

FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCES

EXPERT SYSTEM RULESET PORTABILITY USING THE
LANGUAGE ABSTRACTION FOR RULE-BASED KNOWLEDGE-
SYSTEMS (LARK) ENGINE

By

KENNETH LLOYD AYERS III

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Fall Semester, 2008

The members of the Committee approve the Thesis of Kenneth Lloyd Ayers III defended on October 28th, 2008.

R. C. Lacher
Professor Directing Thesis

Daniel G. Schwartz
Outside Committee Member

Sara F. Stoecklin
Committee Member

Approved:

David Whalley, Chair, Department of Computer Science

The Office of Graduate Studies has verified and approved the above named committee members.

For Amy. My one constant in a world of variables.

ACKNOWLEDGEMENTS

I must thank Dr. R. C. Lacher for serving as my adviser, and even more for being an excellent instructor over the years. There is something to be said for a professor who can easily explain algorithmic runtimes, and make it not just interesting, but compelling. Dr. Sara Stoecklin has my appreciation for teaching me that software engineering is a process, not a program, and for patiently fielding my questions in class. I am indebted to Dr. Stephen Leach for introducing me to expert systems and planting the seed that inspired this thesis, as well as Dr. Daniel Schwartz for serving on my committee.

Mr. Michael Creamer played an integral part in teaching me that writing is something to be passionate about - thank you Mike for your invaluable lessons and genuine enthusiasm for this craft. I must also recognize Dr. Kelley Kline for spending her time looking over this (seemingly) endless manuscript and providing an original viewpoint, as well as some much needed editing expertise.

I can't thank my family enough for their support and encouragement. Sorry for all the late nights, Amy, and thanks for your boundless understanding, legendary patience and forgiveness for my theft of all your caffeinated drinks.

-Kenny

TABLE OF CONTENTS

List of Tables	vii
List of Figures.....	viii
Abstract.....	ix
1. EXPERT SYSTEMS.....	1
1.1 Overview	1
1.2 Production Rules and Inferencing.....	3
1.3 Brief History	4
2. RULESETS.....	6
2.1 Ruleset Portability and LARK	6
2.2 CLIPS.....	7
2.2.1 CLIPS Facts.....	7
2.2.2 CLIPS Rules	12
2.2.3 CLIPS Variables	14
2.2.4 CLIPS Deftemplate.....	18
2.2.5 CLIPS Functions.....	20
2.3 M.1.....	20
2.3.1 M.1 Facts.....	21
2.3.2 M.1 Meta-Facts	22
2.3.3 M.1 Rules	24
2.3.4 M.1 Certainty Factor Computation	28
2.3.5 M.1 Variables	29
2.4 Natural Rule Language (NRL) Constraint Language.....	31
2.4.1 NRL Referencing	31
2.4.2 NRL and Application to Lark Natural Language	33
3. RULESET PORTABILITY EFFORTS	36
3.1 OMG PRR	36
3.1.1 Production Rules and Rulesets	36
3.2 W3C Rule Interchange Format (RIF).....	37
3.3 RuleML	37
3.3.1 RuleML Hierarchy and Tags.....	37
3.3.2 RuleML Hierarchy Reduction.....	39
3.3.3 RuleML Schema Specification 0.91.....	41
3.3.4 Functional RuleML.....	42
4. LANGUAGE ABSTRACTION FOR RULE-BASED KNOWLEDGE-SYSTEMS (LARK) ENGINE.....	44

4.1 Purpose	44
4.2 Scope	44
4.3 Objectives and Success Criteria	45
4.4 Definitions, Acronyms, Abbreviations	46
4.5 Functional Requirements	46
4.6 Non-functional Requirements	49
4.7 LarkML	49
4.7.1 Specification	49
4.8 Language Mapping	52
4.8.1 Translating LarkML to CLIPS	52
4.8.2 Translating LarkML to M.1	55
4.8.3 Translating LarkML to Lark Natural Language	60
4.8.4 CLIPS Rule Parsing and Conversion to LarkML	60
4.8.5 CLIPS Fact Parsing and Conversion to LarkML	70
4.8.6 M.1 Rule Parsing and Conversion to LarkML	74
4.8.7 M.1 Fact Parsing and Conversion to LarkML	83
4.8.8 LarkML Language Mapping Considerations and Exceptions	85
4.9 LARK Engine Class Implementation	87
4.9.1 RuleSet Class	87
4.9.2 RuleTranslator Class	87
4.9.3 RuleSmasher Class	88
5. CONCLUSIONS AND FUTURE DIRECTION	90
5.1 Conclusions	90
5.2 Future Direction	90
APPENDIX A – LARKML LANGUAGE DEFINITION	91
APPENDIX B – LARK NATURAL LANGUAGE XSL	93
APPENDIX C – CLIPS XSL	96
APPENDIX D – M.1 XSL	98
APPENDIX E – LARKML COMPATIBILITY	101
REFERENCES	103
BIOGRAPHICAL SKETCH	104

LIST OF TABLES

Table 1: CLIPS Data Types	11
Table 2: Certainty Factor Computations for Positive Numbers.....	28
Table 3: Certainty Factor Computations for Negative Numbers	28
Table 4: Certainty Factor Computations for Negative and Positive Numbers	29
Table 5: NRL Syntax (Boley).....	33
Table 6: NRL If-Then Syntax (Boley).....	34
Table 7: RuleML Rule-Tag Mapping	38
Table 8: RuleML Rule Application Direction	41
Table 9: LarkML Compatibility Matrix.....	101

LIST OF FIGURES

Figure 1: RuleML Hierarchy	38
Figure 2: LARK Engine Interface.....	47
Figure 3: LARK Engine File Menu	47
Figure 4: LARK Engine Output Format Dropdown.....	48
Figure 5: LARK Engine Output Ruleset.....	48
Figure 6: RuleSet Class Diagram.....	87
Figure 7: RuleTranslator Class Diagram	88
Figure 8: RuleSmasher Class Diagram.....	89

ABSTRACT

This thesis describes the Language Abstraction for Rule-based Knowledge-systems (LARK) Engine. The goal of this engine is to process various expert system rulesets and generate the required semantics for multiple production systems – thus creating true portability for expert systems such as M.1 and CLIPS. Specifically, LARK provides ruleset translation from Lark Markup Language (LarkML, an XML language defined herein), to CLIPS and M.1 expert system rules, as well as an implementation of rules written in natural language. LARK also demonstrates the ability to parse and convert basic CLIPS and M.1 rules to LarkML.

In addition to describing the LARK Engine, this thesis also outlines an overview of significant expert system, UML, and business ruleset portability efforts. Ruleset portability is quickly evolving as the combined efforts of many organizations push the technology forward. Significant ruleset portability efforts include the Production Rule Representation (PRR) as defined by the Object Management Group (OMG), the Rule Interchange Format (RIF) as specified by W3C, the Rule Markup Language (RuleML) Initiative composed of a large group of industry and academia participants, and the Natural Rule Language (NRL), an effort sponsored by SourceForge.

1. EXPERT SYSTEMS

“Professor Edward Feigenbaum of Stanford University, an early pioneer of expert systems technology, has defined an expert system as ‘...an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution’.” (Giarratano and Riley 5)

Chapter one of this thesis provides an overview of expert systems. This is essential for understanding both the M.1 and CLIPS expert systems, which are explained in detail in chapter two. Chapter three serves to outline the ruleset portability efforts of the technology community. Before reading chapter four, which describes the LARK Engine, the reader should have an understanding of the importance of both M.1 and CLIPS rulesets, as well as the flexibility that ruleset portability brings to the expert system programming paradigm. Chapter five provides some concluding remarks and sets a vision for the future development efforts of the LARK Engine.

1.1 Overview

Expert systems exist to supply automated solutions to problems in response to users' input of facts or information (Giarratano and Riley 6). Typically these systems solve problems that are outside the solution space of algorithms. Expert systems emulate the problem solving abilities of human experts by harnessing a knowledge base of rules that are chained together in search of solutions. The solution discovery process is accomplished by using an inference engine to process facts and information supplied by users, facilitated by the knowledge-base. The knowledge-base of an expert system is the abstraction of expert level knowledge from the problem domain the expert system was built to address.

An essential part of the definition of an expert system is a well-defined problem domain. Expert systems have not proven successful as general problem solving tools; their true power is demonstrated in their successful application as problem-solving tools for discrete problem-spaces. If a given problem or problem-space can be solved by an algorithm, an expert system is an inappropriate solution. Expert systems are best applied to ill-structured problems for which there are no efficient conventional algorithms and programming solutions such as medical diagnostic systems and warfare scenario advisement. (Giarratano and Riley 23)

Expert Systems are pervasive. From applications in data mining, facial-recognition, discovery systems, medical diagnostic systems, agricultural, Federal employment laws, and business law, expert systems are used in every sector of modern business, government, and academia. (Giarratano and Riley 4, 804, 805) (PC AI Artificial Intelligence)

The automation of solution discovery processes yields distinct advantages over the use of human experts. Expert systems are not subject to the biological constraints that limit human experts. They will never experience fatigue, sickness, or death – all significant causes of expertise unavailability. Furthermore, expert systems can be placed

in dangerous environments where human experts should not be placed, such as robotic factories or warzones. These systems produce reliable, consistent, comprehensive results and provide the stability of a permanent solution that is always as good as the knowledge-base originally provided (hardware failures and software bugs notwithstanding). Expert systems have the ability to produce explanations of the steps taken to produce a solution to a given problem, thus allowing for the review of how the solution was made and increasing assurance of solution accuracy. The ability of these systems to describe the rules used to make a conclusion is known as the **explanation facility**. Finally, because expert systems are the automated expression of knowledge, assuming an accurate and relevant knowledge-base, their solution production is bound only by the algorithms used within the inference engine and the available computing power. This means that in terms of solution production per unit time, expert systems out-produce human experts by orders of magnitude. The production level an expert system is only limited by the software and hardware architectures' ability to serve multiple users simultaneously, while a single human expert can only solve a single problem at a time. Giarratano and Riley posit that "In a very real sense, an expert system is the mass production of expertise." (Giarratano and Riley 8, 9)

The development of an expert system can be simplified into three essential steps: the expertise and knowledge acquisition process, encoding the garnered expertise into an expert system, and a system review by the expert(s) of the problem domain being addressed.

The initial step of expertise and knowledge acquisition consists of a knowledge engineer working with one or more experts to discover and document the full breadth and depth of their expertise for a given problem domain. The knowledge engineer then must encode the abstracted expertise into the expert system. This second step builds the knowledge-base of the expert system, the set of rules that when combined with the inference engine and information from users, allows the system to produce solutions for a problem domain. Finally the subject matter expert reviews the prototype system and gives an evaluation of both the accuracy of the solutions and the explanations provided by the explanation facility. It should be noted that the amount of information an expert or experts have about a given problem domain (known as the knowledge domain), is almost invariably smaller than said problem domain. The exception to this rule is in problem domains that are so small as to be of little use in practical applications. (Giarratano and Riley 7, 9, 10)

The knowledge acquisition process requires significant effort. In the past, when knowledge engineers have converted the abstracted expertise into production expert systems, they have applied to only a specific (often single) software architecture. For example, when a ruleset is developed for the M.I expert system shell, the rule syntax only works for that specific program. Given a functional ruleset (knowledge-base) portability standard, this expertise can become software architecture agnostic. This is an essential extension of functionality, because if rulesets can become portable, all the effort expended in garnering expertise and creating rules can now be used in multiple expert systems.

The aim of this thesis is to briefly document the current efforts to develop ruleset portability and demonstrate a mechanism by which a language portability tool can create ruleset implementations that works with different software architectures. The Language-

Abstraction for Rule-based Knowledge-systems (LARK) Engine demonstrates the ability to parse and convert expert system rulesets in a variety of ways, and specifically deals with the M.1 and CLIPS expert system shells. M.1 and CLIPS rules may be parsed and converted to LarkML, an XML-based language developed to allow for ruleset portability. Additionally, the LARK Engine demonstrates the ability to transform LarkML rules via XSL stylesheets to M.1, CLIPS, and a simplified implementation of natural language rules: Lark Natural Language (LNL is defined in Appendix B).

1.2 Production Rules and Inferencing

The production rules that define the knowledge-base of an expert system are expressed in an **If - Then** format. The **If** portion of the production rule consists of zero or more conditions (also known as patterns or the antecedent) that must be met before the rule is activated. Also known as the **Left Hand Side** (LHS) of the production rule, the conditional patterns are the mechanism by which the inference engine is able to identify temporally relevant rules. The **Then** portion of the rule, also known as the **Right Hand Side** (RHS) consists of one or more actions to take when the **If** portion of the rule is matched by the inference engine. When the inference engine matches the antecedent, the rules are placed on the **agenda**, which is a prioritized list of rules and actions which must be executed. The execution of these actions to match current facts with patterns is referred to as **activation**. (Giarratano and Riley 30) A few simple examples of production rules are presented below:

RuleName: Eat

IF

Philosopher is hungry.

THEN

Eat.

Tip waitress.

RuleName: Think

IF

Philosopher is awake.

Philosopher is at work.

THEN

Think.

RuleName: Discovery

IF

Philosopher is thinking.

THEN

Assert: Philosopher has idea.

RuleName: Write

IF

Philosopher has idea.

THEN

Write.

RuleName: Breathe

IF

THEN

Breathe.

Rule **Eat** demonstrates a single **If** conditional statement that results in the execution of two actions – “Eat.” and “Tip waitress.” Rule **Think** demonstrates multiple conditions that must be met before the Think action is executed. The **Discovery** rule demonstrates the ability of a rule to create a new fact (or conditional pattern): “Philosopher has idea.” This action is referred to as an assertion. When the **Write** rule encounters this new conditional pattern and the inference engine discovers the matching antecedent, the rule is added to the agenda, and the **Write** action executes. Finally the **Breathe** rule demonstrates the ability to define no **If** conditional statements to create a rule that always executes. Because there are no LHS conditions to meet, the rule is implicitly activated and the **Breathe** rule is placed on the agenda, a good thing for our philosopher.

