A SLDNF based Formalization for Updates and Abduction

Sai Kiran Lakkaraju

A thesis submitted for the degree of Master of Science (Honours)-Computing and Information Technology at University of Western Sydney

July 2001
PLEASE NOTE

The greatest amount of care has been taken while scanning this thesis,

and the best possible result has been obtained.
Except where otherwise indicated, this thesis is my own original work.

Sai Kiran Lakkaraju
18 July 2001

© Sai Kiran Lakkaraju

Typeset in Palatino by \TeX and \TEX\textsf{2e}.
To the all pervading force.
Abstract

Knowledge representation and inference have been the backbone of Artificial Intelligence. Storing knowledge with the goal for simulating human reasoning and/or behaviour has taken many forms. Logic programming is one of the most popular and widely used knowledge representation tools. Logic programming with deduction/induction/abduction as the reasoning technique is serving numerous fields of Artificial Intelligence. In dynamic domains where there is a constant change in knowledge, updating the knowledge base is a crucial point to keep the knowledge base in stable condition. The change in the knowledge is due to a simple fact/rule.

This thesis investigates the issues in updating the knowledge base. Two types of normal logic program based updates are considered, simple fact based updates where the knowledge base is updated by a simple fact, rule based updates where the knowledge base is updated by a rule. A SLDNF based procedural approach is proposed to implement such updates.

This thesis also investigates the issues involved in simple fact based abduction and rule based abduction. In this study it is observed that updates are closely related to abduction. A SLDNF based procedural approach to perform simple fact/rule based update and abduction is proposed as a result of this study.
Contents

Acknowledgements .......................................................... vii

Abstract ............................................................................. ix

1 Introduction ........................................................................ 1
  1.1 The Basis for This Work ................................................. 1
  1.2 Knowledge Representation ............................................ 2
    1.2.1 What Is Knowledge Representation ....................... 2
    1.2.2 Characteristics of Knowledge Representation .......... 4
    1.2.3 Knowledge Representation Tools ............................ 6
  1.3 Reasoning ...................................................................... 9
    1.3.1 Deduction .................................................................. 10
    1.3.2 Induction .................................................................. 10
    1.3.3 Abduction .................................................................. 11
  1.4 Updates ......................................................................... 12
    1.4.1 Formula Based Updates ......................................... 13
    1.4.2 Model Based Updates .............................................. 14
    1.4.3 Logic Program Updates .......................................... 16
  1.5 Structure of the Thesis .................................................. 17

2 Basic Notions ...................................................................... 19
  2.1 Introduction .................................................................... 19
2.2 Conventions .................................................. 19
2.3 Logic Programs ........................................... 20
2.4 Semantics .................................................... 23
  2.4.1 Stable Model Semantics ............................. 25
2.5 Linear Derivation .......................................... 25
2.6 Linear Resolution with Selection Function .......... 26
2.7 Closed World assumption(CWA) ..................... 30
2.8 Negation as Failure ....................................... 30
2.9 SLDNF ........................................................ 31

3 Rule Based Updates ...................................... 35
  3.1 Introduction .............................................. 35
  3.2 Definitions ................................................ 36
  3.3 The Approach ............................................. 37
    3.3.1 Insertion ............................................. 38
    3.3.2 Deletion ............................................. 40
  3.4 Algorithm ................................................ 43
  3.5 Correctness and Minimal Change .................... 49
  3.6 Summary .................................................. 53

4 Rule Based Abduction .................................. 55
  4.1 Introduction .............................................. 55
    4.1.1 The Notion of Abduction ......................... 56
    4.1.2 Abduction in Logic ................................. 59
  4.2 Definitions .............................................. 60
4.3 The Approach .............................................. 62
4.4 Algorithm ..................................................... 68
4.5 Summary ...................................................... 73

5 Update and Abduction ........................................ 75
  5.1 Introduction .................................................. 75
  5.2 Updates ....................................................... 75
  5.3 Abduction ..................................................... 78
  5.4 Why Abducibles? ............................................. 80
  5.5 Abduction Can Be Directly Used for Updates .......... 81
  5.6 The Difference ............................................... 82
  5.7 Summary ...................................................... 88

6 Conclusion ...................................................... 89
  6.1 Summary of The Thesis ..................................... 90
  6.2 Contributions ............................................... 92
    6.2.1 Rule Based Updates .................................... 92
    6.2.2 Rule Based Abduction .................................. 92
    6.2.3 Unified Approach ....................................... 92
  6.3 Future Work .................................................. 92

Bibliography .......................................................... 93
Chapter 1

Introduction

1.1 The Basis for This Work

The field of AI is an attempt to construct machines of their own mind. So basically AI is an *reverse engineering of psychology*. As an AI researcher one has to understand the process of mind, how mind thinks, plans, predicts and has to implement them. None of the present researchers, philosophers believe that intelligence depends on any thing immaterial or supernatural, thus they are all materialists in at least the minimal sense of supposing that matter, suitably selected and arranged, suffices for intelligence. According to the present science mind is nothing but matter in motion. In such case the one and only work of the mind is to think. Here thinking means storing knowledge, processing knowledge, or *knowledge representation* and *reasoning*. By the time of Aristotle we came to know about deduction and induction, as the reasoning techniques. Then C.S.Pierce introduced abduction, another form of reasoning. Computers program which is capable of representing knowledge and use an inference mechanism to answer the quarries can be considered as intelligent? Depending on the applications intelligent system designers are using one of the reasoning tool to develop the inference engine. Now a days the field of
AI is diverging very rapidly that each of these three methods were emerging as new branches, but we strongly feel that if one wants to design a machine with its own brain, the inference engine of that machine should be capable of applying these three modes of inference. In our way of defining such an inference engine, in this thesis we present a procedural approach for updates and abduction, an algorithm which actually can do Abduction and Updates.

In this chapter we briefly describe what is knowledge representation, discuss about some knowledge representation tools, we discuss about various inference mechanisms then non-monotonic reasoning, finally updates.

1.2 Knowledge Representation

1.2.1 What Is Knowledge Representation

Reasoning is a process that goes on internally, while most things it wishes to reason about exist only externally. A program (or person) engaged in planning the assembly of a car, for instance, may have to reason about entities like a wheel, steering, breaks, doors, engine, etc., yet such things exist only in the external world. It will need to know a large number of facts and to know about the processes that cause those facts to change. It will need to know about the objects, which those facts concern and the relationships in which those objects stand to one another. If it is intended to solve problems of some sort then it need to know about the problem and what counts as a solution to the problem, together with some strategies for solving such problems. If we consider the information about the external objects is knowledge, the first problem one
(program/person) encounters while reasoning is the way to represent knowledge. The study of knowledge representation concerns the ways in which we might go about encoding them in a computer program.

Knowledge functions as a proxy inside the reasoner, a stand-in for the things that exist in the world. Operations on and with representations substitute for operations on the real thing, i.e., substitute for direct interaction with the world. In this view reasoning itself is in part a surrogate for action in the world, when we can not or do not (yet) wants to take that action. Conversely, action can substitute for reasoning. The basic idea is that intelligence, whatever it involves, does at least involve knowing things, and exploiting them so as to respond appropriately to a given situation. Therefore it seems reasonable to suppose that if we wish to make an intelligent computer system we must have a way of getting it to know things, and that involves finding a way of representing the things we wish to know so that they can be encoded with in the computer system.

Definition 1 Knowledge Representation is a well defined set of syntactic and semantic conventions that represents the objects in the real world.

By syntax we mean that which specifies a set of rules for combining symbols so as to form valid expressions. By semantics we mean the specification of how such expressions are to be interpreted. They must be well defined with respect to the domain, the pictorial representation of a car is better than a technical report of a car for a expert vision system and a technical report of a car is most useful for a expert robot driver instead of a pictorial representation. For various domains in the real world various knowledge representation techniques
are developed.

1.2.2 Characteristics of Knowledge Representation

Often there will be a number of representations available for a single situation. We need to consider the criteria which we can use to assess the various possibilities. Some of these criteria will relate to things which the representation must have if it is to do what is requires of it, and others will relate to desirable aspects of the representation. The former group of criteria may be termed as Criteria of adequacy and if the representation fails in one of these respects, whatever virtues it may possess when considered on other grounds, will be of no avail, since we will not be able to represent the knowledge we need. The first three criteria of adequacy discussed here is from John McCarthy and Hayes paper "Some philosophical problems from the stand point of artificial intelligence".

- **Metaphysical adequacy** A Representation must be Metaphysical adequate. By this we mean that there is no contradiction between the facts we wish to represent and our representation of them. To use McCarthy's example, we could not produce an adequate representation of physical objects in the world if we tried to represent them as a collection of non-interacting particles. Since they do interact with one another there is a fundamental contradiction between represented and representation which renders the representation metaphysically inadequate. this criterion does not determine the representation we must use. At this level the representations are mainly useful for constructing general theories.
- **Epistemic adequacy** Since we need to derive observable consequences from the representations metaphysical adequacy is not enough, this introduces Epistemic adequacy, which says that the representation must provide us with the ability to express the facts that we wish to express.

- **Heuristic adequacy** This third criterion was somewhat tentatively proposed by Hayes and McCarthy, and was meant to suggest the need for the representation to be itself capable of expressing the reasoning that is gone through in solving a problem. This is the most difficult to fulfil, and it is not clear that any of the representations proposed till now satisfy it completely.

- **Computational tractability** The criteria discussed above would hold good for the representation of knowledge for any purpose. If the knowledge is to be incorporated in a computer system, we need to take account of Computational tractability. This requires that we are able to manipulate the representation efficiently with in a computer system.

- **Expressiveness** The above criteria represent conditions that a representation must fulfil if it is to be used at all. Expressiveness is one important consideration that can enable us to choose amongst different representations. Though it is comparative it can be evaluated using the following factors.
  
  - Lack of ambiguity.
  - Clarity.
  - Uniformity.
- Notational Convenience.
- Relevance.
- Declarativeness.

1.2.3 Knowledge Representation Tools

1. *Programming Languages*: A computer is of little use without programs to run on it, and the writing of programs requires a programming language. A mathematical theory - the theory of computability - shows that if a programming language has certain basic features it can perform any computation that can be carried out on any machine. Most of the computations that are required in modeling intelligent behaviors are non-numerical they require the manipulation of abstract symbols standing not for the numbers, but for such things as people, places, words and sentences, and the relation between them. A special class of programming languages has been developed that are particularly suitable for non-numeric computations. They are called list-processing languages. LISP, POP-11, PROLOG are some of the list processing languages.

2. *Predicate calculus (Logic)*: Predicate calculus is a logical system invented by German mathematician and logician Gottlob Frege (1879/1972). It was designed to formalize arguments in which the premises can contain more than one instance of the quantifiers *every* and *some*. There are many different, but equivalent, ways of writing down predicate calculus. In particular when predicate calculus is used, it must be modified to suit
the list notation of the programming language used. Predicate calculus is a language in which states of affairs can be described. Sentences, or well-formed formulae, of this language are made up from predicates, constant terms, Variables, logical connectives and quantifiers. Predicate calculus provides a uniform method for representing facts in a database that all facts are treated in the same way. Such representations favor general procedures for inferring new information. With in a large data base, a uniform proof procedure can be very inefficient. It has difficulty finding the required conclusion among the possible conclusions.

3. Semantic networks: One of the earliest methods used to represent the information needed in a high-level cognitive task was semantic networks. The networks were called semantic because they encoded information about meaning. When similar networks are used for other purposes they are referred to as associative networks. A semantic network is an example of the kind of structure that mathematicians called a (labeled)graph. It consists of a set of nodes connected by links. In the earliest networks the nodes stood for concepts - roughly - word meanings - and the links for relations between them. The most important links were class inclusion links and feature specification links. In subsequent research, networks were used to represent facts about the world conveyed in dialogues and text. Semantic networks provide a non-uniform way of representing information, because the relations represented by links are treated differently from other relations. Reasoning in semantic networks can be done by spreading activation.
4. *Frames, Scripts*: *Frame system theory* proposed by Minsky in 1975. Minsky proposed that there should be a different but related description of the object for each point of view. A *frame* is a knowledge structure that represents some aspect of the world, which might be, for example, an object, an event or a sequence of events. A frame can represent either a specific object or a type of object. A frame representing a specific object is derived from one representing the corresponding type of object by making a copy of it and filling in information that is 'missing' information that is true of the specific object, but not necessarily of the type of object.

One way that frames are organized is into an inheritance hierarchy of the kind found in semantic networks. Frames provide a declarative method of representing knowledge, but a system that uses frames usually has other information represented procedurally.

Schank and Abelson(1977) developed the concept of a *Script*, which is an application of Minsky's frame idea to one aspect of language understanding. A script encodes information about a stereotype sequence of events, such as a visit to a restaurant. When such sequences of events for the basis of a narrative, some of the events in the sequence may not be mentioned.

5. *Production systems*: Knowledge representation systems such as semantic nets, frames and scripts encode information in an essentially declarative form. However some types of knowledge are more naturally thought as sets of rules. A *production system* is a set of rules, called *productions*, they embodies knowledge about a particular domain and which can be
used for making inferences in that domain. In AI production systems are
also known as rule-based systems. A production states that if certain set
of conditions holds then a certain set of actions can be performed. The
terms actions and condition are interpreted broadly. Production systems
are modular. New productions, corresponding to new bits of knowledge
can be added to them without effecting the way of the production sys-
tem works. Productions may share conditions, so it may happen that,
when an action is performed, more than one production is applicable.
The most natural way of using productions to make inferences is forward
chaining. Forward chaining starts from a set of initial conditions and per-
forms the actions that they warrant. This step creates a new set of condi-
tions in which further productions are triggered. This process continues
until the required conclusion is reached. Productions can also be used
for backward chaining. In backward chaining the goal to be achieved or
the statement to be proved is taken as the starting point.
Production systems have been proposed as psychological models of hu-
man cognition.

1.3 Reasoning

Reasoning is the process that generates the knowledge. Through rigorous rea-
soning only one can generate more knowledge. Answering Why?, What? and
how? is the usual way a human gains knowledge. The three known reasoning
techniques are discussed below.
1.3.1 Deduction

*Deduction* or *Proof* is a form of reasoning which uses the principles of logic.

**Example 1** All the beans from this bag are white
These beans are from this bag.  
These beans are white ●

In the above example the first two sentences are *premises* and the third is a *deductive conclusion* from those premises.

In general if some knowledge and a set of rules are given then one may be able to infer some new knowledge, from the existing knowledge, using the rules. The new knowledge is the deductive consequence of the existing knowledge. Deductive consequences have the nice property that if the premises are true then the deductive consequences must also be true. Since these consequences are truth preserving one can be just as confident in the conclusions as the grounds for making them.

1.3.2 Induction

*Induction* in contrast is a method of reasoning that is not necessarily be truth preserving.

**Example 2** These beans are from this bag.  
These beans are white.  
All the beans from this bag are white ●

Inductive reasoning is essentially finding some generalization that describes a current set of observations. But inductive conclusions cannot be accepted with
certainty, simply on the basis of acceptance of the premises, since some future observations may conflict with the generalization.

1.3.3 Abduction

As a third style of reasoning, philosopher Pierce introduced Abduction. Pierce describes abduction as the "probational adoption of a hypothesis" as explanation for the results, according to known laws". It is however a weak kind of inference, because we cannot say that we believe in the truth of the explanation, but only that it may be true[2],

Example 3 All the beans from this bag are white
These beans are white.
These beans are from this bag.

