester.

The outcome a of the test confirms a hypothesis H with respect to a background knowledge Σ if and only if $\Sigma \wedge A \wedge H$ is satisfiable and $\Sigma \wedge A \vDash H \supset a$. The outcome a refutes H with respect to Σ if and only if $\Sigma \wedge A \wedge H$ is satisfiable and $\Sigma \wedge A \vDash H \supset \neg a$.

2.3.1 Test Generation using Abduction

This section summarises the idea of test generation as it will be used later for finding relevant literals that can eliminate revision suggestions. Test generation can be used for reasoning about models against a set of hypotheses. These hypotheses could represent the specification of the model, and finding tests for them could identify inconsistencies within the model. In this project we will use the set of hypotheses to represent the different suggestions revisions that were learn and using test generation to identify literals that represents the inconsistencies between the revision suggestions, as their truth value could redefine the hypothesis space if it reject some of them.

Tests are generated to meet certain reasoning objective regarding a set of hypotheses HYP with respect to a background theory. These objectives could be to confirm or dismiss certain hypotheses, or for discriminating a hypothesis space.

This problem can be solved using abduction (Section 2.4.1) to find the test that once added to Σ will remove certain hypothesis H from HYP, in other words test (A, o) such that $\Sigma \cup A \cup o \vDash \neg H$.

We concentrate on finding tests that rejects hypotheses as this would redefine the hypothesis set, while confirming tests will not lead to any changes of the hypothesis space.

2.3.2 Test Characterisation

As described in Section 2.3.1 test generated are aimed to have certain objective. For this project the tests that we want are ones that will discriminate the hypothesis space. Here we will describe characteristics of tests that can be used as objection for this type of test generation.

In the following sections on test characterisations Σ represents the the background knowledge, HYP is a set of hypotheses, and (A, o) is a test.

Discriminating Tests

Discriminating tests have the following characteristics:

1. $\Sigma \wedge A \wedge H$ is satisfiable for all $H \in HYP$

2. $A \wedge o$ is an abductive explanation for $\bigvee_{H_i \in HYP} \neg H_i$
3. $A \wedge \neg o$ is an abductive explanation for $\bigvee_{H_i \in HYP} \neg H_i$
4. $\Sigma \not\models \bigvee_{H_i \in HYP} \neg H_i$ (For mutually exclusive tests, there is no need to discriminate them)

Thus, by the second and third condition, regardless of whether o or $\neg o$ is observed, the hypothesis space will be discriminated.

Relevant Tests

The conditions for discriminating test can often be too strong, thus it is often the case that such tests cannot be found. The characterisations below describes relevant tests. This type of tests ensure that hypothesis space can be narrowed down if a is observed, but does not guarantee the same for $\neg a$.

1. $\Sigma \wedge A \wedge H$ is satisfiable for all $H \in HYP$.
2. $A \wedge o$ is an abductive explanation for $\bigvee_{H_i \in HYP} \neg H_i$ (Some hypotheses from HYP are be rejected by the test)
3. $\Sigma \not\models \bigvee_{H_i \in HYP} \neg H_i$ (The hypotheses are rejected by the test and not the background theory)
4. $A \wedge o$ is not an abductive explanation for $\neg H_i, \forall H_i \in HYP$ (There remains some hypotheses approved by the test)

Example

We will be using a general example to illustrate the difference between discriminating and relevant test. Taking a hypothesis set $HYP = \{p, q, r\}$, and the following background knowledge:

$$\begin{array}{l} p \leftarrow s \\ q \leftarrow \neg s \\ r \leftarrow \neg t \end{array}$$

The test $(\{\}, s)$ is a discriminating and relevant as if it is observed to be true then q is rejected, while if it is false p is rejected. $(\{\}, t)$ on the other hand can only reject r if it is observed to be true and does not reject any hypotheses if it is observed to be false.

2.4 Abduction

The definitions that we are using for abduction is adapted from [8]. Abduction was first introduced by Charles S. Pierce, an American philosopher, as part of his trichotomy of inference, consisting of deduction, induction and abduction.

In his definition abduction is understood as inferring plausible explanation E, for some observation O and that the presence of E implies O.

In artificial intelligence, abduction is better associated with the definition given by Gilbert Harman, another American philosopher, as:

> *"The inference to the best explanation"*

This takes into account the fact that there could be many possible hypotheses for an observation. Therefore it is important to be able to select the hypothesis that is the best explanation for the observation, before making the inference.

In logic programming, abductive reasoning can be formalised in a logic programming abductive framework [8].

2.4.1 Abductive Logic Programming (ALP) Framework

This is a triplet $< P, A, IC >$ where:

- P is a logic program.
- A is the set of abducible predicate symbols. These represent the form of the plausible hypotheses.
- IC is the integrity constraint, a set of closed formulae. It is the condition that must be made true by the hypothesis.

The computation for the abductive explanation of a logic program with such a framework, using negation as failure and not classical negation can then be explained using stable models.

2.4.2 Stable Model

For a logic program P and M, which is a set of all possible grounded clauses from the set of all possible atoms. Let P_M be the set of ground Horn clause from grounding P and deleting:

- Any clause with the literals of the form *not* l in its body, where $l \in M$
- All literals of the form *not* l from the body of clauses, where $l \notin M$

M is then the stable model for P if M is the minimal model (one without any negation) of P_M.

Example

Take for example, the following program P:

$p(X, Y) \leftarrow q(X, Y), not\ r(X)$
$q(2, 1)$
$q(2, 2)$

$r(1)$

The grounded instances of P are:

$p(1,1) \leftarrow q(1,1), not\ r(1)$
$p(1,2) \leftarrow q(1,2), not\ r(1)$
$p(2,1) \leftarrow q(2,1), not\ r(2)$
$p(2,2) \leftarrow q(2,2), not\ r(2)$
$q(2,1)$
$q(2,2)$
$r(1)$

Taking $M = \{q(2,1), q(2,2), r(1)\}$, we can construct P_M as:

$p(1,2) \leftarrow q(1,2)$
$p(2,2) \leftarrow q(2,2)$
$q(2,1)$
$q(2,2)$
$r(1)$

Thus M is the stable model of P as it is the minimum model of P_M.

2.4.3 Generalised Stable Model

For a logic programming abductive framework $< P, A, IC >$ and a set of abducibles $\Delta \subseteq atoms(A)$. The set $M(\Delta)$ of ground atoms is a generalised stable model for $< P, A, IC >$ if and only if it is a stable model for the logic program $P \cup \Delta$ and the integrity constraint IC, where $\Delta = A \cap M(\Delta)$.

2.4.4 Abductive Explanation

Given a logic programming abductive framework $< P, A, IC >$, Δ is an abductive explanation for an unit clause observation q, if there is a generalised stable model $M(\Delta)$ in which q is true.

2.5 Inductive Logic Programming (ILP)

Inductive Logic Programming is a merge between inductive learning and logic programming. Logic programming is used for its expressive power to represent the background knowledge, positive and negative examples, and the hypothesis, while inductive learning is used to find the hypothesis, given the background knowledge and examples.

The hypothesis H derived by the program is required to be complete with respect of the background knowledge B and the positive examples E^+, and consistent with respect to the background knowledge and negative examples E^-.

2.5.1 Inductive Learning with Background Knowledge

This explanation of inductive learning is adapted from the description given in chapter 1 of [3]. Given a set of background knowledge B, a set E^+ of positive and a set E^- of negative training examples, find a hypothesis H expressed in some description language L, such that H is complete and consistent with respect to the background knowledge B and the training examples $E^+ \cup E^-$:

- Every positive example $e \in E^+$ is covered by $H \cup B$
- No negative example $e \in E^-$ is covered by $H \cup B$

The hypotheses are rules of the form $r \leftarrow l_1, \ldots, l_n$ where r is the head of the rule is an atom, and $l_1, \ldots, l_n$ is the body, a conjunction of literals l_1 to l_n, with n ≥ 0.

Example

Consider this example from [4], with the following background knowledge B and training example sets E^+ and E^-:

$$\begin{aligned} B = \{&net(s(X)) \leftarrow nat(X),\\ &even(0),\\ &nat(0)\} \end{aligned}$$

$$E^+ = \{odd(s(s(s(s(s(0))))))\}$$

$$E^- = \{odd(s(s(0))), odd(s(s(s(s(0)))))\}$$

Adding the hypotheses $odd(X) \leftarrow X = s(Y), even(Y)$ and $even(X) \leftarrow X = s(Y), odd(Y)$ to the background knowledge would ensure that $odd(s(s(s(s(s(0))))))$ is covered by the new background knowledge, while $odd(s(s(0)))$ and $odd(s(s(s(s(0)))))$ are not.

2.5.2 Mode Declarations

The mode declarations, or language biases, specifies how the hypothesis of the learning problem could be constructed by declaring a schema for the hypothesis head and body. As well as defining the structure of the hypothesis, it also narrows the search space of the hypothesis.

The definition that we are using to describe mode declarations is from [4]. The mode declarations consists of head declarations *modeh*(s) and body declarations *modeb*(s), corresponding to the allowable formats for the head and body of the rule. s is a *schema*, a ground literal containing *placemarkers*. A placemarker can be of the form '+*type*', '-*type*' and '#*type*', corresponding to input, output and constants respectively, where *type* is a constant.

The rules can be generated by combining instances of the schemas, obtained by replacing each placemarkers $p_1, \ldots, p_n$ by a variable or a constant of the expressible type. More specifically each placemarker p_i, where $1 \leq i \leq n$, can be replaced by:

- A variable X_i of type t that occurred in the head of the rule if $p_i = +t$
- A variable X_i of type t that either occurred in the head of the rule or any proceeding literal of the body if $p_i = -t$
- A constant c of type t if the $p_i = \#t$

Example

For the rules $odd(X) \leftarrow X = s(Y), even(Y)$ and $even(X) \leftarrow X = s(Y), odd(Y)$ to be derivable, the mode declarations required are:

$m1 : modeh(odd(+nat))$
$m2 : modeh(even(+nat))$
$m3 : modeb(odd(+nat))$
$m4 : modeb(even(+nat))$
$m5 : modeb(+nat = s(-nat)))$

The rule $odd(X) \leftarrow X = s(Y), even(Y)$ is constructed by using the head declaration $m1$ and body declarations $m4$ and $m5$. The variable X is the input variable from the rule head, while Y is the output from $m5$, as specified by the mode declaration with the type of both X and Y being natural number.

2.5.3 Inductive Logic Programming Framework

A framework for an inductive logic program, can be seen as a triplet $< E, B, M >$ where:

- E is the set containing all positive examples E^+ and negative examples E^-
- B is the background knowledge of the program
- M is the set of mode declarations defining the hypothesis space

2.5.4 TAL and ASPAL

TAL (Top-directed Abductive Learning) is an inductive learning tool that uses abductive search to find hypotheses [4]. We use it indirectly through its ASP implementation ASPAL (ASP Abductive Learning) [1]. This is done through translating the ILP problem into an abductive logic programming problem, ASP can then be used to implement the abductive framework for solving the problem.

The ASP implementation retains the benefits from the previous system, such as the ability to reason about negation during the learning process and multiple predicate learning, as well as making the problem compatible to the ASP representation of the normative framework.

Unlike TAL, ASPAL will not produce rules as results, but would output instead literals which are an encoding of the rule. For example, the encoding of the rule $odd(X) \leftarrow X = s(Y), even(Y)$ in ASPAL would be:

```
rule(0, 0, (m1, (e), (e), (0))).
rule(0, 1, (m5, (e), (0), (1))).
rule(0, 2, (m4, (e), (1), (e))).
```

Where `e` represents an empty list. This literals above corresponds to the following format:

$$rule(RId,\ Level,\ (ModeName,\ Constants,\ InputVars,\ OutputVars)).$$

Where RId is the identification number for the rule and $Level$ defines the ordering of predicates in the rule, $ModeName$ correspond to a unique name of the mode declaration used to construct the predicate, $Constants$ is the list of constants values in the rule (for example if the predicate is $gender(X, female)$, then the constant list would be $(female)$). The two lists $InputVars$ and $OutputVars$ contains the index of the input and output variables of the predicate as ordered by their addition to the rule from left to right, thus in $odd(X) \leftarrow X = s(Y), even(Y)$, the variable X has the index 0 while Y has the index 1.

The transformation of an ILP problem into an ALP one is discussed later in Chapter 4 where we will be looking at alternative ASP implementations of TAL.