Inference engines may operate using two different methodologies: backward and forward chaining. In forward chaining, an inference engine starts with facts or information input by the user, and attempts to chain through the production rules until a conclusion can be made. Backward chaining consists of starting with conclusive information, and moving backwards through the rules to derive initial facts. It is correct to infer these are analogous operational methodologies, only reversed.

1.3 Brief History

The original impetus for the development of expert systems in the 1960s was the need to solve complex problems in relatively narrow problem domains. At that time, it was discovered that the size of the problem domain is in direct inverse proportion to an expert system’s successful development and implementation. As problem domains increase in size and complexity, the probability of an expert system being developed and successfully implemented decreases.

The XCON expert system built to configure DEC computer systems in the 1970s constituted the first successful commercial expert system implementation. Since the greater commercial expansion of expert systems in the 1980s, they have become pervasive in all industries where the problem space can be well defined. Users of expert systems include the U.S. Department of Safety & Labor, OSHA, the National Science Foundation, U.S. Geological Survey, NASA, Tokyo Nissan, and the Australian Taxation Office. This list is a minute sampling, but serves to demonstrate the incredible breadth of industries using expert systems. (Giarratano and Riley 805)

As problem domains grow, expert systems approach the extreme of becoming generalized problem solving engines. Generalized problem solving expert systems have been unsuccessful to date, as they lack well defined problem and solution spaces. Expert

systems built upon well defined problem domains however, have been very successful.
(Giarratano and Riley 2, 5)

2. RULESETS

For the purpose of this thesis, rulesets may be defined as the abstraction of the knowledge of the operational boundaries and functionality of a problem domain. Rulesets may consist of the rules that constitute the knowledge base of an expert system, the constraints of a UML model, or the general understanding of a business process. Rulesets may be expressed in myriad formats, including (but not limited to) expert system shells languages such as CLIPS and M.1, or they be defined in a more human readable format using the Natural Rule Language.

In the operational context of an expert system, rulesets contain the rules that constitute the knowledge-base. These rules allow the expert system to process the data and information input by users. By applying the ruleset to the data via the inferencing engine, the expert systems are able to make determinations about the data, and provide solutions to problems.

While the most basic of rules in rulesets contain the familiar **If-Then** format, the semantics of ruleset implementations vary greatly, especially as rules become more complex. This section serves to explore some of the subtleties of the M.1 and CLIPS ruleset semantics and how they relate to the conversion from LarkML to their respective ruleset implementations. The primary goal of the LARK Engine is to provide ruleset portability. That is, the ability to convert LarkML rules to multiple ruleset formats such as CLIPS, M.1, and LNL. The details in these subsections serve as a springboard for the definition and implementation of XSL stylesheet transformations from LarkML to each respective ruleset implementation.

2.1 Ruleset Portability and LARK

Ruleset portability was not originally a consideration in the first expert systems. While expert systems of all types have proliferated since the need for these systems first emerged in the 1960s, the need to exchange rulesets was not seriously recognized and addressed until the last decade. Efforts to provide ruleset portability began in earnest at this time. Significant efforts to provide software agnostic, portable rulesets include the Rule Markup Language Initiative (RuleML est. 2001), the World Wide Web Consortium's Rule Interchange Formation (W3C RIF est. 2005), and the Object Management Group's Production Rule Representation (OMG PRR est. 2002). While RuleML and RIF seek to create portability of production rules, the PRR initiative focuses primarily on creating a portability standard for business rules.

Some expert system shells such as CLIPS are portable in the sense that they may be installed on multiple operating systems, however CLIPS does not currently use rulesets that may be read by other expert system shells. For the purpose of this thesis, "portability" should be understood to mean the following: a mechanism by which one or more rules is presented in a format that can be read by multiple expert system shells for implementation to their respective knowledge bases. Because expert system shells currently use differing syntactical formats, an intermediary engine must exist to facilitate ruleset portability. To this end the LARK Engine has been developed to translate LarkML (an XML-based ruleset standard) into multiple formats that are readable by

different expert system shells. The current goal is to allow translation from this open XML standard into M.1 and CLIPS readable formats along with an LNL output that allows the rules to be more easily read and understood by non-technical viewers.

2.2 CLIPS

CLIPS is an expert system shell released in 1986, originally developed by the Software Technology Branch of NASA. CLIPS supports multiple programming paradigms: heuristic (rule-based), procedural, and object-oriented. The shell portion of CLIPS contains the fact-list (a container for all current information and data), the instance-list (all objects current being used), the knowledge-base (the active ruleset), and the inferencing engine, which is based on the Rete Algorithm. Charles Forgy developed the Rete Algorithm in part in his 1979 doctoral thesis – “On the efficient implementation of production systems.” (Forgy 25)

CLIPS uses a traditional expert system architecture, matching facts with rules through the use of an inferencing engine. CLIPS also allows for the use of the object-oriented programming model. This unique implementation can pattern match class instances with rules. Object-oriented programming in CLIPS allows for “the five generally accepted features of object-oriented programming: classes, message-handlers, abstraction, encapsulation, inheritance, polymorphism.” (CLIPS Architecture Manual vi)

This section focuses primarily on discussing the details of the heuristic mechanisms available in CLIPS, including an overview of rules, facts, basic functions, variables, and template syntax. The details contained in this section do not serve as a comprehensive reference for CLIPS. The syntax detailed below consists of the majority of what must be addressed in the conversion of a basic ruleset from LarkML to a CLIPS ruleset implementation.

2.2.1 CLIPS Facts

CLIPS facts constitute the information and data input by users as well as the results of rules that are activated during the execution of the CLIPS shell. Facts can be input manually by the user. Also, the action portion of rules that are executed can contain fact assertions. Facts are made of one or more fields with associated values. These fields can be considered slots for either variables (discussed later, in section 2.2.3) or constant values. Facts are contained inside matching opening and closing parentheses. CLIPS facts are stored in temporary memory, which can be viewed by issuing a (facts) command. Facts are entered into CLIPS via the (assert) command:

```
CLIPS> (assert (myFact))
<Fact-1>
```

Now to view our fact, we'll execute the (facts) command:

```
CLIPS> (facts)
f-0      (initial-fact)
```

```
f-1      (myFact)
For a total of 2 facts.
```

In this example, `(myFact)` is a fact with one field. Because this fact was asserted without any predefined type information CLIPS implicitly assigns a data type, however data types may be explicitly assigned using `Deftemplates`, a method for creating fact templates in CLIPS (discussed in section 2.2.4 CLIPS `Deftemplates`). By default, upon the execution of the CLIPS shell, the `(initial-fact)` fact is issued. This serves to provide a starting point for CLIPS programs. Facts may not be nested inside of other facts.

The most basic data type for a field is the **symbol** type. Symbols consist of one or more printable ASCII characters and are commonly delimited by white space (tabs, spaces, carriage-returns, and line-feeds). An example fact showing a varied diet is shown below, delimited by spaces:

```
(favorite-foods toads birds dark-chocolate)
```

This fact contains four symbol fields. It is common practice to use a symbol at the beginning of a fact to describe the information contained within – our example informs us that this is a list of `favorite-foods`. The `favorite-foods` fact is considered **ordered**. The significance of this is that another fact showing the same favorite foods in a different order is not considered the same fact in CLIPS:

```
CLIPS> (assert (favorite-foods toads birds dark-chocolate))
<Fact-1>
CLIPS> (assert (favorite-foods toads birds dark-chocolate))
FALSE
CLIPS> (assert (favorite-foods birds toads dark-chocolate))
<Fact-2>
CLIPS> (facts)
f-0      (initial-fact)
f-1      (favorite-foods toads birds dark-chocolate)
f-2      (favorite-foods birds toads dark-chocolate)
For a total of 3 facts.
```

In the preceding example, we see that issuing the same `(assert)` command twice in a row returns a `FALSE` response upon the second issuance. This is the mechanism by which CLIPS informs us that we already have an equivalent fact existing in the fact-list. When we reorder the favorite food list CLIPS allows the assertion command because as an ordered fact, it is considered different.

Special characters in CLIPS act as delimiters and thus cannot be used in symbols, or have restricted usage in symbols. White space, parentheses, the dollar sign “\$”, pipe symbol “|”, opening carat “<”, tilde “~”, and semicolon “;” act as delimiters. The less-than carat “<” can start a symbol, while the question mark “?” and dollar-sign question-mark combination “\$?” can only be used inside a symbol. The latter two character combinations (? and \$?) denote the beginning of a variable declaration (discussed in

section 2.2.3). Some delimiters cannot be used in any part of a symbol, these include: “&”, “|”, and “~”.

CLIPS contains two number data types: integers and double-precision floating point numbers (commonly referred to as **floats**). Integers are non-decimal whole numbers with a range that varies according to the architecture of the machine running CLIPS. Integers scale from -2^{n-1} to $2^{n-1}-1$ where n is the number of bits used by the operating system. 32-bit integers scale from -2,147,483,648 to 2,147,483,647 (Giarratano 10). Floats are double-precision; they occupy two adjacent memory locations in an operating system, thus 32-bit operating systems support 64-bit precision floats while 64-bit operating systems use 128-bit precision floats. Both number types must be preceded by a symbol within the body of the fact. Below we first assert a number by itself, which CLIPS rejects, and then we put a symbol in front of our number which is accepted:

```
CLIPS> (assert (4))
```

```
[PRNTUTIL2] Syntax Error: Check appropriate syntax for first field of a RHS pattern.
```

```
CLIPS> (assert (myNumber 4))  
<Fact-1>
```

The **string** data type is used in CLIPS, and starts and ends with double quotes. Strings may contain zero or more of any type of ASCII characters, including the delimiters that are restricted for use within symbols. Like the number data types, strings must be preceded by a symbol:

```
CLIPS> (assert ("chocolate"))
```

```
[PRNTUTIL2] Syntax Error: Check appropriate syntax for first field of a RHS pattern.
```

```
CLIPS> (assert (health-food "chocolate"))  
<Fact-1>
```

With the addition of the symbol `health-food`, we assert that “chocolate” is a `health-food` fact.

The ability to identify the location of facts in the fact-list during run time is a powerful tool in CLIPS. This is done with the use of the **fact-address**. To view the address of a fact, execute the `(facts)` command:

```
CLIPS> (assert (horse)(cat)(moose)(mousse))  
<Fact-4>
```

```
CLIPS> (facts)  
f-0      (initial-fact)  
f-1      (horse)  
f-2      (cat)  
f-3      (moose)  
f-4      (mousse)
```

For a total of 5 facts.

Our (horse) fact is at fact-address 1, while our (cat), (moose), and (mousse) facts are at fact-addresses 2, 3, and 4, respectively. To manipulate our facts using their addresses, we can use the (retract) command. Below facts at addresses 1 and 4 are retracted:

```
CLIPS> (retract 1 4)
CLIPS> (facts)
f-0      (initial-fact)
f-2      (cat)
f-3      (moose)
For a total of 3 facts.
```

An important note is that the remaining fact addresses do not change upon the removal of facts preceding their address.

Classes are a data type abstraction used in the object-oriented programming paradigm. **Instance-names** and **instance-addresses** are both used with classes in CLIPS, the former being analogous to a handle name to manipulate an instance of a class. The latter is used in a similar fashion as the fact-addresses to reference instances. For the purpose of this thesis, the instance related variables are of deprecated importance as we are focusing mainly on the basic elements required for the heuristic programming paradigm. Similarly the **external-address** data type is not used in this paper.

Table 1: CLIPS Data Types

Data Type	Examples
symbol	cat c@t cat\$ <at
string	“cat” “” “>◇◇◇◇<” “\$?345”
fact-address	N where N = the number assigned to the fact in the fact-list
instance-name	class instance name, non-applicable
instance-address	class instance address, non-applicable
external-address	non-applicable
float	7.0 30.00
integer	1 44

In CLIPS some facts need to be issued upon the execution of the (reset) command. For example, by default the (initial-fact) is always present after a (reset). The (deffacts) (read as “define facts”) command can be used to create additional default facts:

```
CLIPS> (deffacts cat "Cat Facts"  
        (cat-legs 4)  
        (cat-tails 2))
```

To view all the (deffacts), we issue the (list-deffacts) command:

```
CLIPS> (list-deffacts)  
initial-fact  
cat  
For a total of 2 deffacts.
```

We see that our cat deffact is now in the list of deffacts, along with the initial-fact deffact. To demonstrate the value of our new cat deffact, we assert a few facts, then execute the (reset) command which clears out the fact-list, then reassert all the deffacts:

```

CLIPS> (assert (dog-legs 3)(dog-tails 1))
<Fact-4>
CLIPS> (facts)
f-0      (initial-fact)
f-1      (cat-legs 4)
f-2      (cat-tails 2)
f-3      (dog-legs 3)
f-4      (dog-tails 1)
For a total of 5 facts.
CLIPS> (reset)
CLIPS> (facts)
f-0      (initial-fact)
f-1      (cat-legs 4)
f-2      (cat-tails 2)
For a total of 3 facts.

```

Upon the issuance of the `(reset)` command, all facts are flushed, and the deffacts are reasserted. The `dog` facts are now gone from our fact list. Of note, the `(clear)` command flushes all facts, rules, and deffacts. Also the `(undefacts)` (undefined facts) command can be used to remove deffacts from the deffacts list.

