Chemistry Studio: An Intelligent Tutoring System

Ashish Gupta, IIT Kanpur, ashgupta@cse.iitk.ac.in
Akshay Mittal, IIT Kanpur, amittal@cse.iitk.ac.in
Amey Karkare, IIT Kanpur, karkare@cse.iitk.ac.in
Sumit Gulwani, Microsoft Research, Redmond, sumitg@microsoft.com
Ashish Tiwari, SRI Interntional, tiwari@csl.sri.com
Rupak Majumdar, UCLA, rupak@cs.ucla.edu

***Abstract*—In this paper, we present the prototype of an automated education system, aimed at helping the students in their learning process. Our main motivation is to introduce a dynamic component in such learning systems as compared to existing systems which are primarily static in nature. We aim to automatically solve problems, in the domain of Periodic Table and its properties, with proper logic and reasoning, generate solutions and explanations in accordance with the interest and knowledge of the student. The project involves formulation of a logical system for the Periodic Table in Prolog with an interface in Yield Prolog. The system instead of being based on direct lookup paradigm, breaks a complex problem into certain basic facts by relating it to the underlying logic and provides a smooth flow of reasoning from the base argument up to the final solution. We aim to convert it to interactive system where the student can ask for preferential solution taking into account the knowledge base of a specific individual. The project's main features include logic formulation, problem solving in logic, hint generation and problem generation. We present a statistical evaluation showing that the system solves a large number of questions for each template that is created.**

***Index Terms*—periodic tables; logic; problem solving;**

I. Introduction

The current tutoring systems (for subjects such as Chemistry, Physics, Mathematics) rely on an existing database of facts which are used to solve questions. As a result, proper reasoning behind the solving of questions, from the perspective of a student, is missing. Just knowing the answer to a question does not help in the learning process of the student. The flow of reasoning from the base argument up to the final solution of the question is required. On these lines, we have been working on developing a system which solves a question by giving simple and complete reasoning comprehensible to the student.

The subject of study, that we are aiming at, is the Periodic Table from Chemistry. Students are initiated with the study of Periodic Table from grade 9^{th} onwards till grade 12^{th}. The simplicity and structured nature of the subject makes it an ideal candidate for the formulation of logic and problem solving. Our system can identify the various facts present in the periodic table such as atomic number of an element, group number of an element, and so on. Modeling of predicates in the domain of periodic table makes the system capable of handling the reasoning behind the satisfaction of certain properties of elements, such as *ionization energy, electron affinity, electronegativity*, and so on.

Consider the following scenario. A professor gives a problem (in the domain of Periodic Table) to his class. Before solving the problem, the students are expected to have a knowledge of certain basic facts about the elements and properties involved in the problem statement. This includes an idea about the *atomic number* of elements and their position in the periodic table which affects various properties. Using these basic facts, the students then solve the complex problem, giving sound reasoning for their solution to the professor. This is exactly what our system aims at. Whenever a complex problem is encountered, instead of directly looking up the values of involved properties in the database, our system breaks it down to basic facts by relating it with other properties, and then with the in-built knowledge of the basic facts, it builds up the explanation for the solution of the problem.

The remainder of the paper is organized as follows. Section II introduces the logical formulation of the entities in a Periodic Table. Section III describes the prolog database created from the logical formulation. Section IV elaborates the various buckets of questions that the system is currently able to solve. In Section V, we describe the system that we have developed. Section VI provides a statistical evaluation of the system. Section VII concludes the work done so far and presents scope for future work.

II. Logic Formulation

The project is divided between problem solving (PS) and natural language processing (NLP) teams. A logic base for the system was required since this logic will act as intermediate language up to which NLP team would work and from where the problem solving team would start. To ensure independence between the teams, the logic base on which the system would be based had to be decided at an early step.

Logic formulation began with an initial process of identification of the major components in the Periodic Table

- Unary Predicates - correspond to general properties of the elements in the periodic table. The name of the predicate correspond to the respective property (Table I).
- Binary Predicates - correspond to the properties of compound (or entities containing two elements) (Table I).
- Unary Functions - correspond to functions returning corresponding value for the input arguments (Table I).
- Variables (Dependencies) - are the properties and attributes of elements present in the periodic table along

TABLE I: Components in the Periodic Table used for Logic Formulation

Unary Predicates	Unary Functions
AlkaliMetals	FirstIonizationEnergy
AlkalineEarthMetals	AtomicRadius
Transition Metals	IonicRadius
Metalloids	AtomicNumber
Non-Metals	GroupNumber
Halogens	MetallicCharacter
NobleGases	ElectronAffinity
RareEarthElements	ElectronicConfiguration
IsGasAtATP	QuantumNos
IsLiquidAtSTP	Color
IsSolidAtSTP	Conductance
IsSpinPaired	Reactivity
BinaryPredicates	OrbitalsInPeriod
	OxidationState
IonicBond	
CovalentBond	

TABLE II: Components in the Periodic Table used for Logic Formulation

Trends (Movement m, UnaryFunction F, Change c)
Trend(down, atomicNumber, +ve)
Trend(right-down, atomicNumber, +ve)
Trend(left-down, size, +ve)
Trend(right-up, FirstIonizationEnergy, +ve)

with the basic attributes on which they are dependent.

- Trend - As we move along a specific direction in the periodic table, the properties show certain specific trend consistently which are captured by the trend predicate (Table II).

III. Prolog Database

Following from the previous section, we construct a set of facts and rules in the logic programming language **Prolog**. For each predicate, we discover its relations with all the possible predicates and facts; thus building a directed graph where each predicate may depend on other predicates and in turn is depended on by others. Every directed edge present in the graph represents a relation from one predicate to the other and hence becomes a part of the explanation process for a question whenever the edge is traversed. Presence of more than one path between two predicates indicates the presence more than one solutions for a question whenever these two predicates are involved. This key idea shall be at later stages of the project to solve questions where a student desires a solution with a different comprehensible explanation. Figure 1 shows a part of the directed graph that is constructed using the predicates. An edge from property A to property B implies that B is dependent on A.