It seems however that the idea of Abduction can tracked back to the 17Th. century in the logical writings of G.W.Laibniz, who was the first to notice the insufficiency of the traditional (Aristotelian) modes of reasoning, either purely deductive or purely inductive. He tried to define a new mechanism for hypothesis formation that would take from both induction and deduction.
In general the process of abduction is defined as given a set of logic program P, and a fact / rule G is observed, the process of abduction is to provide the explanation for the observation within the knowledge known to the logic program P. So the explanation can be either a fact or a rule. Explanations are often restricted to belong to a special pre specified domain specific class of sentences called abducibles.
Abduction is currently used and successful in various domains such as semiotics, linguistics.

1.4 Updates

Ordinary logical inference is monotonic in the sense that when a new formula is added to a knowledge base all the deductions permitted before the formula is added are still valid. Adding new formulas never decrease the set of proved theorems. But there are many examples of commonsense reasoning by humans that seem to be non-monotonic. We often make default inferences—ones that barring knowledge to the contrary, we are willing to assume to be true. But, if new, contradictory knowledge arrives, we then must retract the default inference. There are many systems and logical mechanisms that have been proposed for capturing this non-monotonic phenomenon.

In non-monotonic reasoning updates play a vital role as they are necessary to keep the knowledge base updated. When dealing with modifications to a knowledge base represented by a propositional theory, three kinds of abstract frame works have been distinguished

1. Formula Based Updates.

2. Model Based Updates.

3. Logic Program Updates.
1.4.1 Formula Based Updates

Ginsberg and Smith proposed a Possible Worlds Approach (PWA) for reasoning about actions [40; 41]. The knowledge base is realized as a set of logical formulas. This approach involves keeping a single description of the world that is changed when an action is performed. The basic idea to representing such a change is to construct the nearest world to the current one in which the consequences of the action hold. This approach represented that theory revision.

Formally suppose given a set $S$ of logical formulas, Which is treated as a partial world describing the current situation in the question, a set of protected sentences $P$, which is a subset of $S$ and describes domain constraints in the question. If one wishes to add some new facts, a set of formulas $C$ to the current world, then a possible world for $C$ in $S$ is defined to be any subset $T \subseteq S \cup C$ such that:

1. $C \subseteq T$,

2. $P \cap S \subseteq T$,

3. $T$ is consistent.

4. $T$ is maximal with respect to these constraints.

They argued that PWA provided the solution to the frame problem, ramification problem and qualification problem. The collection of all possible worlds for $C$ in $S$ is denoted as $W(C, S)$. Ginsberg and Smith also gave an algorithm to construct the possible worlds for a set $C$. 
1.4.2 Model Based Updates

Winslett [42] proved that the PWA failed to solve the frame problem and ramification problem if it is allowed to operate with incomplete information. Instead Winslett proposed a model based update mechanism called Possible Models Approach (PMA).

Let $T$ be the world description, Where some formulas of $T$ was treated as the protected formulas. The PMA considers the possible states of the world to be the models of $T$. To reason about the effect of performing an action with post condition $S$, the PMA considers the effect of the action on each possible state of the world, that is on each model of $T$. The PMA changes the truth valuations of the atoms in each model as little as necessary in order to make both $S$ and the protected formulas of $T$ true in that model. The possible states of the world after action is performed are all those models thus produced. In the PMA, models are simply considered to be subsets of herbrand base of the language. The formal definitions of the PMA are follows.

Models $M1$ and $M2$ differ on an atom $p$ if $p$ appears in exactly one of $M1$ and $M2$. Then the set of models produced by incorporating into $M$ is defined as following.

Let $M$ be a model of $T$ and $S$ a set of formulas. Incorporate($M, S$) is the set of all models $M'$ such that

1. $S$ and the protected formulas of $T$ re true in $M'$,

2. no other model $M''$ satisfying 1 is such that the set of differing atoms between $M'$ and $M$. 
The possible states of the world resulting from applying an action with post condition $S$ are given by $U_{\text{EModels}(T)} \text{ Incorporate}(M,S)$

The following simple example presented in [43] illustrates the difference between PMA and PWA.

**Example 4** Let $T = \{p \lor q\}$ be the description of the world representing that the door is open ($p$) or the window is open ($q$). Note that there is no protected formula in $T$.

Suppose an action makes the door closed ($\neg p$). According to PWA, since $\neg p$ is consistent with $p \lor q$, the minimal change on the world descriptions gives the resulting description of the world as $T' = \{\neg p, p \lor q\}$. From $T'$, we have that $T' \vdash q$. However, if the window is originally closed, why does the action of closing the door cause the window to open? It seems that the PWA presents an unreasonable result for the frame problem in this example.

Let us see what will happen if we use the PMA to this example. There are three possible states of $T$:

$$\{p, q\}, \{\neg p, q\} \text{ and } \{p, \neg q\}.$$ 

Hence, the corresponding possible resulting states after performing the action are:

$$\{\neg p, q\}, \{\neg p, q\} \text{ and } \{\neg p, \neg q\}.$$ 

Obviously, these possible states present a reasonable result: after closing the door the door becomes closed and the position of the window is unchanged.

Note that the PMA presented above cannot represent the performance information in reasoning about change. To overcome this disadvantage, Winslett extended the PMA to prioritized possible models approach (PPMA).
1.4.3 Logic Program Updates

It is a convention to express rule based knowledge systems as logic programs. The update of such logic programs take place either by adding or removing literals or rules. In this way the logic program updates can be classified into two types again *simple fact updates* where the knowledge base is updated by simple facts, *Rule based updates* where the knowledge base is updated by a rule or a logic program [45]. The first work on simple fact update was due to Marek and Truszcynski [44]. Generally they addressed the following problem: given a finite set of ground atoms as the knowledge base $\mathcal{P}$ and a set of ground atoms $\mathcal{r}$ as the updating rules. what is the resulting knowledge base after updating $\mathcal{P}$ with $\mathcal{r}$. For Example, given a knowledge base $\mathcal{P} = \{A, D\}$ and a set of update rules $r$:

\[
\begin{align*}
\text{in} (C) & \leftarrow \text{in} (A), \text{out} (B), \\
\text{out} (D) & \leftarrow \text{in} (C), \text{out} (B),
\end{align*}
\]

Where A,B,C,D are ground atoms, Then after updating $\mathcal{P}$ with $\mathcal{r}$, According to Marek and Truszcynski’s approach we would expect to have a resulting knowledge base $\mathcal{P}' = \{A, C\}$.

The main restriction posed in this model, the completeness of the knowledge base is released by Baral. In Marek and Truszcynski’s approach the knowledge base is represented by a finite set of ground literals. But in the real world situations, it would be more convenient and flexible to represent a knowledge base as a rule based system, which is a combination of rules and facts.
1.5 Structure of The Thesis

This thesis is organized as follows

* In chapter 2, for the sake of completeness and self containment, we briefly explain the language and some of the most common definitions used throughout this thesis and then we briefly explain what is SLDNF and the background of SLDNF.

* In chapter 3, we present a proof procedure based on SLDNF for updates.

* In chapter 4, we present a proof procedure for Abduction.

* In chapter 5, we discuss the relation between the updates and the abduction.

* In chapter 6, we conclude and provide hints on future developments.
Introduction
Chapter 2

Basic Notions

2.1 Introduction

We present the basic notation used throughout this thesis such that this thesis is complete and self-contained. Most of the definitions are widely in use, while some of them may be changed such that they suit the context. If not explicitly given here, definitions of logic programming terms can be found in foundations of logic programming by J.W.Lloyd [19].

2.2 Conventions

The following conventions are used to represent terms, clauses and so on.

- **Variables**: italicized lower case letters \(x, y, z, \ldots\).

- **Constants**: italicized lower case letters \(a, b, c, \ldots, c_1, c_2, \ldots\), or an italicized string starting with an upper case letter like \(Yan, Kiran \ldots\).

- **Predicates**: italicized upper case letters \(P, Q, R, \ldots\) or an italicized string starting with an upper case letter like \(Happy, Cs, \ldots\).

- **Atoms or Literals**: italicized upper case letters \(A, B, L, M, \ldots\).
- A symbol \( \leftarrow \), which is used to represent a rule.

- Connectives \( \text{not}, \leftarrow, \rightarrow, \lor, \land, \neg \), where \( \text{not} \) represents \textit{nagation as failure} (weak negation).

- Zapf chancery type face string starting with an upper case letter, \( \mathcal{P} \).

Lemmas, Theorems, Figures are numbered by chapters.

\section*{2.3 Logic Programs}

We start by giving the definition of well-formed formulas for the first order predicate logic.

- A variable is a \textit{term}.

- An individual constant is a \textit{term}.

- If \( f \) is a \( n \)-ary function symbol and \( t_1, \ldots, t_n \) are terms, then \( f(t_1, \ldots, t_n) \) is a term.

If \( P \) is an \( n \)-ary predicate symbol and \( t_1, \ldots, t_n \) are terms then \( P(t_1, \ldots, t_n) \) is an atomic formula or atom.

A \textit{literal} is an atom or a negated atom.

A well-formed formula is inductively defined as follows:

- An atomic formula is a well-formed formula.

- If \( F \) and \( G \) are formulas, then so are \( \text{not } F, \neg F, F \land G, F \lor G, F \rightarrow G \) and \( F \leftrightarrow G \).
- If \( F \) is a formula and \( x \) is a variable, then \( (\forall x F) \) and \( (\exists x F) \) are formulas.

The first order language \( L \) of the first order alphabet \( L \) comprises of the set of all formulae constructed from the symbols of the alphabet.

An expression is a finite sequence of such symbols.

Terms and Literals are the only well-formed expressions.

A program clause is a formula of the form

\[
A \leftarrow A_1, \ldots, A_m, \textit{not}A_{m+1}, \ldots, \textit{not}A_n,
\]

Where \( A, A_1, \ldots, A_m, \textit{not}A_{m+1}, \ldots, \textit{not}A_n \) are atoms. \( A \) is called the head, \( A_1, \ldots, A_n \) are called the body of the clause.

A clause with an empty body is called a unit clause.

If there is no part of \( \textit{not}A_{m+1}, \ldots, \textit{not}A_n \) in a clause, that clause is called definite or Horn clause.

A definite program is a set of definite clauses.

In a definite program, the set of all program clauses with the same predicate symbol \( P \) is called the definition of \( P \).

A definite goal is a clause of form \( \leftarrow B_1, \ldots, B_m \), where each \( B_i \) is called a sub goal.

There is an obvious association of a logical formula to a logic program \( \mathcal{P} \). One simply considers \( P \) to be a conjunction of the formulas

\[
\forall x : A \lor \neg L_1 \lor \cdots \lor \neg L_m
\]
corresponds to the clauses

\[
A \leftarrow L_1, \ldots, L_m
\]
A normal logic program\(^1\) is a finite set of rules of the form (2.1). In this thesis we mainly concentrate on normal logic programs.

In a logic program \( \mathcal{P} \), a rule without body \( A \leftarrow \) is called a fact. A rule \( r \) or program \( \mathcal{P} \) is ground if no variable occurs in it. It should be also noted that any free variable in a rule is assumed to be universally quantified.

A substitution \( \varphi \) is a mapping of variables over terms, denoted by \( \{x_1/t_1, \ldots, x_n/t_n\} \), where each \( x_i \) is a variable, each \( t_i \) is a term distinct from \( x_i \), and the term \( t_i \) is also called a binding for \( x_i \). \( \varphi \) is called a ground substitution if each \( t_i \) is a ground term. \( x_1, \ldots, x_n \) are called variables of substitution and \( t_1, \ldots, t_n \) are called terms of substitution.

Let \( \varphi = \{x_1/t_1, \ldots, x_n/t_n\} \) be a substitution and \( E \) be an expression. The application of \( \varphi \) to \( E \), denoted by \( E \varphi \), is the expression obtained by simultaneously replacing each occurrence of the variable \( x_i \) in \( E \) by the term \( t_i \). In this case \( E \varphi \) is called a instance of \( E \) by \( \varphi \). If \( E \varphi \) is ground then \( E \varphi \) is called a ground instance of \( E \). Also, \( E \) is referred to as generalization of \( E \varphi \).

A unifier of two expressions \( E \) and \( E' \) is a substitution \( \varphi \) such that \( E \varphi \) is syntactically identical to \( E' \varphi \). If the two atoms do not have a unifier then they are not unifiable. A unifier \( \varphi \) is called a most general unifier (mgu) for the two atoms if for each unifier \( \alpha \) there exists a substitution \( \beta \) such that \( \alpha = \varphi \beta \).

Given a program \( \mathcal{P} \), the Herbrand Universe of \( \mathcal{P} \), denoted as \( \mathcal{H} \mathcal{U}(\mathcal{P}) \), is defined to be the set of all ground terms that can be constructed from the function symbols occurring in program \( \mathcal{P} \). If \( \mathcal{P} \) does not contain any constant symbols, we will assume one in \( \mathcal{P} \) and then is included in \( \mathcal{H} \mathcal{U}(\mathcal{P}) \).

\(^1\) We also simply call it a logic program if there is no confusion in the context.
The Herbrand Base of a program \( \mathcal{P} \) is the set of \( \{ P(t_0 \cdots t_n) | P \text{ is an n-ary predicate symbol occurring in the program } \mathcal{P} \text{ and } t_0 \cdots t_n \in \mathcal{U}(P) \} \).

The ground version of a normal logic program \( \mathcal{P} \), denoted as \( G(\mathcal{P}) \), is the set (possibly infinite) of ground rules obtained from \( \mathcal{P} \) by substituting each variable with all possible elements of its Herbrand Universe \( \mathcal{H} \mathcal{U}(\mathcal{P}) \). A ground version of a predicate symbol \( P \) in a program \( \mathcal{P} \), denoted as \( G(\mathcal{P}, P) \), is the set of all rules in \( G(\mathcal{P}) \) which have predicate symbol \( P \) in their heads or simply the definition of \( P \).

### 2.4 Semantics

There are several ways to give a meaning to a logic program. Declarative semantics or model theoretic semantics of a program deals with the interpretation of a program including meaning, logical implications and truth. If there exists an interpretation \( I \) which evaluates a closed formula \( G \) to be True, then we say \( I \) satisfies \( G \) or \( I \) is a model of \( G \). The declarative semantics for a program can be defined by selecting one or more of its models. These models determine which substitution of a given goal is correct in the context of the program.

Fix point semantics defines the meaning of a program to be input/output relation which is minimal fixpoint of transformation \( T_p \) associated with the program \( P \). This operator builds the intended model of the program in a step-by-step process.

Van Emden and Kowalski investigated the semantics of Horn logic as a programming language and compared the declarative semantics of a program

\[ \]
with the classical model-theoretic semantics, operational semantics with classical proof-theoretic semantics and fixpoint semantics by fixpoint operator $T_p$ corresponding to a program $P$.

For the case of Definite clauses, it has been pointed out by Emden and Kowalski [9] that the interpretation of all Herbrand models is again a Herbrand model, which is the least one. The least Herbrand model equals to the least fixed point of a function $T_p$ introduced in [9] where $T_p$ maps subsets of the HB to subsets of HB. Procedural semantics (or operational semantics) of a program deals with well-formed formula and their syntax, axioms, rules of inference and proofs in the language of the program. This semantics defines the input/output relations computed by a program in terms of individual operations evoked by the program inside the machine. Procedural semantics refers to a computational method, called proof procedure, for obtaining the meaning of a program.

**Example 5** Consider a program $P$:

\[
\begin{align*}
\text{Math(Kiran)} & \leftarrow, \\
\text{Crazy}(x) & \leftarrow \text{Math}(x).
\end{align*}
\]

1 Declarative Semantics - A fact that is in the minimal model of $P$ is taken as a True.

2 Procedural Semantics - A fact, which can be derived from $P$, is taken as True.

Then the declarative and procedural semantics coincide and represented by the set \{ Math(Kiran), Crazy(Kiran) \}.
2.4.1 Stable Model Semantics

Stable model semantics is more general than the declarative or fixpoint semantics. Informally, when one assumes some set of default literals to be true, and all the others to be false, some consequences are followed according to the declarative or fixpoint semantics. If the consequences completely corroborate the hypotheses made, then they form a stable model.