2.2.2 CLIPS Rules

The CLIPS knowledge base consists of rules that are matched to facts during execution. The Rete algorithm-based inference engine matches facts to the left hand side (LHS) of rule patterns, which contain the rule conditions. Upon a match of the LHS pattern, the rule is added to the agenda which will execute the right hand side (RHS) actions of the rules in order of **salience** (execution priority) which is either explicitly defined in the rule body or implicitly by the CLIPS shell. Because the CLIPS processor executes activated rules according to their ordering on the agenda, salience has a direct effect on the order in which rules are executed. CLIPS rules follow the traditional **if (conditions) then (actions)** format. The `defrule` (read as “define rule”) command is used to create new rules, using the following pattern (note the semi-colon delimiter is used to signify the beginning of a CLIPS comment):

```

(defrule [module name]::[rule name] "[rule comment]"
  [pattern(s) 0...N]                ; LHS pattern(s)
=>                                  ; "then"
  [action(s) 1...N])                ; RHS action(s)

```

We see that the module and rule name follows the `defrule` command, which can be followed optionally by a rule comment describing some information about the rule. The `MAIN` module is the default container, however new modules may be declared using the `(defmodule [module name])` (read as “define module”) command.

Next we define the LHS of the rule, the patterns that must be matched for the rule to be activated. In CLIPS we have the option of defining between zero and N patterns. CLIPS specifically recognizes rules without any LHS patterns, and assigns the (initial-fact) pattern automatically. Because of that feature, any rule with zero patterns in the LHS is automatically sent to the agenda for execution upon the issuing of the (run) command, assuming the default (initial-fact) deffact has not be modified or removed. The “=>” keyword denotes that the LHS of the rule is done being defined, and the RHS begins. Rules must have at least one action defined, otherwise there is nothing to execute if the LHS patterns are matched. With this framework in mind, we now define our example rule:

```
CLIPS>(defrule MAIN::barbecueHand "BBQ Hand Safety"
                                           ;ruleheader
      (bbq-temp hot)                       ; pattern 1
      (hand-position on-bbq)               ; pattern 2
=>
      (assert (hand-action remove-from-bbq))) ; action 1
```

The defrule command is followed by the rule header: the module MAIN, the rule name and rule comment. Our LHS has two patterns, and is followed by one action. Parenthetical notation is used to wrap each element in the rule.

It is a recommended practice to not rely upon manual salience declarations in heuristic systems as you can quickly turn what is supposed to be an unstructured rule-based system into a procedural program. The inference engine should be allowed to make the salience declarations implicitly, however manual declarations are possible. Salience declarations can be made in the range of -10,000 to 10,000, where the larger number denotes a higher order of precedence. The salience declaration must precede the pattern definitions in the rule:

```
CLIPS> (defrule MAIN::barbecueBeer "BBQ Beer Operation"
        (declare (salience 100))
        (bbq-temp hot)
        (hand-position ~on-beer)
=>
        (assert (hand-action place-on-beer)))
```

Our barbecueBeer rule salience is declared as 100, thus its execution order once it is activated and placed upon the agenda will precede all other rules with a salience less than 100. By default CLIPS implicitly gives each activated rule not containing an explicit salience declaration a salience of 0. The second pattern of this rule contains the tilde, a **connective constraint**, which can be read as “not.” Thus, this pattern is matched when hand-position is not on-beer. The ability to use connective constraints like “not” and others described in the proceeding section (2.2.3 CLIP Variables) can save a good deal of superfluous LHS pattern definitions and rule definitions.

Rules may be undefined via the use of the `(undefrule)` (undefine rule) command. To remove our `barbecueBeer` rule, we would issue the following command: `(undefrule barbecueBeer)`.

2.2.3 CLIPS Variables

The focus of this section is CLIPS variables that bind to fact slots and fact addresses. This focus does not address additional types such as instance addresses. The purpose of this section is to detail solely the mechanisms needed to convert an abstract ruleset to the heuristic ruled-based programming model available in CLIPS. Object oriented mechanisms are not addressed.

Single-slot variables are declared in rules using the following syntax: `?[variable name]`, where `[variable name]` is a symbol consisting of one or more printable ASCII characters. Variables bound to fact slots allow for functionality such as printing out all or part of a rule, as well as identifying specific pattern sections for additional processing. To bind a variable to a fact slot, the `CatName` rule uses the `?catName` variable:

```
CLIPS> (defrule CatName "Print cat name"
        (myCat ?catName)
=>
        (printout t ?catName " is my cat." crlf))
```

The CLIPS inferencing engine will match the `(myCat ?catName)` pattern with all facts that begin with the symbol `myCat` and contain one additional slot value:

```
CLIPS> (assert (myCat "Horse")(myCat 35)(myCat Beer))
<Fact-3>
CLIPS> (run)
Beer is my cat.
35 is my cat.
Horse is my cat.
```

Three facts were asserted using a string, integer, and a symbol, respectively. Each was bound to the `?catName` variable at runtime as our `CatName` LHS condition matched each fact pattern. After the inferencing engine recognized the matches, the rule action was placed upon the agenda and then executed three times – once for each match.

Fact-addresses are used to manipulate facts in the fact-list. Fact-addresses are assigned to variables, and can then be modified according to the RHS rule actions. The `<-` symbol is used to bind patterns' fact-addresses to variables:

```
CLIPS> (defrule RemoveCat
        ?DoomedCat <- (myCat ?PoorKitty)
=>
        (printout t "Goodbye " ?PoorKitty crlf);
        (retract ?DoomedCat))
```

Upon execution, this rule is matched to all facts containing the `myCat` symbol followed by exactly one blank slot. The fact-address of each matched pattern is assigned to the `?DoomedCat` variable. The RHS actions consist of printing out a message to terminal window, and then retracting the fact address of the matched LHS pattern. To demonstrate the functionality, the `(watch facts)` function will be enabled. This function prints notification of fact assertions and retractions to the terminal:

```
CLIPS> (watch facts)
CLIPS> (assert (myCat Mousse)(myCat BurBerry))
==> f-1      (myCat Mousse)
==> f-2      (myCat BurBerry)
<Fact-2>
CLIPS> (run)
Goodbye BurBerry
<== f-2      (myCat BurBerry)
Goodbye Mousse
<== f-1      (myCat Mousse)
CLIPS>
```

Upon assertion of our `myCat` facts, CLIPS notifies us that they have been added to the fact list. When the CLIPS `(run)` command is executed, the `RemoveCat` rule matches both of the fact patterns, prints notification to the terminal of the match, and the facts are removed from the fact-list.

The wildcard character `?` does not necessarily need to be used to assign a variable, it can be used solely to facilitate pattern matching. For example to match all cats with exactly two names, the following rule would be used:

```
CLIPS> (defrule RemoveTwoNameCats
        ?DoomedCat <- (myCat ? ?)
=>
        (retract ?DoomedCat))
```

Facts to demonstrate slot number matching:

```
CLIPS> (assert (myCat Maynard James)(myCat Discrete
Math)(myCat Bill))
```

Execution:

```
CLIPS> (run)
<== f-5      (myCat Discrete Math)
<== f-4      (myCat Maynard James)
CLIPS> (facts)
f-0      (initial-fact)
f-6      (myCat Bill)
```

Because the LHS pattern was built to match exactly two slots after the myCat symbol, the (myCat Discrete Math) and (myCat Maynard James) facts were matched and retracted while the (myCat Bill) fact was not.

CLIPS allows for multi-field wildcards using the \$? keyword. Multi-field wildcards can be assigned to variables, or used solely for pattern matching. When used on the RHS of a rule, the \$ character is removed:

```
CLIPS> (defrule RemoveAllCats
        ?DoomedCat <- (myCat $?PoorKitty)
=>
        (printout t "GoodBye " ?PoorKitty crlf)
        (retract ?DoomedCat))

CLIPS> (assert (myCat Maynard James)(myCat Discrete
Math)(myCat Bill))

CLIPS> (run)
GoodBye (Bill)
<== f-3      (myCat Bill)
GoodBye (Discrete Math)
<== f-2      (myCat Discrete Math)
GoodBye (Maynard James)
<== f-1      (myCat Maynard James)
```

The \$?PoorKitty wildcard variable matches all patterns starting with the myCat symbol, thus the RHS print and retraction actions are executed for each of the three facts. Note that for the LHS, we use \$?PoorKitty for variable assignment, whereas the RHS action uses ?PoorKitty the same variable.

Connective constraints may be used to add Boolean logic to pattern matching in CLIPS. The tilde “~” is equivalent to the “not” Boolean operator, the pipe “|” is equivalent to the “or” operator, and finally the ampersand “&” is used to bind variables used in conjunction with the previously listed operators. The “~” operator only affects the preceding slot, while “|” modifies both the previous and preceding slots.

```
CLIPS> (defrule BuryProperCat
        (Cat ?CatName ~dead)
=>
        (assert (Do-not-bury ?CatName)))

CLIPS> (defrule ReviveCat
        (Cat ?CatName comatose|sick)
=>
        (assert (Take-to-Vet ?CatName)))

CLIPS> (assert (Cat Dexter sick)(Cat Frankenstein
comatose)(Cat Bill dead))
```

```

CLIPS> (run)
==> f-4      (Do-not-bury Frankenstein)
==> f-5      (Take-to-Vet Frankenstein)
==> f-6      (Do-not-bury Dexter)
==> f-7      (Take-to-Vet Dexter)

```

All Cat patterns not containing the dead symbol are matched to the LHS of BuryProperCat, which then asserts the proper Do-not-bury fact for the ?CatName variable. The ReviveCat rule matches all Cat patterns containing either comatose or sick symbols, and asserts a Take-to-Vet fact for each.

Finally, the ampersand binding constraint can be used to grab variables being compared with the other connective constraints:

```

CLIPS> (defrule CatAnalysis
        (Cat ?CatName ?Issue&~dead)
=>
        (printout t ?CatName " is currently " ?Issue " and
is not dead." crlf))

```

```

CLIPS> (assert (Cat Dexter sick)(Cat Frankenstein
comatose)(Cat Bill dead))

```

```

CLIPS> (run)
Frankenstein is currently comatose and is not dead.
Dexter is currently sick and is not dead.

```

The ampersand constraint binds the slot proceeding the ?CatName variable to the ?Issue variable. The “&” allows the binding of variables when the other operators are used. Similar syntax is used for the “or” Boolean function:

```

CLIPS> (defrule CatAvailability
        (CatStatus ?CatName ?Status&Married|Dating)
=>
        (printout t ?CatName " is " ?Status " and is thus
unavailable" crlf))

```

```

CLIPS> (assert (CatStatus Jack Married)(CatStatus Dextah
Dating)(CatStatus Murry Single))
<Fact-3>

```

```

CLIPS> (run)
Dextah is Dating and is thus unavailable
Jack is Married and is thus unavailable

```

Additional variable functionality is available in the CLIPS expert system shell, however this overview addresses what is needed to convert a simple LarkML ruleset to a functional implementation of CLIPS rules.

2.2.4 CLIPS Deftemplate

CLIPS allows for fact templates to be defined via the `(deftemplate)` (read as “define template”) command. Fact templates ensure that the proper data type is associated with variable slots. To declare a deftemplate, the command is issued, followed by the name, a comment, and then 1 to N slot definitions:

```
(deftemplate [deftemplate name] "[deftemplate comment]"
  ([slot type] [slot 1 name]
   (type [slot data type])
   (default [default slot value])
   (range [N])|(allowed-[data type] [allowed values])
   ...
  ([slot type] [slot 1 name]
   (type [slot data type])
   (default [default slot value])
   (range [M N])|(allowed-[data type] [allowed values])
```

The deftemplate name and slot names consist of one or more printable ASCII characters with the same restrictions as the CLIPS symbol data type. The deftemplate comment is a string value. The slot types may be defined as single-slots which contain exactly one value or multislots which may contain between 0 to arbitrary number N values. The default slot value allows for the definition of a value that will be assigned to the fact upon the assertion of a null slot value. The **number** data type may be used to designate either an integer or float data type, however to further restrict the types of values asserted, specifying the float or integer type is preferred. The allowable data types for each slot are detailed in section 2.2.1 CLIPS Facts.

A range of values may be specified for number variables using the range keyword from starting number M to ending number N. A list of allowed of values may also be specified for a given data type (which must be specified using the plural forms of the data type name after the allowed keyword e.g.: allowed-numbers, allowed-floats, allowed-symbols). The use of the range and allowed values restrictions are optional, and mutually exclusive.

The use of the `DERIVE?` keyword for a default value may be used to force CLIPS to provide an empty value of the appropriate type for a given data type, unless the `allowed-[data type]` modifier is used, in which case the first allowed data type value will be selected by default. A string type will have an empty string (“”), a symbol will have a nil value, and a number will have either a 0 or 0.0 according to whether it is an integer or float.

```
CLIPS> (deftemplate car
```

```

"Vehicle Information"
(slot make
  (type STRING)
  (default ?DERIVE)
  (allowed-strings "Nissan" "Chevy" "Toyota"))
(slot model
  (type STRING)
  (default "300ZX"))
(slot year
  (type INTEGER)
  (default 1990)
  (range 1908 2010))
(multislot features
  (type SYMBOL)
  (default ?DERIVE)))

```

To use this template, we assert an example fact:

```

CLIPS> (assert (car (model "240Z")(year 1970)(features
Manual_Transmission AC)))
<Fact-1>

```

CLIPS adds this fact to memory, take note that we did not specify a (make) for this fact. We are relying on CLIPS to provide the default value:

```

CLIPS> (facts)
f-0      (initial-fact)
f-1      (car (make "Nissan") (model "240Z") (year 1970)
(features Manual_Transmission AC))
For a total of 2 facts.

```

Upon the assertion of any fact with an incorrect range or allowed-[data type] value, CLIPS returns an error message. The enforcement of the constraints designated in deftemplates means that CLIPS is able to provide some level of basic data validation:

```

CLIPS> (assert (car (make "Ford")(model "truck")(year
1970)(features Manual_Transmission AC)))

```

```

[CSTRNCHK1] A literal slot value found in the assert
command
does not match the allowed values for slot make.

```

Because the “Ford” value is not in our allowed-strings slot definition, the fact assertion is rejected.