2.6 Answer Set Programming (ASP)

Answer Set Programming is a form of declarative problem solving paradigm that has emerged from the fields of reasoning, knowledge representation, and logic programming. It is aimed to be used for representation and reasoning tasks. Problems are solved by constructing a grounded instance of the program, then answer sets are constructed by solving the grounded program for stable models.

We will be using an ASP system, iClingo, for learning revision suggestions of a normative framework, as well as in test generation for generating answer sets containing relevant literals.

For the test generation, as we would like to identify all relevant literals to find the one that is most relevant, its bottom-up approach for solving problem is suitable for this task. Furthermore, it is a powerful constraint solver which makes it suitable for solving the abductive tasks in this project.

Its compatibility to the representation used for the normative framework and the learning task are also beneficial to the project as there would be less complications when transforming the program used for the learning task into the one used for test generation.

2.6.1 iClingo

iClingo is an incremental ASP system, part of the "Potsdam Answer Set Solving Collection", a set of tools for ASP programming developed at University of Potsdam. Both grounder and solver operates incrementally. At each state, the grounder will produce ground rules from the current state while avoiding the previously ground rules, the solver then accumulate the ground rules and computes the stable set from them.

2.6.2 Input Language of iClingo

The input language has various features and is explained in detail in the user's guide [10]. Here we will highlight the features that are relevant to the project.

Rules, facts, and integrity constraints are defined in the following format:

Rule	A :- L_1,...,L_n.
Fact	A.
Integrity Constraint	:- L_1,...,L_n.

The head of the rule, A, is an atom. Each L in the body is a literal of the form B or `not` B, and ',' represents a conjunction. Facts are rules with empty body. Anonymous variable can be defined using '_', and each individual occurrence of such variable is treated as a fresh variable (for example `rain(D) :- after(_,D)`, will be treated as `rain(D) :- after(X,D)`). Rules must be safe, with all variables in the rule's head expected to appear in at least one positive literal in the body. Intuitively, for the integrity constraint to hold, $L_1, ..., L_n$ must be false as it represents $\perp \leftarrow L_1, ..., L_n$, a rule where the head is false.

Note that **negation** of an atom expressed as `not` B is default negation (negation as failure). Thus `not` B will hold if B is not in the stable model. **Classical negation** may also be expressed by adding '-' in front of an atom then adding an integrity constraint so that B and -B cannot both be true.

The choice rule can be declared in the following form:

```
{a,b}.
```

This indicates that two atoms `a` and `b` can be arbitrarily chosen to be included in the answer set. Furthermore, cardinality constraint and rule body may be added:

```
0 {a,b} 1 :- c.
```

The code above indicate that if c is true then from the set of atoms a and b, either one of them must true, else neither of them are.

2.6.3 Other features of interest

Since the stable model contains all atoms that are true, it will be useful to be able to select the subset of the answer that we are interested in, making the result easier to understand. This can be done using `#hide` and `#show` statements as illustrated in the next example:

```
#hide.
#hide owner/2.
#show person/1.
```

In the code above `#hide` will suppress all atoms in the answer set, while `#hide owner/2` will suppress all *owner*/2 atoms. On the other hand `#show person/1` will include all *person*/1 atoms to the answer set. So using the following code for the base program:

```
person(pam).
person(pop).

pet(X) :- cat(X).
pet(X) :- fish(X).

cat(neo).
fish(chip).
fish(gon).

owner(neo,pam).
owner(chip,pam).
owner(gon,pop).
```

Should no `#hide` and `#show` statements are added to the program, the grounded instances of all clauses in the program will be included in the answer set as follows:

```
{ person(pam), person(pop),
  cat(neo),
  fish(chip), fish(gon),
  owner(neo,pam), owner(chip,pam), owner(gon,pop),
  pet(neo), pet(gon), pet(chip) }
```

Should the statements `#hide` and `#show person/1` be added to the program, then we will only have two literals in the answer set:

```
{ person(pam), person(pop) }
```

Conditions may also be added to `#hide` and `#show` statements, as shown in the following code:

```
#hide owner(X,Y): fish(X).
#show owner(X,Y): cat(X).
```

Here, all fish owners will be hidden, while the cat owners will be shown. Note that in this case if X satisfies both fish(X) and cat(X), then the atom will be included to the answer set. For example, if we make `cat(gon)` true, then adds the statements above as well as `#hide` to our base program, the answer set produced will be:

```
{ owner(neo,pam), owner(gon,pop) }
```

Note that one-line comments in ASP is preceded by the symbol '%', while multi-line comments are enclosed within '%*' and '*%'.

Chapter 3

Norms Revision

3.1 Overview

3.1.1 Current Procedure

As illustrated in Figure 3.1, the framework described in [1] revise norms by capturing the expected behaviour of the system through use cases. These use cases are used in an ILP learning task $< E, B, M >$ where:

- E is the use cases that must be covered by the revised framework
- B is the partial normative framework
- M is the mode declaration defining all possible forms a revision suggestion can take

This is then fed to the learner to compute all revision suggestions for the normative framework. The revisions could be in terms of introducing new rules, and removing or changing the conditions of existing rules. These revision suggestions are returned to the designer, who will choose one of them to apply to the normative framework. The process can then be repeated using the revised norms and new use cases until the designer is satisfied with the normative framework.

The approach however, does not provide the designer with any information on the differences between each revision suggestions. While each revision will ensure that the revised normative framework acquires the behaviour captured by the use cases, each one can also implies additional behaviours, intended or not by the designer.

3.1.2 Modified Procedure

To address the problem of selecting one revision suggestion over other results from the learner, we returned to the idea of capturing the behaviour of the

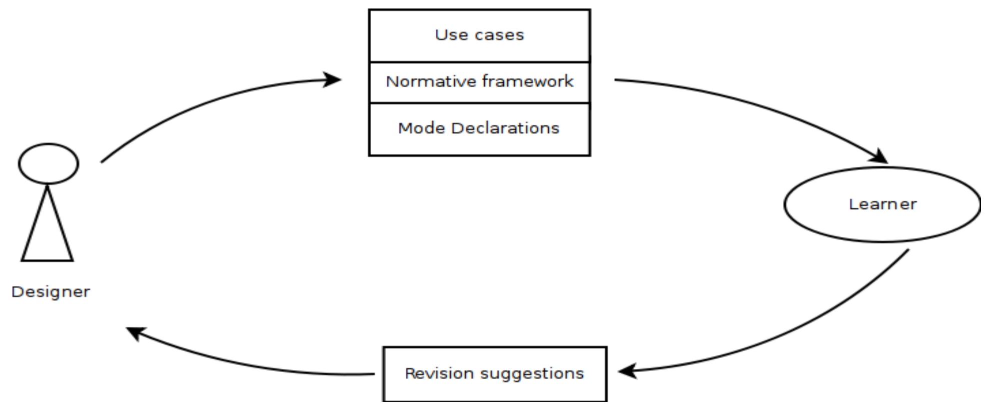

Figure 3.1: Current procedure for norms revision using use cases

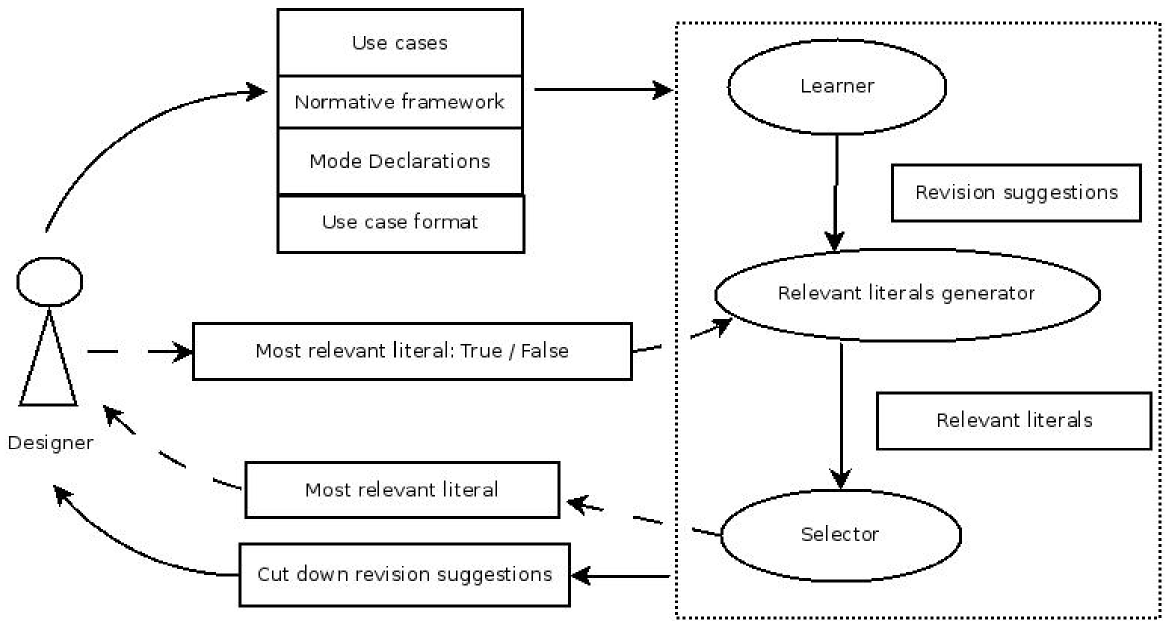

Figure 3.2: Proposed procedure for norms revision allowing literals to be added to the use cases after the learning has been applied

system through use cases. As the use cases supplied by the designer is often incomplete, we can find within the undefined literals, relevant literals that can be used to reject some revision suggestions. By enquiring the designer about the truth value of these literals, they can be used as a selection criteria query for the different suggestions learnt by the learner.

This idea has resulted in the proposed framework shown in Figure 3.2, where the designer can expand their use cases after the learning has been applied. This is done iteratively as shown by the dotted line in Figure 3.2. Results from the learner are used to generate relevant literals. These are literals the can be added to the use cases while at the same time their truth value may be used for dismissing some but not all revision suggestions, in effect acting as *tests* for potentially rejecting the revision suggestions. This is so we do not ask the designer about literals whose truth value have no effect on the revision suggestions.

In addition to identifying the relevant literals, we need to rank them in order of number of revision suggestions it can reject. This is to ensure that as much information is gained from asking about the literal as possible, resulting in the most discriminated revision suggestions. This will also keep the number of enquiry needed to be answer by the designer at a minimum, to ensure that the procedure is carried out efficiently. The answer from the designer can then be used to determine if any of the revision suggestions will be discarded.

The steps needed to be taken to identify the most relevant is illustrated in Figure 3.3. Firstly, the revision suggestions from the learner must be transformed into hypotheses so that we can use them for specifying objective for the relevant literals generation. This program is then fed into an ASP solver to generate all answer sets containing relevant literals. Some post processing is then required to extract the relevant literals and associate them to the hypotheses they reject. Lastly, by knowing all the hypotheses rejected by each relevant literal, they can be ranked against each other to find the most relevant literal.