Definition 2 Let \( P \) be a logic program, \( I \) be a model of \( P \) and \( I \subseteq HB(P) \). The set \( Stable_I P \) be obtained from \( ground(P) \) by following two steps:

- Delete each rule from \( ground(P) \) that has a negative literal of the form \( \neg A \) in its body and \( A \in I \), and

- Delete all negative literals in the bodies of the remaining rules.

2.5 Linear Derivation

Definition 3 Let \( S \) be a set of clauses and \( C \) be a member of \( S \). Then a linear derivation of \( C_n \) from \( S \) with top clause \( C \) is a finite sequence \( C_0 = C, C_1, \ldots, C_n \) of clauses such that for each \( i = 0, 1, \ldots, n - 1 \):

- \( C_{i+1} \) is a resolvent of \( C_i \) called a near parent and another clause \( B_i \) called a side clause and

- each \( B_i \) is

  - either in \( S \) and in that case \( B_i \) is the input parent of \( C_{i+1} \).

  - or an ancestor of each \( C_j (0 \leq j < i) \) and in that case \( B_i \) is the far parent of \( C_{i+1} \).
In the above definition of linear derivation, each clause \( C_{i+1} \) is said to have been obtained by linear resolution. A linear refutation is a linear derivation of \( \emptyset \). Linear derivation is constrained by the fact that a new clause is always derived from the preceding clause of the deduction by resolving against an earlier clause of the deduction. Linear resolution is complete.

For linear derivations from a set \( S \) of clauses with common clause \( C \), the whole search space is represented as a tree, called a **search tree**.

A **search tree** \( T \) is defined as follows

- The root of \( T \) is labeled by \( C \).
- Suppose \( N \) is a node in \( T \) and the nodes from the root to \( N \) are labeled by \( C_1, C_2, \ldots, C_n, C_{n+1} \) a node labeled by \( C_{n+1} \) is an immediate descendant of \( N \).

### 2.6 Linear Resolution with Selection Function

*Linear resolution with selection function (or SL-resolution)* (Kowalski and Kuehner, 1971) is a restricted form of linear resolution. The main restriction is affected by a selection function that chooses from each clause one single literal to be resolved upon in that clause. SL - resolution operates on **chains** rather than clauses and hence strictly is not a form of resolution. It does however employ ideas of unification and resolution.

Let \( G \) be a goal \( \leftarrow A_1, \ldots, A_m, \ldots, A_k \) and \( C \) be the clause \( A \leftarrow B_1, \ldots, B_q \). \( G' \) is derived from \( g \) and \( C \) using mgu \( \emptyset \) if the following holds.

(a) \( A_m \) is an atom, called the **selected atom**, in \( G \).
§2.6 Linear Resolution with Selection Function

(b) $\emptyset$ is a most general unifier of $A_m$ and $A$.

(c) $G'$ is the goal $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\emptyset$.

A SLD derivation of $G$ consists of a (finite or infinite) sequence $G_0 = G, G_1, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of variants of program clauses and a sequence of $\emptyset, \emptyset_1, \ldots$ of mgs such that each $G_{i+1}$ is derived from $G_i$ and $C_{i+1}$ using $\emptyset_{i+1}$. A single derivation step is called SLD-resolution.

A computation rule uniquely determines which atom is selected for every goal in the derivation.

A SLD derivation is infinite unless, for some goal $G_i$ in the derivation, there is no next goal. There are two cases. The SLD-derivation is successful if $G_i$ is empty. In this case it is called an SLD-refutation for the initial goal $G$. The second case occurs when the derivation is finitely failed. A derivation fails finitely if for some $G_i$ no head of a clause unifies with the selected atom of $G_i$.

A SLD-derivation is fair if every atom that appears in the derivation is chosen at some step.

SLD-resolution is a refinement of the resolution inference rule given by Robinson. An inference rule is called correct or sound if only valid formulas are inferred. It is called complete if all valid formulas can be inferred. Resolution is sound and complete for logical formulas in clausal form. Soundness of SLD-resolution is directly implied by soundness of resolution. While SLD-resolution is incomplete for clauses in general, it is complete for horn-clauses. SLD-refutations are linear proofs, while proofs are trees for resolution in general. So SLD-resolution is much more efficient, but negative information cannot be expressed in Horn-clauses.
The success set of a definite program $\mathcal{P}$ is the set of all $A \in HB_p$ such that the goal $G$ has an SLD-refutation.

A SLD-tree for a goal $G$ with a program $\mathcal{P}$ is the tree satisfying the following:

(a) Each node of the tree is a (possibly empty) definite goal.

(b) The root node is $G$.

(c) Let $A_1, \ldots, A_m, \ldots, A_k$ be a node in the tree and suppose that $A_m$ is the selected atom. Then for the each program clause $A \leftarrow B_1, \ldots, B_q$ such that $A_m$ and $A$ are unifiable with mgu $\varnothing$, the node has a child

$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\varnothing$.

(d) Nodes that are the empty clause have no children.

Each branch of the SLD tree is a derivation. Branches corresponding to successful derivations are called success branches, branches corresponding to infinite derivations are called infinite branches, and branches corresponding to failed derivations are called failure branches.

A leaf node of a SLD tree is either the empty goal or a goal that immediately fails since the selected atom is not unifiable with any program clause.

The example 6 given Lloyd explains the importance of the computational rule. The computational rule has a great influence on the size and structure of the SLD-tree.

**Example 6** consider the following program $\mathcal{P}$
§2.6 Linear Resolution with Selection Function

Let the goal $G$ be $\leftarrow P(X, b)$ Figure 2.1 shows two different SLD-trees for a single goal. The left tree is finite where as the right tree is infinite. Both of them however have two success branches corresponding to the answers $\{X/a\}$ and $\{X/b\}$.

A search rule is a strategy for searching SLD-trees to find success branches. A search rule is called fair if it guarantees that each success branch on the SLD-tree will eventually be found.

It is not possible to express negative information with horn clauses. They state what is true but not what is false. SLD-resolution applied to Horn clauses cannot deduce negative information. To be precise, let $P$ be a definite program.
and $A \in HB_p$. Then we cannot prove that not $A$ is a logical consequence of $P$. The reason is that $P \cup \{A\}$ is satisfiable having $HB_p$ as a model.

2.7 Closed World assumption (CWA)

Given a program $P$ and $A \in HB(P)$, the closed World Assumption states that the negation of $A$ may be inferred from $P$, if $A$ is not a logical consequence of $P$. The CWA is consistent for definite programs. This means that each ground atom $A \in HB(P)$ is either True or False.

In general, the Closed World Assumption states that all information that is not true in the program is considered as false.

The Closed World Assumption may not be consistent for the normal programs.

**Example 7** Suppose $P = \{A \leftarrow \neg B\}$. This can be rewritten as $A \lor B$. Neither $A$ nor $B$ is a logical consequence of $P$. Therefore CWA infers $\neg A$ as well as $\neg B$ and these two negated facts together with the $P$ are inconsistent.

2.8 Negation as Failure

First order logic is only semi-decidable. While it is possible to recursively enumerate all ground atoms $A$ that are implied by a given program $P$, there is no way which, for given $A$ and $P$, decides in a finite number of steps whether or not $A$ is a logical sequence of $P$. If $A$ is not a logical consequence, a proof attempt may loop forever. Therefore in practice a weaker rule has to be applied.

The **Negation as Failure** rule says that not $A$ can be inferred that if $A$ is in the finite failure set of $P$. The finite failure set of $P$ is the set of all $A \in HB(P)$ for
which there exists a finitely failed SLD-tree for \( \mathcal{P} \cup \{ \leftarrow A \} \). If \( \mathcal{P} \cup \{ \leftarrow A \} \) has a finitely failed SLD-tree then \( A \) is not a logical consequence of \( \mathcal{P} \) and hence not \( A \) can be inferred from \( \mathcal{P} \).

## 2.9 SLDNF

SLDNF resolution is essentially the SLD resolution augmented with negation as failure (NAF) rule.

Let \( \mathcal{P} \) be a program and \( G \) be a goal. A SLDNF-derivation of \( \mathcal{P} \cup \{ G \} \) consists of a sequence \( G_0 = G, G_1, \cdots \) of goals, a sequence \( C_1, C_2, \cdots \) of variants of clauses of \( \mathcal{P} \) or negative literals, and a sequence \( \phi_1, \phi_2, \cdots \) of substitutions satisfying the following:

1. If the selected literal from \( G_i \) is an atom then \( G_{i+1} \) is derived from \( G_i \) and \( C_{i+1} \) using \( \phi_{i+1} \).

2. If \( G_i \) is \( \leftarrow L_1, \cdots, L_m \) and the selected literal \( L_n \) in \( G_i \) is a ground negative literal \( \text{not} \ A \) and there is a finitely failed SLDNF tree for \( \mathcal{P} \cup \{ \leftarrow A \} \), then \( \phi_{i+1} \) is the identity substitution, \( C_{i+1} \) is \( \text{not} \ A \) and \( G_{i+1} \) is \( \leftarrow L_1, \cdots, L_{n-1}, L_{n+1} \cdots, L_m \).

3. If the sequence \( G_0, G_1, \cdots \) is finite, then either
   - the last goal is empty, or
   - the last goal is \( \leftarrow L_1, \cdots, L_m \) and the selected literal \( L_n \) is positive and there is no program clause whose head unifies with \( L_n \) or
the last goal is \( \leftarrow L_1, \ldots, L_m \) and the selected literal \( L_m \) is a ground negative literal \( \text{not} A \) and there is an SLDNF refutation of \( \mathcal{P} \cup \{ \leftarrow A \} \).

A SLDNF derivation is finite if it consists of a finite sequence of goals, otherwise it is infinite. A SLDNF derivation is successful if it is finite and the last goal is the empty goal. A successful SLDNF derivation is indeed a SLDNF-refutation. An SLDNF derivation is failed if it is finite and the last goal is not the empty goal. Let \( \mathcal{P} \) be a program and \( G \) be a goal. A SLDNF tree for \( \mathcal{P} \cup \{ G \} \) is a tree satisfying the following:

1. Each node of the tree is a goal or an empty clause.

2. the root node is \( G \).

3. Let \( \leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m \ (m \geq 1) \) be a non-leaf node in the tree and suppose that a positive literal \( L_k \) is selected. Then this node has a descendant for each clause \( A \) in \( \mathcal{P} \) such that \( L_m \) and \( A \) are unifiable where \( \emptyset \) is an mgu of \( L_m \) and \( A \).

4. Let \( \leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m \ (m \geq 1) \) be a non-leaf node in the tree and suppose that the selected literal \( L_m \) is a ground negative literal of the form \( \text{not} A \) and there is a finitely failed SLDNF-tree for \( \mathcal{P} \cup \{ \leftarrow A \} \). Then the single descendant of the node is \( \leftarrow (L_1, \ldots, L_{i-1}, L_{i+1} \ldots \land L_m) \)

5. Let \( \leftarrow L_1, \ldots, L_m \ (m \geq 1) \) be a leaf node in the tree and suppose that the literal \( L_m \) is selected, the n either

(a) \( L_m \) is positive and there is no clause in \( \mathcal{P} \) whose head unifies with \( L_m \), or
(b) $L_m$ is a ground negative literal not A and there is an SLDNF refutation of $\mathcal{P} \cup \{ \leftarrow A \}$.

(c) A node which is the empty clause has no descendants.

In a SLDNF-tree, a branch which terminates at an empty goal is a success branch, a branch which does not terminate is an infinite branch and a branch that terminates at a non-empty goal is a failure branch. An SLDNF tree for which every branch is failure branch is indeed a finitely failed SLDNF tree. Each branch of an SLDNF-tree corresponds to an SLDNF derivation. When a positive literal in an SLDNF resolution is selected, SLD resolution is used to derive a new goal. When a ground negative literal is selected a recursive process is established to apply NAF rule.

**Example 8** Consider a program $\mathcal{P}$:

\[
\begin{align*}
\text{Math(Kiran)} & \leftarrow, \\
\text{Crazy}(x) & \leftarrow \text{Sci}(x), \text{notCs}(x), \\
\text{Happy}(x) & \leftarrow \text{Crazy}(x), \text{Math}(x), \\
\text{Crazy}(x) & \leftarrow \text{Phi}(x). 
\end{align*}
\]

The SLDNF tree for $\mathcal{P} \cup \{ \leftarrow \text{Crazy}(Kiran) \}$ presented in Fig (2.2) shows and the goal G be $\{ \leftarrow \text{Crazy}(Kiran) \}$. ■
\[ \text{Figure 2.2: SLDNF tree for } \mathcal{P} \cup \{\leftarrow \text{Crazy}(Kiran)\} \]
3.1 Introduction

In many situations knowledge based systems are represented as logic programs comprising facts and rules. Facts express the current state of the knowledge base and are usually changeable, while rules express constraints about the domain and hence are usually unchangeable. However in dynamic domains, a knowledge base has to be updated not only because some new facts occur, but also because some old rules are no longer suitable to characterize new domain constraints. Therefore, it is important to have a more powerful update mechanism to maintain a knowledge base to a stable state whenever some information, which may be a fact or a rule, is combined into the current knowledge base. Recent research on update has shown that rule based update provides some new insights for knowledge base updates in more flexible domains [5; 10; 11; 25].

We consider the problem of updating a knowledge base, where the knowledge base is realized as a (normal) logic program, which represents both facts and rules, with a fact or a rule. First we consider the problem of deleting a fact from a logic program and inserting a fact into a logic program. In this general set-
ting, the update problem is as follows, given a logic program and a fact or rule, which is (not) a logical consequence of the program, find a way to change the program so that the formula is no longer (is) a logical consequence of the updated program. Then we consider the problem of updating a knowledge base with a rule. Basically, the rule-based update addresses the following problem: given an initial knowledge base and an update rule, how to update the initial knowledge base such that whenever the body of the update rule is achieved, in the initial knowledge base, the head of the update rule is then achieved in the resulting knowledge base. We propose a SLDNF based procedure to implement this kind of update. We also prove a correctness theorem for our approach and investigate semantic properties to characterize the underlying update formalization.

Our general approach to the update problem consists of generating all choices for each update. By having all of the choice, we expect that at least one of them will give an update program which satisfies the constraints (if there any exists).

### 3.2 Definitions

An **update rule** is a rule with one of the following two forms:

\[ Q \leftarrow P_1, \ldots, P_m, \text{not} P_{m+1}, \ldots, \text{not} P_n, \]  

\[ \text{not} Q \leftarrow P_1, \ldots, P_m, \text{not} P_{m+1}, \ldots, \text{not} P_n. \]
Rule (3.1) is called the inserting update rule, while rule (3.2) is called the deleting update rule. Note that the inserting update rule has the same form of the rule in a knowledge base, and the deleting update rule allows the negation as failure sign not to be occurred in the head of the rule. In our formalism, an update is a transformation on knowledge base. Informally, if a knowledge base \( \mathcal{P} \) is updated by an inserting update rule (3.1), it means that if \( P_1, \ldots, P_m \) can be proved from \( \mathcal{P} \), and \( P_{m+1}, \ldots, P_n \) cannot be proved from \( \mathcal{P} \), then the resulting knowledge base \( \mathcal{P}' \) should be able to prove \( Q \). Furthermore, we also require that \( P_1, \ldots, P_m \) can be proved from \( \mathcal{P}' \) and \( P_{m+1}, \ldots, P_n \) cannot be proved from \( \mathcal{P}' \). When we say a atom \( P \) can be proved from a knowledge base \( \mathcal{P} \), denoted as \( \mathcal{P} \vdash P \), we mean that \( P \) can be derived from \( \mathcal{P} \) through a SLDNF resolution [19].