CLIPS deftemplates allow for the definition of a standardized schema for fact slot names, data types, acceptable values and ranges. This standardization is not just a

primitive form of data validation, but more importantly a mechanism by which abstract definitions of facts can be enforced. The LARK Engine and LarkML do not currently support the use of deftemplates.

2.2.5 CLIPS Functions

The CLIPS execution environment provides both basic arithmetic and extended scientific math functionality. Additionally the CLIPS environment has implemented a substantial set of environment functions. It is important to note that LarkML does not provide comprehensive definitions for the entire set of extensive math and environmental functions available in CLIPS. LarkML provides conversion definitions for an essential subset of functions required to translate basic rulesets, and is extensible to allow for more complex rulesets and functionality translation in future implementations.

For details on the available functions in CLIPS as well as a comprehensive overview of the CLIPS software architecture used, refer to the CLIPS Architecture Manual. (1992 CLIPS Architecture Manual)

2.3 M.1

The M.1 expert system shell was originally copy written by Teknowledge, Inc. in 1984. M.1 is a traditional expert system architecture: an inferencing engine combined with a knowledge base containing facts, meta-facts, rules, and other implementation specific structures. This expert system shell and inferencing engine operates in a relatively simple fashion, seeking expression values that are defined as **goals**. Alternatively, during a consultation (an instance of a user interacting with the M.1 execution shell) a user may specify a goal manually through the use of the **find** command. The M.1 inferencing engine iterates sequentially through the entries in the knowledge base as it seeks to find entries that can be used to derive goal expression solutions. While some rules may not directly evaluate the specific goal expression being sought, the expression evaluated in that rule may be required to be known before a goal can be determined. When this happens, M.1 searches the knowledge base for an entry that allows for the computation of this new expression. The seeking of other expressions to evaluate goal expressions in a cascading fashion is known as **backchaining**, this is discussed in more depth later in section 2.3.3 M.1 Rules.

When the M.1 inferencing engine finishes execution and is able to make conclusory statements about the consultation, M.1 displays the goal values derived and an explanation of how each goal was found. This explanation facility allows for users to determine how each goal was determined. Additionally this explanation mechanism acts as a debugging tool for knowledge engineers as it allows them to validate the rules used to determine a goal.

During a consultation as M.1 encounters expressions in the knowledge base that are considered **relevant** (expression relevance determination is explained in section 2.3.3 M.1 Rules) to the goal being sought, M.1 uses one of four basic mechanisms for evaluating expressions. For functions that are built into the M.1 engine, such as arithmetic functions, M.1 immediately computes the expression and evaluates the result. M.1 stores all evaluated expressions previously sought in a cache. If an expression is

present in the cache, M.1 uses the entry and the expression evaluation is finished. If an expression cannot be immediately calculated or found in the cache, M.1 searches the knowledge base for entries that can be used to facilitate the evaluation of the expression being sought. Because M.1 searches the knowledge base sequentially, it must be noted that the knowledge base entry order is important. If multiple knowledge base entries exist that allow M.1 to calculate an expression, the first entry encountered will be used. Finally if M.1 is unable to determine the result of a goal expression using the aforementioned methods, the inference engine will query the user and ask them to supply an answer. If the user is unable to provide this information, they may enter an answer indicating the solution to the expression being sought is **unknown**. Regardless of the result of an expression evaluation attempt, M.1 makes a cache entry of the result (the unknown result being no exception).

2.3.1 M.1 Facts

Facts are entries in the M.1 knowledge base that give some a value to an expression, possibly including a **certainty factor** – a degree of certainty of fact truth using a sliding scale from -100 to 100. For a fact to be true the certainty factor must be 20 or greater; this value is known as the belief threshold. A certainty factor of zero denotes that there is no evidence for or against the fact. Certainty factor grades of 100 and -100 denote certain truth and certain falsehood, respectively. When facts are not given certainty factor modifiers, the M.1 engine automatically assigns an implicit rating of 100.

Most M.1 facts follow the following format:

```
[label] : [expression] = [value] cf [integer value of cf]
```

Using this format, a few example facts would be:

```
fact-1:   factors cf 20
fact-2:   food = salad cf -30
fact-3:   drink = coffee cf 90
fact-4:   cat = dog
```

Fact labels are optional, however if no label is attached to a fact, M.1 automatically assigns a label of kb-[integer] (“kb” standing for knowledge base) where [integer] is incremented for each knowledge base entry. The [value] and [expression] slots can either be a number or what could be compared to the symbol data type in CLIPS, one or more printable ASCII characters. The inclusion of the certainty factor keyword cf and integer value is optional. The presence of the cf keyword denotes the existence of a certainty factor modifier, ranging from 100 to -100. The = [value] modifier is optional; if a fact in the knowledge base does not contain the = symbol, M.1 automatically appends a value of = yes after the [expression] portion of the fact entry.

As expressions are calculated during the execution of the M.1 consultation, cache entries are created for each expression. The successful search for an expression results in the following cache entry:

```
[expression] = [value] [cf%] because [label-1]...[label-N]
```

The [expression], and [value] slots in the cache entry are the same as the knowledge base fact entries, while the [label] semantics have been slightly modified. All labels of knowledge base entries that modify the value of the expression are listed after the because keyword. The [cf%] slot shows the certainty factor as a percentile. Cache entries log the corresponding knowledge base entries that result had bearing on their calculations, thus given the following knowledge base entries:

```
goal = cats.  
fact-1: cats cf 50.  
fact-2: cats cf 15.
```

The correlating cache entry after consultation execution would be:

```
cats = yes (57%) because fact-2 and fact-1
```

The certainty factor calculation formula is not immediately obvious; this is described in section 2.3.4 M.1 Certainty Factor Computation.

2.3.2 M.1 Meta-Facts

Instead of serving to provide factual information to a consultation instance, M.1 meta-facts serve to instruct the engine on how to go about the finding the information being sought. A plethora of meta-facts exist, this section serves to provide a succinct overview of those commonly used.

As facts are added to the cache, multiple contradictory expression values that are 100% certain may only co-exist if these expressions are declared as **multivalued**. A **multivalued** meta-fact may be declared as such:

```
multivalued(meat).
```

The M.1 engine implicitly assigns the **singlevalued** trait to expressions in the absence of an explicit `multivalued` declaration, but for the example below the declaration is included for demonstrative purposes:

```
singlevalued(food).  
multivalued(bbq-meat).  
goal = food.  
goal = bbq-meat.  
fact-1: food = balut.  
fact-2: food = squid-salad.
```

```
fact-3:    bbq-meat = crocodile-ribs.
fact-4:    bbq-meat = frog-legs.
```

Upon the execution of a consultation for these meta-facts, goals, and facts, the following results are concluded:

```
food = balut (100%) because fact-1.
bbq-meat = crocodile-ribs (100%) because fact-3.
bbq-meat = frog-legs (100%) because fact-4.
```

Because the food expression is `singlevalued`, the consultation only reports a single value. As soon as a `singlevalued` expression is found to have a 100% certainty, all other proceeding values for that expression are dropped. Because the `bbq-meat` expression is `multivalued`, we are able to see contradictory expression values, even when they are 100% certain. The `multivalued` trait means no contradictory values are dropped, even when they are completely certain.

The question meta-fact gives the M.I engine a mechanism to ask the user a question during the consultation. The use of additional meta-facts can provide the user with a friendly menu that facilitates easier interaction:

```
goal = meat.
question(meat)="What kind of meat will you be barbecuing?".
legalvals(meat)=[ribs, steak, sausage, burgers].
automaticmenu(meat).
enumeratedanswers(meat).
```

First `meat` is set as the goal. The `question` meta-fact allows the use of a text-based user query, while the `legalvals` meta-fact specifies legal values for the user to respond to the query with. The `automaticmenu` meta-fact will force M.I to list the entries in the `legalvals` meta-fact to inform the user of the possible choices they can make. Finally the `enumeratedanswers` meta-fact will add numbers to the menu, so that selections can be made by entering just a number, instead of having to type out the ASCII character sequence of the corresponding `legalvals` choice. Once the preceding entries are loaded into our knowledge base and a consultation instance is executed using the `go` command, we are presented with the following menu:

```
M.1>go

What kind of meat will you be barbecuing?
  1.  ribs
  2.  steak
  3.  sausage
  4.  burgers
>> 1
Meat = ribs (100%) because you said so.
```

M.1 recognizes the `meat` goal then asks the user a question with legal values listed in an automatic, enumerated menu. The user responds with the selection of `ribs` by entering a 1. This was accomplished without the use of any facts or rules, just meta-facts and a single goal.

2.3.3 M.1 Rules

M.1 rules follow a format similar to CLIPS, although it should be noted there is a subtle difference in the operation of the two expert system shells. While CLIPS seeks to match patterns with the LHS of rules using the Rete algorithm, M.1 pattern matches with expressions whose solutions are searched out sequentially in the knowledge base, which can result in a cascading **backchaining** effect, discussed later in this section. Additionally M.1 is geared towards calculating expressions while CLIPS focuses on pattern matching.

M.1 rules are expressed using the following format:

```
[rule label]:
  if
    [premise clause 1]...[premise clause N]
  then
    [conclusion clause 1]...[conclusion clause N]
```

Premise clauses consist of one or more expressions that are evaluated by the M.1 engine to determine if the rule can be evaluated to be true. If a rule premise evaluates as true, the rule is said to **succeed**. Conversely if a rule premise evaluates as false, the rule **fails**. The testing of a rule premise is known as the **invocation** of said rule. Multiple premise clauses may be joined using Boolean operators:

```
bbqrule-1:      if bbq = true and
                 meat = ribs
                 then delicious = true
```

M.1 uses the conjunction (`and`), disjunction (`or`) and negation (`not`) Boolean operators. Premises in M.1 generally following four formats:

```
[expression]
[expression] = [value]
[expression] cf [CF]
[expression] = [value] cf [CF]
(Teknowledge, Inc. 4-11)
```

Similar to the syntax for M.1 facts discussed in section 2.3.1 M.1 Facts, rule premises are expressions that must either be calculated or searched out in the M.1 cache for entries. Expressions are evaluated as true if their certainty factor is evaluated as being 20 or greater.

Once the M.1 engine identifies an expression to seek out a solution for via the use of a goal, or the backchaining effect, rules are selected for evaluation based on two factors: the order in which they are encountered and whether or not they are considered relevant. Rule relevance means that an expression being sought appears within a conclusion clause in the RHS of the rule. M.1 assumes that the evaluation of the rule containing the expression being sought will allow some determination to be made regarding the value of said expression. The following example knowledge base shows a goal of `best-drink`:

```
goal = best-drink.  
rule-1: if 1 = 1 then best-drink = cold-drink.  
rule-2: if 1 = 1 then best-drink = warm-drink.  
rule-3: if 1 = 1 then best-steak = rib-eye.
```

Of the three possible rules the M.1 engine could select to attempt to evaluate the goal, only `rule-1` and `rule-2` are considered relevant because their conclusions includes a value assignment for the `best-drink` expression. Because the M.1 engine gives precedence to the first knowledge base entry (and because the goal is implicitly singlevalued), the conclusion of the consultation is as follows:

```
best-drink = cold-drink (100%) because rule-1.
```

M.1 encountered `rule-1` first and was able to make a conclusion about the value of `best-drink` based on the RHS. Had the goal been declared multivalued, the results would be:

```
best-drink = cold-drink (100%) because rule-1.  
best-drink = warm-drink (100%) because rule-2.
```

The multivalued and singlevalued meta-facts are discussed at length in the previous section, 2.3.2 M.1 Meta-Facts.

An essential part of the operation of the M.1 engine is the cascading testing of multiple rules in response to the evaluating of a single expression. The “chain of events in which seeking the value of one expression causes M.1 to invoke a relevant rule, and consequently to seek the value of another expression found in the premise of the rule, is called *backchaining* and is fundamental to the operation of the M.1 inference engine.” (Teknowledge, Inc. 4-11). The following knowledge base demonstrates this operation:

```
goal = drink.  
rule-1: if bbq  
       then drink = beer.  
rule-2: if weather = sunny  
       then bbq.  
fact-1: weather = sunny.
```

M.1 first notes that the goal of this consultation is the value of the `drink` expression. M.1 then identifies `rule-1` as being relevant because the `drink` expression is assigned a value in the conclusion. The `bbq` expression is now sought out, as its calculation is a prerequisite for the evaluation of `rule-1`. The `bbq` expression is assigned a value in the conclusion of `rule-2`. Before the `bbq` expression can be evaluated in `rule-2`, there must be an evaluation of the `weather` expression. Finally, `fact-1` gives us a value for `weather`, which allows `rule-2`, then `rule-1` to finalize their evaluations and conclusions. This short example demonstrates the characteristic backchaining that often occurs during the execution of an M.1 consultation.

When M.1 rule premise clauses are joined using a conjunction (`and`) Boolean operator, the first failed premise clause causes the entire rule to fail. M.1 does not seek to prove or disprove any remaining premises after encountering a failure. This subtlety is important to note as the disjunction (`or`) Boolean operator causes M.1 to execute multiple evaluations in an attempt to prove or disprove a rule.

M.1 rules containing multiple rule premise clauses joined by disjunction have a significant effect on the final certainty factor of the expression being evaluated. M.1 treats the multiple expression solution paths as independent verification of the expression being evaluated:

```
goal = meat.  
rule-1: if (1 = 1) or (2 = 2)  
      then  
      meat = ribeye cf 50.
```

```
M.1>go
```

```
meat = ribeye (75%) because rule-1.
```

The calculation of certainty factors is discussed at length in section 2.3.4 M.1 Certainty Factor Computation, however a generalization can be made at this juncture to emphasize a point: when multiple rules evaluate as true that contain conclusions with positive certainty factors for the same expression, the certainty factor for the expression contained in the cache is greater than or equal to the certainty factor of each rule conclusion. This is demonstrated above as M.1 evaluated first the `(1=1)` premise clause to be true, then adds a cache entry to for `meat = ribeye cf 50`. M.1 then performs an action known as **backtracking**, wherein the engine discovers additional paths to prove the validity of the remaining premises. In this example, the `(2=2)` premise clause is evaluated as true, resulting in a modification of the `meat` cache entry, increasing the certainty factor to 75. The use of multiple premise clauses in rules combined via disjunction(s) is functionally equivalent to using multiple independent rules with the same conclusion clauses, however the use of the disjunction operator allows for more succinct syntax.