3.2 Relevant Literal

Use cases (T, O) [1] for a normative framework revision consist of a trace T of observable events caused by the agents, and set O of violations (E_{viol} in Section 2.2.2). Following the definition of relevant tests in Section 2.3.2, we will characterise the relevant literal l, such that for a partial normative framework Σ and set HYP of hypotheses describing revision suggestions:

1. $\Sigma \wedge T \wedge O \wedge H$ is satisfiable for all $H \in HYP$ (All revision suggestions exhibit the behaviour conveys be the use case)

2. $T \wedge O \wedge l$ is an abductive explanation for $\bigvee_{H_i \in HYP} \neg H_i$ (l can falsify at least one revision suggestion)

3. $\Sigma \wedge T \wedge O \nvDash \bigvee_{H_i \in HYP} \neg H_i$ (All revision suggestions are consistent with the the normative framework and use case)

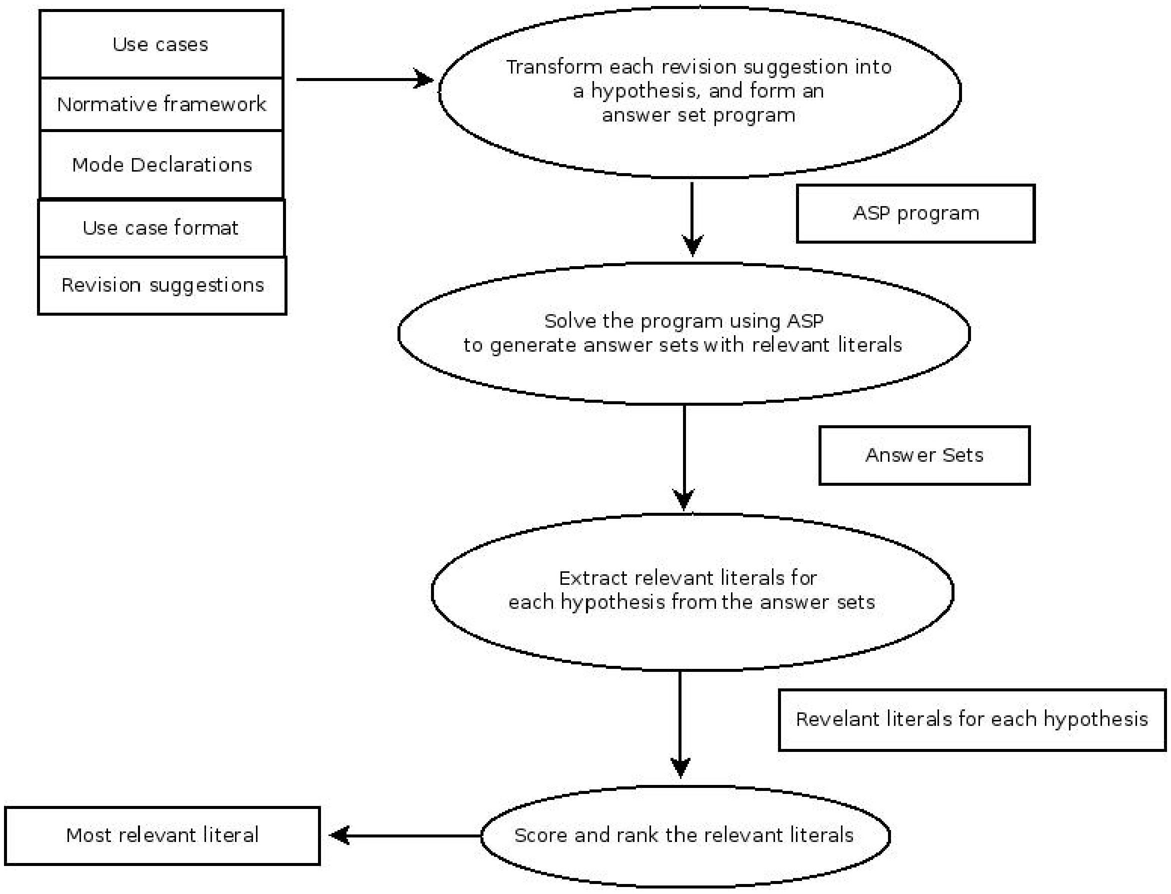
Use cases
Normative framework
Mode Declarations
Use case format
Revision suggestions
Transform each revision suggestion into a hypothesis, and form an answer set program
ASP program
Solve the program using ASP to generate answer sets with relevant literals
Answer Sets
Extract relevant literals for each hypothesis from the answer sets
Revelant literals for each hypothesis
Score and rank the relevant literals
Most relevant literal

Figure 3.3: Steps for finding the most relevant literal

4. $A \wedge O \wedge l$ is not an abductive explanation for $\neg H_i$, $\forall H_i \in HYP$ (Not all revision suggestions are rejected by l)

Note that the first and third condition is already satisfied by the correctness of the learner, thus the conditions that we need to add as objectives for abducing relevant literals are the second and fourth conditions.

3.3 Relevant Literals Generation

As seen in Section 2.3.1 we can form an abductive problem for generating relevant literal, using our characterisation of the literals as the objective of the problem. However, we now have the question of how to translate the revision suggestions into a set of hypotheses for the computation of relevant literals. Using a simple, non normative framework as an example to illustrate our proposed approach, shown in Listing 3.1:

```
%----------------BACKGROUND------------------%
bean(b1).
bean(b2).
bean(b3).

bag(whitebag).
bag(blackbag).

colour(white).
colour(black).

beancolour(b1, white).
beancolour(b2, black).
beancolour(b3, white).

special(b1).
special(b2).

poisonous(b2).
poisonous(b3).
```

Listing 3.1: Example background

The program in Listing 3.1 contains background facts about three white or black beans. In addition to colour, each bean can have addition properties of some being special or poisonous. The background also contain information about two bags and their colours.

There are three way to modify the program, to ensure that `in(blackbag,b2)` is true. This is can be done by adding one of the following rules to the program:

1. $in(blackbag, B) \leftarrow beancolour(B, black)$
2. $in(blackbag, B) \leftarrow special(B)$
3. $in(blackbag, B) \leftarrow poisonous(B)$

The question now is to find the literals that can discriminate between these suggested changes, to find the criteria for discarding each one of them. We should form for each hypothesis to represent cases where the revision would be false. Thus, for each rule in H_i of the form $rule_j(v_1, \dots, v_m) \leftarrow l_1, \dots, l_n$ where $n \geq 1$, we can add to the given partial model the following two rules:

- $\neg hyp(i) \leftarrow not\ rule_j(v_1, \dots, v_m), cond_{i,j}(v_1, \dots, v_m)$
- $cond_{i,j}(v_1, \dots, v_m) \leftarrow l_1, \dots, l_n$

Where i distinguish one revision suggestion from another, j for indexing rules within the revision suggestion, and $v_1, \dots, v_m$ are for linking variables in the rule head to its conditions, with $m \geq 0$.

For the example and the three rules addition, this would result in the program segment in Listing 3.2 given:

```
%-----------------HYPOTHESES------------------%
hyp_id(1).
hyp_id(2).
hyp_id(3).

hyp(H) :- hyp_id(H), not -hyp(H).

% H1: Black beans are in the blackbag
% in(blackbag, B):- beancolour(B, black).
-hyp(1) :- not in(blackbag, B), cond_1(B).
cond_1(B) :- beancolour(B, black).

% H2: Special beans are in the blackbag
% in(blackbag, B) :- poisonous(B).
-hyp(2) :- not in(blackbag, B), cond_2(B).
cond_2(B) :- special(B).

% H2: Poisonous beans are in the blackbag
% in(blackbag, B) :- special(B).
-hyp(3) :- not in(blackbag, B), cond_3(B).
cond_3(B) :- poisonous(B).

%-----------------ABDUCIBLES------------------%
% A bean can be in one bag only
0 { in(BAG, BEAN) : bag(BAG) } 1 :- bean(BEAN).
```

```
26
27 -in(BG,B) :- bag(BG), bean(B), not in(BG,B).
```

Listing 3.2: Program segment of possible rules to add to background

Note that lines 2-4 in Listing 3.2 are auxiliary information that are for reasoning about these hypotheses, they represent the unique identifying number for each hypothesis. Closed world assumption is also added for the hypotheses by the rule in line 6. The last three lines is needed to make the literals `in/2` abducible so we can generate all the relevant literals. We also added closed word assumption for it, as this is will make it easier for us to extract relevant literals that are negative instances of `in/2`.

We can add the constrains on the hypotheses to ensure that the answer acquired will contain relevant literals, as shown in our relevant literal characterisation in Section 3.2.

As mentioned earlier, we will ignore the fist and third condition as they are satisfied through the correctness of the learner. So the main conditions for relevant literals are:

1. $T \wedge O \wedge l$ is an abductive explanation for $\bigvee_{H_i \in HYP} \neg H_i$
2. $A \wedge O \wedge l$ is not an abductive explanation for $\neg H_i$, $\forall H_i \in HYP$

These can be added as integrity constraints in ASP as shown in the following Listing 3.3:

```
 1 %------------------CONSTRAINT-------------------%
 2
 3 :- hyp(1), hyp(2), hyp(3).
 4 :- -hyp(1), -hyp(2), -hyp(3).
 5
 6 %----------------------------------------------%
 7
 8 #hide.
 9 #show in/2.
10 #show -in/2.
11 #show hyp/1.
12 #show -hyp/1.
```

Listing 3.3: Hypothesis objective

Note that lines 8-12 in Listing 3.3 are additional clauses for readability of the results, hiding all predicates whose definition remains unchanged. Running the completed program in iClingo will produce answer sets with relevant literals. The first three answer sets from this program is shown in Listing 3.4. Although each answer set contains relevant literals for rejecting the falsified hypotheses, not all literals in the set are relevant, for example literals `in/2` with `whitebag` as its predicate are irrelevant as the rule changes have no effect on them. This shows that we will still need to carry out some post processing of these results to identify the most relevant literals. Furthermore, we have only considered the addition of rules in this example while there are more ways in which a model could be revised.

```
Answer: 1
in(blackbag,b3) in(blackbag,b2) -in(blackbag,b1)
-in(whitebag,b3) -in(whitebag,b2) -in(whitebag,b1)
-hyp(2) hyp(3) hyp(1)

Answer: 2
in(blackbag,b3) in(blackbag,b2) in(whitebag,b1)
-in(blackbag,b1) -in(whitebag,b3) -in(whitebag,b2)
-hyp(2) hyp(3) hyp(1)

Answer: 3
in(blackbag,b2) in(blackbag,b1) -in(blackbag,b3)
-in(whitebag,b3) -in(whitebag,b2) -in(whitebag,b1)
-hyp(3) hyp(2) hyp(1)

...
```

Listing 3.4: Example result of relevant literals generation

3.4 Relevent Literals Generation and Norms Revision

There are three ways of refining the normative framework:

1. Adding a new rules.
2. Deleting an existing rules.
3. Adding or deleting conditions of existing rules.

Our method of describing revision suggestion as an hypothesis in Section 3.3 can only handle the first case of adding new rules. For deleting or changing conditions of existing rules, we need to take into consideration the rules that will not be in those revision suggestion but can be in all other ones that do not delete or modify them.

To handle the last two cases, we can treat the deleted or modified rules as new rules for all other revision suggestions that do not modify them, representing how they remain unchanged in those revision suggestions. By removing the rule from the original background knowledge, the revision suggestions that modify the rule can treat its revised version as new rules to add to the model. For example, suppose the rule R_1 was in the the background knowledge in Listing 3.1:

$$\mathrm{R}_1\text{: } in(whitebag, B) \leftarrow special(B), beancolour(B, white)$$

Then if we want to choose between these two possible revisions such that `in(whitebag,b2)` is true:

- Changing R_1 to: $in(whitebag, B) \leftarrow special(B)$
- Adding a new rule: $in(whitebag, B) \leftarrow poisonous(B)$

From the background knowledge Σ we remove from it all deleted or modify rules R (in this case only R_1). $\Sigma' = \Sigma - R_1$ represents the core background that is common to all revision suggestions for the program. Any revision suggestions that modified or kept R_1 unchanged can add the respective versions of it as new rule addition by the suggestion.

The hypotheses representation for these example revision suggestions are shown in figure 3.5, with R_1 deleted from the background (thus the background remains the same as in Listing 3.1), and added to the second revision suggestion as a new rule. This makes it possible to keep the translation to hypotheses the same, while at the same time making it possible to generate use cases for all types of revisions.

Note that should there be another revision suggestion which simply deletes R_1 from the hypothesis without adding or modifying any rules, then Σ' already represents the program as revised by the revision.

```
%-----------------HYPOTHESES------------------%

% H1: Modifying rule
-hyp(1) :- not in(whitebag, B), cond_1(B).
cond_1(B):- special(B).


% H2: Add new rule
-hyp(2) :- not in(whitebag, B), cond_2_1(B).
cond_2_1(B) :- poisonous(B).

% Deleted rule added to hyp(2)
-hyp(2) :- not in(whitebag, B), cond_2_2(B).
cond_2_2(B) :- special(B), beancolour(B, white).
```

Listing 3.5: Example hypotheses for modifying existing rule

The Listing 3.6, shows the answer sets produced by the hypotheses in Listing 3.5, using the background knowledge in Listing 3.1. There were only four answer sets produced and thus it can be worked out that `hyp(1)` is falsified whenever `in(whitebag,b3)` and `-in(whitebag,b1)` holds, while `hyp(2)` is falsified when `-in(whitebag,b3)` and `in(whitebag,b1)` holds.

```
Answer: 1
in(whitebag,b3) in(whitebag,b2) -in(blackbag,b3)
-in(blackbag,b2) -in(blackbag,b1) -in(whitebag,b1)
-hyp(1) hyp(2)
Answer: 2
in(whitebag,b3) in(whitebag,b2) in(blackbag,b1)
-in(blackbag,b3) -in(blackbag,b2) -in(whitebag,b1)
-hyp(1) hyp(2)
Answer: 3
in(whitebag,b2) in(whitebag,b1) -in(blackbag,b3)
-in(blackbag,b2) -in(blackbag,b1) -in(whitebag,b3)
-hyp(2) hyp(1)
Answer: 4
in(blackbag,b3) in(whitebag,b2) in(whitebag,b1)
-in(blackbag,b2) -in(blackbag,b1) -in(whitebag,b3)
-hyp(2) hyp(1)
```

Listing 3.6: Example result for relevant literals generation with rule modifying hypotheses

3.5 Finding Relevant Literals

To be able to score and rank the relevant literals, they need to be extracted from the answer sets of the relevant literals generation. Furthermore, for them to have any meaning, we also need to know what revision suggestions they eliminate.