On the other hand, if a knowledge base \( \mathcal{P} \) is updated by a deleting update rule (3.1), it means that if \( P_1, \ldots, P_m \) can be proved from \( \mathcal{P} \), and \( P_{m+1}, \ldots, P_n \) cannot be proved from \( \mathcal{P} \), then the resulting knowledge base \( \mathcal{P}' \) should not be able to prove \( Q \), while \( P_1, \ldots, P_m \) still can be proved from \( \mathcal{P}' \) and \( P_{m+1}, \ldots, P_n \) cannot be proved from \( \mathcal{P}' \).

### 3.3 The Approach

We introduce the basic idea of our update approach by illustrating several examples in this section. Either insertion or deletion may cause an update.

Given a knowledge base \( \mathcal{P} \) (i.e. a normal logic program) and an update rule \( r \) (3.1) or (3.2), our idea of updating \( \mathcal{P} \) with the rule \( r \) can be described by the following steps. First, by using the SLDNF resolution, we try to prove that
\( P_1, \ldots, P_m \) are derivable from \( \mathcal{P} \), and \( P_{m+1}, \ldots, P_n \) are not derivable from \( \mathcal{P} \) based on the principle of \textit{negation as finite failure} [19]. If some of \( P_i \) (1 \( \leq \) \( i \leq m \)) cannot be derived from \( \mathcal{P} \), or some of \( P_j \) (\( m + 1 \leq j \leq n \)) can be derived from \( \mathcal{P} \), then the update process stops and no any change will be made on \( \mathcal{P} \).

If the first step succeeds, we go to the next step of the update. If the update rule \( r \) is an inserting rule (3.1), we add some rule into \( \mathcal{P} \) or delete some rule from \( \mathcal{P} \) to form \( \mathcal{P}' \) such that \( Q \) can be derived from \( \mathcal{P}' \) through a SLDNF proof.

If \( r \) is a deleting update rule (3.2) with head \textit{not} \( Q \), on the other hand, we need to find all possible SLDNF proofs of \( Q \) from \( \mathcal{P} \), then for each of such proofs, we remove some rule from \( \mathcal{P} \) or add some rule into \( \mathcal{P} \) such that \( Q \) cannot be derived from the revised program. We check for the derivability of the body of the rule in the revised program. After this, the resulting knowledge base \( \mathcal{P}' \) is formed.

### 3.3.1 Insertion

To illustrate the above procedure clearly, let us consider the following example.

**Example 9** Consider a knowledge base \( \mathcal{P}_1 \) consisting of following facts and rules:

\[
\begin{align*}
\mathit{Math}(\mathit{Kiran}) & \leftarrow, \\
\mathit{Crazy}(x) & \leftarrow \mathit{Sci}(x), \mathit{not} \mathit{Cs}(x), \\
\mathit{Happy}(x) & \leftarrow \mathit{Crazy}(x), \mathit{Math}(x), \\
\mathit{Crazy}(x) & \leftarrow \mathit{Phi}(x).
\end{align*}
\]
§3.3 The Approach

\[ \text{Math(Kiran)} \leftarrow \]

\[ \text{Figure 3.1: SLDNF tree for } \mathcal{P}_1 \cup \{\text{Math(Kiran)}\} \]

Intuitively, the above knowledge base presents the following facts and knowledge: Kiran is a mathematician; if someone is a scientist but not evidence to show that he/she is a computer scientist, then he/she is crazy; if someone is crazy and is a mathematician, he/she is then happy; if someone is a philosopher, then he/she is crazy.

Now suppose a request is to update this knowledge base by the following update rule:

\[ \text{Sci(Kiran)} \leftarrow \text{Math(Kiran)}, \]

which means that if Kiran is a mathematician, then it should be presented that Kiran is also a scientist.

Clearly this is an inserting update rule. According to the update process described, we should first try to prove Math(Kiran) from \( \mathcal{P}_1 \). Figure (9) shows that Math(Kiran) is derivable from \( \mathcal{P}_1 \). The next step, obviously, is to add some rule into \( \mathcal{P}_1 \) such that fact Sci(Kiran) can be derived from the new formed knowledge base through a SLDNF tree. For this particular example, is it quite reasonable from our common sense that we can simply add rule \( \text{Sci(Kiran)} \leftarrow \) into \( \mathcal{P}_1 \) to form the new knowledge base \( \mathcal{P}_2 = \mathcal{P}_1 \cup \{\text{Sci(Kiran)} \leftarrow \} \)

The above example shows how an inserting update rule can be handled through the SLDNF resolution. It should be noted, however, it may not be the only case
to add some rule into the initial knowledge base in order to achieve the head of an inserting update rule. For instance, if we have a knowledge base $\mathcal{P}$ consisting of following rules:

\[
\begin{align*}
A & \leftarrow, \\
C & \leftarrow B, \\
C & \leftarrow \text{not} D, \\
D & \leftarrow.
\end{align*}
\]

We want to update this knowledge base with an inserting update rule $C \leftarrow A, \text{not} E$. Ignoring the detail, it is easy to see that the body of this update rule is achieved by the knowledge base. To achieve the head $C$, however, there are three possible ways: add facts $C \leftarrow$ or $B \leftarrow$ into $\mathcal{P}$, or remove fact $D \leftarrow$ from $\mathcal{P}$. From a proof theoretic viewpoint, we consider a SLDNF tree for $\mathcal{P} \cup \{ \leftarrow C \}$ (3.2). If there exists a successful branch, then it means that $C$ is already derivable from $\mathcal{P}$ and no any change on $\mathcal{P}$ is needed. Otherwise, for every finitely failed branch, we make it succeed by adding/removing minimal number of facts into/from $\mathcal{P}$. Therefore, in the revised SLDNF tree, each successful branch presents a possible way to achieve the resulting knowledge base. This is shown by the following SLDNF tree.

### 3.3.2 Deletion

Now we describe our idea of handling deletion.

**Example 10** Example (9) continued. Suppose that after complete the update presented in example (9), we obtain a knowledge base $\mathcal{P}_2$ as follows:
\section*{The Approach}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{tree.png}
\caption{Revised finitely failed SLDNF tree for $\mathcal{P} \cup \{\leftarrow C\}$}
\end{figure}

\begin{equation}
\begin{align*}
\text{Sci}(Kiran) & \leftarrow, \\
\text{Math}(Kiran) & \leftarrow, \\
\text{Crazy}(x) & \leftarrow \text{Sci}(x), \neg \text{Cs}(x), \\
\text{Happy}(x) & \leftarrow \text{Crazy}(x), \text{Math}(x), \\
\text{Crazy}(x) & \leftarrow \Phi(x).
\end{align*}
\end{equation}

Consider an update of $\mathcal{P}_2$ with a deleting update rule
\begin{equation}
\text{notHappy}(x) \leftarrow \text{Crazy}(x), \neg \text{Cs}(x). 
\end{equation}

Intuitively, this deleting update rule expresses that for any $c \in \mathcal{U}(\mathcal{P}_2)$, if $\mathcal{P}_2 \vdash \text{Crazy}(c)$ and $\mathcal{P}_2 \not\vdash \text{Cs}(c)$, then the updated knowledge base $\mathcal{P}_2' \vdash \text{Happy}(c)$. Since the only element in $\mathcal{U}(\mathcal{P}_2)$ is Kiran, this deleting update rule actually has one ground form

\begin{equation}
\text{notHappy}(Kiran) \leftarrow \text{Crazy}(Kiran), \neg \text{Cs}(Kiran).
\end{equation}

According to our previous discussion, we should first prove $\text{Crazy}(Kiran)$ from $\mathcal{P}_2$. The SLDNF tree for $\mathcal{P}_2 \cup \{\leftarrow \text{Crazy}(Kiran)\}$ presented in Figure (3.3) shows $\mathcal{P}_2 \vdash \text{Crazy}(Kiran)$ Now we need to show $\mathcal{P} \not\vdash \text{Cs}(Kiran)$. To show this, we only need to construct a finitely failed SLDNF tree for $\mathcal{P}_2 \cup \{\leftarrow \text{Cs}(Kiran)\}$, i.e. (3.4). So far, we have shown that the body of update rule is achieved by $\mathcal{P}_2$. The next step is to delete $\text{Happy}(Kiran)$ from $\mathcal{P}_2$ to obtain the result-
Rule Based Updates

\[ \text{\texttt{<- Crazy(Kiran)}} \]
\[ \text{\texttt{<- Sci(Kiran), not Cs(Kiran)}} \]
\[ \text{\texttt{<- Phil(Kiran)}} \]

\[ \text{\texttt{<- not Cs(Kiran)}} \]
\[ \text{\texttt{Failed}} \]
\[ \text{\texttt{(<- Cs(Kiran) finitely failed)}} \]

**Figure 3.3:** SLDNF tree for \( \mathcal{P}_2 \cup \{\text{<- Crazy(Kiran)}\} \)

\[ \text{\texttt{<- Cs(Kiran)}} \]
\[ \text{\texttt{Failed}} \]

**Figure 3.4:** Finitely failed SLDNF tree for \( \mathcal{P}_2 \cup \{\text{<- Cs(Kiran)}\} \)

...ing knowledge base \( \mathcal{P}'_2 \). Similarly as discussed earlier, to achieve this purpose, we construct a SLDNF tree for \( \mathcal{P}_2 \cup \{\text{<- Happy(Kiran)}\} \). Then for each successful branch in the SLDNF tree, we make it fail by adding/removing minimal number of rules into/from \( \mathcal{P}_2 \). Clearly, from the revised SLDNF tree shown in (3.5), there are also three possible ways to achieve \( \mathcal{P}'_2 \): remove rule \( \text{Happy(x) \leftarrow Crazy(x), Math(x)} \) from \( \mathcal{P}_2 \), or rule \( \text{Crazy(x) \leftarrow Sci(x), notCs(x)} \) from \( \mathcal{P}_2 \), or remove fact \( \text{Math(Kiran) \leftarrow} \) from \( \mathcal{P}_2 \). It should be noted that we cannot remove fact \( \text{Sci(Kiran) \leftarrow} \) from \( \mathcal{P}_2 \) or add fact \( \text{Cs(Kiran) \leftarrow} \) into \( \mathcal{P}_2 \) in order to make node "\( \text{<- Sci(Kiran), notCs(Kiran), Math(Kiran)} \)" or node "\( \text{<- notCs(Kiran), Math(Kiran)} \)" in (3.3) fail because this will cause the body of update rule (3.3) is not achievable in the resulting knowledge base \( \mathcal{P}'_2 \).
3.4 Algorithm

Based on the ideas presented previously, in this section we describe the formal procedures for our rule based update. As it has been described in section 3.3, our approach is based on the SLDNF resolution. In the SLDNF resolution proof, the negation is proved from a logic program by finite failure. However, it is well known that it is possible that a SLDNF tree may include infinite branch. In this case, no result is able to be proved from the SLDNF resolution proof. For instance, given a logic program \( \mathcal{P} = \{ A \leftarrow \mathit{notB}, B \leftarrow \mathit{notA} \} \), no finite SLDNF tree exists for \( \mathcal{P} \cup \{ \leftarrow A \} \). To avoid this problem, in our update context, we assume that during the update, each SLDNF tree is finite in the sense that each branch in the SLDNF tree is finite.

To simplify our following description, we also introduce some useful notions about SLDNF trees. Observing a SLDNF tree described previously, e.g. fig-
ure 3.3, if we use $n_0, \cdots, n_i, \cdots$ to denote all nodes in a SLDNF tree, a SLDNF tree can be actually presented by a set of branches $n_0 \rightarrow n_1 \rightarrow \cdots, \cdots, n_0 \rightarrow n_k \rightarrow \cdots$. Each branch in the SLDNF tree starts from the root node $n_0$. Consider the SLDNF tree in 3.3, it is quite clear that it can be described as two branches $n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow \Diamond$ and $n_0 \rightarrow n_3 \rightarrow$ Failed, where nodes $n_0, n_1, n_2, n_3$ are $\leftarrow$ Crazy(Kiran), $\leftarrow$ Sci(Kiran), $\neg$notCs(Kiran), $\leftarrow$ notCs(Kiran) and $\leftarrow$ Phi(Kiran) respectively. Node $\Diamond$ indicates the end of a successful branch, and Failed indicates the end of a finitely failed branch. We also call a segment of a branch starting from the root node a path in a SLDNF tree.

Algorithm 1. Checking($\mathcal{P}, r$)

**Function:** Checking the body of an update rule $r$ in logic program $\mathcal{P}$.

**Input:** A logic program (knowledge base) $\mathcal{P}$ and an update rule $r$ (3.1) or (3.2), where the body of $r$ consists of $P_1, \cdots, P_m, \neg P_{m+1}, \cdots, \neg P_n$.

**Output:** A Boolean value True or False.

**Begin**

For each $i$ $(1 \leq i \leq m)$, $T_i$ = a finite SLDNF tree for $\mathcal{P} \cup \{ \leftarrow P_i \}$;

If there exists some $T_i$ where there is no successful branch,

then return Checking($\mathcal{P}, r$) = False;

For each $j$ $(m+1 \leq j \leq n)$, $T_j$ = a finite SLDNF tree for $\mathcal{P} \cup \{ \leftarrow P_j \}$;

If there exists some $T_j$ where there is no finitely failed branch,

then return Checking($\mathcal{P}, r$) = False;

Return Checking($\mathcal{P}, r$) = True;

**End.**
The function of algorithm Checking\((\mathcal{P}, r)\) is to check if the body of the update rule \(r\) is achieved by \(\mathcal{P}\). This is necessary when we perform the rule based update on \(\mathcal{P}\) with \(r\). It should be noted that in our update context, we do not restrict an update rule to be ground. For a non-ground update rule, as we mentioned earlier, we need to consider its all ground forms which are obtained by substituting variables in the update rule with elements of program universe \(\mathcal{U}(\mathcal{P})\). This consideration will be also adopted in our following update algorithms.

Algorithm 2. InsertUpdate\((\mathcal{P}, r)\)

Function: Update \(\mathcal{P}\) with an inserting update \(r\).

Input: A logic program \(\mathcal{P}\) and an inserting update rule \(r\) with form (3.1).

Output: A logic program \(\mathcal{P}'\).

Begin

If Checking\((\mathcal{P}, r)\) = False, then \(\mathcal{P}' = \mathcal{P}\), and return \(\mathcal{P}'\);

Let \(\mathcal{P}' = \mathcal{P}\) and \(Path = \emptyset\);

Loop

\(T = \) a finite SLDNF tree for \(\mathcal{P}' \cup \{ \leftarrow \mathcal{Q}\}\);

If there is a successful branch in \(T\) and Checking\((\mathcal{P}', r)\) = True,

then return \(\mathcal{P}'\);

If there is a successful branch in \(T\) and Checking\((\mathcal{P}', r)\) = False,

then \(\mathcal{P}' = \mathcal{P}\) and \(Path = \emptyset\);

If all branches in a SLDNF tree for \(\mathcal{P} \cup \{ \leftarrow Q\}\) have been selected,

then return \(\mathcal{P}' = \mathcal{P}\);
Select a finitely failed branch $B$ in $T$ such that $Path \subseteq B$:

$n_0 \rightarrow \cdots \rightarrow n_i \rightarrow \text{Failed};$

(1) If $n_i$ has form $\leftarrow P_{i1}, [\text{not}]P_{i2}, \cdots$,

then $\mathcal{T}' = \mathcal{T}' \cup \{P_{i1} \leftarrow \}$ and

$Path = Path \rightarrow n_{i+1}$, where $n_{i+1}$ is $\leftarrow [\text{not}]P_{i2}, \cdots$;

(2) If $n_i$ has form $\leftarrow \text{not}P_{i1}, [\text{not}]P_{i2}, \cdots,$

$\mathcal{T}' = \text{DeleteUpdate}(\mathcal{T}', \text{not}P_{i1} \leftarrow);$  

EndLoop

End.