The negation (`not`) Boolean operator may be added to a premise clause to negate the evaluation of an expression contained within parentheses. There is a subtlety to this functionality – because some expressions can evaluate to yes, no, and unknown, the

negation of a (a==yes) fact does not necessarily have the same result as a (a==no) fact:

```
goal = rule-1-success.  
goal = rule-2-success.  
noautomaticquestion(food).  
rule-1: if not (food == yes) then rule-1-success = yes.  
rule-2: if (food == no) then rule-2-success = yes.
```

M.1>go

```
Rule-1-success = yes (100%) because rule-1.  
Rule-2-success was sought but no value was concluded.
```

Note that expressions that are sought with no satisfactory conclusion are added to the cache with the following conclusion: [expression] was sought but no value was concluded. Because the (food == yes) premise clause is unknown, it evaluates to a 0 CF of yes. The negation of this unknown value is yes. Because the value of food is unknown, the premise (food == no) is simply unknown. Strangely, but in step with the initial assertion that a negation of an unknown value is equivalent to yes, if the value of (food == no) is negated it also results in a yes result:

```
goal = rule-1-success.  
goal = rule-2-success.  
noautomaticquestion(food).  
rule-1: if not (food == yes) then rule-1-success = yes.  
rule-2: if not (food == no) then rule-2-success = yes.
```

M.1>go

```
Rule-1-success = yes (100%) because rule-1.  
Rule-2-success = yes (100%) because rule-2.
```

The only allowed Boolean operator in the conclusion of an M.1 rule is the conjunction operator, the disjunction and negation operators may not be used. Also, only one expression may be assigned a value in the conclusion of a rule; however it may be assigned multiple values, as such:

```
rule-1: if bbq  
      then drink = lemonade cf 50  
      and  drink = water cf 40.
```

Because the initial conclusion clause contains the drink expression, only the drink expression may be modified in any additional conclusory clauses contained within the rule-1 conclusion.

2.3.4 M.1 Certainty Factor Computation

M.1 resolves the issue of multiple certainty factors existing in the cache by combining them. The calculation strategies for certainty factors are selected according to whether or not the respective certainty factors are greater or less than zero. When both certainty factors are positive, the following formula is used:

$$\text{CF-Noted} = \text{CF1} + \text{CF2\% of } (100 - \text{CF1})$$

(Teknowledge, Inc. 17)

Table 2, below calculates the values for two sets of certainty factor values using the formula for combining two positive certainty factor values:

Table 2: Certainty Factor Computations for Positive Numbers

CF1	CF2	CF2%	100 - CF1	Formula	CF-Noted
60	60	60%	40	$60 + (.6 * 40)$	84
50	30	30%	50	$50 + (.3 * 50)$	65

To calculate two negative certainty factors, M.1 uses the following formula:

$$\begin{aligned} \text{CF-noted} &= -(|\text{CF1}| + |\text{CF2\% of } (100 - |\text{CF1}|)|) \\ &= \text{CF1} + \text{CF2\% of } (100 + \text{CF1}) \end{aligned}$$

(Teknowledge, Inc. 18)

Table 3, below calculates the values for two sets of certainty factor values using the formula for combining two negative certainty factor values:

Table 3: Certainty Factor Computations for Negative Numbers

CF1	CF2	CF2%	100 + CF1	Formula	CF-Noted
-60	-60	-60%	40	$-60 + (-.6 * 40)$	-84
-50	-30	-30%	50	$-50 + (-.3 * 50)$	-65

M.1 may also combine positive and negative certainty factors using the following formula:

$$\begin{aligned} \text{CF-Noted} &= (\text{CF1} + \text{CF2}) * 100 / (100 - A) \\ A &= \min(|\text{CF1}|, |\text{CF2}|) \end{aligned}$$

Per the second expression above, A is the lesser of the absolute value of the two certainty factors being calculated. Table 3, below calculates the values for two sets of

certainty factor values using the formula for mixed negative and positive certainty factor values:

Table 4: Certainty Factor Computations for Negative and Positive Numbers

CF1	CF2	A	100 / (100-A)	Formula	CF-Noted
-60	60	60	2.5	$(-60 + 60) * 100 / (100 - 60)$	0
-50	30	30	1.43	$(-50 + 30) * 100 / (100 - 30)$	-28

The combination of any positive or negative certainty factor with a certainty factor of zero leaves the original unchanged. There remain additional possible combinations of certainty factors, such as the combination of unknown and known premises that is not detailed here.

2.3.5 M.1 Variables

M.1 uses variables in a fashion similar to CLIPS. M.1 variables are not used for assigning value to a container as is done in a procedural programming language. In procedural programming languages, a variable is instantiated as a container for information and then data is put in that container, for example an integer or a string. In M.1 (and CLIPS) variables are instead considered wildcard slots. M.1 variables allow for a multiplicity of rules to be simplified into a single rule, via the use of these wildcard slots. CLIPS allows for the use of variable type declarations to restrict the types of rules and facts found in CLIPS rulesets, while M.1 does not provide this granular type control. Before the following examples, it should be noted that M.1 is case-sensitive. Variables are declared by starting the variable name with a capitalized character, or an underscore character:

```
kb-1:    goal = best-steak.
kb-2:    deliciousness = ribeye.
kb-3:    if deliciousness = MYSTEAK
         then best-steak = MYSTEAK.
```

```
M.1> go
```

```
best-steak = ribeye (100%) because kb-3.
```

The value being sought from the goal declared in kb-1, best-steak, is found in the RHS side of kb-3, which causes M.1 to backchain to kb-2 to discover the value of deliciousness. This instantiates the MYSTEAK variable to the value of deliciousness found in kb-2: ribeye. Finally, M.1 applies ribeye to the wildcard slot MYSTEAK in the RHS of kb-3, and M.1 is able to conclude that best-steak = ribeye.

M.1 allows for the creation of **compound expressions** consisting of two or more terms via the use of the hyphen operator (-). Compound expressions allow part of the

pattern being created to be specified by the rule it is found in, and then adding on a wildcard slot, as such:

```
kb-1:    goal = best-meat.
kb-2:    deliciousness = preferred-toadlegs.
kb-3:    if deliciousness = preferred-GOODMEAT
         then best-meat = GOODMEAT.
```

M.1> go

```
best-meat = toadlegs (100%) because kb-3.
```

The value being sought from the goal declared in kb-1 is best-meat. Because kb-3 contains best-meat in the RHS portion of the rule, a solution for the LHS is sought in the knowledge base. The preferred- portion of the condition of the kb-3 entry sets a constant part of the pattern being sought, while the GOODMEAT portion is the wildcard slot. The preferred- prefix is then found in kb-2, and the toadlegs portion is now instantiated as the value of the GOODMEAT wildcard slot. The wildcard is now toadlegs, and is assigned to best-meat, the goal.

M.1 allows for the use of **anonymous variables**. Denoted by the use of a single underscore character “_”, the anonymous variable allows for M.1 to substitute multiple values into anonymous variable wildcard slots. This is different from non-anonymous variables because non-anonymous variables always instantiate the same value to a given wildcard. The example knowledge base below serves to demonstrate this functionality:

```
kb-1:    goal = bbq-time.
kb-2:    beer = cold.
kb-3:    grill = hot.
kb-4:    if beer = _ and
         grill = _
         then bbq-time.
```

M.1> go

```
bbq-time = yes (100%) because kb-4.
```

The goal set in kb-1 is bbq-time, and is found in the RHS of kb-4. M.1 seeks out values for the LHS patterns and finds two different possible values. Because an anonymous variable was used in the two LHS conditions of kb-4, the conditions are able to be met using two different values, found in kb-2 and kb-3. Had the same non-anonymous variable been used in this instance in the two kb-4 conditions, the rule would not have evaluated as true. The value of the anonymous variable is not to confirm that a specific value is assigned to a given expression, but that a value exists. A natural language interpretation of kb-4 would be “If the beer expression contains any value, and the grill expression contains any value, then it is indeed bbq-time.”

This has served as a non-comprehensive introduction to M.1 variable functionality, sufficient for understanding how these variables can be used in the LARK engine, detailed in remainder of this paper. Additional information about M.1 variables can be found in the M.1 Reference Manual. (Teknowledge, Inc.)

2.4 Natural Rule Language (NRL) Constraint Language

The Natural Rule Language (NRL) Constraint Language specifies business rules for constraining models. Developed by Christian Nentwich and Rob James, NRL is part of an open specification sponsored by SourceForge. NRL is planned to consist of a few different languages, however the current implementation has only the Constraint Language, which serves as NRL's core. The goal of this language is to be able to specify constraint rules that are machine parseable, but that are also readable and understandable by non-programmers. NRL rules are easily readable, but it must be noted that they are not freeform. NRL uses a specific grammar that must be followed to ensure machine-readability. This language is specifically geared towards constraining UML models; however its application can be generalized beyond UML to include XML schemas in addition to Java code.

NRL provides an example of a natural language implementation of a strict grammar used to describe UML constraints as rules. One of the goals of the LARK engine is to express expert system rules in an easily understood natural language. NRL is relevant to this thesis because it provides an example of a natural language used to describe model constraint rules. This overview of NRL serves to provide the reader with an understanding of the concepts important to the expression of abstract rules as natural language rules.

The LARK engine produces rules in a natural language, Lark Natural Language (LNL) specified for the purpose of communicating a small subset of rule functionality. It would not be feasible for LARK to instead produce NRL Constraint Language rules, given the scope of this thesis, however there are key concepts and considerations that can be gleaned from the NRL implementation for the development of LNL. This author recognizes the value in surveying previous successful efforts of a natural language rules implementation, but also the value in specifying a language for a specific purpose.

The NRL Constraint Language is fully defined in EBNF notation in the Natural Rule Language Version 1.0 Working Draft (Boley), so it would be of little value to delve into a full description of each element, however this section covers the high points, relevant to the development of LNL.

2.4.1 NRL Referencing

NRL rules constrain the UML model elements known as **classifiers**, which in turn contain attributes. Classifier attributes may be static, are named, and have associated types. As with all UML model elements, classifiers inherit attributes from their respective parent elements.

To constrain a UML model, NRL Constraint Language rules reference model elements or model element attributes. Each NRL rule and property contains context

definitions that define the element or attribute being constrained. Constraints are defined by applying restrictions to model elements or model element attributes.

The follow production defines the syntax for the four common reference types found in NRL: model element, attribute, static attribute and variable references.

```
ModelPath ::= (PackageReference)? elementOrAttribute::ModelElementName  
            ( "." attribute:ModelElementName)*"
```

(Boley)

`ModelPath` consists of an optional `PackageReference` (used for fully qualified references, defines the container for a given entity), a required element `ModelElementName` (a reference to either a model element or an element attribute), and if further resolution is needed to reference an attribute, an additional `ModelElementName` (which contains the name of the attribute being referenced).

Model element references use an element's name to reference model elements. Elements may be referenced simply using their name, however if there are two elements in the same context using the same name, this results in an error. Alternatively an element can be referenced using a fully qualified name. Valid model references include:

```
Ribeye -  
Meats::Steaks::Ribeye -
```

`Ribeye` is a model element contained in the `Steaks` package, which is in turn contained in the `Meats` package. The first reference specifies the model element simply by using its name. The second reference uses the qualified name by first referencing the `Meats` package, then the `Steaks` package, and finally the `Ribeye` model element.

Attribute references are used to reference a model element attribute, in a relative fashion. Attribute references start with a model element, which then use the dot syntax to "navigate" to either an attribute of said element, or an attribute of a related element. Another methodology for referencing attributes is to simply use their name, with the current context as the assumed starting point. Valid attribute references include (but are not limited to):

```
Meat.Sauce -  
Sauce -
```

The first reference assumes that the current context contains a model element `Meat` which in turn contains the attribute `Sauce`. The second reference is only valid when the current context contains an attribute `Sauce`. An error occurs if the current context does not contain the elements and attributes being referenced.

Alternatively known as absolute references, static attribute references are used to specify a reference to a model element's static attribute. Valid static attribute references include:

```
MeatType.MEATCONST -  
Meats::Steaks::SteakType.STEAKCONST -
```

Similar in format to the model element references, but with the addition of a static attribute qualifier, these static attribute references assume a model element `MeatType` containing a static attribute `MEATCONST`. `SteakType` is contained within package `Steaks`, which is in turn contained in the `Meats` package.

Variable type references refer to attributes using a variable as the starting point as opposed to a model element, and are otherwise the same as the attribute reference type.

2.4.2 NRL and Application to Lark Natural Language

The NRL Constraint Language defines two kinds of constraints: rules and properties. Both rules and properties may be evaluated to have a Boolean result of true or false. In NRL, properties are known as rule fragments, as they are not intended to be evaluated like rules. Constraints, quantifiers and variables play an essential role in the definition of both rules and properties. The productions of each are described below using EBNF notation:

Table 5: NRL Syntax (Boley)

Production	Syntax
Rule	<pre>Rule ::= Context "Rule" id:DoubleQuotedString Constraint Context ::= "Context:" ModelReference</pre>
Property	<pre>Property ::= Context "Property" id:DoubleQuotedString Constraint</pre>
Constraint	<pre>Constraint ::= IfThenStatement</pre>
Quantifier	<pre>ExistsStatement ::= (Enumerator ("of the")?)? ModelReference ("has" "have" "is" "are") ("present" SimpleOrComplexConstraint) Enumerator ("has" "have" "is" "are") SimpleOrComplexConstraint Enumerator ::= ("at least" "at most" "exactly")? ("one" "two" "three" "four" "no" "none" IntegerNumber)</pre>
Variable	<pre>VariableDeclaration ::= "Let" VariableDeclarationList ", " "then" Constraint ";" VariableDeclarationList ::= SingleVariableDeclaration ("and" VariableDeclarationList)? SingleVariableDeclaration ::= VariableName ("be" "represent") Expression</pre>

It should be noted that each production contains additional elements that must be defined in EBNF format for a comprehensive understanding of all the possible rule and property permutations. For the comprehensive reference, please refer to the Natural Rule Language Version 1.0 Working Draft (Boley).