As iClingo allows us to specify what predicates symbols are included in the answer sets it produce. As shown in Section 3.3, we can choose for only the hypotheses predicates HYP, and the abducible predicates, and their negations, to be included in the answer set results ANS. Finding relevant literals for hypothesis $h \in HYP$ can be done by subtracting answer sets where h is not rejected from the ones that do reject it, as described in Algorithm 1.

In Algorithm 1, the set subtraction must be done one by one as we do not want to miss the case where two or more literals are needed to reject a hypothesis. However, the resulting sets will still have irrelevant literals in them, thus we need to find distinct minimal subsets of the differences before combining it to the result of the algorithm. The returned result is a set containing sets of relevant literals for the given hypothesis. We took this approach, rather than returning a set of literals, so that we can distinguish between literals that is powerful enough to reject the hypothesis by itself, and those that need additional literals to do so.

For example if we take the answer sets in the Listing 3.6 as an example for ANS, and `hyp(1)` for h, and $HYP = \{$ `hyp(1)`, `hyp(2)` $\}$ to find the relevant literals for h using the Algorithm 1. We first go through each answer set to find one that includes `-hyp(1)` (line 4-5).

Taking the first answer set S_1 where the hypothesis is falsified, we then find the answer sets that do not falsify the hypothesis (line 10-11), for each of such answer set (S_3 and S_4), union it with the set of predicates that were used for identifying hypotheses and their negation (line 13), then add the set difference between S_1 and the union sets to $DIFF$.

The two sets in $DIFF$ after line 16 would be:

$$S_1 - S_3 \cup HYP \cup \{\neg h : HYP\} = \{in(whitebag, b3), \neg in(whitebag, b1)\}$$
$$S_1 - S_4 \cup HYP \cup \{\neg h : HYP\} = \{in(whitebag, b3), \neg in(whitebag, b1)\}$$

Thus the smallest subsets in $DIFF$ found after line 26 is only $\{in(whitebag, b3), \neg in(whitebag, b1)\}$, and applying the algorithm to S_2 would produce the same result thus this set of two literals are the only relevant relevant literals for `hyp(1)`, and as they are returned in the same set, it is their conjunction which is required to reject `hyp(1)`.

Although the constraints added to the relevant literals generation program prevent answers where none of the hypotheses is rejected to be produced (to ensure that all answer sets in the result contain relevant literals), when using this algorithm to identify relevant literals, in some case it is beneficial to relax the constraint so that there are more answer sets for comparison. For example, in the case where only one hypothesis can be rejected. If we use the same

Algorithm 1 Finding relevant literals of a hypothesis

Input: answer sets ANS, hypothesis predicate h, and the set of hypothesis predicates HYP

```
1:
2: $REV = \emptyset$
3:
4: for all $S_i \in ANS$ do
5:   if $\neg h \in S$ then
6:
7:     {Find the difference between $S_i$ and other answer sets that do not
       contain relevant literals of $h$}
8:     $DIFF = \emptyset$
9:
10:    for all $S_j \in ANS$ do
11:      if $\neg h \notin S$ then
12:
13:        $NREV = S_j \cup HYP \cup \{\neg h : HYP\}$
14:        $DIFF = DIFF \cup \{S_i - NREV\}$
15:      end if
16:    end for
17:
18:    {Find the smallest subsets of the sets in $DIFF$}
19:    $LIT = \emptyset$
20:
21:    for all $D \in DIFF$ do
22:      $LIT = LIT - \{L : LIT | L \supset D\}$
23:      if $D \notin LIT$ and $\nexists L : LIT(L \subset D)$ then
24:        $LIT = LIT \cup \{D\}$
25:      end if
26:    end for
27:
28:    {Add this set of relevant literals to the result}
29:    $R = \bigcup_{L_i \in LIT} L_i$
30:    $REV = REV \cup \{R\}$
31:  end if
32: end for
33:
34: return $REV$
```

background knowledge as in Listing 3.1, and try to find the differences between two revision suggestions, each one adding one of the following rules:

- $in(whitebag, B) \leftarrow beancolour(B, white).$
- $in(whitebag, B) \leftarrow beancolour(B, white), special(B).$

The answer sets produced from the relevant literals generation to differentiate the addition of these rules are shown in Listing 3.7. While `hyp(1)` can be falsified, its relevant literals cannot be found as there is no answer sets where `hyp(1)` is true that can be used for the set comparison when applying the Algorithm 1. However, the algorithm would be able to identify the relevant literals should the condition placed on the relevant literals generation is relaxed to include answer sets where both `hyp(1)` and `hyp(2)` are true.

```
Answer: 1
in(blackbag,b2) in(whitebag,b1) -in(blackbag,b3)
-in(blackbag,b1) -in(whitebag,b3) -in(whitebag,b2)
-hyp(1) hyp(2)
Answer: 2
in(whitebag,b1) -in(blackbag,b3) -in(blackbag,b2)
-in(blackbag,b1) -in(whitebag,b3) -in(whitebag,b2)
-hyp(1) hyp(2)
Answer: 3
in(whitebag,b2) in(whitebag,b1) -in(blackbag,b3)
-in(blackbag,b2) -in(blackbag,b1) -in(whitebag,b3)
-hyp(1) hyp(2)
Answer: 4
in(blackbag,b3) in(blackbag,b2) in(whitebag,b1)
-in(blackbag,b1) -in(whitebag,b3) -in(whitebag,b2)
-hyp(1) hyp(2)
Answer: 5
in(blackbag,b3) in(whitebag,b1) -in(blackbag,b2)
-in(blackbag,b1) -in(whitebag,b3) -in(whitebag,b2)
-hyp(1) hyp(2)
Answer: 6
in(blackbag,b3) in(whitebag,b2) in(whitebag,b1)
-in(blackbag,b2) -in(blackbag,b1) -in(whitebag,b3)
-hyp(1) hyp(2)
```

Listing 3.7: Example relevant literal generation output with where relevant literals cannot be extracted

3.6 Scoring and Ranking Relevant Literals

We want to narrow down the hypothesis space by asking the designer about the truth value of the relevant literals. It would be preferable that as much information as possible is gained from each answer from the designer. Thus the literal that is picked for questioning the designer should be able to reject as many revision suggestions as possible. Ideally, it should have the similar characteristics of discriminating test, being able to reject some revision suggestions regardless of its truth value.

To decide which literal to pick, we will need a way to score and rank them. We will be adopting a greedy approach by identifying the literals that reject the maximum number of suggested revisions independently of their truth value.

Thus, for each relevant literal l with that rejects number of n revision suggestions when it is true, and rejects m suggestions when it is false, the score $minimum(n, m)$ will be used to compare it to other literals. This minimum between the two values is used to ensure that our scoring gives discriminating literals higher score than those that are as discriminating.

For a set L of positive of relevant literals and set S of scores, with each $l_i \in L$ and corresponding set of scores $s_i \in S$, where $s_i = minimum(n_i, m_i)$, the most relevant literal is the a literal l whose score s is equal to the maximum score of S, thus: $s = maximum(\{s_i | s_i \in S\})$.

For example, consider the answer sets shown in Listing 3.8 which are some the output of literals generation where the each revision suggestion adds the following rule to our bean and bag example:

- Revision Suggestion 1: $in(whitebag, B) \leftarrow notbeancolour(B, white)$.
- Revision Suggestion 2: $in(whitebag, B) \leftarrow poisonous(B)$.
- Revision Suggestion 2: $\neg in(whitebag, B) \leftarrow beancolour(B, white)$.

```
Answer: 1
in(blackbag,b2) -in(blackbag,b3) -in(blackbag,b1)
-in(whitebag,b3) -in(whitebag,b2) -in(whitebag,b1)
-hyp(1) -hyp(2) hyp(3)
Answer: 2
in(blackbag,b2) in(blackbag,b1) -in(blackbag,b3)
-in(whitebag,b3) -in(whitebag,b2) -in(whitebag,b1)
-hyp(1) -hyp(2) hyp(3)
Answer: 3
in(blackbag,b3) in(blackbag,b2) -in(blackbag,b1)
-in(whitebag,b3) -in(whitebag,b2) -in(whitebag,b1)
-hyp(1) -hyp(2) hyp(3)
Answer: 4
in(blackbag,b3) in(blackbag,b2) in(blackbag,b1)
-in(whitebag,b3) -in(whitebag,b2) -in(whitebag,b1)
-hyp(1) -hyp(2) hyp(3)
```

```
...
```

Listing 3.8: Example result from relevant literals generation

Applying our algorithm for identifying relevant literals we can find out the literals required for rejecting each hypothesis:

- `hyp(1)` rejected by $\neg in(whitebag, b1)$
- `hyp(2)` rejected by $\neg in(whitebag, b3)$
- `hyp(2)` rejected by $\neg in(whitebag, b2)$
- `hyp(3)` rejected by $in(whitebag, b1)$

Then we give each literal a score based on the number of hypotheses it rejects, treating the cases of positive and negative separately:

- $in(whitebag, b1)$: 1.0
- $\neg in(whitebag, b1)$: 1.0
- $in(whitebag, b2)$: 0.0
- $\neg in(whitebag, b2)$: 1.0
- $in(whitebag, b3)$: 0.0
- $\neg in(whitebag, b3)$: 1.0

Then the minimum score for each literal will be:

- $in(whitebag, b1)$: 1.0
- $in(whitebag, b2)$: 0.0
- $in(whitebag, b3)$: 0.0

Thus, the literal that would be chosen as the most discriminating from these three would be $in(whitebag, b1)$ as whatever its truth value is, it will reject one hypothesis, while the other two will only reject a hypothesis when they have certain truth values.

We also use fractions when score literals that are needed in conjunction to discriminate a hypothesis. For example if instead if being able to discriminate `hyp(2)` by itself, $\neg in(whitebag, b3)$ and $\neg in(whitebag, b2)$ are needed in conjunction with each other to have enough power to reject `hyp(2)`. Then we would give each one of them a score of a half, to represent their strength in reject the hypothesis.

Chapter 4

ASP implementation of TAL

4.1 Preliminary Notations

These are the notations that were used in [1] for mode declarations. We will be using them in this chapter for the translation of mode declaration into top theories.

Given a head declaration *modeh*(s) or body declaration *modeb*(s):

- **id** is the unique identification of the mode declaration
- $\mathbf{s}^v$ is the literal obtained from s by replacing all of its placemakers with different variables $X_1, \ldots, X_n$
- $\mathbf{type(s}^v\mathbf{)}$ is the sequence of literals $t_1(X_1), \ldots, t_n(X_n)$ where each t_i is the type of variable X_i
- $\mathbf{con(s^*)}$ is the list of the variables that replace constant placemarkers '#*type*' in s, in order of appearance from left to right
- $\mathbf{inp(s^*)}$ is the list of the variables that replace input placemarkers '+*type*' in s, in order of appearance from left to right
- $\mathbf{out(s^*)}$ is the list of the variables that replace output placemarkers '-*type*' in s, in order of appearance from left to right

For example, for the mode declaration
$modeb(move(+block, +location, -location, \#fuel, -time))$:

- **id** could be any unique name for identifying the mode declaration from other mode declarations, for example if the predicate is not used in another mode declaration then *move* could be used for identifying this mode declaration.