Knowing the values of certain properties (such as *atomic number, period number, atomic size*, and so on) of an element is often required and essential. Table III shows the predicate

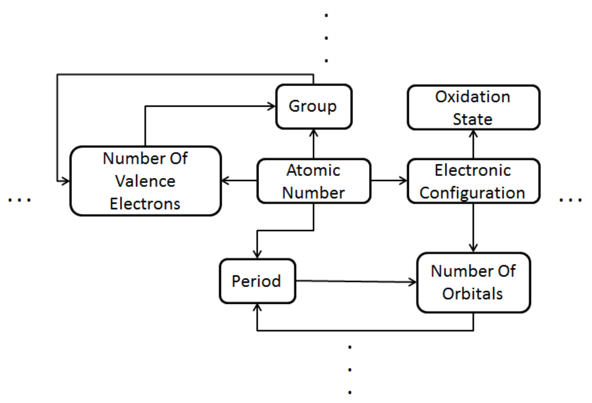


Fig. 1: Directed Graph between the Predicates of the Prolog Database

TABLE III: Predicate *data* with Examples

data(atomicNumber, element, groupNumber, periodNumber, atomicMass, FIE, EA)
data(1, 'H', 1, 1, 1.008, 1312, 72)
data(2, 'He', 18, 1, 4.003, 2372, 0)
data(3, 'Li', 1, 2, 6.941, 520, 60)
data(4, 'Be', 2, 2, 9.012, 899, 0)
data(5, 'B', 13, 2, 10.81, 801, 27)
⋮

data which relates an element to its various properties in the following order - *atomic number, element name, group number, period number, atomic mass, first ionization energy and electron affinity*. As the database will be extended, more property values will be encoded into this predicate. Using the *data* predicate, the other properties can be obtained with ease as shown in Table IV.

IV. Templates

There is a vast pool of questions available in the domain of Periodic Table. Solving each question separately by encoding its logic into the database would be fruitless and inefficient. Thus, there is need to cluster similar questions into a common bucket. We term a bucket containing similar questions as a *template*. The system that we propose in this paper builds logic for these *templates* in the prolog database and uses them to solve multiple questions present in the bucket corresponding to the *template*. For evaluation purposes, we show later that the number of *templates* encoded in the database is significantly less than the number of benchmark questions solvable by the system. For each of the identified *templates* there exist a logical formulation of the question. The system proceeds by parsing this logical formulation into a prolog goal which is satisfied using the prolog database developed in Section III. The logic has been developed by a mutual discussion between the PS and the NLP teams. Currently, we have found and encoded the following *templates* -

TABLE IV: Using *data* Predicate to obtain other properties

data(atomicNumber, element, groupNumber, periodNumber, atomicMass, FIE, EA)

atomicNumber(Ele,Value) :- data(Value, Ele, _, _, _, _, _)
group(Ele,Value) :- data(_, Ele, Value, _, _, _, _)
period(Ele,Value) :- data(_, Ele, _, Value, _, _, _)
atomicMass(Ele,Value) :- data(_, Ele, _, _, Value, _, _)
fie(Ele,Value) :- data(_, Ele, _, _, _, Value, _)
electronAffinity(Ele,Value) :- data(_, Ele, _, _, _, _, Value)
⋮

A. Template 1: Same

While solving periodic table related problems, students often encounter questions which involve finding elements which satisfy some fixed/given properties. Sometimes, it is a single property which needs to be satisfied or a conjunction of properties. In either case, the question is associated with a bound variable whose domain is restricted. This domain is utilized to find the appropriate binding for the variable. In some cases, the question does not provide information about the domain, in which case, the whole periodic table is searched to find the binding for the variable. The system proceeds by giving a binding to the unknown value and then matching it with the known value (computed from the question).

Example 1: *What is the atomic number of Calcium ?*
(a)21 (b)18 (c)19 (d)20
The system works by analyzing the logical formulation of the question which is -

Same(AtomicNumber(Ca), X), X in choices

In the above logical formulation of the problem, the *Same* predicate has two arguments, both of type *num*. The aim of this predicate is to equate the values of its arguments. It does so by iterating over different bindings of *X*, given in the choices, and then tries to satisfy the predicate *Same* i.e. whether the atomic number of *Ca* is same as the corresponding binding of *X*. If the predicate is *true*, then the corresponding binding of *X* is the solution to the problem. Our system generates the following prolog goal to be satisfied using the prolog database generated in Section III

:- iter([21, 18, 19, 20], X), atomicNumber('Ca', V0), X is V0.

V0 is a temporary variable introduced by the system to match the binding of the atomic number of *Ca* with the binding of *X*. The resulting binding of *X* at the successful satisfaction of the goal is *20*. ◀

It is to be noted that this template covers the most basic questions in the domain of periodic table. This template shall be used very often in the following templates, hence a clear understanding of the questions corresponding to this bucket is important. An interesting example which involves conjunction of multiple conditions is shown below.

Example 2: *What is the atomic number of the element in group 2 and period 3 ? (a)13 (b)14 (c)12 (d)15*
The logical representation of the goal is -

And(And(Same(Group(X), 2), Same(Period(X), 3)), Same(AtomicNumber(X), Y)), Y in choices

In this logical representation, *And* is not a predicate that belongs to our prolog database. It is a representation of the conjunction of conditions to be satisfied. In Prolog, this conjunction is represented as a comma-separated list of predicate goals. Hence, the prolog goal computed by our system is

:- iter([13, 14, 12, 15], Y), group(X, V0), V0 is 2, period(X, V1), V1 is 3, atomicNumber(X, V2), V2 is Y.

As discussed in the previous example, *V0* and *V1* are temporary variables introduced by the system to check the bindings for the corresponding values of the group number and the period number of the unknown element *X*. The resulting binding of *Y* obtained at the successful satisfaction of the goal is *12*. ◀

B. Template 2: Max/Min

Questions concerning periodic table properties often involve maximizing (or minimizing) a given property over a domain of elements, provided along with the question. In this query, each element of the domain is considered as a candidate for the solution and its relation with other elements is computed using the rules for the given property. The logical formulation of the *Max* predicate takes two arguments. One is the property to be maximized and the second is the domain of entities to be used for finding the maximum. In this case, dynamic passing of the predicates is involved (the intricacies are explained in Section V). The property which is to be maximized is called dynamically from the *max* predicate.

Example 3: *Which of the following elements has the highest first ionization energy ? (a)Na (b)F (c)He (d)Ag*
For this question, the corresponding logical formulation would be

Max(FirstIonizationEnergy, X in choices)

Here the first argument denotes the property which is to be maximized and the second argument is the domain of the *Max* predicate. The prolog goal computed for this logical formula by the system is

:- iter(['Na', 'F', 'He', 'Ag'], X), takeout(X, ['Na', 'F', 'He', 'Ag'], L), max(firstIonizationEnergy, L, X).