Rules consist of a context declaration, which defines the model being referenced, an id (which can be considered a rule name), and finally a constraint element. Constraint elements are further identified as if-then statements, which have a few permutations. Similar to NRL rules, property declarations consist of a context element, and id, and a constraint. A rule using a property is defined below:

```
Context: Steak
Property "delicious"
steakmeat is equal to 'Ribeye'
```

```
Context: Steak
Rule "Delicious Rule"
If the Steak is "delicious" then the value is 25.
```

As shown above, in rule "Delicious Rule" the constraint portion of an NRL rule contains an if-then statement. The production for an `IfThenStatement` is below:

Table 6: NRL If-Then Syntax (Boley)

Production	Syntax
<code>IfThenStatement</code>	<code>IfThenStatement ::= "if" IffStatement "then" IfThenStatement "else" IfThenStatement IffStatement</code>
<code>IffStatement</code>	<code>IffStatement ::= ImpliesStatement ("only if" IffStatement)?</code>
<code>ImpliesStatement</code>	<code>ImpliesStatement ::= OrStatement ("implies" ImpliesStatement)?</code>
<code>OrStatement</code>	<code>OrStatement ::= AndStatement ("or" OrStatement)?</code>
<code>AndStatement</code>	<code>AndStatement ::= LogicalStatement ("and" AndStatement)?</code>
<code>LogicalStatement</code>	<code>LogicalStatement ::= ExistsStatement NotExistsStatement ForallStatement VariableDeclaration "(" Constraint ")" Predicate</code>

The productions in addition to the `IfThenStatement` implement useful predicate logic, such as the `IffStatement`, which is defined as meaning “if and only if” (Boley). The `ImpliesStatement` implements the predicate logic concept of logical implication. The `OrStatement` and `AndStatement` allow for Boolean disjunctions and conjunctions, respectively. Finally the `LogicalStatement` production allows for further complexity using variables, additional constraints, predicates, and existence declarations.

The Natural Rule Language Version 1.0 Working Draft details the following examples of valid constraints:

```
"If the Trade is "internal" then its value is less than 1000000  
else its value is less than 10000
```

```
value > 1000 and value < 10000
```

```
value > 10000 implies the Trade is "internal"
```

```
(value < 1000000 and the Trade is "internal") or (value < 1000  
and the Trade is "external")"
```

(Boley)

The `IfThenStatement` demonstrates the classic if-then structure familiar to expert system rules, and contains some important concepts that need to be considered in the specification of Lark Natural Language. The Boolean functionality shown in the `OrStatement` and `AndStatement` productions as well as the ability to declare and reference variables (per the `VariableDeclaration` production, not shown above) are essential to NRL. LNL must also be able to express the notion of numeric quantification similar to the `Quantifier` production defined in Table 5, but because numeric comparisons are easily understood by the layman, LNL does not have the facility to express numbers and comparisons in English.

While the NRL specification contains some important considerations for the development of LNL, it should be noted that LNL only seeks to express expert system rulesets in an easily readable language and does not seek to be machine parseable. Because of that stipulation, LNL grammar can be more relaxed than NRL. Indeed LNL's sole purpose is to express LarkML rules in a format that is more easily understood by non-technical readers.

3. RULESET PORTABILITY EFFORTS

Ruleset portability for business rules, model-constraint rules, expert system rules, and related areas has been in ongoing development by the technology community. This section provides a succinct overview of a few of the significant community efforts for ruleset portability, and provides some background on the current state of affairs for portability efforts. A more in-depth overview of the RuleML portability effort is given, as this XML-based language bears a close functional relationship to the LarkML language and LARK Engine.

3.1 OMG PRR

The Object Management Group, Inc. (OMG) was founded in 1989 as a non-profit organization with the goal of defining and maintaining computer industry standards to facilitate the portability, interoperability and reusability of enterprise applications in varied, networked environments. The OMG consortium allows for open membership, including commercial vendors, the academic community, government, and technology users. (Taylor et al ix)

The OMG Production Rule Representation (PRR) standard is in development to meet the needs of a portability solution for rulesets used in business and software systems, as well as other OMG standards. PRR facilitates the exchange of business rules between modeling tools by providing a standard production rule representation that industry vendors are familiar with. PRR “provides a standard production rule representation that can be used as the basis for other efforts such as the W3C Rule Interchange Format and a production rule version of RuleML” (Taylor et al 1). Currently only forward chaining and procedural production rules may be modeled via PRR, however the standard is extensible. Future PRR versions may include ability to express additional types of rulesets, such as backward chaining (Taylor et al 1, 5).

3.1.1 Production Rules and Rulesets

The basic atomic element of the PRR standard is the **production rule**. As discussed previously in section 1.2, production rules are expressed in an if-then logical structure:

```
if [condition] then [action-list]
(Taylor et al 6)
```

When a specific condition is met, the corresponding action or actions following the if-statement are executed. It must be noted that the order in which the rules are encountered can affect their order of execution. Production rules are housed within **production rulesets**. Production rulesets are logical containers for production rules, and provide “a means of collecting rules related to some business process or activity as a functional unit, a runtime unit of execution in a rule engine together with the interface for rule invocation.” (Taylor et al 6, 7)

PRR variables come in two different types: standard variables and rule variables. Standard variables have types and their scope is the ruleset in which they are defined. Standard variables have single values (that may be collections) and can be modified as the result of a rule action.

Rule variables may be defined either within a rule or a ruleset. Rule variables are assigned type, and optionally may have a filter applied to their data source which sets the domain of the rule variable.

The PRR standard is full defined in “OMG document bmi/2007-03-05 Production Rule Representation.”

3.2 W3C Rule Interchange Format (RIF)

The W3C Rule Interchange Format (RIF) Working Group’s mission is to create a standardized, extensible format (or language) for rules. This rule interlingua will facilitate the portability of rulesets between applications and rule-based knowledge-systems. The RIF working group recognizes that the rulesets of the various applications and rule engines are quite diverse, thus the ability to extend the original, core language is an integral consideration specified in the RIF charter. The RIF working group charter specifies the need for a core language, standard extensions, and non-standard extensions.

3.3 RuleML

Composed of a network of over 40 industry partners and academia, the RuleML Initiative seeks to create a rule markup language that is adopted by the greater technology community. The RuleML Initiative plans on taking the significant existing rule markup standards and working with the open network of partners to converge them into a widely adopted, shared standard. The RuleML initiative is leveraging standards from many sectors with preexisting rule markup languages including engineering, commerce, law, and the internet representing diagnosis rules, business rules, legal rules, and access authentication – respectively. The current implementation of RuleML recognizes three types of rules: natural, formal, and semi-formal. The goal of the initiative is to convert these rule types into a standard XML format. (Boley)

The RuleML Initial Steps 1.0 document, originally specified in 2002 by Harold Boley defined the issues surrounding the RuleML Initiative. This document began the process of giving structure to the RuleML markup language, as well as bringing to light issues, tasks, and challenges facing the RuleML initiative. (Boley)

3.3.1 RuleML Hierarchy and Tags

The RuleML design is dominated by a hierarchy of rule types, as illustrated below:

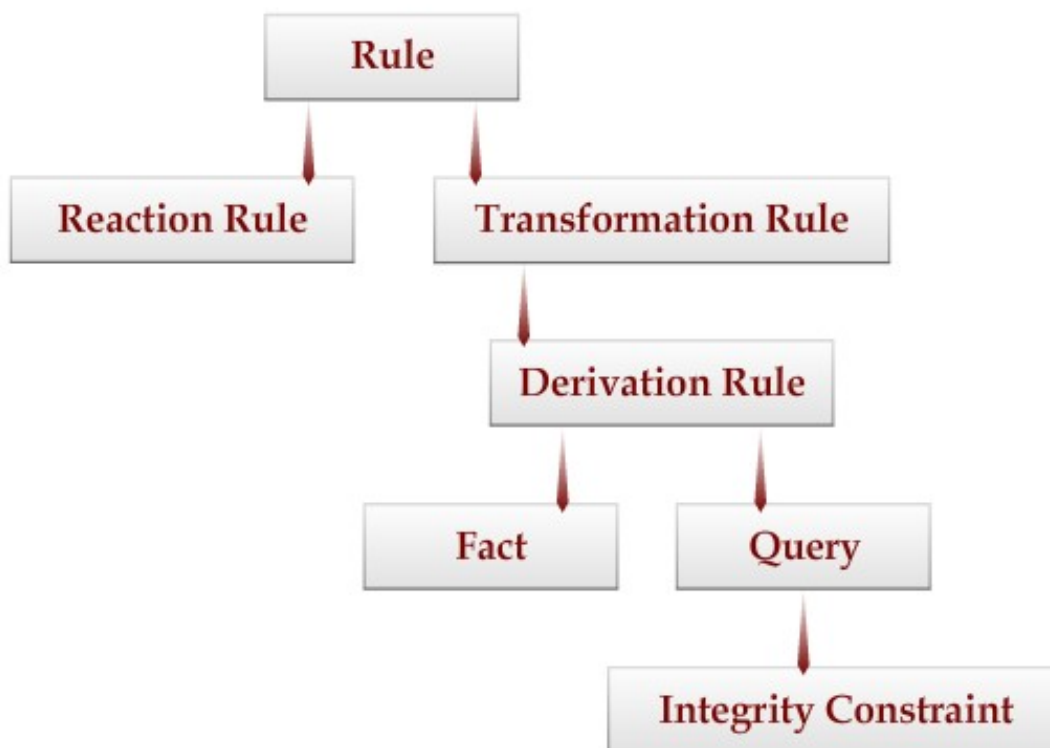


Figure 1: RuleML Hierarchy

The specification of RuleML rule-types is ongoing. The above diagram provides only a top-level view of the rule type hierarchy and reduction tree. Rules consist of reaction rules and transformation rules. Transformation rules consist of derivation rules, which are further subdivided into facts and queries. Finally queries can be specified as integrity constants.

Each of the RuleML rule-types uses tags, as specified below:

Table 7: RuleML Rule-Tag Mapping

Rule Type	Tag
rules	rule
reaction rules	react
transformation rules	trans
derivation rules	imp
facts	fact
queries	query
integrity constraints	ic

3.3.2 RuleML Hierarchy Reduction

Each rule in the tree can be reduced to a parent node. Integrity constants produce variable bindings, but can be reduced to “closed” queries that do not. As queries are reduced to derivation rules, they implicitly receive a “false” conclusion disjunction or variable bindings that may derive a conclusion. To reduce a fact to a derivation rule, the fact receives a “true,” or empty, conjunction of premises. As derivation rules are reduced to transformation rules, they return an unqualified “true” upon success. Transformation rules which contain conditional event triggers are reduced to have event triggers always activated when they are reduced to general rules. Reaction rules which return values can be reduced to general rules which do not return a value. As each rule type is reduced and travels up the hierarchy of rule types, essential, defining characteristics are either stripped off, or significantly reduced such that they are properly typed for the specific level of rule hierarchy.

The RuleML Initiative defines the following reduction scenarios:

“Reaction rules:

```
<react>
  <_event> trigger </_event>
  <_body>
    <and> prem1 ... premN </and>
  </_body>
  <_head> action </_head>
</react>
```

reducible to:

```
<rule>
  <_event> trigger </_event>
  <_body>
    <and> prem1 ... premN </and>
  </_body>
  <_head> action </_head>
  <_foot> empty </_foot>
</rule>
```

Transformation rules:

```
<trans>
  <_head> conc </_head>
  <_body>
    <and> prem1 ... premN </and>
  </_body>
  <_foot> value </_foot>
</trans>
```

reducible to:

```

<rule>
  <_event> active </_event>
  <_body>
    <and> prem1 ... premN </and>
  </_body>
  <_head> conc </_head>
  <_foot> value </_foot>
</rule>

```

Derivation rules:

```

<imp>
  <_head> conc </_head>
  <_body>
    <and> prem1 ... premN </and>
  </_body>
</imp>

```

reducible to:

```

<trans>
  <_head> conc </_head>
  <_body>
    <and> prem1 ... premN </and>
  </_body>
  <_foot> true </_foot>
</trans>

```

Facts:

```

<fact>
  <_head> conc </_head>
</fact>

```

reducible to:

```

<imp>
  <_head> conc </_head>
  <_body>
    <and> </and>
  </_body>
</imp>

```

Queries:

```

<query>
  <_body>
    <and> prem1 ... premN </and>
  </_body>
</query>

```

reducible to:

```

<imp>
  <_body>
    <and> prem1 ... premN </and>
  </_body>
  <_head> bindings( var1, ..., varK ) </_head>
</imp>

```

Integrity constraints:

```

<ic>
  <_body>
    <and> prem1 ... premN </and>
  </_body>
</ic>

```

reducible to:

```

<query kind="closed">
  <_body>
    <and> prem1 ... premN </and>
  </_body>
</query>”

```

(Boley)

These rule reductions are examples, and are not the only possible reductions for the given rules. For additional reduction examples, see the RuleML Design site, <http://www.ruleml.org/>.