- If $\mathbf{s}^v$ is the literal $move(X1, X2, X3, X4, X5)$
- $\mathbf{type(s^v)}$ would be the sequence of literals $block(X1), location(X2), location(X3), fuel(X4), time(X5)$
- $\mathbf{con(s^*)}$ would be the list of variables $(X4)$
- $\mathbf{inp(s^*)}$ would be the list of variables $(X1, X2)$
- $\mathbf{out(s^*)}$ would be the list of variables $(X3, X5)$

4.2 Overview

Inductive Logic Programming (Section 2.5) is used to learn the revision suggestions of the normative framework. The ILP learning tool TAL [4] is implemented using Prolog. In [1], to make it compatible to representation of the normative framework, the ILP task $< E, B, M >$ (Section 2.5.3) is translated to a Abductive Logic Programming task $< P, A, IC >$ (Section 2.4.1).

The example set E can be intuitively translated into integrity constraint IC. The mode declarations M, however, is translated into a top theory T. The top theory is constructed via the translation in [1] as follows.

- For each head declaration $modeh(s)$ and its unique identifier **id**, add the following rule to T

  $$\begin{aligned}\mathbf{s}^v \leftarrow\ & \mathbf{type(s^v)},\\ & rule(RId, 0, (\mathbf{id}, \mathbf{con(s^*)}, ())),\\ & rule\_id(RId),\\ & body(RId, 1, \mathbf{out(s^*)})\end{aligned}$$

- The following rule is in T (this acts as a marker for the end of the rule, for terminating the recursion)

  $$body(RId, L, \_) \leftarrow rule(RId, L, last)$$

- For each body declaration $modeb(s)$ and its unique identifier **id**, add the following rule to T

  $$\begin{aligned}body(RId, L, Inputs) \leftarrow\ & \mathbf{type(s^v)},\\ & rule(Rid, L, (\mathbf{id}, \mathbf{con(s^*)}, Links)),\\ & link\_variables(\mathbf{inp(s^*)}, Inputs, Links),\\ & \mathbf{s}^v,\\ & level(L),\\ & level(L+1),\\ & body(RId, L+1, (\mathbf{out(s^*)}, Outputs))\end{aligned}$$

$rule\_id(RId)$ and $level(L)$ are used to specify how many rules and rule conditions (levels) can be abduced, and
$link\_variables((a_1, \ldots, a_m), (b_1, \ldots, b_n), (o_1, \ldots, o_m))$ is true if for each a_i there exist b_j such that unifies with a_i, and $o_i = j$. $link\_variables/3$ is used to links

the input variables of the predicate to the list of variables from previous mode predicates in the rule. For example, if an instance of the partial rule being constructed is $aunt(susan, rob) \leftarrow sister(susan, jane)$, and we are trying to add $mother(+person, +person)$ to the end of the rule, then the list of inputs that will be given to $link\_variables/3$ is $(susan, rob, jane)$. Should we want to match $jane$ to the first argument of $mother(+person, +person)$ and rob to the second argument, then $link\_variables((jane, rob), (susan, rob, jane), (2, 1))$, assuming the first index to be 0, must be true.

T and the background knowledge B now fit into the abductive framework as the logic program $P = T \cup B$, and $rule/4$ and $rule/3$ are the only abducible predicate symbols in A. By implementing this in ASP, the rule is constructed by abducing the $rule/4$ atom associated to the rule head, then test out each definition of $body/3$ atom, trying to one where it can abduce other $rule/4$ atoms that can be added to the rule body.

4.2.1 Limitations

This representation of the of the mode declarations can very easily leads to an explosion in the number of grounded instances of clauses in the program, which takes a great amount of time and space to ground and can lead to the solver giving up on the program completely.

The problem is present in any implementation of ILP problems using ASP, as rules that are learnt have to be encoded by the literals abduced. This requires a large amount of information to be added to the program for encoding the rules.

For large problems such as the normative framework, the explosion of grounded instances often make the program unsolvable. To reduce the grounded instances and make the program solvable, the program would need to be simplified. For example, in some cases by removing the argument RId completely from the program, the ASP solver was able to generate the learning results. However, such removal of information not only makes it impossible to work out what rule head does each body literal corresponds to, it can also limits the search space (by removing RId, the number of rules abducible would decreases, as RId is used for specifying the number of rules abducible).

To address the problem of state space explosion, we considered alternative ways to implement the top theory of TAL in order to find the one that will reduce the grounding needed as much as possible.

4.3 List Approach

In order to make the result easier to interpret, we tried to represent the partial rules as a lists, and construct the new rule by adding bodies to the list. We change the translation of mode declarations to top theory such that:

- For each head declaration $modeh(s)$ and its unique identifier **id**, add the following rule to T

$$\begin{aligned}\mathbf{s}^v \leftarrow\ & \mathbf{type(s^v)},\\ & \quad \mathit{body}(\mathbf{out(s^*)}, (\mathbf{id}, \mathbf{con(s^*)}, ()))\end{aligned}$$

- The following rule is in T (similarly to the original translation, this acts as the base case to terminate the rule construction)

  $$\mathit{body}(\mathit{Inputs}, \mathit{PRule}) \leftarrow \mathit{rule}(\mathit{PRule})$$

- For each body declaration $\mathit{modeb}(s)$ and its unique identifier **id**, add the following rule to T

  $$\begin{aligned}\mathit{body}(\mathit{Inputs}, \mathit{PRule}) \leftarrow\ & \mathbf{type(s^v)},\\ & \mathit{link\_variables}(\mathbf{inp(s^*)}, \mathit{Inputs}, \mathit{Links}),\\ & \mathbf{s}^v,\\ & \mathit{append\_body}(\mathit{PRule}, (\mathbf{id}, \mathbf{con(s^*)}, \mathit{Links}), \mathit{NewPRule}),\\ & \mathit{append\_variables}(\mathit{Inputs}, \mathbf{out(s^*)}, \mathit{Outputs}),\\ & \mathit{body}(\mathit{Outputs}, \mathit{NewPRule})\end{aligned}$$

Keeping the transformation of the ILP task into an ALP one, with the exception of only $\mathit{rule}/1$ being in the only abducible predicate symbol, $\mathit{append\_body}/3$ and $\mathit{append\_variables}/3$ both adds elements to the end of the partial rule and variables list respectively, while $\mathit{link\_variables}/3$ retains the same function as in the original translation.

By using a list to represent the rules, we remove the need of using $\mathit{rule\_id}(\mathit{RId})$ and $\mathit{level}(L)$. Limiting the length of the rule is done by utilities functions, such as limiting $\mathit{append\_body}/3$ to only being able to add new bodies to partial rules of certain length, and specifically defining the allowed formats of the partial rules.

As lists are not included in the language for ASP, we have to define the list representation and operations ourselves. A too general definition of lists, such as ones of unlimited length, would only result in more instances explosion, thus we defined lists of all allowable sizes explicitly. For example, for rules of with up to two body literals, the following rules needed to added to the ASP program for representing the partial rule list and how to append body literals to them:

```
%-Partial rule
partial_rule((H)) :- head_lit(H).
partial_rule((H, B0)) :- head_lit(H), body_lit(B0).
partial_rule((H, B0, B1)) :- head_lit(H), body_lit(B0),
                             body_lit(B1).

%-Append rule body literal
append_body(B0, (H), (H, B0)) :- head_lit(H), body_lit(B0).
append_body(B1, (H, B0), (H, B0, B1)) :- head_lit(H), body_lit(B0),
                                         body_lit(B1).
```

Similar rules are used to define the variable lists, as well as additional rules needed for looking up indexes of variables within the list:

```
%-Variable Lists
vars(V0) :- v(V0).
vars((V0,V1)) :- v(V0), v(V1).
vars((V0,V1,V2)) :- v(V0), v(V1), v(V2).

append_var( V1, (V0), (V0,V1) ) :- v(V1), v(V0), vars((V0,V1)).
append_var( V2, (V0,V1), (V0,V1,V2) ) :- v(V2), vars((V0,V1,V2)).

%---Variables Index lookup
link( V0, (V0, V1, V2), 0 ) :- vars((V0, V1, V2)).
link( V1, (V0, V1, V2), 1 ) :- vars((V0, V1, V2)).
link( V2, (V0, V1, V2), 2 ) :- vars((V0, V1, V2)).

link( V0, (V0, V1), 0 ) :- vars((V0, V1)).
link( V1, (V0, V1), 1 ) :- vars((V0, V1)).

link( V0, (V0), 0 ) :- vars(V0).
```

The output of this implementation is aimed to be easier for interpretation as each rule is represented by a single list and do not need to be constructed by different predicates.

4.3.1 Limitations

Although it is easier to read off the result from this approach, it leads to an even greater state space explosion problem. This is due to the increase in the search space of the rule, as a consequent of using a list to represent it. If we compare it to the original representation, for a number of r values for RId, l values for rule levels, n total instances for the pair of link indexes and constants lists, then for h and b numbers of head and body declarations, the number grounded $rule/4$ instances (ignoring the end of rule marker) is:

$$(h + b) \cdot (r \cdot l \cdot n)$$

As the format of the predicate is $rule(RId, Level, (\mathbf{id},\ Constants, Indexes))$, thus there would be a total $(h + b)$ possible values for **id**, and per each **id** there are a total n instances for the two lists in $(\mathbf{id}, Constants, Indexes)$, each of such instance can be at any level in the rule body with any rule id (RId).

However, with this list representation, if the maximum number of body literals in the rule is k, then the number of grounded instances of this list is:

$$(h \cdot n) \cdot (\textstyle\sum_{i=1}^{k} (b \cdot n)^i)$$

For each head declaration, there are the total of n number instances due to the constant and index lists for representing the head literal of the list, and each head instance could be combined with any combinations of instances of body literals. Just like for each head declaration, each body will have n number of instances, and they could be arranged in any permutation after the head literal and there is no restriction on whether each instance can be repeat, or any ordering restrictions, so for each length *len* of the list of body predicate, there are $(b \cdot n)^{len}$ permutations of the instances that could be added to the head literal.

This leads to an extremely steep increase in the number of grounded atom as the maximum length of the rule increases. Furthermore, this will also create in a chain of increases in grounded instances of other predicates as the partial rule is used as part of predicates such as *body*/2 and *append_body*/3.

4.4 Double Negation Approach

We describes here the translation from [15], which employs double negation for rule learning. The top theory is constructed from the mode declarations using the following translation.

- For each head declaration *modeh*(s) and its unique identifier **id**, add the following rule to T

$$\begin{aligned}\mathbf{s}^v \leftarrow\ & \mathbf{type(s^v)},\\ & \mathit{head\_lit}(\mathbf{id}, RId, \mathbf{con(s^*)}),\\ & \mathit{not\ body}(\mathbf{b}_1, (\mathbf{id}, RId), \mathbf{out(s^*)}, \mathbf{out(s^*_{b_1})}),\\ & \mathit{not\ body}(\mathbf{b}_2, (\mathbf{id}, RId), (\mathbf{out(s^*)}, \mathbf{out(s^*_{b_1})}), \mathbf{out(s^*_{b_2})}),\\ & \ldots\\ & \mathit{not\ body}(\mathbf{b}_n, (\mathbf{id}, RId), (\mathbf{out(s^*_{b_{n-2})}}), \mathbf{out(s^*_{b_{n-1}})}), \mathbf{out(s^*_{b_n})})\end{aligned}$$

  Where each b_i is the unique identifier of each body declaration and s_{b_i} corresponds to the schema of the body declaration with unique identifier b_i.

- For each body declaration *modeb*(s) and its unique identifier **id**, add the following rule to T

$$\begin{aligned}\mathit{body}(\mathbf{id}, (\mathbf{id}_{head}, RId), \mathit{Inputs}, \mathbf{out(s^*)}) \leftarrow\ & \mathbf{type(s^v)},\\ & \mathit{link\_variables}(\mathbf{inp(s^*)}, \mathit{Inputs}, \mathit{Links}),\\ & \mathit{body\_lit}(\mathbf{id}, (\mathbf{id}_h ead, RId), \mathbf{con(s^*)}, \mathit{Links}),\\ & \mathit{not}\ \mathbf{s}^v\end{aligned}$$

Again the only difference between this method and the other previous ones is the translation for the top theory and the set of aducible predicate symbol. With this approach two predicate symbols *head_lit*/3 and *body_lit*/4 are abducible.

We slightly modify the translation given in [15] by passing the **id** of the head literal as input to *body*/4. This makes it easier to match up the rule bodies to their respective head. This representation also allows smaller values to be used for RId as each rule head can be uniquely identified by the combination of its **id** and RId, while each rule body can be identified by their their **id**, RId, and its rule head's $\mathbf{id}_h ead$. Thus, Rid can be used here to specify the number rules allowed with the same head. This helps decrease the search space as smaller values can be used for RId.