The *max* predicate defined in the prolog database takes in another predicate as one of its arguments which it uses as the maximising parameter. This predicate is satisfied only when the element bounded to *X* has the property value greater than or equal to all the elements present in the remaining list of choices *L*. This way, by iterating over each element in the list of choices, the element for which the property takes maximum value is obtained. It is to be noted here that the value of the

TABLE V: Rules determining variation of First Ionization Energy in the Periodic Table

```
1) firstIonizationEnergy(X, Y, '+') :- trendPeriod(X, Y, '='), !,
trendGroup(X, Y, '+'), !.
2) firstIonizationEnergy(X, Y, '+'):- trendGroup(X, Y, '='), trend-
Period(X, Y, '-'), !.
```

first ionization for any of the elements is not looked up while computing the maximum. Only the base level facts are looked up. Table V shows the prolog rules governing the variation of *First Ionization Energy* with the elements. The theoretical interpretation of the rules in the domain of periodic table is as follows

- Rule 1: *First Ionization Energy* increases along(right) a period
- Rule 2: *First Ionization Energy* decreases down the group

The system solely uses these rules to determine the relationship between the *First Ionization Energy* of different elements. Based on these rules, resulting binding of *X* at the successful satisfaction of the goal is *He*. This is because the *First Ionization Energy* increases along a period and decreases down the group and *He* is the top-left corner element in the periodic table. On manually seeing the corresponding value in the database, we find that *He* has the ionization value *2372 kJ/mole* which is in fact maximum among the given elements. ◀

The problem statement may require that certain other condition(s) to be satisfied by the domain elements before the maximizing process is commenced. These conditions tend to restrict the domain further. To handle them, we devise a mechanism where these conditions are mentioned as a Filter-function. The filter is applied to the initial domain and the restricted domain is passed to the *max* predicate. The restricting of the domain happens in the interface (discussed in Section V).

Example 4: *Which element of group 17 has the highest first ionization energy? (a)Na (b)F (c)Cl (d)I*
The domain specified by the question includes four elements but according to the question only three of these (*F, Cl, I*) satisfy the condition of belonging to the group 17. Hence when computing the element with maximum first ionization energy, *Na* needs to be removed from the domain passed to the max-predicate. Hence, the logical formula for this question is

Filter(Same(17, Group(Y)), [Na, F, Cl, I], FilterChoiceList), Max(FirstIonizationEnergy, FilterList)

FilterList contains the new filtered and valid domain of elements for computing the maximum. The dynamic computing of the filtered list, in prolog itself, is a non-trivial task. Hence for this purpose the system pre-computes the *FilterList* (in the interface) before generating the prolog goal for the question. This results in

:- FilterList is ['F', 'Cl', 'I'], iter(FilterList, X), takeout(X, FilterList, L), max(FirstIonizationEnergy, L, X).

The resulting binding of *X* at the successful satisfaction of the goal is *F* since the *First Ionization Energy* decreases down a group, in this case group 17. ◀

Note that the comparison function is not solely a look-up based function. It depends on rules which are representative of the pre-knowledge of a student. As a result, instead of directly looking up the values of the property, say *First Ionization Energy (IE)*, the system proceeds by determining how the *ionization energy* depends on some of the basic properties, (in this case *Atomic Number*). We then look up the value of *atomic number* since that information is expected to be known by the student rather than the information about a more complex property such as IE, if he is faced with such a problem.

C. Template 3: ForAll

The forall problem is related to finding if a given list of properties for an element or a set of elements implies another property i.e.

$$A(x,y), B(x,y), C(x,y) \rightarrow D(x,y)$$

This problem has a naive solution as follows. For every combination of two elements from the domain, if at any point we get a combination such that the antecedent is satisfied while the consequent is not, then the output for this query is false. If, after traversing through the database, such a combination is not found then the output is *true*. Instead of following this computationally expensive approach, we propose an approach in which the system tries to find if the consequent can be implied using antecedent by a series of interconnected relations. For a better explanation, consider the following example:

Suppose we wish to decide if

$$\forall x : A(x), A1(x), A2(x) \rightarrow B(x)$$

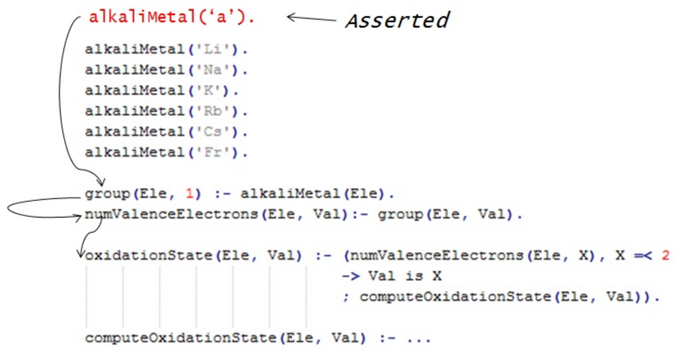
Then, we assert *A(a), A1(a), A2(a)* as three new facts where *a* is a new constant introduced into the system (its life spans only this question). Since *a* is a new constant, no existing fact from the database would be used, only rules will be traversed. Hence at this stage on querying *B(a)*, if the solution is true, then the *forall* formula is valid, else not. The constant *a* stands as a representative of all the elements in the database and on asserting the antecedents, the *forall* query gives a correct solution. If the database consists of the following rules

$$A(x), A2(x) \rightarrow P(x)$$

$$A1(x), A2(x) \rightarrow R(x)$$

$$R(x) \rightarrow Q(x)$$

$$P(x), Q(x) \rightarrow B(x)$$



```
alkaliMetal('a').        Asserted

alkaliMetal('Li').
alkaliMetal('Na').
alkaliMetal('K').
alkaliMetal('Rb').
alkaliMetal('Cs').
alkaliMetal('Fr').

group(Ele, 1) :- alkaliMetal(Ele).
numValenceElectrons(Ele, Val):- group(Ele, Val).

oxidationState(Ele, Val) :- (numValenceElectrons(Ele, X), X =< 2
                             -> Val is X
                             ; computeOxidationState(Ele, Val)).

computeOxidationState(Ele, Val) :- ...
```

Fig. 2: Sample Prolog Database

and the query is

$$\forall x : A(x), A1(x), A2(x) \rightarrow B(x)$$

Then we assert *A(a), A1(a), A2(a)* and we query *B(a)*. Prolog will say *true* as it can a find a logical path between the asserted facts and the goal predicate. Hence the *forall* formula is valid. This proves the *forall* formula without enumeration. On the other hand, if we had followed the first approach and checked for validity of the forall formula by querying its negation, i.e.

$$query : -A(X), A1(X), A2(X), not(B(X))$$

then Prolog would have answered the query by enumerating all possible values for *X*, testing for each and reporting the result either at the first failure or after the complete scan of the database.