RuleML rules have a preferred application direction, depending upon the rule-type:

Table 8: RuleML Rule Application Direction

Rule Type	Preferred Application Direction
reaction rule	forward – actions are performed only after all events/conditions are fulfilled
transformation rule	backward
derivation rule	bidirectional – depending on rule set, either direction may be optimal
facts / unit clauses	non-applicable
queries	bidirectional
integrity constants	bidirectional – forward is typically preferred for optimal performance, however backward may be used for specific cases

3.3.3 RuleML Schema Specification 0.91

The RuleML XML Schema specification 0.91 was published August 24th, 2006. To facilitate a flexible implementation approach, RuleML uses a modular schema that

allows sublanguages to be extended independent of each other. The RuleML Schema Specification 0.91 currently only defines derivation rules. Reaction rules are not yet specified, as the standard is still evolving and the RuleML Initiative is attempting to move in small, deliberate steps towards defining a schema that can be proven in a simple fashion before adding additional complexity. (Boley)

3.3.4 Functional RuleML

Functional RuleML was added to the RuleML Specification Schema 0.91. According to the Functional RuleML specification: “This develops RuleML into a Relational-Functional or Functional-Logic Markup Language (cf. RFML) that can be regarded as being composed of Relational RuleML plus Transformation Rules (Oriented-Equality Definitions of Functions).” (Boley et al)

Using a boolean XML attribute “in” (interpreted), Functional RuleML defines the notion of a interpreted versus uninterpreted expressions:

*“Uninterpreted functions **denote** unspecified values when applied to arguments, not using function definitions.*

*Interpreted functions **compute** specified returned values when applied to arguments, using function definitions.”* (Boley et al)

Interpreted functions use the following syntax:

```
<Expr>
  <Fun in="yes">Result-Of</Fun>
  <Ind>PartA</Ind>
  <Ind>PartB</Ind>
</Expr>
```

Because this function is defined as interpreted (in="yes"), upon execution it returns a result. In contrast to the interpreted functions are the uninterpreted functions which exist to define expressions that are not computed to return values:

```
<Expr>
  <Fun in="no">Result-Of</Fun>
  <Ind>PartA</Ind>
  <Ind>PartB</Ind>
</Expr>
```

By default any expression with an undefined interpreted attribute is assumed to be uninterpreted.

In addition to the use of the interpreted tag, Functional RuleML defines the number of returned arguments for an interpreted function using the value (“val”) tag with the following possible parameters: “0..” and “1”. The use of the “1” (val="1") parameter value means that the function will return exactly one result, while the use of the “0..” parameter value denotes a possible return of zero to many results upon the execution of

the interpreted function. Our first example shows an interpreted function that returns exactly one result:

```
<Expr>
  <Fun in="yes" val="1">Addition-Result</Fun>
  <Ind>2</Ind>
  <Ind>5</Ind>
</Expr>
```

Using the assumption that this is a simple addition function, it makes sense to return only one result. An appropriate use of the “0..” parameter value would be in a function where there is some ambiguity regarding the number of returns, such as the names of the Computer Science students in a graduating class:

```
<Expr>
  <Fun in="yes" val="0..">Graduating-Students</Fun>
  <Ind>FSU</Ind>
  <Ind>Class-Of-2008</Ind>
  <Ind>Computer-Science</Ind>
</Expr>
```

Because there could be zero to many graduates in the FSU Computer Science program in 2008, the appropriate “val” parameter value is “0..”.

Functions may be nested; however uninterpreted functions cannot be wrapped around interpreted functions. Any number of interpreted functions may dominate any number of interpreted and uninterpreted functions, as long as the uninterpreted functions comprise the inner layer of nesting (and do not dominate any interpreted functions). To continue our example of the FSU Computer Science graduating class of 2008, we demonstrate a two-layer nesting of interpreted functions:

```
<Expr>
  <Fun in="yes" val="1">Student-Count</Fun>
  <Expr>
    <Fun in="yes" val="0..">Graduating-Students</Fun>
    <Ind>FSU</Ind>
    <Ind>Class-Of-2008</Ind>
    <Ind>Computer-Science</Ind>
  </Expr>
</Expr>
```

We are first returning the graduating students, then making a count of them using our “Student-Count” function – which uses the “1” parameter value for the “val” attribute because there will be exactly one return value once the students are counted.

Please refer to the RuleML site for additional information: <http://www.ruleml.org>.

4. LANGUAGE ABSTRACTION FOR RULE-BASED KNOWLEDGE-SYSTEMS (LARK) ENGINE

The LARK Engine is written in C# using the Microsoft .NET Framework 2.0.

4.1 Purpose

A general overview of CLIPS and M.1 in sections 2.2 and 2.3 (CLIPS and M.1, respectively) shows the similarities and a few differences in these expert system shells. With these similarities in mind, along with some of the lessons learned from the current portability efforts, the LARK Engine aims demonstrate a very specific functionality: to provide ruleset portability between the CLIPS and M.1 expert system shells, and finally to express LarkML rules in a format easily understood by non-technical audiences.

The purpose of the Language Abstraction for Rule-Based Knowledge-Systems (LARK) Engine is to provide ruleset portability. The current implementation of the LARK Engine demonstrates the ability to translate LarkML rules to CLIPS, M.1, and Lark Natural Language (LNL) rules. Additionally LARK can parse CLIPS and M.1 rulesets and convert them to the LarkML XML standard (as defined in Appendix A). LARK may be extended in the future to provide conversion for other types of languages.

The M.1 engine is a relatively old expert system shell, and is not as relevant in the computing world as newer expert system shells. The decision to include this as one of the two shells to provide portability to was two-fold. M.1 rulesets are no less relevant now than they were when M.1 enjoyed more popularity. The LARK Engine can parse M.1 rulesets, and put them in the portable LarkML syntax that can be transformed into rulesets for more modern expert system shells. M.1 rulesets don't have to expire with the M.1 shell; neither do they need to be translated by hand. The LARK Engine provides an automated way to extract and express these rulesets as a malleable XML syntax. The second reason for including M.1 is to demonstrate that the LARK Engine can parse significantly different ruleset syntaxes. M.1 rulesets use a sentence structure that is similar to natural language while CLIPS uses a more abstract syntax.

4.2 Scope

Different expert system implementations invariably have features that are not supported by all other expert systems. The goal of the LarkML language was to capture a small subset of shared features between M.1 and CLIPS, and provide a mechanism by which they can be translated. Exceptions wherein LarkML features cannot be used in either M.1 or CLIPS exist, and are discussed later in this section.. It should also be noted that the entire implementation of LarkML can be expressed as LNL, however the current implementation of the LARK Engine does not allow for LNL rules to be parsed and converted to LarkML. The conversion between LarkML and LNL rules is one way.

The LARK Engine converts rules defined in LarkML to CLIPS, M.1 and LNL rules. Additionally, the LARK Engine is able to read and convert most simple CLIPS and M.1 rules and facts into LarkML. System specific features and functions are not converted into LarkML, as the goal of this system is to provide ruleset portability

between M.1 and CLIPS. A comprehensive conversion engine is beyond the scope of this thesis, but the LARK engine sets the groundwork for a more extensive engine.

Section 4.7 discusses the LarkML language, and gives a comprehensive overview of the features and types of rules that can be created using this standard. Section 4.8 details the mapping of LarkML rules to M.1, CLIPS, and LNL rulesets. Additionally this section discusses the CLIPS and M.1 rule formats that may be parsed by the LARK Engine and converted to LarkML.

An essential consideration to make is the concept of equivalent rules. Because of differences in syntax, and the way in which rules are expressed in different languages, it can be difficult to ensure that a rule translated from CLIPS to LarkML then transformed into M.1 syntax has equivalent meaning. Even more so, it should be noted that because of expert system shell differences, it is possible some rules simply cannot be expressed in another language without significant changes to the structure and possibly rule meaning. This can be seen in a simple example:

```
(defrule findparents
(find-parents-of ?child )
(parent-of ?myvar1 )
?mybind <- (pattern2)
=>
(retract ?mybind ))
```

This CLIPS rule has three patterns in the LHS, then one action in the RHS. Expressing this rule in M.1 presents a fundamental challenge. First of all, M.1 style rules are expressions with single terms. The first two patterns found in this CLIPS rule have two terms: one constant and one variable. The third LHS pattern presents another issue. This pattern binds a fact (`pattern2`) to a variable, so that the RHS side of the rule can retract said fact. M.1 does not have the functionality to bind facts to variables, and then retract them. Equivalent rules can be defined, however it will take significant effort on the part of the programmer to massage the translated rulesets even after the LARK Engine executes a conversion between the two languages. It is within the scope of the LARK Engine to do a best-effort translation, but not to create turn-key rulesets going from one expert system shell to another.

4.3 Objectives and Success Criteria

To be successful, the LARK Engine must be able to load LarkML .xml files and then convert them to CLIPS, M.1, and LNL rulesets. Conversely the LARK Engine must be able to load and parse CLIPS and M.1 rules and convert them into LarkML rulesets. Finally the LARK Engine must be able export all converted rules to text files.

4.4 Definitions, Acronyms, Abbreviations

Table 9: LARK Engine Definitions, Acronyms, and Abbreviations

Term / Phrase	Definition
LARK Engine	The Language Abstraction for Rule-Based Knowledge-Systems (LARK) Engine is a parsing tool to convert from LarkML compliant rulesets to M.1 and CLIPS implementation rulesets as well as Natural Language rules.
implementation ruleset	For the purposes of this thesis, the phrase “implementation ruleset” refers to any ruleset a rule-based knowledge-system can read and execute
rule-based knowledge-system	expert system
production rule	atomic element of LarkML standard
production ruleset	container for LarkML production rule
LarkML	Lark Markup Language, defined in EBNF format in Appendix A. LarkML is the ruleset abstraction language for the LARK Engine.
LNL	Lark Natural Language: a non-normative language used to express LarkML rules in a manner more easier understood by non-technical audiences
Regex(es)	Regular Expression(s)

4.5 Functional Requirements

The LARK Engine must demonstrate the following functionalities: the ability to load LarkML, CLIPS, and M.1 ruleset files (.xml, .clp, and .txt filetypes, respectively) translating LarkML rulesets to a language of choice (M.1, CLIPS, or LNL), translating M.1 and CLIPS into LarkML, and exporting these ruleset translations to text files.

This section details how the LARK Engine meets these requirements, and illustrates how each is met with a brief introduction to the GUI. A graphic of the LARK Engine interface is shown below for reference:



Figure 2: LARK Engine Interface

The LARK Engine loads LarkML, CLIPS, and M.1 files via the file menu:

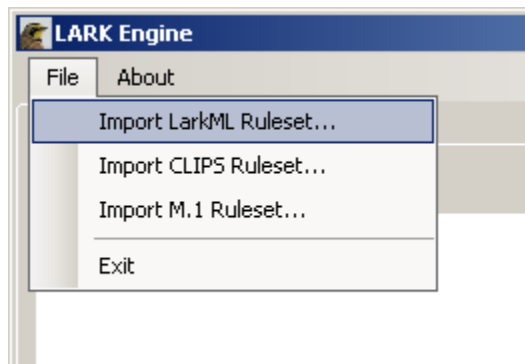


Figure 3: LARK Engine File Menu

Upon execution, the LARK Engine reads the files found in the “transforms” directory, which must be located in the same directory as the LARK executable. The files found in the “transforms” directory are .XSL stylesheets, used to transform LarkML rules to one of three languages: CLIPS, M.1, and LNL. Additional .XSL files can be added to this directory, but care must be taken to ensure they are well-formed. These .XSL file names are populated in the “Output Format” dropdown:

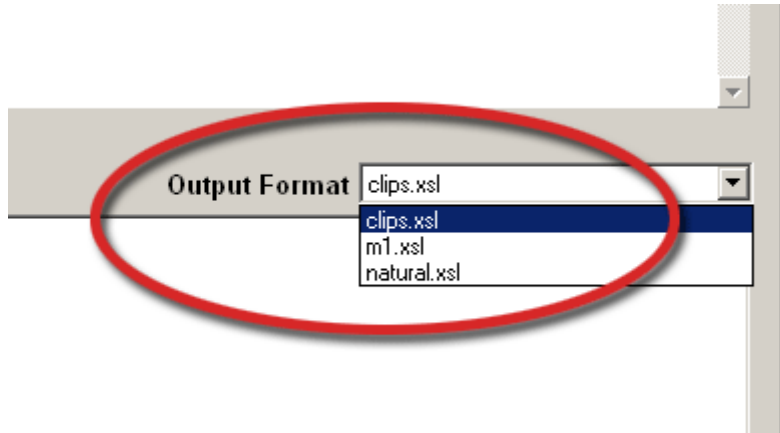


Figure 4: LARK Engine Output Format Dropdown

As LarkML files are loaded via the File menu, the currently selected output format .XSL file is applied to the LarkML ruleset, and the results are displayed in the “Output Ruleset” text box:

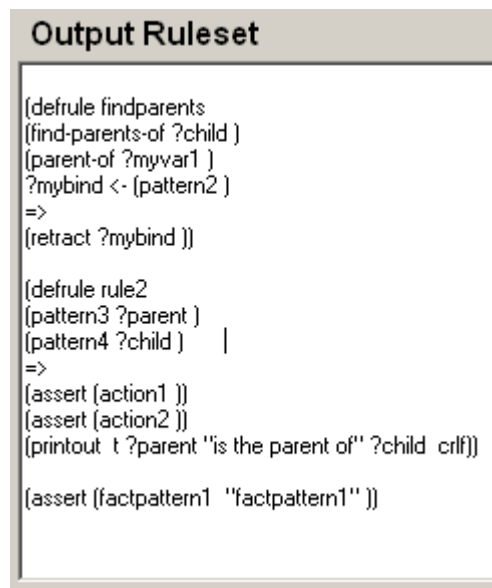


Figure 5: LARK Engine Output Ruleset

The LARK Engine handles the conversion to LarkML and from LarkML on two separate tabs, as can be seen at the top of Figure 5, bearing the respective labels “LarkML

-> Rules” and “Rules -> LarkML.” Each tab has a button allowing the export of the ruleset found in the bottom text box, again show in the bottom left-hand corner of Figure 5.