Before we perform the update on knowledge base $\mathcal{T}$, we need to check if the body of the update rule $r$ has been achieved. Our update on $\mathcal{T}$ starts from the SLDNF tree for $\mathcal{T} \cup \{\leftarrow Q\}$ where $Q$ is the head of $r$. If there is a successful branch in the SLDNF tree, then the process stops. Otherwise, we need to add or remove some rules from $\mathcal{T}$ so that the goal $\leftarrow Q$ can be achieved. It is worth to note that if a sub goal $\leftarrow \text{not}P$ fails in some branch of the SLDNF tree, we need to call algorithm $\text{DeleteUpdate}(\mathcal{T}', \text{not}P \leftarrow)$, which will be described below, in order to achieve the sub goal $\leftarrow \text{not}P$. Finally, after change $\mathcal{T}$ to $\mathcal{T}'$ such that $\mathcal{T}' \vdash Q$, we need one more step to check if the body of $r$ is still achieved from $\mathcal{T}'$. This is important since under some situations, to achieve the head of $r$ may lead to a contradiction in the resulting knowledge base. For instance, updating a knowledge base with an inserting update rule like $Q \leftarrow \text{not}Q$ will never succeed. In this case, no change will occur in the initial knowledge base.

\footnote{Notation $[\text{not}]$ indicates that the negation as failure sign $\text{not}$ may or may not occur.}
Algorithm 3. DeleteUpdate\( (\mathcal{P}, r) \)

Function: Update \( \mathcal{P} \) with a deleting update \( r \).

Input: A logic program \( \mathcal{P} \) and a deleting update rule \( r \) with form (3.2).

Output: A logic program \( \mathcal{P}' \).

Begin

If \( \text{Checking}(\mathcal{P}, r) = \text{False} \), then \( \mathcal{P}' = \mathcal{P} \), and return \( \mathcal{P}' \);

Let \( \mathcal{P}' = \mathcal{P} \);

\( T = \) a finite SLDNF tree for \( \mathcal{P}' \cup \{ \leftarrow Q \} \);

Loop

If there is no successful branch in \( T \) and \( \text{Checking}(\mathcal{P}', r) = \text{True} \),

then return \( \mathcal{P}' \);

If there is no successful branch in \( T \) and \( \text{Checking}(\mathcal{P}', r) = \text{False} \),

then return \( \mathcal{P}' = \mathcal{P} \);

Let \( \mathcal{P}' = \mathcal{P} \) and \( T = \) a finite SLDNF tree for \( \mathcal{P}' \cup \{ \leftarrow Q \} \);

For each successful branch \( B \) in \( T \),

1. If there exist two nodes \( n_i \) and \( n_{i+1} \) in \( B \) such that \( n_i \rightarrow n_{i+1} \),

   where \( n_i \) and \( n_{i+1} \) have forms \( \leftarrow P_{i1}, [\neg]P_{i2}, \cdots \) and \( \leftarrow [\neg]P_{i2}, \cdots \)

   respectively, then \( \mathcal{P}' = \mathcal{P}' - \{ P_{i1} \leftarrow \} \);

2. If there exist two nodes \( n_i \) and \( n_{i+1} \) in \( B \) such that \( n_i \rightarrow n_{i+1} \),

   where \( n_i \) and \( n_{i+1} \) have forms \( \leftarrow \neg P_{i1}, [\neg]P_{i2}, \cdots \) and \( \leftarrow [\neg]P_{i2}, \cdots \)

   respectively, then \( \mathcal{P}' = \mathcal{P}' \cup \{ P_{i1} \leftarrow \} \);

3. If there exist two nodes \( n_i \) and \( n_{i+1} \) in \( B \) such that \( n_i \rightarrow n_{i+1} \),

   where \( n_i \) and \( n_{i+1} \) have forms \( \leftarrow P_{i1}, [\neg]P_{i2}, \cdots \) and

   \( \leftarrow [\neg]P_{i11}, \cdots, [\neg]P_{i1q}, [\neg]P_{i2}, \cdots \)
respectively in which a rule \( r: P_1 \leftarrow [not]P_{i1}, \ldots, [not]P_{i1q} \)

is used to derive node \( n_{i+1} \), then \( \mathcal{P}' = \mathcal{P} - \{ r \} \);

\( T = \) a finite SLDNF tree for \( \mathcal{P}' \{ \leftarrow Q \} \);

EndLoop

End.

Example 11 Consider a knowledge base \( \mathcal{P} \) consisting of following facts and rules:

\[
A \leftarrow B, \\
A \leftarrow notB, \\
B \leftarrow C,
\]

and an inserting update rule \( r_1: C \leftarrow A \). Suppose we update \( \mathcal{P} \) with \( r_1 \). Firstly, by using algorithm \textbf{Checking}(\( \mathcal{P}, r \)), it is clear that \( A \) is derivable from \( \mathcal{P} \). So \( \textbf{Checking}(\( \mathcal{P}, r \) ) = \text{true} \). Then by algorithm \textbf{InsertUpdate}(\( \mathcal{P}, r_1 \)), we simply obtain a unique resulting knowledge base \( \mathcal{P}' = \mathcal{P} \cup \{ C \leftarrow \} \). Note that the body \( A \) in \( r_1 \) can still be proved from \( \mathcal{P}' \). Now we consider to update \( \mathcal{P}' \) with a deleting update rule \( r_2: notA \leftarrow \). Consider the following SLDNF tree for \( \mathcal{P}' \cup \{ \leftarrow A \} \) as shown in 11. To make the successful branch fail, there are several options according to our algorithm. Firstly, if we choice node "\( \leftarrow C \)" as \( n_i \) and "\( \leftarrow \)" as \( n_{i+1} \), then to make this branch fail, we may remove fact \( C \leftarrow \) from \( \mathcal{P}' \) according to the case (1) in Algorithm 3. However, this does not achieve our purpose that \( \mathcal{P}'' \not\vdash A \), where \( \mathcal{P}'' = \mathcal{P}' - \{ C \leftarrow \} \). This is clear that even if rule \( A \leftarrow B \) cannot used to derive \( A \) in \( \mathcal{P}'' \), \( A \) still can be proved from \( \mathcal{P}'' \) through rule \( A \leftarrow notB \). According to Algorithm 3, we have to consider other options then. The only way available for us is to choice node "\( \leftarrow A \)" as node \( n_i \) and
node \( \leftarrow B \) as node \( n_i+1 \). Then from the case (3) in the algorithm, we need to remove rule \( A \leftarrow B \) from \( \mathcal{P}' \). Therefore, after updating \( \mathcal{P}' \) with the deleting update rule \( r_2 \), we obtain the resulting knowledge base \( \mathcal{P}'' \):

\[
A \leftarrow \text{not} B,
B \leftarrow C,
C \leftarrow.
\]

This example also shows that it is not always the situation of adding or removing simple facts from the initial knowledge base in order to achieve the update.

\[\blacksquare\]

3.5 Correctness and Minimal Change

We have proposed a procedural approach to implement rule based update. Now a natural question is: in which sense can we justify that our approach is correct? It is clear that the question can only be answered based on some semantics of the logic program. Here we use Gelfond and Lifschitz's stable
model semantics [8] to characterize our rule based update approach. Let \( \mathcal{P} \) a normal logic program not containing negation as failure sign not and G the
program base of \( \mathcal{P} \), i.e. G is the set of all ground atoms in the language of \( \mathcal{P} \). A
stable model of \( \mathcal{P} \), denoted as \( S_{\text{model}}(\mathcal{P}) \), is the smallest subset of G such that for
each rule \( A_0 \leftarrow A_1, \ldots, A_m \) in \( \mathcal{P} \), if \( A_1, \ldots, A_m \in S_{\text{model}}(\mathcal{P}) \), then \( A_0 \in S_{\text{model}}(\mathcal{P}) \).

Now we consider \( \mathcal{P} \) to be a normal logic program. For any subset \( G' \) of G, let
\( \mathcal{P}^{G'} \) be the logic program obtained from \( \mathcal{P} \) by deleting (i) each rule that has a
formula not \( A \) in its body with \( A \in G' \); and (ii) all formulas of the form not \( A \)
in the bodies of the remaining rules in \( \mathcal{P} \). Then we define a stable model of \( \mathcal{P} \),
denoted as \( S_{\text{model}}(\mathcal{P}) \), iff \( G' \) is a stable model of program \( \mathcal{P}^{G'} \), i.e. \( S_{\text{model}}(\mathcal{P}) = G' \).

Given a logic program \( \mathcal{P} \) and a ground atom \( A \), we say that \( A \) is entailed by \( \mathcal{P} \)
under stable model semantics, denoted as \( \mathcal{P} \models_{\text{s}} A \), if \( A \) is in every stable model
of \( \mathcal{P} \). In the case that \( A \) includes variables, \( \mathcal{P} \models_{\text{s}} A \) means that every ground
instance of \( A \) is in each stable model of \( \mathcal{P} \). The relationship between the stable
model semantics and SLDNF proof is revealed by the following lemma.

**Lemma 1** Let \( \mathcal{P} \) be a normal logic program and \( A \) an atom. Then \( \mathcal{P} \models A \) implies
\( \mathcal{P} \models_{\text{s}} A \).

**Proof:** Since \( \mathcal{P} \models A \), it follows that \( \mathcal{P} \cup \{ \leftarrow A \} \) has a finite SLDNF tree which has
a successful branch. To simplify our presentation, we consider the proposition
program case, while the general case can be easily extended.

Assume \( A \leftarrow B_1, \ldots, B_m, \text{not} B_{m+1}, \ldots, \text{not} B_n \) is a rule in \( \mathcal{P} \) which \( A \) is derived.

Then the SLDNF tree of \( \mathcal{P} \cup \{ \leftarrow A \} \) must have a branch as showed in the figure 3.7. The SLDNF tree showed in 3.7 actually shows that \( B_1, \ldots, B_m \) can be
derived from $\mathcal{P}$ and $B_{m+1}, \ldots, B_n$ cannot be derived from $\mathcal{P}$ finitely. From the soundness of SLDNF it follows that for any interpretation in which each rule of $\mathcal{P}$ is satisfied, $A$ is satisfied. On the other hand in each stable model of $\mathcal{P}$ each rule of $\mathcal{P}$ is satisfied. Therefore $A$ is also satisfied in each stable model of $\mathcal{P}$ that is $\mathcal{P} \models_s A$.

The result can be directly followed from the construction of finite SLDNF tree and from the property of stable model semantics. Note that the other way does not hold due to the way of handling negation in SLDNG as finitely failed while in stable model semantics any thing failed to be proved (not only finitely) is viewed as its negation. Now our correctness result for our SLDNF rule based update approach is described as follows.

**Theorem 1 (Correctness)** Given a knowledge base $\mathcal{P}$ and an update rule $r$ with a form (3.1) or (3.2). Let $\mathcal{P}'$ be a resulting knowledge base after updating $\mathcal{P}$ with $r$ using
procedures described in section 3.4. Suppose that \( P' \neq P \). Then the following results hold.

1. \( P \models_s P_1, \ldots, P \models_s P_m, P \not\models_s P_{m+1}, \ldots, P \not\models_s P_n; \) and

2. \( P' \models_s P_1, \ldots, P' \models_s P_m, P' \not\models_s P_{m+1}, \ldots, P' \not\models_s P_n, \) and \( P' \models_s Q \) if \( r \) is an inserting update rule, while \( P' \not\models_s Q \) if \( r \) is a deleting update rule.

Proof: The proof of the above theorem is based on Lemma 1 and algorithms presented in section 3.4. Firstly from algorithm 2 (insert update) and algorithm3(delete update) and the condition \( P' \neq P \), it follows that \( P \models P_1, \ldots, P \models P_m, P \not\models P_{m+1}, \ldots, P \not\models P_n; \) and

\( P' \models P_1, \ldots, P' \models P_m, P' \not\models P_{m+1}, \ldots, P' \not\models P_n, \) from lemma 1 that is \( P \models P_1, \ldots, P \models P_m, P \not\models P_{m+1}, \ldots, P \not\models P_n; \) and \( P' \models P_1, \ldots, P' \models P_m, P' \not\models P_{m+1}, \ldots, P' \not\models P_n. \)

Secondly, algorithm 2 will ensure \( P' \models Q \) if \( r \) is an inserting update rule and algorithm 3 will ensure \( P' \not\models Q \) if \( r \) is a deleting update rules. Again from lemma 1, that means \( P' \not\models Q \) respectively.

Now we consider the minimal change property of our update approach. The following theorem shows that our update satisfies a minimal change in terms of the set inclusion criterion on knowledge base. (Minimal Change Property)

Let \( P \) be a knowledge base and \( r \) an update rule with form (3.1) or (3.2), and \( P' \) a resulting knowledge base after updating \( P \) with \( r \) using procedures described in section 4. Then there does not exist another knowledge base \( P^* \) such that (i) \( P^* \models P_1, \ldots, P^* \models P_m, P^* \not\models P_{m+1}, \ldots, P^* \not\models P_n; \) (ii) \( P^* \models Q \) if \( r \) is an inserting update rule (3.1), or \( P^* \not\models Q \) if \( r \) is a deleting update rule (3.2); and (iii) \( P^* \subseteq P' \).

\(^2\)This ensures that some change on \( P \) is made in terms of this update.
3.6 Summary

In this chapter, we proposed a new approach to deal with rule based update. We also investigated the correctness and minimal change properties for our update formulation. Compared with previous methods, the major advantage of this approach is that our update is SLDNF procedure based and hence it is possible to be implemented with reasonable computational cost. Our update formalism is also more expressive than other methods because in our formalism a knowledge base is represented by a normal logic program while it was usually presented as a finite set of ground atoms/literals in our methods, e.g. [5; 10; 25].
Rule Based Updates
4.1 Introduction

Charles Sanders Peirce introduced the concept of abduction as a mode of reasoning. The fundamental question Peirce addressed was how synthetical reasoning is possible. Very much influenced by the philosophy of Immanuel Kant, Peirce’s aim was to extend his categories and correct his logic: “According to Kant, the central question of philosophy is ‘How are synthetical judgements a priori possible?’ But antecedently to this comes the question how synthetical judgements in general, and still more generally, how synthetical reasoning is possible at all. When the answer to the general problem has been obtained, the particular one will be comparatively simple. This is the lock upon the door of philosophy.” (CP, 5.348, quoted in Hookway (1992), page 18). Peirce proposes abduction to be the logic for synthetic reasoning, a method to acquire new ideas. That is, an ampliative mode of inference. In contrast, deduction is analytic, an explicative mode of inference, since the information in the conclusion is already suggested in the premises. Induction is also considered by Peirce to be an ampliative mode of inference, though not of a purely synthetic kind (CP, 5.170). The notions of logical inference and of validity that
Peirce puts forward however, go beyond our present understanding of what logic is about.

4.1.1 The Notion of Abduction

C.S. Peirce was the first philosopher to give to abduction a logical form. However, his notion of abduction is a difficult one to unravel. On the one hand, it is entangled with many other aspects of his philosophy, and on the other hand, several different conceptions of abduction evolved in his thought. We will point out a few general aspects of his theory of inquiry, and later concentrate on some of its more logical aspects. The development of a logic of inquiry occupied Peirce’s thought since the beginning of his work. In the early years he thought of a logic composed of three modes of reasoning: deduction, induction and hypothesis(2), each of which corresponds to a syllogistic form, illustrated by the following, often quoted example (CP 2.623):

Example 12 Deduction

     Rule : All the beans from this bag are white
     Case : These beans are from this bag.
     Result : These beans are white.