Example 5: *Alkali metals belong to group 1 and usually have a +1 oxidation state.*
Suppose the prolog database consists of the facts/rules as shown in Figure 2. The facts contain the alkali metals. According to the *oxidationState* rule, if the number of valence electrons in an atom is less than 3, then the oxidation state is same as the number of valence electrons. Otherwise, there is a complex rule to compute the oxidation state (not elaborated for want of space). The logical formulation of this question hence becomes -

ForAll(LHS(AlkaliMetal(X)), RHS(And(Same(1, Group(X)), Same(1, OxidationState(X)))))

The logical formula contains *ForAll* as the main node with two children - *LHS* (representing the *antecedent*) and *RHS* (representing the *consequent*). As discussed above, we assert the antecedent by introducing a new constant *a* into the database and then we form the goal query with the consequent. This results in the following prolog goal -

:- assertz(alkaliMetal('a')), V0 is 1, group('a', V0), V1 is 1, oxidationState('a', V1).

Once the fact *alkaliMetal('a')* is asserted, one can see the sequence of steps that the system automatically follows in order to prove the goal. This shows the efficiency introduced into the system in solving the goal. Using the naive approach would have involved checking for each of the alkali metals present in the database. Thus, the result of the above prolog goal is *true*. ◀

It is to be noted that for our approach to work, the database should be exhaustive i.e. if there is a relation (however deep leveled and interconnected it may be), it should be present in our system. If the relation is arbitrary just resulting from lookup values (which should not be the case since it is asked to a student) then we will also not be able to solve it since the student is also not expected to solve it.

D. Template 4: Trend

As we move down/up a column of periodic table (viz. a group) or move left/right across a row of periodic table (viz. a period), a lot of properties show specific trends which can be captured by the logical system. We model a trend of a property using three arguments. The direction of motion, the predicate (viz. property) of concern and the trend that the property shows (viz. increases/decrease/remains constant). The name of the predicate specifies which property is being talked about. The first-two arguments for the predicate stand for the two elements between which the relation is to be found out. The third argument is the change itself. For example, a trend concerning electronegativity is modeled as

- electronegativity('p','q','+')
  electronegativity(p) $>$ electronegativity(q))
- electronegativity('p','q','-')
  electronegativity(p) $<$ electronegativity(q)
- electronegativity('p','q','=')
  electronegativity(p) = electronegativity(q)

Kindly note that in the above modeling of the trend, the new constant elements *p* and *q* are chosen along the direction of movement. If the direction is *down* the group then *p* lies above *q*. Also note that the direction of movement is not encoded in the above modeling. With each direction along the periodic table, we are required to assert certain statements before the system goes on to find a binding for the bound variable. For example, the problem statement desires of student to find the trend *T* for a property *P* when one moves in a direction *D*. This necessitates the use of certain facts (associated with each movement) to be asserted before the goal is queried using the binding of trend. For example, in the case when *D* is *down the group*, the system asserts the facts that down the group, the atomic number increases and the group number remains constant. Table VI shows the assertions made for each direction. These are the most basic facts with the movement in the particular direction and are essential for high level prolog goals. Then the goal is queried and the corresponding binding is obtained. If the goal is true, we display success along with the derivation tree and also retract the asserted facts. Otherwise, we retract the asserted facts, assert new facts based on other trend and query the goal again. This continues till either a successful binding is

TABLE VI: Assertions made with each direction

Direction	Assertions Made
down the group	atomicNumber('p', 'q', '-'), group('p', 'q', '=')
up the group	atomicNumber('p', 'q', '+'), group('p', 'q', '=')
right a period	atomicNumber('p', 'q', '-'), period('p', 'q', '=')
left a period	atomicNumber('p', 'q', '+'), period('p', 'q', '=')
⋮	⋮

TABLE VII: Assertions/Retractions made dynamically with each direction

assertFact('down') :- assertz(trendAtomicNumber('p', 'q', '-')), assertz(trendGroup('p', 'q', '=')).
assertFact('right') :- assertz(trendAtomicNumber('p', 'q', '-')), assertz(trendPeriod('p', 'q', '=')).
⋮
retractFact('down') :- retract(trendAtomicNumber('p', 'q', '-')), retract(trendGroup('p', 'q', '=')).
retractFact('right') :- retract(trendAtomicNumber('p', 'q', '-')), retract(trendPeriod('p', 'q', '=')).
⋮

obtained or none of the options satisfy the goal query.

Since any of the three *T, P* or *D* can become the bound variable in a question, we have three kinds of modeling of the Trend-template.

1) Property-P Unknown: In this case, the direction of movement in the periodic table and the trend which a property is supposed to follow is known. The unknown parameter is the property/predicate which follows the known trend in the given direction.

Example 6: *Which of the following properties decreases down the group ? (a)Electron Affinity (b)Electronegativity (c)First Ionization Energy (d)Metallic Character*
The logical formulation of this question is

Trend(down, X, decreases), X in Choices

As explained earlier while iterating over the bindings for *X*, we first need to encode the direction *down* in the database. This is done using the assertions mentioned in Table VI. This results in the following prolog goal being generated by the system

:- iter(['ElectronAffinity', 'Electronegativity', 'FirstIonizationEnergy', 'MetallicCharacter'], X), assertz(trendAtomicNumber('p', 'q', '-')), assertz(trendGroup('p', 'q', '=')), G=..[X, 'p', 'q', '+'], (G -> retract(trendAtomicNumber('p', 'q', '-')), retract(trendGroup('p', 'q', '=')); retract(trendAtomicNumber('p', 'q', '-')), retract(trendGroup('p', 'q', '='))).

The above prolog goal may look complicated but it is not. Here the goal makes a dynamic call to the binding predicate of *X* and checks if it shows the expected trend. In either case, the asserted statements must be retracted from the database. The resulting bindings of *X* at the successful satisfaction of the goal are *Electronegativity* and *FirstIonizationEnergy* since both the properties decrease down the group due to increase in the number of orbitals. ◄

2) Trend-T Unknown: In this case, when the property and the direction (up/down/left/right) are mentioned, the trend (i.e. increase or decrease or constancy) is unknown. We iterate over each of the increase, decrease or constancy options producing a binding for the variable and querying the goal. If the goal returns true, then the corresponding binding of the variable is valid.

