**Student Tracking System:**

**A Brief Documentation**

**Informatics Lab**

**November 23, 2014**

**Prolog Expert System:**

Based on the application needed to track students- in an attempt to study their respective progress-and the functionality of the expert systems, we found out that there is a good match. For this, Prolog, a rule-based expert system, was chosen to be implemented. In particular, we designed a framework implementing a set of rules that can check/examine if a student satisfies the requirements associated with a particular degree program.

**The Rules Format and the Requirement Hierarchy Structure:**

The rules are intended to be designed in a way that fits the hierarchical structure of the requirement. The basic requirement unit is the *course* (a course can be a *class* or a *lab*). A set of *courses* can satisfy a requirement. For example, *‘Social Behavior‘* is one of the requirements that are satisfied by a set of *classes* and *labs*. These *classes* and *labs* can be inbounded under the ‘*Social Behavior’* requirement class (we should differentiate between the class as a hierarchical level and the class as a course). Thus to examine the *satisfibility* of the requirement at different class levels, different types of rules should be implemented. For example, we might want to know if the student has satisfied (or not) the ‘*Social Behavior’* requirement regardless of the names of the *classes* or *labs* associated with this requirement. In addition, we might want to know more details about the names and types (*class* or *lab*) of the courses taken even if the student did not satisfy the ‘*Social Behavior’* requirement (for example, a student might have taken 2 courses out of 3 required). Such types of *satisfibility* levels (or we can call it *abstraction* level) can be achieved with different rules format. In other words, we should design the syntax of the rules in a hierarchical structure that takes into consideration the different levels of abstraction (see Figure 1). One way to do this is by defining the syntax of the head of the Prolog rules as following:

*satisfy Field1 Field2 (Field3,….,..) :-*

Field1: The name of the degree program/department/college/institution. Field1 should be abbreviated w.r.t “Reference\_DegreeProgram\_Abbreviation” document.

Field2: The element in the Field1 under question. It can be a ‘Requirement’, ‘Course’, etc.

Field3: The name of Field2. For example, ‘MATH 162’ is a name of a ‘Course’; ‘Humanities’ is a name of a ‘Requirement’.

Lab

Class

Fine Arts

Foreign Language

Physical Natural

Social Behavior

Math

English

Core requirement

Completeness

requirement

EE requirement

**Figure 1: Requirement hierarchical structure**

**The Names of the Requirement Classes:**

The names of the classes at different levels should be unified. The names are illustrated as following:

1. Names of classes at level 0: ‘Course’ (class or lab)
2. Names of classes at level 1: ‘Humanities’,’ Socialbehavior’, ‘Physicalnatural’, ‘English’, ‘Mathematics’,’Finearts’, ‘Foreignlanguage’, ‘Trackelective’, ‘Technicalelective’, ‘Completeness’, etc. (we will add more when needed)
3. Names of classes at level 2: ‘Core’, ‘Group’, etc.

**Rules’ Syntax and Applications**

The syntax of the rules depends on the type of application needed. For example, if the application requires only examining if a student has satisfied his/her degree plan excluding any details about the courses taken, then the rules can be implemented with a high level of abstraction. This can be defined as a decision problem where the solution is either YES or NO (satisfied or not satisfied). Moreover, the application might require more information about the name of the ‘Course’ or ‘Requirement’ taken even if the student has not satisfied his/her degree plan. Then rules with low abstraction level should be implemented. The following examples will elaborate more:

1. Assume it is required to know if a student has satisfied his core requirements regardless of the core type (i.e. humanities, social behavior, etc.). Then we can implement a rule with high abstraction level to examine that:

satisfyRequirement(Core,…,..) :-

1. Assume now it is required to know if an EE student has satisfied his ‘*Completeness*’ requirements. Then we should implement a rule with a lower abstraction level (compared to the example of part (a)):

satisfyEERequirement(‘Completeness’,…,..) :-

1. Assume now it is required to know if a student has satisfied his *‘MATH 162’* course. Then we should implement a rule with a low abstraction level:

satisfCourse(‘MATH 162’,…,…) :-

N.B. The syntax of the rules in these examples follows the format illustrated previously.