If the repetition of a rule body with the same body declaration is allowed, then more *not body* clauses can be added to he chain in the head declaration translation.

This approach allows greater restriction on the search space, for example, different sets of rule body can be associated to different rule heads. Furthermore, in most cases where repetition of bodies from the same body declaration is not allowed, this method will not generate redundant result from different permutations of the bodies in the rule.

4.4.1 Limitations

Consider the translation of each head declaration:

$\mathbf{s}^v \leftarrow \mathbf{type(s}^v\mathbf{)},$
$\quad \mathit{head\_lit}(\mathbf{id}, RId, \mathbf{con(s}^*\mathbf{)}),$
$\quad \mathit{not\ body}(\mathbf{b}_1, RId, \mathbf{out(s}^*\mathbf{)}, \mathbf{out(s}^*_{b_1}\mathbf{)}),$
$\quad \mathit{not\ body}(\mathbf{b}_2, RId, (\mathbf{out(s}^*\mathbf{)}, \mathbf{out(s}^*_{b_1}\mathbf{)}), \mathbf{out(s}^*_{b_2}\mathbf{)}),$
$\quad \ldots$
$\quad \mathit{not\ body}(\mathbf{b}_n, RId, (\mathbf{out(s}^*_{b_{n-2})}\mathbf{)}, \mathbf{out(s}^*_{b_{n-1}}\mathbf{)}), \mathbf{out(s}^*_{b_n}\mathbf{)})$

As the input of each *body*/2 atom is constructed from outputs of previous two atoms, should any mode declarations have output variables they must be given priority to be the first in such a chain. This is to ensure that subsequent atoms are not missing any variables that could decide if *body_lit*/4 predicate is abduced or not. This would also mean that if there are more than one body declarations with output variable, then this clause for the head declaration must be repeated for different permutations of such bodies.

4.5 Other Optimisations

- The problem of redundant results due to permutations of rule bodies can be avoided by forcing an ordering on them. This works well when all body declarations only have constants and input variables, but is not so straight forward when some have output variables, as the ordering could make the literals with outputs be forced behind others that need them as inputs.
- One of the main reason for the problem of the state space explosion is the constants, variables and indexes lists. For example, in the original approach, for a number of r values for RId, l values for rule levels, n indexes for linking variables, and k possible constants, suppose there is no restriction on the length of the constants and index lists, the number of grounded $rule/4$ instances per mode declaration is:

$$r \cdot l \cdot \sum_{i=1}^{n} n^i \cdot \sum_{i=1}^{k} k^i$$

As the number of constants possible is n, the maximum length for the list must also be n, and for each length len of the list there are n^{len} permutations of constants, this reasoning also apply for the indexes list. Each constants list could be associated with any of the indexes lists, thus the total number of instances from these lists are $\sum_{i=1}^{n} n^i \cdot \sum_{i=1}^{k} k^i$. Each one of such lists combination could have any RId and be at any level of the rule.

These lists also increases the instances of other predicates, such as $body/3$, $link\_variables/3$, and any other utility functions needed for constructing or editing such lists. Thus, a more rigid representation for these would help with reducing the number of grounded instances of the program. For example, by imposing a fixed size on the list which would reduce the number of instances permutations from $\sum_{i=1}^{n} n^i$ to $n^{len_{max}}$.

4.6 Examples

We will use each approach to solve the same problem in order to show the difference in their encoding and the results produced. For the ILP problem that we will be solving, we will be using the bean and bags example in Listing 4.1. The background contains facts describing three black and white beans, two bags with the colour black or white.

```
%----------------Background----------------%
bean(b1).
bean(b2).
bean(b3).

bag(whitebag).
bag(blackbag).

colour(white).
colour(black).

bagcolour(whitebag,white).
bagcolour(blackbag,black).

beancolour(b1, white).
beancolour(b2, black).
beancolour(b3, white).

poisonous(b1).
special(b1).
```

Listing 4.1: Example background knowledge of ILP task

The example set contains a negative `in(whitebag, b2)` example and a positive example `in(whitebag, b1)`. In all approaches, this will be added as an integrity constraint, as shown in Listing 4.2.

```
%-----------------Example-----------------%
example :- in(whitebag, b1), not in(whitebag,b2).
:- not example.
```

Listing 4.2: Positive example to be covered by the learning result

Lastly, we will be using the following mode declarations:

- in: $modeh(in(+bag, +bean))$
- bgc1: $modeb(bagcolour(+bag, -colour))$
- bgc2: $modeb(bagcolour(+bag, \#colour))$
- bec: $modeb(beancolour(+bean, +colour))$

These are chosen to show the difference between mode declarations with different types of placemarkers.

4.6.1 Using Original Method

The Listing 4.3 shows the top theory constructed using the first translation from Section 4.2.

```
%----------------Top Theory---------------%
%modeh(in(+bag,+bean))
in(BG,BE) :- bag(BG), bean(BE), rule_id(RId),
             rule(RId, 0, (in, (e), (e))),
             body(RId, 1, (BG,BE)).

body(RId,L,Inputs) :- vars(Inputs), rule(RId, L, last)
    .

%modeb(bagcolour(+bag,-colour))
body(RId,L,Inputs) :- link( BG, Inputs, IO ),
                    bag(BG), colour(CO),
                    bagcolour(BG, CO),
                    append_var(CO, Inputs, Outputs),
                    rule(RId, L, ( bgc1, (e), (IO))),
                    body(RId, L + 1, Outputs).

%modeb(bagcolour(+bag,#colour))
body(RId,L,Inputs) :- link( BG, Inputs, IO ),
                    bag(BG), colour(CO),
                    bagcolour(BG, CO),
                    rule(RId, L, ( bgc2, (CO), (I0))),
                    body(RId, L + 1, Inputs).

%modeb(beancolour(+bean,+colour))
body(RId,L,Inputs) :- link( BE, Inputs, I0 ),
                    link( CO, Inputs, I1 ),
                    bean(BE), colour(CO),
                    beancolour(BE, CO),
                    rule(RId, L, ( bec, (e), (I0,I1))),
                    body(RId, L + 1, Inputs).
```

Listing 4.3: Example top theory constructed using the original approach

The main difference between the format of each mode declarations in the top theories depends on their placemarkers, such as needing to add output variables to the existing variable lists if there are output placemarkers, and keeping adding the constants or linking indexes to the abduced $rule/3$ according to the constant and input placemakers of the mode declaration.

The Listing 4.4 shows the abducible predicate symbol added to the program. We specify for precisely four of these predicates to be abduced as it is the minimum rule length needed for the program to produce any solution.

```
%----------------Abducibles---------------%
4 { rule( RId, L, PRULE ) : rule_id(RId) : level(L) :
    prule(PRULE) } 4.
```

Listing 4.4: Abducible predicate symbol for the ALP task

The results of the completed program is shown in Listing 4.5, e is the symbol used for empty lists, and variable indexes starts from zero. This result can be translated to the rule $in(BG, BE) \leftarrow bagcolour(BG, CO), beancolour(BE, CO)$ by comparing the output to the mode declarations, and matching the value of each variable index to the order in which the variables are added to the rule. In these representation, the indexes used for the variables starts at 0, thus the indexes one and two in `rule(1,2,(bec,e,(1,2)))` refers to the second and third variables in the rule.

```
Answer: 1
rule(1,3,last)
rule(1,2,(bec,e,(1,2)))
rule(1,1,(bgc1,e,0))
rule(1,0,(in,e,e))
```

Listing 4.5: Result of learning using the original approach

4.6.2 Using the List Approach

Following the translation of the mode declarations that was defined in Section 4.3, the result is the top theory in Listing 4.6.

```
%----------------Top Theory---------------%
%modeh(in(+bag,+bean))
in(BG,BE) :- bag(BG), bean(BE),
             body( (BG,BE), (in, (e), (e)) ).

body(Inputs, PRule) :-  vars(Inputs), rule(PRule).

%modeb(bagcolour(+bag,-colour))
body(Inputs, PRule) :- link( BG, Inputs, I0 ),
     bag(BG), colour(CO),
     bagcolour(BG, CO),
     append_var(CO, Inputs, Outputs),
     append_body((bgc1, (e), (I0)), PRule, NewPRule),
     body(Outputs, NewPRule).

%modeb(bagcolour(+bag,#colour))
body(Inputs, PRule) :- link( BG, Inputs, I0 ),
     bag(BG), colour(CO),
     bagcolour(BG, CO),
     append_body((bgc2,(CO),(I0)), PRule, NewPRule),
     body(Inputs, NewPRule).

%modeb(beancolour(+bean,+colour))
body(Inputs, PRule) :- link( BE, Inputs, I0 ),
     link( CO, Inputs, I1 ),
     bean(BE), colour(CO),
     beancolour(BE, CO),
     append_body((bec,(e),(I0,I1)), PRule, NewPRule),
     body(Inputs, NewPRule).
```

Listing 4.6: Example top theory constructed using the list approach

The abducible predicate symbol is added in a similar manner to the previous approach as shown in Listing 4.7

```
%----------------Abducibles---------------%
0 { rule( R ) : interm_rule(R) } 1.
```

Listing 4.7: Abducible predicate symbol for the ALP task

As this approach produce a an extremely large search space compared to the other two, to make this program terminate, a lot of optimisation is needed.

This includes a strict ordering on the lists of variables, *Inputs* and *Outputs*, as well as the rule bodies. Unfortunately, this was not enough and one of the mode declarations, *bgc*2, was removed to make the program terminate.

```
Answer: 1
rule(((in,e,e),(bgc1,e,0),(bec,e,(1,2))))
```

Listing 4.8: Result of the ILP task using the list approach

The translation from the result is Listing 4.8 to a rule can be done in a similar manner as the last approach, by comparing each rule body to the corresponding mode declaration and building up a variable list for figuring out how the variables in the rule are linked to each other.

4.6.3 Using Double Negation Approach

Lastly, using the translation of Section 4.4, we can translate the mode declarations to acquire the top theory in Listing 4.9.

```
%----------------Top Theory---------------%

%modeh(in(+bag,+bean))
in(BG,BE) :-
    bag(BG), bean(BE), rule_id(RId),
    head_lit(in, RId, (e)),
    not body(bgc1, (in, RId), (BG,BE), (CO)), colour(
        CO),
    not body(bgc2, (in, RId), (BG,BE,CO), (e)),
    not body(bec, (in, RId), (BG,BE,CO), (e)).

%modeb(bagcolour(+bag,-colour))
body(bgc1, (HId, RId), Inputs, (CO) ) :-
    link( BG, Inputs, I0 ),
    bag(BG), colour(CO),
    body_lit(bgc1, (HId, RId), (e), (I0)),
    not bagcolour(BG, CO).

%modeb(bagcolour(+bag,#colour))
body(bgc2, (HId, RId), Inputs,(e)) :-
    link( BG, Inputs, I0 ),
    bag(BG), colour(CO),
    body_lit(bgc2, (HId, RId), (CO), (I0)),
    not bagcolour(BG, CO).

%modeb(beancolour(+bean,+colour))
```

```
body(bec, (HId, RId), Inputs,(e)) :- link( BE, Inputs,
     I0 ),
     link( CO, Inputs, I1 ),
     bean(BE), colour(CO),
     body_lit(bec, (HId, RId), (e), (I0,I1)),
     not beancolour(BE, CO).

%----------------Abducibles---------------%

0 { head_lit( ID, RId, CLIST ) : head_id(ID) : rule_id
    (RId) : cons(CLIST) } 1.

0 { body_lit( ID, (HId, RId), CLIST, ILIST ) : id(ID)
    : head_id(HId) : rule_id(RId) : cons(CLIST) :
    indexes(ILIST) } 2.
```

Listing 4.9: Example top theory constructed using the double negation approach and abducible predicate symbols

The answer set acquired from this program is shown in listing 4.10. This has the same translation as the results from the previous approaches, $in(BG, BE) \leftarrow bagcolour(BG, CO), beancolour(BE, CO)$.

```
Answer: 1
head_lit(in,1,e)
body_lit(bec,1,e,(1,2))
body_lit(bgc1,1,e,0)
```

Listing 4.10: Result of learning using the double negation approach

4.7 Evaluation

While all three approaches can learn the same rule, there is a significant difference in their performance. We compare each implementation approach using the examples implemented.