Example 7: *Moving down the group, the first ionization energy shows which of the following trends ? (a)Increases (b)Decreases (c)Remains Constant*
As expected, the logical formulation for this question is

Trend(down, FirstIonizationEnergy, X), X in Choices

The corresponding prolog goal generated is

:- iter(['-', '+', '='], X), assertz(trendAtomicNumber('p', 'q', '-')), assertz(trendGroup('p', 'q', '=')), (firstIonizationEnergy('p', 'q', X) -> retract(trendAtomicNumber('p', 'q', '-')), retract(trendGroup('p', 'q', '=')); retract(trendAtomicNumber('p', 'q', '-')), retract(trendGroup('p', 'q', '='))).

Note that in this case the iteration is done over the trends and not the properties. The resulting binding of *X* at the successful satisfaction of the goal is *'+'* which indicates *decreases*. It is true since the *First Ionization Energy* decreases down the group. ◄

3) Direction-D Unknown: In this case, the direction of movement is to found when the trend and the property are given.

Example 8: *The first ionization energy decreases ___ ? (a)Up a Group (b)Left a Period (c)Down a Group (d)Right a Period*
The logical formulation for this question is

Trend(X, FirstIonizationEnergy, Decreases), X in Choices

This case is a bit tricky. The system cannot make assertions about a direction unless it knows the direction. Hence new rules in introduced into the prolog database which act as helper predicates in making assertions and retractions. These are shown in Table VII. Thus the corresponding prolog goal generated is

:- iter(['left', 'down', 'up', 'right'], X), assertFact(X), (firstIonizationEnergy('p', 'q', '+') -> retractFact(X); retractFact(X)).

The resulting binding of *X* at the successful satisfaction of the goal is *'down'* which indicates *down a group*. It is true since the *First Ionization Energy* decreases down the group. ◄

E. Template 5: Order

Order problems require ordering a list of elements present as options according to a given property. We have already explained the basic compare function in the Max-query which takes two elements and returns the order between the two. We apply merge sort in Prolog on the given list of options to obtain the sorted list. Note that in this case too, the system shall take in candidate orderings from the list of choices and validate if they represent the correct sorting order as per the given property.

Example 9: *Which of the following shows the correct order of increasing first ionization energy ? (a)He, Be, H, Li (b)Li, Be, H, He (c)Li, Be, He, H*
For this, the logical formulation of the goal is

Order(FirstIonizationEnergy, Ascending), X in Choices

To find the correct ordering among the given choices, we iterate over each option, find the correct order w.r.t. the property *First Ionization Energy* and match it with the original order. If they are same then the corresponding option represents the correct order of *First Ionization Energy*. Otherwise, the next option is chosen as candidate. Hence the prolog goal for this logical formula is

:- iter([['He','Be','H','Li'], ['Li','Be','H','He'], ['Li','Be','He','H']], X), order(X, firstIonizationEnergyProperty, T1), T1 = X.

The *order* predicate is an implementation of the *merge sort* algorithm. Hence its first argument list *X* is sorted in ascending order w.r.t the property *First Ionization Energy* and bound to the variable *T1*. The resulting binding of *X* at the successful satisfaction of the goal is *['Li','Be','H','He'].* ◄

True/False or Multiple Choice Questions

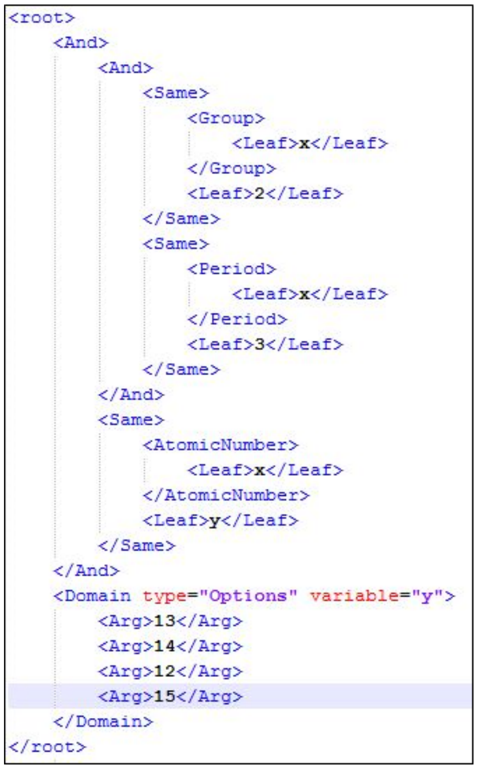
Having developed the basic prolog database, the true/false or multiple choice questions can be modeled in a very related manner. The true/false question involves querying the database using a free variable while the multiple choice question involve using a bound variable using options available from the choices and checking which option is correct.

Example 10: *Zn belongs to group 12 and period 4.*
As expected, this *true/false* question does not have any choice list. The system recognizes this question as belonging to *template 1*. Hence the logical formulation is -

And(Same(12, Group(Zn)), Same(4, Period(Zn)))

There is no bound variable in the logical clause. Hence, the system constructs a prolog goal which just needs to be test for truth or falsity.

:- V0 is 12, group('Zn', V0), V1 is 4, period('Zn', V1).



```
<root>
    <And>
        <And>
            <Same>
                <Group>
                    <Leaf>x</Leaf>
                </Group>
                <Leaf>2</Leaf>
            </Same>
            <Same>
                <Period>
                    <Leaf>x</Leaf>
                </Period>
                <Leaf>3</Leaf>
            </Same>
        </And>
        <Same>
            <AtomicNumber>
                <Leaf>x</Leaf>
            </AtomicNumber>
            <Leaf>y</Leaf>
        </Same>
    </And>
    <Domain type="Options" variable="y">
        <Arg>13</Arg>
        <Arg>14</Arg>
        <Arg>12</Arg>
        <Arg>15</Arg>
    </Domain>
</root>
```

Fig. 3: XML representation of Example 2

Incidentally, the problem statement in the question is indeed *true*. Hence the prolog goal is satisfied and the result is a success. ◄

V. The System Developed

In order to increase the interactivity of the project with the students, the final system, which the two teams (the PS and the NLP teams) aim to build, will take a question (in English) as input from the student and solve it using the system developed. It is clear that a standard is to be maintained between the NLP team and the PS team which shall define the intermediate language in which the question shall be parsed from English and then solved by using the Prolog database. The choice of our intermediate language is **XML (Extensible Markup Language)**. The Figure 3 shows the XML encoding of the question in Example 2. The NLP team builds a tree structure from the question. Every internal node of the tree represents a predicate and the leaves contain elements or values. Using this tree, for the XML standard, each node is represented as a tag and the values are encoded in their inner text. This makes it easier for the system to parse the XML file and detect the logic formula. In future, the two systems can be independently be extended with the XML format as an intermediate standard.