Induction

     Case : These beans are from this bag.
     Result : These beans are white.
     Rule : All the beans from this bag are white

Abduction

     Rule : All the beans from this bag are white
Result: These beans are white.

Case: These beans are from this bag.

Of these, deduction is the only reasoning which is completely certain, inferring its 'Result' as a necessary conclusion. Induction produces a 'Rule' validated only in the 'long run' (CP 5.170), and hypothesis, merely suggests that something may be 'the Case' (CP 5.171). Later on, Peirce proposed these types of reasoning as the stages composing a method for logical inquiry, of which hypothesis, now called abduction, is the beginning:

"From its [abductive] suggestion deduction can draw a prediction which can be tested by induction." (CP 5.171)

The notion of abduction is then enriched by the more general conception of: "the process of forming an explanatory hypothesis" (CP 5.171) and the syllogistic form is replaced by the following often-quoted logical formulation (CP 5.189)

The surprising fact, C, is observed.

But if A were true, C would be a matter of course.

Hence, there is reason to suspect that A is true.

For Peirce, three aspects determine whether a hypothesis is promising: it must be explanatory, testable, and economic. A hypothesis is an explanation if it accounts for the facts, according to the formulation above. Its status is that of a suggestion until it is verified, which explains the need for the testability criterion. Finally, the motivation for the economic criterion is twofold: a response to the practical problem of having innumerable explanatory hypotheses to test, as well as the need for a criterion to select the best explanation amongst the
testable ones. For Peirce, abductive reasoning is essential for every human inquiry. It plays a role in perception: "The abductive suggestion comes to us as a flash" (CP 5.181) As well as in the general process of invention: "It [abduction] is the only logical operation which introduces any new ideas" (CP 5.171).

In all this, abduction is both "an act of insight and an inference" as has been claimed by Anderson (Anderson 1986), who suggests a double aspect of abduction, an intuitive and a rational one.

The notion of abduction in Peirce has puzzled scholars all along. Some have concluded that Peirce held no coherent view on abduction at all (Frankfurt 1958), others have tried to give a joint account with induction (Reilly 1970) and still others claim it is a form of inverted modus ponens (Anderson 1986). A more modern view is found in (Kapitan 1990) who interprets Peirce's abduction as a form of heuristics. An account that tries to make sense of the two extremes of abduction, both as a guessing instinct and as a rational activity is found in (Ayim 1974). We leave here the reconstruction of Peirce's notion of abduction. F. Fann gives a nice concise account of the development of abduction in Peirce, which clearly distinguishes three stages in the evolution of his thought (Fann 1970). Another key reference on Peirce's abduction, in its relation to creativity in art and science is found in (Anderson 1987).
4.1.2 Abduction in Logic

Research on abductive reasoning in AI dates back to 1973 (Pople 1973), but it is until fairly recently that it has attracted great interest. Peirce’s abductive formulation has been the point of departure of recent studies on abductive reasoning, such as logic programming (Kakas et al. 1995), knowledge acquisition (Kakas 90) and natural language processing (Hobbs et al. 1990).

Given a logic program $\mathcal{P}$, and a observation $\mathcal{G}$, the abductive task can be characterized as the problem of finding a set of sentences $\mathcal{E}$ such that

1. $\mathcal{P} \cup \mathcal{E} \in \mathcal{G}$.
2. $\mathcal{P} \cup \mathcal{E}$ is consistent.

The above mentioned two conditions are too weak to capture Peirce’s notation. Moreover, we don’t want to explain one effect in terms of another effect, but only in terms of some cause so we need to restrict $\mathcal{E}$. For this reason explanations are restricted to domain specific class of sentences called abducibles. Cox and Pietrzykowski [46] identified the other desirable properties of the explanations. For instance, the explanation should be basic. In general the process of abduction is defined as given a set of logic program $\mathcal{P}$, and a fact or rule $\mathcal{G}$ is observed, the process of abduction is to provide the explanation for the observation within the knowledge known to the logic program $\mathcal{P}$. The explanation can be either a fact or a rule. Abduction is currently used and successful in various domains such as semiotics, linguistics and artificial intelligence. Abduction plays a vital role in commonsense reasoning,
knowledge representation, and database update. In this chapter we discuss a SLDNF formalism to find the explanations for the observations where the observations can be facts or rules. We mainly concentrate on such situations where observations are rules. When an observation is a rule, we propose that the body part of the rule must be explained first to form a new knowledge base which intern explains the head of the rule. Then the union of the explanations gives the complete explanation for the observation (the rule).

4.2 Definitions

Let us recall some concepts and definitions of logic programs from chapter 2. We extend those definitions here such that they are applicable to abductive logic programs. A rule is of form

$$A_0 \leftarrow A_1, \cdots, A_m, \text{not} B_{m+1}, \cdots, \text{not} B_n,$$

(4.1)

where $A_0, \cdots, A_m, B_{m+1}, \cdots, B_n$ are atoms of language $L$. A normal logic program is a finite set of rules of form (4.1). A rule of form $\leftarrow A_1, \cdots, A_m, \text{not} A_{m+1}, \cdots, \text{not} A_n$ is called a normal goal. In a logic program $P$, a rule without body $A \leftarrow$ is called a fact. A term, atom, literal, rule or program is ground if no variable occurs in it. It should be also noted that any free variable in a rule is assumed to be universally quantified. If $P$ does not contain any constant symbols, we will assume one in $P$.

Now we define a normal abductive logic program to be a pair $< P, A >$, where $P$
§4.2 Definitions

is a normal logic program and $\mathcal{A}$ is the set of abducibles. An abducible is either a fact or a rule. Let $< \mathcal{P}, \mathcal{A} >$ be a normal abductive logic program and $\mathcal{G}$ is a goal (observation). Note that $\mathcal{G}$ can be a literal or a rule. We first consider the case that $\mathcal{G}$ is a literal. If $\mathcal{G}$ is a ground atom (we also call $\mathcal{G}$ is a positive observation), then proper hypothesis is introduced to or removed from the current knowledge base to explain the observation by showing that $\mathcal{G}$ can be derived from the resulting knowledge base. If $\mathcal{G}$ is a negative ground atom (we also call $\mathcal{G}$ is a negative observation), then proper hypothesis is introduced to or removed from the current knowledge base to explain the observation by showing that the corresponding ground atom of $\mathcal{G}$ cannot be derived from the resulting knowledge base.

**Definition 4** A pair $(E, F)$ is an explanation of a positive observation (or negative observation, resp.) $\mathcal{G}$ with respect to an abductive logic program $< \mathcal{P}, \mathcal{A} >$ if

1. $(\mathcal{P} \cup E) - F \vdash \mathcal{G}$, (or $(\mathcal{P} \cup E) - F \not\vdash \overline{\mathcal{G}}$, resp.);

2. $(\mathcal{P} \cup E) - F$ is consistent; and

3. $E \subseteq \mathcal{A}$ and $F \subseteq \mathcal{A} \cap \mathcal{P}$.

**Definition 5** An explanation $(E, F)$ of an observation $\mathcal{G}$ is minimal if for any explanation $(E', F')$ of $\mathcal{G}$, $E' \subseteq E$ and $F' \subseteq F$ imply $E' = E$ and $F' = F$.

---

\(^1\)Here we use $\overline{\mathcal{G}}$ to denote the complementary literal of $\mathcal{G}$. For instance, if $\mathcal{G}$ is not $F$, then $\overline{\mathcal{G}}$ is $\overline{F}$. 

4.3 The Approach

In this section we describe the basic idea of our approach of abductive reasoning by illustrating several examples.

Example 13 Let \( \mathcal{P}, \mathcal{A} \) be an abductive program such that

\[
\begin{align*}
\mathcal{P}: \\
&\text{Bird(tweety)} \leftarrow, \\
&\text{Bird(opus)} \leftarrow, \\
&\text{Broken-Wing(tweety)} \leftarrow, \\
&\text{Ab(x)} \leftarrow \text{Broken-Wing(x)}, \\
&\text{Flies(x)} \leftarrow \text{Bird(x)}, \text{notAb(x)}, \\
\mathcal{A}: \\
&\text{Broken-Wing(x)} \leftarrow.
\end{align*}
\]

Intuitively the above knowledge base presents the following facts and knowledge: tweety, opus are birds, tweety has got a broken wing, if some one has got a broken wing he is abnormal, if some one is a bird and not abnormal then only he can fly. The set of abducibles consists of broken wing \( x \) means all the observations over this program must be explained in terms of broken wing only.

In this situation some one observed that tweety is flying and he wants to explain that fact. Then the explanation for the positive observation \( \mathcal{G} = \text{Flies(tweety)} \) can be derived from the SLDNF tree showed in figure 4.1. which says that if you delete the fact that tweety has got a broken wing, the knowledge base can explain the fact observed. That is, the observation \( \mathcal{G} = \text{Flies(tweety)} \) has an ex-
planning $\langle \mathcal{E}, \mathcal{F} \rangle = (\emptyset, \{\text{Broken-Wing(tweety)} \leftarrow \})$. On the other hand, an explanation $\langle \mathcal{E}, \mathcal{F} \rangle$ for the negative observation $G = \text{not Flies(opus)}$ is $\langle \{\text{Broken-Wing(opus)} \leftarrow \}, \emptyset \rangle$ and can be derived from the SLDNF tree showed in figure 4.2. Note that to explain the fact $\text{not Flies(opus)}$ we have to add the fact $\text{Broken-wing(opus)} \leftarrow$ to the logic program $\mathcal{P}$.

Now we consider the case that the observation is a rule. An observation rule is a rule of the form

$$A_0 \leftarrow A_1, \ldots, A_m, \text{not } B_{m+1}, \ldots, \text{not } B_n,$$

(4.2)
or of the form

\[ \text{not}A_0 \leftarrow A_1, \cdots, A_m, \text{not}B_{m+1}, \cdots, \text{not}B_n. \] (4.3)

The former is called a positive observation rule, while the latter is called a negative observation rule.

**Definition 6** Given an abductive program \( \langle P, A \rangle \), a pair \((E, F)\) is an explanation of the observation rule \(G\) if there exist \(E_1, \cdots, E_n\) and \(F_1, \cdots, F_n\) such that

1. \((P \cup E_1) - F_1 \models A_1, \cdots, (P \cup E_m) - F_m \models A_m, (P_{m+1} \cup E_{m+1}) - F_{m+1} \not\models B_{m+1}, \cdots, (P \cup E_n) - F_n \not\models B_n;\)

2. \((P \cup E) - F \models A_0\) (or \(P \cup E) - F \not\models A_0\) if the head of \(G\) is \(\text{not}A_0\), where \(E_1 \cup \cdots \cup E_n \subseteq E\), and \(F_1 \cup \cdots \cup F_n \subseteq F;\)

3. \(E \cap F = \emptyset\) and \((P \cup E) - F\) is consistent;

4. \(E \subseteq A\) and \(F \subseteq A \cap P.\)

\((E, F)\) is a minimal explanation for \(G\) as there does not exist another explanation \((E', F')\) of \(G\) such that \(E' \subset E\) and \(F' \subset F.\)

Now let us see how we can use the SLDNF resolution proof to achieve abductive reasoning in which an observation is a rule.

**Example 14** Consider an abductive program \(\langle P, A \rangle\) where

\(P:\)

\[ A \leftarrow \]

\[ C \leftarrow F \]
\( \mathcal{A} : \)

\( D \leftarrow C, B \)

\( F \leftarrow \).

Consider the observation \( G : \)

\( D \leftarrow C, \neg E. \)

\( G \) is a positive observation. According to our definition, we need to find \( E_1, E_2 \) and \( \mathcal{F}_1, \mathcal{F}_2 \) such that \((\mathcal{P} \cup E_1) - \mathcal{F}_1 \vdash C, (\mathcal{P} \cup E_2) - \mathcal{F}_2 \not\vdash E \), and \((\mathcal{P} \cup E_1 \cup E_2) - (\mathcal{F}_1 \cup \mathcal{F}_2) \vdash D \). Use the Negation as finitely failure rule of Clark [26], which states that \( \neg Q \) is a consequence of a program \( \mathcal{P} \) if a finitely failed SLD tree for the query \( Q \) w.r.t. \( \mathcal{P} \) exists. The detailed discussion is referred to [16]. Now we have the following revised SLDNF trees for deriving \( C \) and \( \neg E \) respectively. According to our previous discussion, from figure 4.3 we can see that a minimal explanation for \( C \) is \( (\{\mathcal{F} \leftarrow\},\{\emptyset\}) \). Since \( \mathcal{P} \not\vdash E \), the minimal explanation for \( \neg E \) is \( (\{\emptyset\},\{\emptyset\}) \). From figure 4.4 we can see that to achieve \( (\mathcal{P} \cup E) - \mathcal{F} \not\vdash D \), there are three possible explanations:

1. \( (\mathcal{E}, \mathcal{F}) = (\{\emptyset\}, \{D \leftarrow C, B\}) \),

2. \( (\mathcal{E}, \mathcal{F}) = (\{\emptyset\}, \{F \leftarrow\}) \),
3. \((E, F) = (\{\emptyset\}, \{B \leftarrow\})\).

The possible minimal explanations for the negative observation \(notD \leftarrow C, notE\) are

1. \((E, F) = (\{F \leftarrow\}, \{D \leftarrow C, B\})\),

2. \((E, F) = (\{F \leftarrow\}, \{F \leftarrow\})\),

3. \((E, F) = (\{F \leftarrow\}, \{B \leftarrow\})\).

We consider 1 and 3 as the final minimal explanations for \(G\).

Example 15 Consider an abductive program \(<P, A>\) where

\(P:\)

\(Sci(Kiran) \leftarrow,\)

\(Math(Kiran) \leftarrow,\)

\(Crazy(x) \leftarrow Sci(x), notCs(x),\)

\(Happy(x) \leftarrow Crazy(x), Math(x),\)

\(Crazy(x) \leftarrow Phi(x).\)
§4.3 The Approach

\[ \text{Crazy}(kiran) \]
\[ \text{Sci}(kiran), \text{not Cs}(kiran), \text{Phi}(kiran) \]
\[ \circ \]
\[ \text{failed} \]
\[ \text{Cs}(kiran) \]

\begin{center}
Figure 4.5: The SLDNF tree for \( \mathcal{P} \cup \{ \leftarrow \text{Crazy}(Kiran) \} \).
\end{center}

\[ \mathcal{A}: \]

\[ \text{Happy}(x) \leftarrow \text{Crazy}(x), \text{Math}(x), \]

\[ \text{Phi}(Kiran) \leftarrow. \]

Consider the observation \( \mathcal{G} \):

\[ \text{notHappy}(kiran) \leftarrow \text{Crazy}(kiran), \text{notCs}(kiran). \]

According to our definition, we need to find \( \mathcal{E}_1, \mathcal{E}_2 \) and \( \mathcal{F}_1, \mathcal{F}_2 \) such that \( \mathcal{P} \cup \mathcal{E}_1 \vdash \text{Crazy}(Kiran), (\mathcal{P} \cup \mathcal{E}_2) \vdash \text{Cs}(Kiran), \) and \( (\mathcal{P} \cup \mathcal{E}_1 \cup \mathcal{E}_2) \vdash (\mathcal{F}_1 \cup \mathcal{F}_2) \vdash \text{Happy}(Kiran). \) Note any finitely failed SLDNF tree of \( \mathcal{P} \cup \{ \leftarrow Q \} \) implies that \( \text{not} Q \) is derived from \( \mathcal{P} \). The detailed discussion is referred to in [16].