So that the comparison reflects the difference between each implementation of the top theory, we the same mode declarations were used in all implementations, the order in which they are added to the program are also kept the same. All implementations share the following utility functions to make the comparison fair:

```
%---Variables
v(BG) :- bag(BG).
v(BE) :- bean(BE).
v(CO) :- colour(CO).
```

```
%---Variables Lists
vars(V0) :- v(V0).
vars((V0,V1)) :- v(V0), v(V1).
vars((V0,V1,V2)) :- v(V0), v(V1), v(V2).

append_var( V1, (V0), (V0,V1) ) :- v(V1), v(V0), vars((V0,V1)).
append_var( V2, (V0,V1), (V0,V1,V2) ) :- v(V2), vars((V0,V1,V2)).

%---Variables Links
link( V0, (V0, V1, V2), 0 ) :- vars((V0, V1, V2)).
link( V1, (V0, V1, V2), 1 ) :- vars((V0, V1, V2)).
link( V2, (V0, V1, V2), 2 ) :- vars((V0, V1, V2)).

link( V0, (V0, V1), 0 ) :- vars((V0, V1)).
link( V1, (V0, V1), 1 ) :- vars((V0, V1)).

link( V0, (V0), 0 ) :- vars(V0).

%---Indexes Lists
index(0..2).
indexes(e).
indexes(I0) :- index(I0).
indexes((I0, I1)) :- index(I0), index(I1).

%---Constants Lists
cons(C0) :- colour(C0).
cons(e).
```

We base our comparison on the time taken by the solver to ground each program, we did not consider the solving time as for all programs that were successfully grounded the time taken to solve each one is at most 0.2 seconds.

iClingo provides a command line option for for printing out statistical data on successful termination of the solver, we use this to find out the number of grounded atoms in each implementation. We also varied the number of mode declarations to observe how it effects the complexity of the program.

In the Table 4.2 we were not able to find out the number grounded atoms and rules when using four mode declarations in with the list implementation. This is due to the program interruption that we had to use on the solver so that it does not use up all memory of the machine.

From the three tables, Table 4.1, and 4.2, it is clear that the ASP implementation using double negation is the most efficient approach, while the list implementation fares extremely poorly in efficiency. Although we have expected for the list implementation to have poorer results compared to the other two, the dramatic increase in number of grounded atoms from 23234 to 1340865, more than fifty times increase, and the state space explosion that resulted from using only four mode declarations highlights how the approach is not practical

Time taken to ground program (s)			
Implementation Used	Number of mode declarations		
	2	3	4
Original	0.19	0.26	0.43
List	0.37	32.58	$>$ 102.06
Double Negation	0.06	0.08	0.08

Table 4.1: Time spent on grounding of different ASP implementation of TAL

Number of atoms in program			
Implementation Used	Number of mode declarations		
	2	3	4
Original	4998	5506	6014
List	23234	1340865	
Double Negation	2824	3089	3490

Table 4.2: Number of grounded atoms in different implementation of TAL

to use.

The tables also shows that the double negation approach is much more efficient than the original, as reflected in the extremely small increase in grounding time as the number of mode declarations increases, due to the small number of grounded atoms it has.

Chapter 5

Evaluation: Normative Framework Case Study

5.1 Overview

We use the revision problem from [1] as a case study to evaluate the performance of our approach of for finding relevant literals when applied to normative framework revision.

The normative framework that we use is described in by following specification:

- It consists of active agents, each one having ownership of some digital blocks of data that form parts of larger files.
- An agent must share a copy of its blocks of data before being allowed to download another block from another agent.
- Initially, there is only one copy of each block within the agent population, and each one is owned an agent within the population.
- Agents with VIP status can download blocks without any restriction.
- Violations and misuse of an agent are generated when an agent requests for a download without sharing one of its blocks after its previous download.
- A misuse terminates the agent's empowerment to download blocks.

5.2 Normative Framework Revision

To be able to delete or revise rules within the partial normative framework N, it is divided into two parts $N = N_B \cup N_T$, where N_B is parts of the framework that is not revisable and N_T is the part containing the revisable theory. As described in [1], a pre-processing phase is needed so that a standard ILP learner is able to learn hypotheses at the meta-level, so that exceptions of the rules can be learnt. This pre-processing phase consists of:

1. For each rule in N_T, replace its body literals c^i_j, where i is the rule index and j is the index of the body literal within the rule, by the atom $try(i, j, c^i_j)$.
2. Add the condition *not* $exception(i, r_i, v_i)$ to all rules r_i in N_T, where v_i is an optional list of additional variables appearing in the body.
3. For each $try(i, j, c^i_j)$ atom added to a rule, define it to be true either if $del(i, j)$ is true, else it is true when ever the condition c^i_j is true.
4. By adding head declarations for $del(i, j)$ and $exception(i, r_i, v_i)$, if $del(i, j)$ is included in the learning result then it indicates the condition c^i_j to be removed from revised framework, while any hypotheses of the form $exception(i, r_i, v_i) \leftarrow c_1, \ldots, c_n$ learnt indicates conditions that need to be added as exceptions to rule r_i. These exceptions are added by replicating the r_i into n new rules, adding one of the condition c_k, $1 \leq k \leq n$, to each one.

For example, the Listing 5.1 demonstrates how the rule 5 is transformed by this pre-processing phrase.

```
%---rule 5
%---occurred(myDownload(X,B),I) :-
$---  occurred(download(Y,Y,B),I),
%---  holdsat(hasblock(Y,B),I)).

occurred(myDownload(X,B),I) :-
    try(5, 1, occurred(download(Y,Y,B),I)),
    try(5,2, holdsat(hasblock(Y,B),I)),
    not exception(5, occurred(myDownload(X,B),I), Y).

try(5, 1, occurred(download(Y,Y,B),I)) :-
    not del(5,1),
    occurred(download(Y,Y,B),I).

try(5, 1, occurred(download(Y,Y,B),I)) :-
    del(5,1).

try(5, 2, holdsat(hasblock(Y,B),I)) :-
    not del(5,2),
    holdsat(hasblock(Y,B),I).

try(5, 2, holdsat(hasblock(Y,B),I)) :-
    del(5,2).
```

Listing 5.1: Revisable rule 5 transformed to allow learning at meta-level

Thus, if `del(5,1)` is learnt, then the literal `occurred(download(Y,Y,B),I)` will be removed from the rule, while if `exception(5, occurred(myDownload(X,B),I), Y)` ← `isVIP(X)` is learnt, then `not isVIP(X)` will be added as another body literal of rule 5.

5.3 Learning Revision Suggestions

We are using the same case study from the paper [1], but modifying it to fit our purpose of generating multiple revision suggestions for the normative framework. We kept the set of revisable rules N_T to remain the same, consisting of:

- Rule 1

```
initiated(hasblock(X, B), I) :-
    occurred(myDownload(X, B) , I).
```

- Rule 2

```
initiated(perm(myDownload(X, B)), I) :-
    occurred(myShare(X), I).
```

- Rule 3

```
terminated(pow(extendedfilesharing, myDownload(X, B)), I) :-
    occurred(misuse(X), I).
```

- Rule 4

```
terminated(perm(myDownload(X, B2)), I) :-
    occurred(myDownload(X, B), I).
```

- Rule 5

```
occurred(myDownload(X, B), I) :-
    occurred(download(Y, Y, B), I),
    holdsat(hasblock(Y, B), I).
```

- Rule 6

```
occurred(myShare(X), I) :-
    occurred(download(Y, X, B), I),
    holdsat(hasblock(X, B), I).
```

Consequently, the mode declarations contain the head declarations needed for revising these rules as discussed in Section 5.2. Other mode declarations that are included are as follows:

- $modeh(occurred(misuse(+agent), +instant))$
- $modeb(occurred(myDownload(+agent, +block), +instant))$
- $modeb(notoccurred(download(+agent, +agent, +block), +instant))$
- $modeb(isVIP(+agent))$
- $modeb(notholdsat(hasblock(+agent, +block), +instant))$
- $occurred(viol(myDownload(+agent, -block)), +instant)$

The trace of the system and the set of expected violations for the use case, are defined as follows.

$$\begin{aligned}
T = \{&observed(start, i00),\\
&observed(download(alice, bob, x3), i01),\\
&observed(download(charlie, bob, x3), i02),\\
&observed(download(bob, alice, x1), i03),\\
&observed(download(charlie, alice, x1), i04),\\
&observed(download(alice, charlie, x5), i05),\\
&observed(download(alice, bob, x4), i06)\}
\end{aligned}$$

$$\begin{aligned}
O = \{&viol(myDownload(alice, x4), i06)\} \cup\\
\{&not\ viol(myDownload(a, c), i) | a \in Agents, b \in Blocks, i \in Instances,\\
&i \neq i06\}
\end{aligned}$$

From the trace and system specification, the only violation that should occur is the download by *alice* at time $i06$. This is because *alice* failed to share any of her data block with other agents after the previous download at time $i05$. The agent *charlie*, however, does not have the same restriction placed on him as *charlie* is a VIP agent. Domain facts regarding the agents and data blocks within in the system, as well as any agent's VIP status are included in the fixed portion of the of the partial normative framework.

We used the double negation translation of the mode declarations to produce the top theory for for the learning task, as this size of this problem is great enough that at times the top theory from the translation in [1] may fail to produce any results.

Below were the revision suggestions learnt by the learner:

- Revision suggestion 1:

```
%---Rule 4
terminated(perm(myDownload(X, B2)), I) :-
```

```
    occurred(myDownload(X, B), I),
    not isVIP(X).

%---Rule 5
occurred(myDownload(X, B), I) :-
    occurred(download(X, Y, B), I),
    holdsat(hasblock(Y, B), I).
```

- Revision suggestion 2:

```
%---Rule 4
terminated(perm(myDownload(X, B2)), I) :-
    occurred(myDownload(X, B), I),
    not occurred(viol(myDownload(X,B3)),I).

%---Rule 5
occurred(myDownload(X, B), I) :-
    occurred(download(Y, Y, B), I),
    holdsat(hasblock(Y, B), I).

%---Rule 7 (added)
occurred(misuse(X),I) :- isVIP(X).
```

5.4 Generating Relevant Literals

Following the steps in Section 3.3 and Section 3.4, we converted each revision suggestion into a hypothesis and transform the partial normative framework accordingly. The hypotheses are shown in Listing 5.2.

```
%-Sugesstion-1---------------------------

% Revise rule 4
-hyp(1) :- not terminated(perm(myDownload(X,B)),I),
           cond_1_1(X,B,I).
cond_1_1(X,B,I) :- occurred(myDownload(X,B2),I),
                   not isVIP(X).

% Revise rule 5
-hyp(1) :- not occurred(myDownload(X,B),I),
           cond_1_2(X,B,I).
cond_1_2(X,B,I) :- occurred(download(X,Y,B),I),
                   holdsat(hasblock(Y,B),I).

%-Sugesstion-2----------------------------

```

```
% Revise rule 4
-hyp(2) :- not terminated(perm(myDownload(X,B)),I),
           cond_2_1(X,B,I).
cond_2_1(X,B,I) :- occurred(myDownload(X,B2),I),
                   not occurred(viol(myDownload(X,B3))
                       ,I).

% Revise rule 5
-hyp(2) :- not occurred(myDownload(X,B),I),
           cond_2_2(X,B,I).
cond_2_2(X,B,I) :- occurred(download(X,Y,B),I),
                   holdsat(hasblock(Y,B),I).

% Add new rule
-hyp(2) :- not occurred(misuse(X),I),
           cond_2_3(X,I).
cond_2_3(X,I) :- isVIP(X), instant(I).
```

Listing 5.2: The revision suggestions as hypotheses for relevant literals generation

Unlike in simpler example where only one predicate symbol is abducible, as the use case have wide effects on the partial model, the head of all changed rules need to be abducible in the program, these as shown in Listing 5.3.

```
0 { occurred(misuse(X),I) } 1 :-
      agent(X), instant(I).
0 { occurred(viol(myDownload(X,B)),I) } 1 :-
      agent(X), block(B), instant(I).
0 { occurred(myDownload(X,B),I) } 1 :-
      agent(X), block(B), instant(I).
0 { terminated(perm(myDownload(X,B)),I) } 1 :-
      agent(X), block(B), instant(I).
```

Listing 5.3: The abducible predicates for the case study

5.5 Identifying Relevant Literals

Using the exact method from Section 3.3 to generated all possible combinations of relevant literals lead to an explosion in the number of answer sets in the solution.