The input XML file for the system represents the question to be solved. Using the logic formula for the question, we develop a prolog goal which shall be solved for using the database of facts and rules that we developed in Section III.

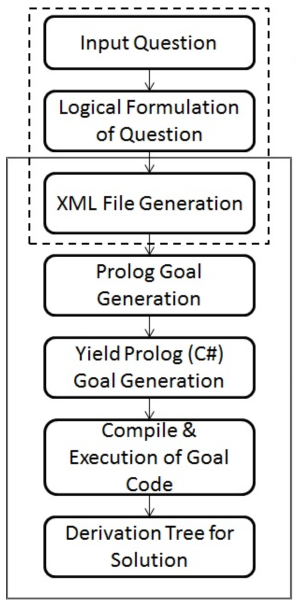
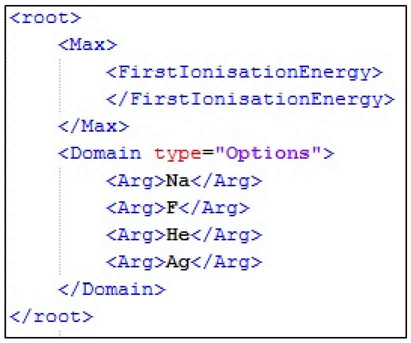

Input Question
Logical Formulation of Question
XML File Generation
Prolog Goal Generation
Yield Prolog (C#) Goal Generation
Compile & Execution of Goal Code
Derivation Tree for Solution
NLP
PS

Fig. 4: Flowchart from XML representation to the Derivation-Tree of the Solution (the Solid Box).

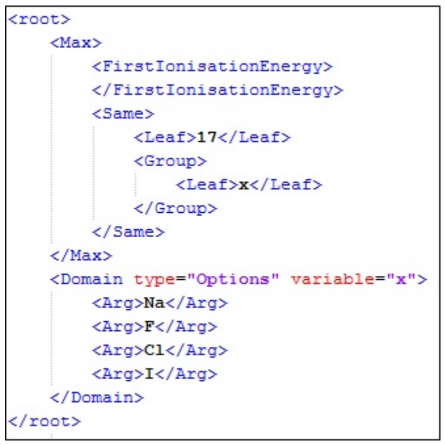


```
<root>
    <Max>
        <FirstIonisationEnergy>
        </FirstIonisationEnergy>
    </Max>
    <Domain type="Options">
        <Arg>Na</Arg>
        <Arg>F</Arg>
        <Arg>He</Arg>
        <Arg>Ag</Arg>
    </Domain>
</root>
```

Fig. 5: XML representation of Example 3

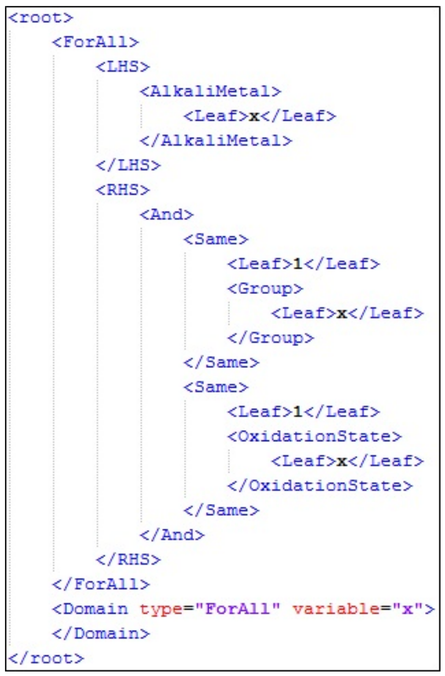
The bindings obtained and the derivation tree for the solution shall be the result. But parsing a XML file in Prolog is a non-trivial process. To simplify things, we have used C# as the *interface* language of our system. The interface parses the XML logic file to generate the prolog goal and feeds it to the prolog database. To simplify things further, it would be nice if the prolog database was itself in C#. So we leverage the Yield Prolog Compiler [1]. This compiler converts each prolog file into a corresponding C# code using the *yield* keyword [2]

Using Yield Prolog, we convert the developed Prolog database to C# database. The generated database is then used as a *template* for all the goal queries. Similar to this, the system generates a prolog goal (explained later) from the XML file and using the compiler obtains the C# code for the goal. It then inserts the goal code into the template database in order to get the bindings for the bound/free variables and hence the solution. Figure 4 shows a flowchart for the whole path from XML file to the reasoning-based solution. To summarize, it represents the following key steps -



```
<root>
    <Max>
        <FirstIonisationEnergy>
        </FirstIonisationEnergy>
        <Same>
            <Leaf>17</Leaf>
            <Group>
                <Leaf>x</Leaf>
            </Group>
        </Same>
    </Max>
    <Domain type="Options" variable="x">
        <Arg>Na</Arg>
        <Arg>F</Arg>
        <Arg>Cl</Arg>
        <Arg>I</Arg>
    </Domain>
</root>
```

Fig. 6: XML representation of Example 4

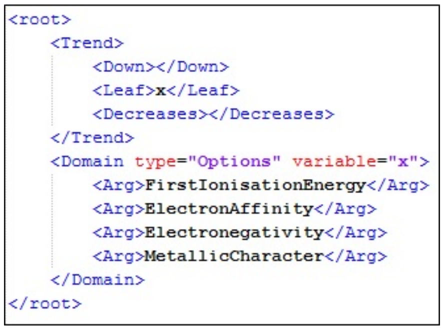


```
<root>
    <ForAll>
        <LHS>
            <AlkaliMetal>
                <Leaf>x</Leaf>
            </AlkaliMetal>
        </LHS>
        <RHS>
            <And>
                <Same>
                    <Leaf>1</Leaf>
                    <Group>
                        <Leaf>x</Leaf>
                    </Group>
                </Same>
                <Same>
                    <Leaf>1</Leaf>
                    <OxidationState>
                        <Leaf>x</Leaf>
                    </OxidationState>
                </Same>
            </And>
        </RHS>
    </ForAll>
    <Domain type="ForAll" variable="x">
    </Domain>
</root>
```