Now we have the following revised SLDNF trees for deriving \( \text{Crazy}(Kiran) \) and \( \text{notCs}(Kiran) \) respectively. According to our previous discussion, from figure 4.5 we can see that a minimal explanation for \( \text{Crazy}(Kiran) \) is \( (\emptyset, \{\emptyset\}) \). Since \( P \not\vdash \text{CS}(Kiran) \), the minimal explanation for \( \text{notCs}(Kiran) \) is \( (\emptyset, \{\emptyset\}) \). From figure 4.6 we can see that to achieve \( \mathcal{P} \cup \mathcal{E} \vdash \mathcal{F} \not\vdash \text{Happy}(Kiran) \), there are four possible explanations:

1. \( (\mathcal{E}, \mathcal{F}) = (\emptyset, \{\text{Happy}(Kiran) \leftarrow \text{Crazy}(Kiran), \text{Math}(Kiran)\}) \),

2. \( (\mathcal{E}, \mathcal{F}) = (\emptyset, \{\text{Sci}(Kiran) \leftarrow \}) \),
3. \((\mathcal{E}, \mathcal{F}) = (\{\emptyset\}, \{\text{Math(kiran)} \leftarrow \})\),

4. \((\mathcal{E}, \mathcal{F}) = (\{\text{Cs(Kiran)} \leftarrow \}, \{\emptyset\})\).

However, explanations 2 and 3 are not satisfying our definition \(\mathcal{F} \subseteq \mathcal{A} \cap \mathcal{P}\),
exploration 4 is not satisfying our definition that \(\mathcal{E} \subseteq \mathcal{A}\). Also, as explanations for both Crazy(Kiran) and notCs(kiran) is \((\{\emptyset\}, \{\emptyset\})\), we can conclude that explanation 1 is the only and final minimal explanation for \(\mathcal{G} \). 

4.4 Algorithm

Based on the ideas presented previously, in this section we describe the formal procedures for our rule based abduction. As it has been described in section 4.3, our approach is based on the SLDNF resolution. In the SLDNF resolution proof, the negation is proved from a logic program by finite failure. However, it is well known that it is possible that a SLDNF tree may include infinite branch. In this case, no result can be proved from the SLDNF resolution proof. For instance, given a logic program \(\mathcal{P} = \{A \leftarrow notB, B \leftarrow notA\}\), no finite SLDNF tree exists for \(\mathcal{P} \cup \{\leftarrow A\}\). To avoid this problem, in this context, we assume that during the abduction process, each SLDNF tree is finite in the sense that each
branch in the SLDNF tree is finite. To simplify our following description, we also introduce some useful notions about SLDNF trees. Observing a SLDNF tree described previously, e.g. Fig.4.5, if we use \(n_0, \ldots, n_i, \ldots\) to denote all nodes in a SLDNF tree, a SLDNF tree can be actually presented by a set of branches \(n_0 \rightarrow n_1 \rightarrow \ldots, \ldots, n_0 \rightarrow n_k \rightarrow \ldots\). Each branch in the SLDNF tree starts from the root node \(n_0\). Consider the SLDNF tree in Fig.4.5, it is quite clear that it can be described as two branches \(n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow \diamond\) and \(n_0 \rightarrow n_3 \rightarrow\) Failed, where nodes \(n_0, n_1, n_2, n_3\) are \(\leftarrow\) Crazy(Kiran), \(\leftarrow\) Sci(Kiran), \(\leftarrow\) notCs(Kiran), \(\leftarrow\) notCs(Kiran) and \(\leftarrow\) Phi(Kiran) respectively. Node \(\diamond\) indicates the end of a successful branch, and Failed indicates the end of a finitely failed branch. We also call a segment of a branch starting from the root node a path in a SLDNF tree.

Algorithm 1. Main(<\(\mathcal{P}, \mathcal{A}\), \(\mathcal{G}\))

Function: Explain the observation \(\mathcal{G}\) in logic program <\(\mathcal{P}, \mathcal{A}\)>

Input: A logic program (knowledge base) <\(\mathcal{P}, \mathcal{A}\)> and an observation \(\mathcal{G}\), where the body of \(\mathcal{G}\) consists of \(P_1, \ldots, P_m, notP_{m+1}, \ldots, notP_n\). \(Q\) is the head of the observation

Output: <\(\mathcal{E}, \mathcal{F}\), \(\mathcal{P}'\)>

Begin

\[\mathcal{E} = \emptyset; \mathcal{F} = \emptyset;\]

For each \(i (1 \leq i \leq m)\) Do Insert Hypothesis (<\(\mathcal{P}, \mathcal{A}\),\(P_i\));

For each \(i (m + 1 \leq i \leq n)\) Do Delete Hypothesis (<\(\mathcal{P}, \mathcal{A}\),\(P_i\));

If the observation is a positive observation

Then Insert Hypothesis(<\(\mathcal{P}, \mathcal{A}\),\(Q\));
If the observation is a negative observation

Then Delete Hypothesis ( < P,A >,Q );

Print ( P', < E,F >);

End.

The function of algorithm Main ( < P,A >,G ) is to split the observation, which is a rule, into a set of literals, deriving the explanation for each literal and updating the program with the explanations starting from the body part of the observation. Once the body part is explained and program is updated, the updated program is used. It should be noted that we do not restrict an observation to be ground. For a non-ground observation, as we mentioned earlier, we need to consider all its ground forms which are obtained by substituting variables in the observation with elements of program universe U(P). This consideration will be adopted in our following algorithms.

Algorithm 2. Insert Hypothesis(< P,A >,G)

Function: Explain the observation G WRT < P,A >

Input: A logic program < P,A > and an observation G

Output: An explanation < E,F > such that ( P ∪ E ) - F |= G

Begin

Let P' = P and Path = ∅;

Loop

T = a finite SLDNF tree for P'∪{← G};

If there is a successful branch

n₀ → n₁ → ⋯ → nᵢ → ◊ in T and nᵢ ∈ A
then \( E_n = n_i, E = E \cup E_n, P' = P' \cup E; \)

Return \((< E, F, >, P');\)

Select a finitely failed branch \( B \) in \( T \) such that \( Path \subseteq B; \)

\( n_0 \rightarrow \cdots \rightarrow n_i \rightarrow Failed; \)

(1) If \( n_i \) has form \( \leftarrow P_{i1}, [not]P_{i2}, \cdots \)

then If \( P_i \subseteq A \) then \( E_n = P_{i1}, E = E \cup E_n, P' = (P' \cup E); \)

Return \((< E, F, >, P');\)

(2) If \( n_i \) has form \( [not]P_{i1}, [not]P_{i2}, \cdots \)

\( P' = \text{DeleteHypothesis}(P', notP_{i1} \leftarrow); \)

EndLoop

End.

The algorithm \textbf{Main} splits the observation into a set of \textit{literals} and passes them to the algorithm \textbf{Insert hypothesis}. If there is a successful branch in the SLDNF tree, then the process stops. Otherwise, we need to add or remove some rules from \( P \) so that the goal \( \leftarrow Q \) can be achieved. It is worth noting that if a sub goal \( \leftarrow notP \) fails in some branch of the SLDNF tree, we need to call algorithm \textbf{DeleteHypothesis}(\( P', notP \leftarrow \)), which will be described below, in order to achieve the sub goal \( \leftarrow notP \). Finally, after change \( P \) to \( P' \) such that \( P' \vdash G \). First, the body of the observation is explained using this algorithm by SLDNF tree for \( P \cup \{ \leftarrow G \} \) where \( G \) is an atom from the body of the observation. Once the body of the observation is explained completely and a new program \( P' = (P \cup \Sigma) - F \) is formed to explain the head of the observation, thereafter from the SLDNF tree for \( P' \cup \{ \leftarrow G \} \) where \( G \) is the head of the observation we will get another set of explanation which combined gives the
explanation for the observation.

Algorithm 3. Delete Hypothesis ($\mathcal{P}, \mathcal{A}, \mathcal{G}$)

Function: Explain the observation $\mathcal{G}$ WRT $\mathcal{P}, \mathcal{A}$

Input: A logic program $\mathcal{P}, \mathcal{A}$ and an observation $\mathcal{G}$

Output: An explanation $\langle \mathcal{E}, \mathcal{F} \rangle$ such that $(\mathcal{P} \cup \mathcal{E}) - \mathcal{F} \not\vdash \mathcal{G}$

Begin

$T$ = a finite SLDNF tree for $\mathcal{P}' \cup \{ \leftarrow \mathcal{G} \}$;

Loop

Let $T$ = a finite SLDNF tree for $\mathcal{P}' \cup \{ \leftarrow \mathcal{G} \}$;

For each successful branch $B$ in $T$,

1. If there exist two nodes $n_i$ and $n_{i+1}$ in $B$ such that $n_i \rightarrow n_{i+1}$,
   
   where $n_i$ and $n_{i+1}$ have forms $\leftarrow P_{i1}, \lnot P_{i2}, \ldots$ and $\leftarrow \lnot P_{i2}, \ldots$
   
   respectively, then $\mathcal{F}_n = P_{i1}$;

   If $\mathcal{F}_n \subseteq \mathcal{A} \cap \mathcal{P}$ Then $\mathcal{F} = \mathcal{F} \cup \mathcal{F}_n, \mathcal{P}' = \mathcal{P}' \cup \mathcal{F}$;

   Return $\langle \mathcal{P}', \langle \mathcal{E}, \mathcal{F} \rangle \rangle$;

2. If there exist two nodes $n_i$ and $n_{i+1}$ in $B$ such that $n_i \rightarrow n_{i+1}$,

   where $n_i$ and $n_{i+1}$ have forms $\leftarrow \lnot P_{i1}, \lnot P_{i2}, \ldots$ and $\leftarrow \lnot P_{i2}, \ldots$

   respectively, then $\mathcal{F}_n = P_{i1}$;

   if $\mathcal{F}_n \subseteq \mathcal{A}$ then $\mathcal{E} = \mathcal{E} \cup \mathcal{E}_n, \mathcal{P}' = \mathcal{P}' \cup \mathcal{E}$;

   Return $\langle \mathcal{P}', \langle \mathcal{E}, \mathcal{F} \rangle \rangle$;

3. If there exist two nodes $n_i$ and $n_{i+1}$ in $B$ such that $n_i \rightarrow n_{i+1}$,

   where $n_i$ and $n_{i+1}$ have forms $\leftarrow P_{i1}, \lnot P_{i2}, \ldots$ and

   $\leftarrow \lnot P_{i1}, \ldots, \lnot P_{i1q}, \lnot P_{i2}, \ldots$

   respectively in which a rule $r: P_{i1} \leftarrow \lnot P_{i1}, \ldots, \lnot P_{i1q}$
is used to derive node \( n_{i+1} \), then \( \mathcal{F}_n = r \);

If \( \mathcal{F}_n \subseteq \mathcal{A} \cap \mathcal{P} \) Then \( \mathcal{F} = \mathcal{F} \cup \mathcal{F}_n, \mathcal{P}' = \mathcal{P}' - \mathcal{F} \);

Return \((\mathcal{P}', <E, F>)\);

\( T \) = a finite SLDNF tree for \( \mathcal{P}' \{ \leftarrow G \} \);

EndLoop

End.

Property: Given an abductive logic program \(<P, A>\) and an observation rule \(G\). If there exists an explanation for \(G\), then the algorithm \(\text{Main}(<P, A>, G)\) will always return an explanation \((E, F)\) for \(G\), and this explanation is minimal. This theorem ensures that our procedure only finds the minimal explanation for an observation with respect to an abductive logic program if there exists some explanations for this observation.

4.5 Summary

In this chapter, we briefly explained the process of abduction, we proposed a SLDNF proof approach for abductive reasoning. Differently from previous approaches, this approach allow the observation to be a rule. Since within our framework, an abductive reasoning is translated into a SLDNF proof, this approach can be easily implemented based on the revision of traditional SLDNF proof procedure.
Rule Based Abduction
Chapter 5

Update and Abduction

5.1 Introduction

The aim of this thesis is to propose a unified procedural approach to deal with updates and abduction based on SLDNF proof procedure. To achieve that we proposed a new method to deal with rule-based updates based on SLDNF proof procedure in chapter 3 and in Chapter 4 we discussed rule based abduction based on SLDNF proof procedure. In this chapter we compare these two methods. We start with a update problem and show how it can be achieved by doing abduction. We then discuss the difference between update and abduction.

5.2 Updates

In general knowledge systems are expressed as logic programs. Such systems consist of two components, facts and rules. Facts express the current state of the knowledge base and are usually changeable, while rules express constraints about the domain and hence are usually unchangeable. However in dynamic domains, a knowledge base has to be updated not only because some
new facts occur, but also because some old rules are no longer suitable to characterize new domain constraints. So one can say a knowledge base is updated when

1. A new fact is added to or removed from the knowledge base which we call simple fact update.

2. A rule is added or removed from the knowledge base which we call rule based update.

**Example 16** Consider a simple knowledge base representing a family.

\[ P: \]
\[
Married(Al, Bo) \leftarrow, \\
Child(Cy, Bo) \leftarrow, \\
Child(Di, Bo) \leftarrow, \\
Male(Al) \leftarrow, \\
Male(Cy) \leftarrow, \\
Female(Bo) \leftarrow, \\
Female(x) \leftarrow not(Male(x)).
\]

Intuitively the above knowledge base presents the following facts and knowledge that Al, Bo are married, Cy, Di are children of Bo. Al, Cy are male and Bo, Di are female. If some one is a female then she can’t be a male. This is a simple family knowledge base. After some time, say Ed got Married to Di, which can be represented by

\[ Married(Ed, Di) \]
One has to add this fact and update the knowledge base to represent the current state of the family tree. This is the example of a simple fact update in general.

After looking at a number of family trees similar to this, one reaches to a decision that if \( x \) is a parent of \( y \) then \( y \) is a child of \( x \), which can be represents by

\[
\text{Parent}(x, y) \leftarrow \text{Child}(y, x)
\]

At this time one has to add this rule into the current knowledge base to represent to new knowledge which is a rule. This is a simple example for the rule-based update.

In Chapter 3 we explained a SLDNF proof procedure to implement such update, either a simple fact update or a rule based update where the update rule is of form

\[
Q \leftarrow P_1, \cdots, P_m, \neg P_{m+1}, \cdots, \neg P_n,
\]

which is called an *inserting rule* or

\[
\neg Q \leftarrow P_1, \cdots, P_m, \neg P_{m+1}, \cdots, \neg P_n.
\]

which is called as a deleting update rule, allows the negation as failure sign \( \neg \) to be occurred in the head of the rule.

In this general setting, the update problem is as follows: given a logic program and a fact or rule, which is (not) a logical consequence of the program, find a way to change the program so that the formula is no longer (is) a logical consequence of the updated program, which is a simple fact update.
The rule-based update addresses the problem: given an initial knowledge base and an update rule, how to update the initial knowledge base such that whenever the body of the update rule is achieved in the initial knowledge base, the head of the update rule is then achieved in the resulting knowledge base.

5.3 Abduction

Conventional abduction first introduced by philosopher Pierce on the other hand tries to explain the observations with the help of the existing knowledge base. He defined *Abduction* as another form of synthetic inference, but the case from a rule to a result. The result from a case to a rule is called as deduction, A rule from the result to a case is called as induction. According to him each argument is composed of three propositions *case result* and *rule*. Peirce further characterized abduction as the "probational adoption of a hypothesis" as explanation for the results, according to known laws. "it is however a weak kind of inference, because we cannot say that we believe in the truth of the explanation, but only that it may be true" [2].

Consider an example drawn from [21].