Nevertheless, there is still ways of finding relevant literals. In order to handle the problem of state explosion, the optimisation tools of iClingo can be used. This works by specifying which predicate should have the smallest or largest number of instances in the answer set as possible. This could be used on the abducible predicates to limit the number of answer sets they generate. In this case, the number of answer sets reduced from 30,000+ to less than 200. However, as this does not guarantee that all combinations of $hyp(X)$ and $\neg hyp(X)$ will be in these answer sets, we cannot use set comparison to extract the relevant literals.

Seeing as we cannot rely on the solver to generate all combinations of $hyp(X)$ and $\neg hyp(X)$ when applying optimisation, we have to direct the search ourselves through constraints on the hypotheses. For example, we can go through each hypothesis one by one, solving the program once for the answer sets that will eliminate the hypothesis and then again to find those that will not eliminate it, before comparing the answers for the relevant literals.

However, such approach will not guarantee that all relevant literals are generated, and in some case the sets comparison will not eliminate all literals that are not relevant. This method can be used as approximation at best, as many answer sets are not included in the result due to the optimisation applied, resulting in a loss of much information.

On the other hand, from the state space explosion, it is clear that the method that we are using in Listing 5.3 for generating all potential relevant literals and heads of the revised or new rules is too general and can disregard the trace in the use case completely. For example, `occurred(myDownload(X,B),I)` can be abduced even if no download have been observed in the trace given by the designer, furthermore the only `occurred(misuse(X),I)` that actually have any meaning to us is the one that can be generated from the new rule in the second revision suggestion, and lastly, `occurred(viol(E),I)` already already defined as rules in the unchanged part of the framework.

This suggests that we can impose further restriction on our generation method, by taking into account the common conditions on the abducible literals in all of the revision suggestions. For each rule head h_i revised by some revision suggestions from the set REV, to find the conditions that we could impose for abducing each h_i we can find the intersection of all of its conditions in the revision where it is included $\bigcap_{c_j \in b_i, (h_i \leftarrow b_i) \in REV} c_i$, as this would be the common conditions on h_i in all suggestions.

How to treat literals such as `occurred(viol(myDownload(X,B)),I)` is debatable, as although they are used in the use case, their definition in the normative framework remain unchanged. So with respect to our original objective of building up the use case, they should not have further constraints placed on them, as their value should be defined by the designer. However, unlike the learning stage, where they are used for finding exceptions within the framework, we are trying to find the differences implied by each revision suggestion. Thus, even if some of these literals can reject some hypotheses but is not derivable by their existing rule, then they neither be derivable by the revised framework. For example, if we have the following hypothesis for the revised rule r:

```
hyp(1) :- not -hyp(1).
-hyp(1) :- not r(X), cond(X).
cond(X) :- p(X,const)
```

With the following rule as background knowledge:

```
p(X,const) :- int(X), even(X).
int(2).
int(3).
```

If the hypothesis is rejected on the grounds that `p(3,const)` is chosen to be true and `r(X)` is false, if the rules for $p/2$ is not changed by any revision suggestion, then `p(3,const)` would still not be derivable in the revised theory. Thus any such predicate with unchanged definition should be removed from the abducible set. This would give us a new specification for generating literals in Listing 5.4.

```
0 { occurred(misuse(X),I) } 1 :-
      agent(X), instant(I), isVIP(X).
0 { occurred(myDownload(X,B),I) } 1 :-
      agent(X), block(B), instant(I),
      occurred(download(X,Y,B),I), agent(Y).
0 { terminated(perm(myDownload(X,B)),I) } 1 :-
      agent(X), block(B), instant(I),
      occurred(myDownload(X, B2), I).
```

Listing 5.4: The revised abducible predicates for the case study

Using the new set of abducible literals, we are able to restrict the number of answer sets generated to only 1152 sets. Applying our algorithm on these sets will identify the following literals:

$occurred(myDownload(charlie, x1), i04)$
$occurred(myDownload(charlie, x3), i02)$
$\neg occurred(misuse(charlie), i00)$
$\neg occurred(misuse(charlie), i01)$
$\neg occurred(misuse(charlie), i02)$
$\neg occurred(misuse(charlie), i03)$
$\neg occurred(misuse(charlie), i04)$
$\neg occurred(misuse(charlie), i05)$
$\neg occurred(misuse(charlie), i06)$

If the designer choose any one these to be true, then the second revision suggestion can be rejected, thus they are all equally relevant.

5.6 Summary

From this case study, we adapted our work to problems of larger domain with normative framework revision. Regarding our addition to the framework revision, our initial attempt has shown us the consequence of approaching the problem naively. By giving the answer solver free reign over abducing literals, the amount of answer sets produced was too much to be processed.

We have discussed whether iClingo's optimisation feature could be used to address the state explosion problem. While this would give us a workable number of answer sets, we concluded that the answers are at best approximation, as too much informative would have been lost.

Lastly, we have shown how the the literal generation can be controlled by restricting the abducible literals to those effected by the revised or new rules only. This provided us a reasonable number of answer sets, from which we were able to identify the relevant literals.

We also identified another weakness in our first attempt of generating relevant literals. By including literals, whose definitions are not changed by any revision suggestion, into the set of abducibles, the relevant literal identified may not be derivable by the framework or any its revisions, thus dismissing revisions based on such literals could be misleading.

For the learner, the program used for learning was implemented using the approach from Section 4.4. In an average from of five runs, the learner took 6.26 seconds to produce the solutions, which is an improvement from using the previous implementations that will lead to a memory overflow.

Chapter 6

Conclusion and Future Work

In this project, we have taken the idea of test abduction and applied it for generating criteria that can be used for dismissing hypotheses of an ILP learner. Although we have directed this project to address a problem raised by a specific past work on normative framework revision, it could also be applied to other types of learning problems. For revising models, we have described how to form hypotheses to represent possible changes in the model by the revision suggestions, and the characterisations needed for a literals to relevant in dismissing these hypotheses. These are then used to form an abductive problem of generating relevant literals and implemented using ASP. We also described an algorithm that can be used to extract the exact relevant literals from the answer set results of relevant literal generation, identifying the hypotheses each one can reject and they can be ranked against each other to find most relevant literal.

In our case study, we applied our work on test generation, and ASP implementation of TAL, to use with normative framework revision. This case study highlights some problems with our method, and we were able to identify ways of improving our method. However, even if we were able to overcome the state explosion in this use case, it still shows how easily can such a problem arise. For example, should there be no common constraints that can be used to restrict the literals generation, then we would have no means of avoiding the state explosion.

Furthermore, during the investigation of the case study, it was apparent that for a well defined problem TAL learns very few hypotheses. This brings up the question of whether it is important to identify the most relevant literal, as it could be more practical to to ask the user about any of the relevant literals found, as this could also avoid the computational cost of finding all relevant literals.

6.1 Future Work

These are some work that we were hoping to explore on normative framework revision and test generation.

6.1.1 Multiple Normative Framework

There is another answer set compatible representation that allows for reasoning with many frameworks. It would be of interest to see if our work could also be applied to such representation and what results would it produce. This could potentially explore the limits of normative framework size that our method would be able to handle.

6.1.2 Generate Use Case From Social Constraints

Initially we wanted to use test generation for creating use cases for the learner, though describing properties by a set of hypotheses. However with the method we have taken for test generation we though that it would not be appropriate to follow through with this approach, as it would imply that the changes needed for the framework is already known, defeating the point of learning the revision suggestions.

As described in Section 2.2.1, normative frameworks have certain social constraints that they try to express. These could be studied and generalised into properties that should be exhibited by all frameworks, for example for empowered agents never raising certain exceptions.

Alternatively a language could be created based on concepts of social constraints, for example we could use a predicate to describe that agents of certain type are not permitted to perform certain actions $\mathit{prohibited}(Type, Action)$. The same could be applied for other constraints such as institutional effect, obligation, and violation, as they are common to all normative frameworks. These could be used for the designer to to describe the behaviours they want in the normative system, with each concepts having a predefined translation into hypotheses such that they could be used by our test generation to produce use cases.

6.1.3 Method for Generating Relevant Literals

Considering the state space explosion problem we had when a revision suggestion revise many rules, we should explore ways to improve our method, or find look at complete different ways for generating relevant literals. This could be by exploring different reasoning techniques and tools. For example, instead of from bottom-up reasoning, could we find all relevant tests using a top-down approach, and would it overcome the problem that we have experienced in the use case.

In our approach with ASP, the grounding of literals was not the main problem, but simply from the number of answer sets it produce. Many of these answer sets have the same relevant literals, thus a way to avoid such redundancy would possibly avoid any state space explosion from occurring.

Another approach could be reasoning at the meta-level to find the differences in the revision suggestions. For example, in the case study, the program was transformed so that the learner can learn rules at the meta-level. By reasoning with the transformed model, the learner is able to find exceptions to the rules. Thus could similar reasoning not be applicable for finding relevant literals, as we are searching for literals that are exceptions for the revision suggestions to use for dismissing them.

6.1.4 Normative Framework Revision Tool

We have concentrated on the theory and method for revising normative framework, one clear extension of this project would be to implement a tool for interactive revision of the normative framework. The system should be able to question the user for further information after learning the revision suggestions to help the user dismiss any unwanted revision suggestions.

Bibliography

[1] D. Corapi, M. De Vos, J. Padget, A. Russo and K. Satoh, *Norm Refinement and Design through Inductive Learning.* 11th International Workshop on Coordination, Organization, Institutions and Norms in Multi-Agent Systems, Domaine Valpre' in Lyon, France, 30th August - 2nd September 2010.

[2] Owen Cliffe, Marina De Vos, and Julian Padget, *Answer Set Programming for Representing and Reasoning about Virtual Institutions.* Seventh Workshop on Computational Logic in Multi-Agent Systems, Japan, May 2006.

[3] Nada Lavrac, and Saso Dzeroski, *Inductive Logic Programming Techniques and Applications.* Ellis Horwood, New York, 1994.

[4] Domenico Corapi, Alessandra Russo, and Emil Lupu, *Inductive Logic Programming as Abductive Search.* Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK.

[5] A. Jones, and M. Sergot, *A formal characterisation of institutionalised power.* Journal of the IGPL, 4(3):429.445, 1996.

[6] Alexander Artikis, Marek Sergot and Jeremy Pitt, *An Executable Specification of an Argumentation Protocol.* In Proceedings of Conference on Artificial Intelligence and Law (ICAIL), pp. 1-11, Edinburgh, 2003.

[7] Stuart J. Russell, and Peter Norvig, *Artificial Intelligence: A Modern Approach (Second Edition).* Pearson Education, 2003, pp 34-38.

[8] S. McIlraith, *Logic-based abductive inference.* Technical Report KSL-98-19, Knowledge Systems Lab, Department of Computer Science, Stanford University, 1998.

[9] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner, *Answer Set Programming: A Primer.* In Sergio Tessaris and Enrico Franconi and Thomas Eiter and Claudio Gutierrez and Siegfried Handschuh and Marie-Christine Rousset and Renate A. Schmidt, editors, 5th International Reasoning Web Summer School (RW 2009), Brixen/Bressanone, Italy, August 30-September 4, 2009, volume 5689 of LNCS, pages 40-110. Springer, September 2009.

[10] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele, *A User's Guide to gringo, clasp, clingo, and iclingo.* Avaliable form http://sourceforge.net/projects/potassco/files/potassco_guide/2010-10-04/

[11] S. Mcllraith, *Generating Test using Abduction.* In Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94) (1994), pp. 449-460.

[12] W. Vasconcelos, M. Kollingbaum, and T. Norman, *Resolving conflict and inconsistency in norm-regulated virtual organizations.* In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07), 14-18 May, 2007. Honolulu, Hawai'i, USA.

Toronto, Canada, pp. 683-690

[13] B. T. R. Savarimuthu, and S. Cranefield, *A categorization of simulation works on norms.* A categorization of simulation works on norms. In Dagstuhl Seminar Proceedings 09121: Normative Multi-Agent Systems, 15-20 March, Leibniz, Germany, 2009, pp. 39-58.

[14] Henrique Lopes Cardoso, and Eugénio Oliveira, *Norm Defeasibility in an Institutional Normative Framework* In M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, editors, ECAI, volume 178 of Frontiers in Artificial Intelligence and Applications, pages 468.472. IOS Press, 2008.

[15] Domenico Corapi, Daniel Sykes, Katsumi Inoue, and Alessandra Russo, *Probabilistic Rule Learning in Nonmonotonic Domains.* In Joao Leite, Paolo Torroni, Thomas Agotnes, Guido Boella, and Leon van der Torre (eds.)