Fig. 7: XML representation of Example 5

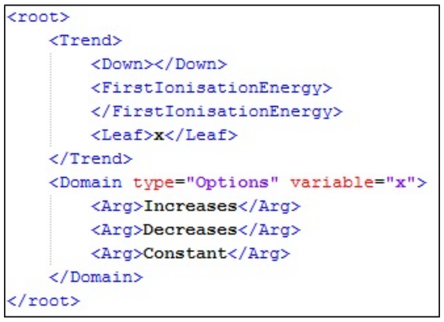
- Generation of Prolog Goal from XML Logical Formulation
- Generation of Yield Prolog Goal from the Prolog Goal
- Compile and Execution of the Yield Prolog Goal to get Solution
- Displaying the Derivation Tree for the Solution

The challenging part is the first step i.e. the generation of goal code in prolog from the XML logic formulation. For this task, each template needs to be handled independently although parts of code are re-used in some templates. For example, *Max* template and the *Trend* template are handled differently but *Max* and the *True/False* templates use the *Same* template for solving sub-queries whenever possible (in Example 4 and Example 10 respectively). The XML representations for the



```xml
<root>
    <Trend>
        <Down></Down>
        <Leaf>x</Leaf>
        <Decreases></Decreases>
    </Trend>
    <Domain type="Options" variable="x">
        <Arg>FirstIonisationEnergy</Arg>
        <Arg>ElectronAffinity</Arg>
        <Arg>Electronegativity</Arg>
        <Arg>MetallicCharacter</Arg>
    </Domain>
</root>
```

Fig. 8: XML representation of Example 6



```xml
<root>
    <Trend>
        <Down></Down>
        <FirstIonisationEnergy>
        </FirstIonisationEnergy>
        <Leaf>x</Leaf>
    </Trend>
    <Domain type="Options" variable="x">
        <Arg>Increases</Arg>
        <Arg>Decreases</Arg>
        <Arg>Constant</Arg>
    </Domain>
</root>
```

Fig. 9: XML representation of Example 7

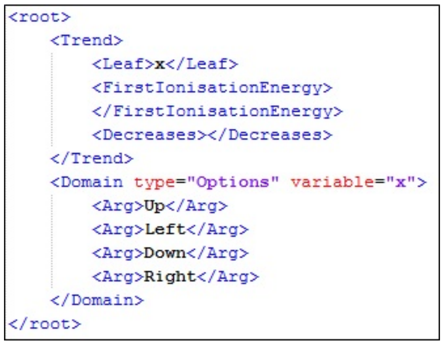
remaining examples are shown in Figures 5, 6-12.

For various problem statements, some assertions (and hence retractions) are to be made while solving a Prolog goal. This presents a problem in Yield Prolog since it does not handle the assertions in a convenient manner. It does not allow for the modification of the existing prolog rules/facts and hence if they are modified (i.e. asserted) then it treats the asserted rules in a separate predicate store rather than together with the existing rules for the same predicate. This affects the relation between static and dynamic calling of predicates. When after an assertion, a static call is made for a predicate, Yield Prolog uses the existing rules for the predicate rather than the asserted ones. Contrary happens with the dynamic calling of the same predicate. To solve this problem of dynamic and static calling of predicates, we modified the Yield Prolog compiler so that it does not create a separate predicate store for the asserted rules/facts. Also, the rules which are allowed to be asserted in the Prolog database must now contain a starting rule which signifies that the dynamic calling of the predicate is permitted. Details are shown in Figure 13.

A working demo of the whole system can be viewed at *http://www2.cse.iitk.ac.in/karkare/propsolve/*.

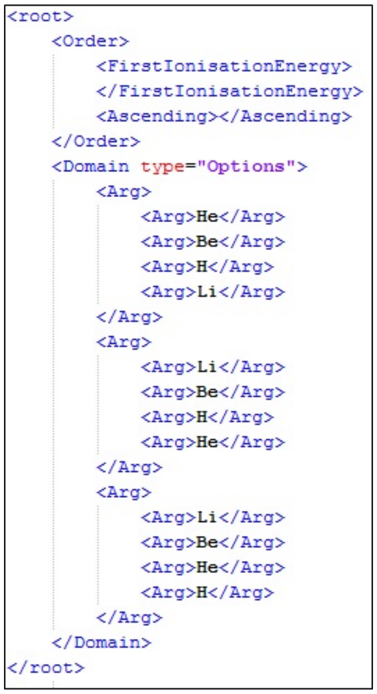
VI. Statistical Evaluation

It has been shown in the previous sections, how, after modeling the periodic table in a prolog database, the logical formulation of a question are solved by our system. In this section, we present an intuitive evaluation metric



```xml
<root>
    <Trend>
        <Leaf>x</Leaf>
        <FirstIonisationEnergy>
        </FirstIonisationEnergy>
        <Decreases></Decreases>
    </Trend>
    <Domain type="Options" variable="x">
        <Arg>Up</Arg>
        <Arg>Left</Arg>
        <Arg>Down</Arg>
        <Arg>Right</Arg>
    </Domain>
</root>
```

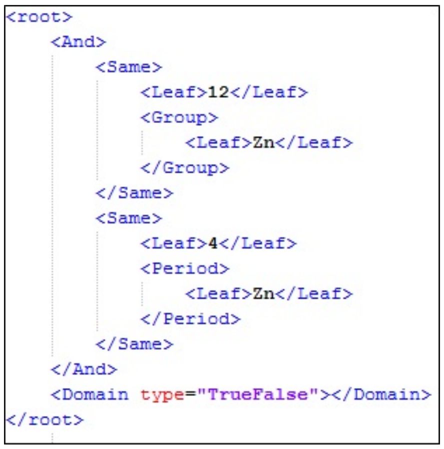
Fig. 10: XML representation of Example 8



```xml
<root>
    <Order>
        <FirstIonisationEnergy>
        </FirstIonisationEnergy>
        <Ascending></Ascending>
    </Order>
    <Domain type="Options">
        <Arg>
            <Arg>He</Arg>
            <Arg>Be</Arg>
            <Arg>H</Arg>
            <Arg>Li</Arg>
        </Arg>
        <Arg>
            <Arg>Li</Arg>
            <Arg>Be</Arg>
            <Arg>H</Arg>
            <Arg>He</Arg>
        </Arg>
        <Arg>
            <Arg>Li</Arg>
            <Arg>Be</Arg>
            <Arg>He</Arg>
            <Arg>H</Arg>
        </Arg>
    </Domain>
</root>
```