**Example 17** Consider the following theory $T$ :

- $\text{grass} - \text{is} - \text{wet} \leftarrow \text{rained} - \text{last} - \text{night}$
- $\text{grass} - \text{is} - \text{wet} \leftarrow \text{sprinklers} - \text{was} - \text{on}$
- $\text{shoes} - \text{are} - \text{wet} \leftarrow \text{grass} - \text{is} - \text{wet}$

If we observe that our shoes are wet, and we want to know why this is so, *rained-last-night* is a possible explanation, i.e. a set of hypotheses that together
with the explicit knowledge in $T$ implies the given observation. Obviously, 
_splinkers-was-on_ is another alternative explanation.

Abductive logic programs are logic programs, with a set of pre-specified, domain specific class of sentences called _abducibles_. Any explanation to the observation is drawn form the set of abducibles. As you go through this chapter you will come to know what is the criteria for selecting the abducibles.

In Chapter 4 we discussed abduction in logic programs, simple fact abduction and rule based abduction. In our formalism abduction not only gives an explanation for the observation which exists in the knowledge base but also tries to provide the possible options of knowledge base changes which intern explains the fact which is observed.

Consider another example as follows.

**Example 18** Let the knowledge base $(\mathcal{P}, \mathcal{A})$ be

$\mathcal{P}$:

\[
\begin{align*}
\text{Passed(Jim)} & \leftarrow, \\
\text{Passed(Sue)} & \leftarrow, \\
\text{Passed(Liz)} & \leftarrow, \\
\text{Failed(x)} & \leftarrow \neg(\text{Passed(x)})
\end{align*}
\]

$\mathcal{A}$:

\[
\begin{align*}
\text{Passed(Jim)} & \leftarrow,
\end{align*}
\]

The meaning of the program is straight and simple. _Jim, Sue, Liz_ took some exams and passed. Some one who did not passed the exam is failed. After some time one noticed that _Jim_ copied from _Sue_ so now _Jim_ is failed. The observation is _Failed(Jim)_ , Our formalism explained in Chapter 4 comes to a
conclusion that if one removes \textit{passed(Jim)} from the original knowledge base, the observed fact can be explained from the knowledge base.

5.4 Why Abducibles?

Why do we need a set of abducibles? Consider a situation where the knowledge base is a time critical system like satellite fault detection and correction system. Once the satellite is launched, if any problem occurs the system got to identify the problem and repair it. If a problem occurs in firing the second stage, the system should not try to start to analyze all the stages, it should only look into the situations which cause the miss fire. The set of abducibles here is the knowledge involved with the second stage firing.

Consider another example of large databases such as libraries or WWW search engines, when you try to find a book which is related to IT and written by John Smith, the set of abducibles here are the books related to IT for the first approximation and if there are number of books written by John Smith then the set of abducibles becomes all those books written by John Smith.

So the set of abducibles is nothing but a special set of knowledge, which is either a subset of the current knowledge, or in some cases which are the knowledge acquired from the other agents from the same domain.

The set of abducibles is always related to the observation and can be effected by the observation. From the above example one can see that our formalism of abduction works in two different ways. First it tries to explain the observation in the conventional way, if there is a basic explanation exists in the current
knowledge base, if the explanation of the observation does not exist in the present knowledge base it updates the current knowledge base to explain the observation. So our formalism can be used for updates and abduction.

Defining the set of abducibles plays an important role in any abductive procedures in general and very particular in our formalism. Say if the set of abducibles does not contain $Passed(Jim)$ then no abductive formalism can explain the observation $Failed(Jim)$.

### 5.5 Abduction Can Be Directly Used for Updates

The set of abducibles poses a constraint on the knowledge base, they actually define the knowledge that can be modified in the current knowledge base. So if there are no restrictions on the knowledge base or if the set of abducibles includes all the knowledge which is directly or indirectly related to the observation then our formalism gives you all the possibilities of the updated knowledge base. Let us consider the example from Chapter 4 again.

**Example 19** Let $< \mathcal{P}, \mathcal{A} >$ be an abductive program such that

\[
\mathcal{P}:
\]

\[
Bird(tweety) \leftarrow,
\]

\[
Bird(opus) \leftarrow,
\]

\[
Broken-Wing(tweety) \leftarrow,
\]

\[
Ab(x) \leftarrow Broken-Wing(x),
\]

\[
Flies(x) \leftarrow Bird(x), notAb(x),
\]

\[
\mathcal{A}:
\]
Broken-Wing(x) ←.

Intuitively the above knowledge base presents the following facts and knowledge: tweety, opus are birds; tweety has a broken wing; if some one has a broken wing then he is abnormal; if some one is a bird and not abnormal then only he can fly. The set of abducibles consists of Broken-Wing(x) means all the observations over this program must be explained in terms of broken wing only.

If some one wants to know why tweety is not flying then conventional abduction and our formalism both gives an explanation saying that because tweety has a broken wing. If some one observed that tweety is flying and he wants to explain that fact. Conventional abduction cannot explain such fact, but our formalism says that if you delete the fact that tweety has got a broken wing, the knowledge base can explain the fact observed.

If one considers the observation as an update rule and implement our formalism developed in Chapter 3 then the updated knowledge base can actually explain the observation Flies(tweety) ■

5.6 The Difference

In the case of the above example our abductive formalism is the same as the update formalism because as we mentioned the set of abducibles in that program includes all the knowledge related to the observation, if the set of abducibles does not include all the knowledge related to the observation then it is possible that our update and abductive formalisms generate different solutions. Consider the following example.
Example 20 Consider a knowledge base $\mathcal{P}$ where

$$
\mathcal{P}:
\begin{align*}
A & \leftarrow \\
B & \leftarrow \\
C & \leftarrow A, notE \\
D & \leftarrow C, B
\end{align*}
$$

Say we want to update it with

$$
\mathcal{G}: notD \leftarrow C.
$$

$\mathcal{G}$ is a deleting update rule. According to our formalism we first verify that whether the body of the rule is derivable from the knowledge base. Clearly the body of the rule is derivable from the existing knowledge base. Then we try to update the knowledge base such that the head the rule is derivable from the knowledge base.

Three possible updated versions of the knowledge base $\mathcal{P}$ can be formed.

1. $\mathcal{P} = \{A \leftarrow, B \leftarrow, C \leftarrow A, notE\}$ after deleting $D \leftarrow C, B$

2. $\mathcal{P} = \{A \leftarrow, B \leftarrow, D \leftarrow C, B\}$ after deleting $C \leftarrow A, notE$

3. $\mathcal{P} = \{A \leftarrow, C \leftarrow A, notE, D \leftarrow C, B\}$ after deleting $B \leftarrow$

There are two other possibilities but we don’t consider them as they violate our requirement that the updated knowledge base must also achieve the body of the update rule.
Let us consider the same update problem and try to implement our abduction formalism by considering the update rule as an observation, and here assume the only abducible is $D \leftarrow C, B$. Now the knowledge base $\mathcal{P}, \mathcal{A}$ looks like

$$\mathcal{P}:$$
$$A \leftarrow$$
$$B \leftarrow$$
$$C \leftarrow A, notE$$
$$D \leftarrow C, B$$

$$\mathcal{A}:$$
$$D \leftarrow C, B$$

Let the observation $\mathcal{G}$ be $notD \leftarrow C$. Our abductive formalism comes up with a solution saying that one has to delete the rule $D \leftarrow C, B$ from the knowledge base to explain the observation. Then the modified knowledge base, which is able to explain the observation, is $\mathcal{P} = \{A \leftarrow, B \leftarrow, C \leftarrow A, notE\}$, which is the first option from our update formalism.

From this example one can see that abducibles pose constraints on the update of the knowledge base. If the set of abducibles is $\{B \leftarrow, C \leftarrow A, notE, D \leftarrow C, B\}$, on the other hand then our formalism comes up with all the possibilities, which are generated by our updating formalism.

Let us take a closer look at the algorithms given in Chapter 3 and in Chapter 4 and compare them to find how our abductive algorithm can be used to do updates and how it resolves the conflicts.
There are three important routines in our algorithm proposed in Chapter 3 for updates.

The main purpose of the algorithm checking is to verify whether the body part of the update rule is achievable from the existing knowledge base. The process can be continued if and only if this routine returns a True value. It means that the body part of the update rule is achievable by the existing knowledge base. Then we have to update the knowledge base such that it achieves the head of the update rule. For that first we find the type of the update rule whether an insert update rule or a deleting update rule. Depending on the type of the update rule appropriate routine is called.

If the update rule is an insert update rule then routine insert update is called. The purpose of this routine is to create a SLDNF tree with the head of the update rule as the goal (head).

Then by resolution proof first it checks whether the head is derivable from the tree if so it returns the knowledge base with out any updates done as the knowledge base it self achieves the update rule.

If the head is not achievable from the exiting knowledge base the this routine

- traces the facts/rules which conflict the goal to be achieved. OR

- traces which facts/rules to be added to the current knowledge base to achieve the goal.

In the case of first option the routine again calls one of the insert update or delete update routines depending on the fact/rule. In the case of second option it adds the fact/rule to the current knowledge base and checks whether the
knowledge base is complete or not (whether the body part is also achievable from the updated knowledge base or not). If so then it returns the updated knowledge base, which is the knowledge base achieves the updated rule. If the updated rule is an deleting updated rule then the routine delete update is called. The purpose of this routine is to create a SLDNF tree with the negation of the head of the update rule as the goal (head).

Then by resolution proof first it checks whether the head is derivable from the tree if so then this routine traces the facts/rules that are making the knowledge base to achieve the goal and tries to make them fail such that the head of the goal is failed. if knowledge base it self cannot derive the head of the rule then this routine returns the current knowledge base as the updated knowledge base as no updates are needed.

If the head is achievable from the exiting knowledge base the this routine

- traces the facts/rules which makes the goal to be achieved. OR

- traces which facts/rules to be added to the current knowledge base to fail the goal.

In the case of first option the routine again calls one of the insert update or delete update routines depending on the fact/rule. In the case of second option it adds the fact/rule to the current knowledge base and checks whether the knowledge base is complete or not (whether the body part is also achievable from the updated knowledge base or not). If so then it returns the updated knowledge base, which is the knowledge base achieves the updated rule.
Now let us see how the algorithm for abduction provided in Chapter 4 incorporates all the functions described above with the capabilities of doing conventional abduction.

The algorithm described for abduction consists of three important routines. The first routine *main* actually does verify the type of the observation made and splits the observation into parts, sends them to appropriate routines, *insert hypothesis* in the case of a *positive observation* and *delete hypothesis* in the case of the *negative observation*. It receives the explanations from the routines and prepares an updated knowledge base and the explanations for the observation. At the end of the process this routine returns the minimal explanation with the updated knowledge base. if knowledge base itself then this routine returns the explanation for the observation and returns the knowledge base with out any modifications as such is not needed.

The type of the observations are the same as the update rules. The *positive observation* is same as the *inserting update rule* and the *negative observation* is same as the *deleting update rule*. The main difference between handling any observation and a update rule is that in the case of an observation we actually try to explain the body of the observation, i.e. we try to update the knowledge base if the body of the observation is not achievable by the current knowledge base but in the case of the updates we check whether the body of the update is derivable from the knowledge base. if the body is not derivable then we just stop the process.
The other routines \textit{insert hypothesis}, \textit{delete hypothesis} are called when a positive/negative observation is made. The function of these two routines are to provide an explanation for the observation and an updated knowledge base which is able to explain the observation. So actually these two routines does the conventional abduction task if the knowledge base is able to provide the explanation for the observation by itself. If not they perform the updation task over the knowledge base so that the knowledge base is able to explain the observation considering the observation as an update rule.

5.7 Summary

In this chapter we showed that our abductive formalism proposed in the previous chapter is capable of doing conventional abduction and updates. By using this approach one can perform simple fact or rule based updates as well as simple fact or rule based abduction. This approach is more expressive than other approaches as this approach deals normal logic program as the knowledge base and fact or rule as the observations while the other approaches use ground literals or atoms as the observations.
Brain, which contributes every thing to make human beings different from the other living beings is still unexplored. What we know about brain is, it does store data/knowledge in various forms and perform search, modification, provide solution/explanation to the situation/observation. Number of researchers in AI tried to simulate the tasks performed by a human being. Each one concentrated on a particular task so each person proposed a different kind of data/knowledge representation tools/techniques. But the reasoning criteria is either one of the three known inferencing mechanisms Deduction, Induction and Abduction. When one wants to simulate a human being one might have to use all the knowledge representation techniques available. The reasoning mechanism must use a logic that can do the three types of inferencing, without any one of these three the reasoning part is not complete or quite laborious. Consider a knowledge base which has very large data, if one does abduction task before doing deduction task then the size of the knowledge base to be searched will be reduced a lot depending on the number of the individual facts defined in the knowledge base. In the same way if there is a mechanism which can perform automated knowledge acquisition from the database/examples gathered by the source(entity) from various sit-
uations/co sources(entities) then the size of the knowledge base can be drastically reduced. So it is a must for any machine that simulates human to perform a combination of inferencing techniques to obtain a fast and accurate solution/decision. This thesis is a part of proposing such a unified inferencing engine, in which we proposed an algorithm that can perform Abduction, deduction, and updates.

6.1 Summary of The Thesis

Chapter 1 discusses about the knowledge representation, characteristics of knowledge representation and knowledge representation tools as there are various knowledge representation tools available for various applications in AI. It briefly discusses about reasoning mechanisms. As updates is one of the main topics of this thesis Chapter 1 briefly discusses the types of updates proposed depending on various knowledge representation tools. This chapter gives the back ground to understand the purpose of this thesis and about knowledge bases. The chosen representation tool through out this thesis is normal logic programs because we think logic programs work well in various fields rather than any other knowledge representation tool. But the proposed algorithm can be used in conjunction with any of the knowledge representation tools described here. Then Chapter 2 explains the syntax and semantics used through out this thesis, some of the definitions from the earlier work on logic programs and updates were explained/ modified to suit to the current context. To solve the problem of expressing the negative information there
are two suggested solutions closed world assumption and negation as failure. Negation as failure is selected in this thesis. As we are interested in a procedural approach so that one can easily implement and incorporate this logic we preferred SLDNF approach. Chapter 3 purely concentrates on updates, It explains what are updates, types of updates, fact updates, rule based updates. As the fact updates are nothing but a subset of rule based updates, we concentrated on rule based updates and proposed how a rule-based update can be achieved. For this purpose we explained two forms of rules that our algorithm can deal with. In general these two types of rules are enough to represent most situations because one can encode negative information using these rules. Chapter 4 discusses about conventional abduction which is a fact based explanation to any simple fact observation. We extended the definition of conventional abduction to a rule based explanation for any rule based observation. Again here we defined two types of observations such that negative observations can also hold well. This chapter discusses an algorithm which can do rule based explanation and update. Chapter 5 discusses the relation between the proposed rule based abduction and rule based update as there are a lot of similarities between the two. It explains how the algorithm for rule based abduction can be used to perform rule-based updates. Using some examples this chapter explains the constraints or the limited outputs when using the abductive algorithm for updates and explains in which situations the output of both algorithms are the same, in which situations the output is different and why. It also briefly explains why we need abducibles.
6.2 Contributions

The main contributions of this work include

6.2.1 Rule Based Updates

We extended the updates mechanism to rule based from a simple fact based update.

6.2.2 Rule Based Abduction

We extended the conventional fact based abduction to a rule based abduction that can explain negative information.

6.2.3 Unified Approach

We proposed a unified procedural approach to perform updates and Abduction which is more flexible, accurate and reduces the search/update time when used as a inference machine.

6.3 future Work

The only method of inference that is not covered by this thesis is induction. In Knowledge acquisition and machine learning induction plays a very important role. So it is intended to incorporate the inductive task in this formalism such that the underlying this reasoning algorithm becomes a fast, accurate, self-learning and self-corrective inference engine.
Bibliography