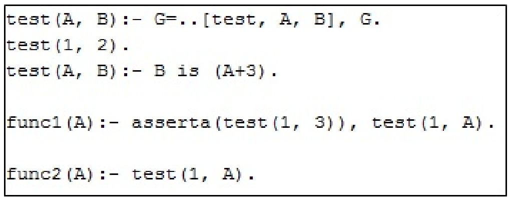
Fig. 11: XML representation of Example 9

- the number of question solved by the system versus the number of predicate rules defined and used by the system. If the latter is of the same order as the number of questions solved then the system is in a way modeling each question into the prolog database. This is extremely inefficient and not scalable if we are to solve a thousand questions in future. However, this is not the case with our system. We have a collection of 100 questions (can be viewed at *http://www2.cse.iitk.ac.in/karkare/propsolve/BTP/BenchMark.xlsx*) of Periodic table taken from *TMH Tata McGraw-Hill Education Book - Chemistry for 11*. The current system is able to solve 70 out of these 100 questions i.e. approximately 70%. Also, the number of predicate rules present in the prolog database is 20. It clearly shows that the questions are not specifically modeled into the database and a general trend/template is identified among the questions.



```
<root>
    <And>
        <Same>
            <Leaf>12</Leaf>
            <Group>
                <Leaf>Zn</Leaf>
            </Group>
        </Same>
        <Same>
            <Leaf>4</Leaf>
            <Period>
                <Leaf>Zn</Leaf>
            </Period>
        </Same>
    </And>
    <Domain type="TrueFalse"></Domain>
</root>
```

Fig. 12: XML representation of Example 10



```
test(A, B):- G=..[test, A, B], G.
test(1, 2).
test(A, B):- B is (A+3).

func1(A):- asserta(test(1, 3)), test(1, A).

func2(A):- test(1, A).
```

Fig. 13: Dynamic Calling and Asserting of a Predicate

The accuracy, 70%, of our system can be attributed to several factors. Firstly, we have a limited set of predicates which are currently modeled into the prolog database. The questions which are not solved, do not belong to the existing templates and their predicates have not yet been modeled. Thus, as the number of rules and the templates will increase, the number of solvable questions shall increase. Secondly, we have not yet collected a vast database. The set of 100 questions represents the first 100 out of about 500 questions in the Periodic Table Chapter of the TMH book. Hence this set is not representative of the whole domain of questions available in the Periodic Table. The evaluation gives a general idea about the efficiency of the system and its capability in solving questions with proper reasoning.

Another evaluation metric that we can use the gauge the efficacy of our approach is the *proof length* of the solution. Some proofs are long and some short. Often it happens that the long proofs are much easier for a student to understand as compared to the compact short proofs. Hence *proof length* combined with the *ease of understanding* can be used a evaluation metric of how efficient our system is in providing explanation to the problems.

The system generates proofs for the solutions by giving bindings to bound variables. This process is self-terminating and the system will either generate the proof for the solution or prompt the user with no solution. This is attributed to the fact that special attention has been paid while formulating the predicate rules in the prolog database. Presence of cycles among the predicate may cause the system to loop forever, hence detecting and removing such cycles is important. If there is no cycle present, then the system will eventually follow the path along every branch of a predicate rule. In case, the none of the clauses present in the predicate rules is satisfied then the system looks at the actual values of the predicate values. This is unavoidable since and indicates that the knowledge of the student is not sufficient to solve the problem and actual values are required.

VII. Conclusion and Future Work

In this paper, we present the prototype of an automated education system. We introduce a dynamic component in such learning systems which are primarily static in nature. We embed certain basic facts and rules in the system which it uses to logically solve complex problems and present the user with a coherent explanation. The statistical evaluation of our system shows that it provides logically correct proofs for solutions to the problems and contribute to a student's knowledge in a much better way as compared to the existing systems.

Besides the problem solving approach proposed in this paper, we plan to achieve the following as a part of the B.Tech Project.

- *Derivation Tree*: We wish to improve the way in which the system prints the derivation path to a particular solution. A trivial strategy could to include print statements at the end of each rule. But this techniques suffers from an additional display of those paths which are cut off from the actual derivation tree. We plan to keep record of the path followed up to a predicate and maintain the further path only if the predicate is satisfied.
- *Hint Generation*: Since the system models the actual thinking process which happens during solving a problem, the system is capable of generating step-wise explanation and give hints to students at a intermediate steps rather than just giving the final answer. The system solves the problem and at the same time generates many parallel solutions. We plan to represent the explanation as a tree with an option to the student to expand the node of the tree to get further explanation.
- *Problem Generation*: Since the system is based on few basic templates and conjunction operators with an underlying factual and relational database, it will also be possible to generate new problems based on a combination of existing templates. The system can consider the depth of an explanation as a heuristic to measure the hardness of the problem. Also, some predicates may be assigned an initial hardness score and the overall hardness of the problem may be derived from the hardness of the component predicates used in its explanation. Further, the hardness may be used to generate problems of varying difficulty.

- *Future Interactive System*: Modeling the initial knowledge of the student. Instead of traversing all possible paths for a particular solution, a good idea is to include the knowledge which the student already has. Using these facts as the base facts, the system can be optimized to find only the explanations relevant to the student.
  - We can bias the system to choose the rules preferable to a student by asserting the base facts in a particular path (explanation) to a student.
  - We need to *Assert* the basic facts about the given elements (*atomic number, same period*, and so on, just the same way as students would assume before tackling the problem).
- *Assertions*: Assuming certain basic knowledge of the students with respect to the questions (eg - *atomic number, valence electrons*). We can assert these basic facts to limit the derivation tree and not reach the database values in solving the problem.
- *Exceptions*: We plan to model the system to handle the exceptions.
- *Handle Conflicting Rules*: It might be a possibility that two rules give opposite outcomes to a particular. In this situation, a weight metric needs to be defined which can compute the solution based on the weights.
- *Similarity Metric*: Currently, all the properties values are either exactly same or different. There must exist a similarity metric between any two values. This will vary within an ϵ range.

References

[1] http://yieldprolog.sourceforge.net/.

[2] http://msdn.microsoft.com/en-us/library/9k7k7cf0(v=vs.80).aspx.