4.6 Non-functional Requirements

The LARK engine is compatible with the versions of Windows that support the .NET 2.0 Framework, which at the time of this writing includes Windows 2000 Service Pack 3, Windows 98, Windows 98 SE, Windows ME, Windows Server 2003, and Windows XP Service Pack 2. This is not a comprehensive listing, but serves to cover the more popular Windows versions. To ensure robust performance, a minimum of 512 megabytes of memory, and a 2.0 gigahertz Pentium processor or better is recommended.

4.7 LarkML

The challenge of developing a language as a standard for rulesets to be transformed into multiple expert system languages is ensuring that all necessary features are included, without focusing too deeply on proprietary functionality present only in specific expert system shells. LarkML was developed to allow basic rules to have a standard, portable definition that can be transformed into the M.1, CLIPS, and Lark Natural Language specifications. It should be noted that LarkML was not developed to allow all possible rule formats to be defined in an abstract fashion. There must be a compromise between rule complexity and the return on the time and effort investment in defining the LarkML standard. The LarkML language specification defined in Appendix A seeks to create functionality for basic ruleset translation.

The LarkML language should be considered an open standard specifically developed for use with the LARK language conversion engine, but certainly usable for other applications. Because it is an XML-based markup language, LarkML is portable to any XML parsing engine. This allows any production rulesets that are parsed, and converted to LarkML by the LARK Engine to be transformed to other languages, using XSL transforms. LarkML is specified in XML to provide a simple mechanism for ruleset translation using a mature, pre-existing technology: XSL transformations.

4.7.1 Specification

The Extended Backus-Naur Form (EBNF) of the LarkML language is defined in Appendix A. This section serves to discuss each element of the LarkML language.

All well-formed XML documents begin with a declaration specifying the version of XML being used as well as the encoding:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Each LarkML ruleset is broken up into two sections, Rules and Facts:

```
<LarkML-Ruleset>  
  <Rules>...</Rules>
```

```
<Facts>...</Facts>
</LarkML-Ruleset>
```

Facts must contain a Name and between 0..N FactPatterns. FactPatterns may contain a FactPattern-Type attribute which currently only specifies an option of String. Each FactPattern must contain an atomic, which is defined as an alpha character followed by 0..N alphanumeric characters. The following fact demonstrates a common use case of a fact in LarkML:

```
<Fact Name="myfact1">
  <FactPattern>factpattern1</FactPattern>
  <FactPattern Type="String">factpattern2</FactPattern>
  <FactPattern Type="Var">myVariable</FactPattern>
</Fact>
```

Note the use of the fact pattern types String and Var, which allow the LARK Engine to recognize specific types of fact patterns, and modify their semantics as appropriate for each translation.

LarkML facts also have the option of using an <Equals> tag, in addition to using a confidence factor parameter CF to denote the level of assurance of fact veracity, analogous to the use of confidence factors in M.1:

```
<Fact Name="rule-3">
  <FactPattern>drink</FactPattern>
  <Equals CF="90"><Atom>coffee</Atom></Equals>
</Fact>
```

A LarkML Rule consists of four parts: the RuleHeader, LHS, RHS, and the RuleTrailer. The RuleHeader contains the rule Name and opens the rule tag. The RuleTrailer closes the rule tag. An example LarkML rule:

```
<Rule Name="bbqrule1">
  <LHS>...</LHS>
  <RHS>...</RHS>
</Rule>
```

The LHS consists of 0..N LHS-Patterns. LHS-Patterns may contain an optional Name, Pattern-Type, Pattern-Bool, and Negate attributes. The Pattern-Type attribute allows for the use of a Bind LHS-Pattern. The Bind modifier is analogous to the assignment of a fact address to a variable name in CLIPS, and allows for the assertion and retraction of facts in an RHS action. Pattern Bool attributes allows for Boolean operators (and, or) to be applied to each pattern. Contained within the LHS-Pattern are 1..N Atoms. Atoms contain an optional Atom-Type attribute, which allows for the use of either a Var or String modifier. The inside of an Atom element contains a single atomic:

```

<LHS>
  <Pattern>
    <Atom>find-parents-of</Atom>
    <Atom Type="Var">child</Atom>
    <Equals CF="23">old</Atom>
  </Pattern>
  <Pattern Bool="and" Negate="true">
    <Atom>parent-of</Atom>
    <Atom Type="Var">myvar1</Atom>
  </Pattern>
  <Pattern Type="Bind" Name="mybind" Bool="and">
    <Atom>pattern2</Atom>
  </Pattern>
  <Pattern Name="falsepattern" Bool="or">
    <Atom Negate="true">pattern3</Atom>
  </Pattern>
</LHS>

```

The `Equals` tag allows for the declaration of expressions, analogous to those used in the M.1 shell. The `CF` parameter allows for the declaration of a confidence factor, again equivalent to those used in the M.1 expert system shell. CLIPS does not allow the use of confidence factors nor the use of explicit expressions. Because of this, the `Equals` tags are not translated into CLIPS rules. Section 4.8.8 discusses these considerations and other exceptions which should be noted in the translation process.

The RHS, consisting of 1...N RHS-Patterns, is a similar format to the LHS. RHS-Patterns contain an opening `Action` tag, and optional `Action-Type` attribute, and an optional `Pattern-Bool` attribute. The `Pattern-Bool` attribute may be moot, as the actions in the RHS of any rule never have a 'not' or 'or' attribute, however this helps facilitate an easier ruleset translation for M.1 rules. The available `Action-Types` are `Assert`, `Retract`, and `Print`. An example RHS is shown below:

```

<RHS>
  <Action Type="assert">
    <Atom>action1</Atom>
  </Action>
  <Action Type="assert" Bool="and">
    <Atom>action2</Atom>
  </Action>
  <Action Type="Print" Destination="Console" Bool="and">
    <Atom Type="Var">parent</Atom>
    <Atom Type="String">is the parent of</Atom>
    <Atom Type="Var">child</Atom>
  </Action>
</RHS>

```

The examples in this section are not a comprehensive reference, but serve an introduction to LarkML. The formal EBNF specification for LarkML can be referenced in Appendix A. LarkML is not unnecessarily complex; the goal of this language was to provide only the feature-set required to provide basic ruleset portability. The advantage of using an XML language is that it is easily extensible, and future work with the LARK Engine and LarkML will use an extended version of LarkML.

4.8 Language Mapping

This section briefly discusses the process by which LarkML rulesets are translated to each respective language: CLIPS, M.1, and LNL. Because LarkML rulesets are written in XML, the translation process consists of establishing some notion of equivalence between a given LarkML rule and a given expert system rule, then applying an XSL transform to the LarkML ruleset. Each language LarkML is translated to must have an XSL stylesheet that defines which elements of each LarkML rule are used, and where they go for the production ruleset. The subsections herein describe this process.

To derive LarkML syntax from CLIPS and M.1 rulesets, a more complex process must be used. Each production rule must be modified, much in the same way that LarkML rules are massaged into production rules using the XSL transformations, however there is no current infrastructure in place to execute this process. To this end, the LARK Engine executes a series of replacement functions upon the production rulesets until the final result is valid LarkML syntax. The LARK Engine uses regular expressions to match patterns, identify specific rule parts and boundaries, bind relevant rule elements to variables, and then refactor each rule in a step by step process. This refactoring process is described in the following sections.

4.8.1 Translating LarkML to CLIPS

All translations from LarkML to another language are handled by an XSL transformation being applied to the original LarkML source ruleset. The stylesheet used to process the translation to CLIPS syntax is shown uninterrupted in Appendix C, while this section pieces it apart to discuss each step in the transform process.

LarkML rulesets are segmented into two general sections: rules and facts. The strategy for processing LarkML rulesets is to nest XSL templates in an increasingly granular fashion, and then insert the proper CLIPS characters where appropriate. The first template is applied at the root of the XML file:

```
<xsl:template match="/">
  <xsl:apply-templates select="LarkML-
Ruleset/Rules/Rule" />
  <xsl:apply-templates select="LarkML-
Ruleset/Facts/Fact" />
</xsl:template>
```

These XSL tags can be interpreted to mean this: when the root of the XML file is found, apply the template defined for the elements found the LarkML-Ruleset/Rules/Rule tags, then apply the template defined for the elements found in the LarkML-Ruleset/Facts/Fact tags.

The template for the Rule tag is as follows:

```
<xsl:template match="Rule">
  (defrule <xsl:value-of select="@Name" />
  <xsl:apply-templates select="LHS" />
  =<xsl:text disable-output-
  escaping="yes">&gt;</xsl:text>
  <xsl:apply-templates select="RHS" />)
</xsl:template>
```

This template begins CLIPS rules by outputting the (defrule keyword, then writing the value found in the Name parameter of the rule tag. Next the LHS template is applied, followed by the => characters, which denote the transition from the RHS portion of the CLIPS rule to the LHS portion. Next follows the application of the RHS template.

The LHS template consists of applying the Pattern template:

```
<xsl:template match="LHS">
  <xsl:apply-templates select="Pattern" />
</xsl:template>
```

The Pattern template processes each LHS pattern first testing for Bind type patterns, and then outputting the appropriate syntax. Regardless of pattern type, the Atom template is then applied:

```
<xsl:template match="Pattern">
  <xsl:text disable-output-escaping="yes">
  </xsl:text>
  <xsl:if test="@Type='Bind'">?
  <xsl:value-of select="@Name" />
  <xsl:text disable-output-escaping="yes"> &lt;-
  </xsl:text>
  </xsl:if>
  (<xsl:apply-templates select="Atom" />)
</xsl:template>
```

The Atom template tests the Type parameter of the Atom tag, adding the appropriate syntax for Var and String types. The Negate option of the Atom tag allows for the use of functions such as the CLIPS “~” connective constraint. CLIPS variables are preceded by a question mark, while strings are wrapped in quotations. The value found within the Atom tag is either preceded by a question mark, wrapped in quotes, or printed without modification:

```

<xsl:template match="Atom">
<xsl:if test="@Negate = 'true'">~</xsl:if>
<xsl:choose>
<xsl:when test="@Type = 'Var'">?
<xsl:value-of select="." />
</xsl:when>
<xsl:when test="@Type = 'String'">
"<xsl:value-of select="." />"
</xsl:when>
<xsl:otherwise>
<xsl:value-of select="." />
</xsl:otherwise>
</xsl:choose>
<xsl:text disable-output-escaping="yes"> </xsl:text>
</xsl:template>

```

The RHS template consists of applying the Action template:

```

<xsl:template match="RHS">
<xsl:apply-templates select="Action" />
</xsl:template>

```

The Action template has four output possibilities. The Atom templates applied within each Action tag are asserted, retracted, printed using the CLIPS print syntax, or output unmodified. The processing applied is dependent upon the Type parameter of the Action tag:

```

<xsl:template match="Action">
<xsl:text disable-output-escaping="yes">
</xsl:text>
<xsl:choose>
<xsl:when test="@Type = 'assert'">
(assert (<xsl:apply-templates select="Atom" />))
</xsl:when>
<xsl:when test="@Type = 'retract'">
(retract <xsl:apply-templates select="Atom" />)
</xsl:when>
<xsl:when test="@Type = 'Print'">
(printout
<xsl:if test="@Destination='Console'"> t </xsl:if>
<xsl:apply-templates select="Atom" />
<xsl:if test="@Destination='Console'"> crlf
</xsl:if>)
</xsl:when>
<xsl:otherwise>
(<xsl:apply-templates select="Atom" />)
</xsl:otherwise>

```

```
</xsl:choose>
</xsl:template>
```

Fact templates are used to convert LarkML facts to CLIPS fact assertion statements, which contain the FactPattern template application:

```
<xsl:template match="Fact">
  (assert (
    <xsl:apply-templates select="FactPattern" />
  ))
</xsl:template>
```

The FactPattern template has two options, the value found in the FactPattern tag is either wrapped in quotations or output in an unmodified fashion:

```
<xsl:template match="FactPattern">
  <xsl:choose>
    <xsl:when test="@Type='String'">
      "<xsl:value-of select="." />"
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="." />
    </xsl:otherwise>
  </xsl:choose>
  <xsl:text disable-output-escaping="yes"> </xsl:text>
</xsl:template>
```

The clips.xml stylesheet sits in the “transforms” sub-directory of the directory where the LARK.exe executable is found. The LARK Engine was built to be easily extensible; the clips.xml file can be modified even as LARK is running to change the output. Note that the syntax shown above was modified to increase readability. The XSL `<xsl:text disable-output-escaping="yes"> </xsl:text>` is used to introduce whitespace into the output document, as the transformation engine removes whitespace between tags.

4.8.2 Translating LarkML to M.1

Just as in the case of the LarkML to CLIPS transformation, LarkML is transformed to M.1 syntax using an XSL stylesheet. The format and logic of the processing is very similar to the processing described in the previous section as the challenge is essentially the same: reformatting XML to another language via XSL. It should be noted however that the M.1 language has different functionality than CLIPS, and is much more limited. While LHS patterns in CLIPS may contain multiple symbols, M.1 expressions contain only single symbol expression constructions. Because of this, it is difficult to establish a notion of equivalence between CLIPS rules containing single LHS patterns with multiple symbols and an M.1 pattern that can only contain one

symbol. The compromise made to deal with this issue is the adjoining of LHS pattern tags. The M.1 stylesheet appends all the Atom tags together with the use of the dash operator.

Another important note is that M.1 uses case to declare variables. Any symbol containing a capital letter is understood to be a variable in M.1. Because of this language feature, LarkML rulesets that will be transformed to the M.1 language should be analyzed for the usage of capital letters, and modified as needed to prevent the unintended declaration of variables.

The M.1 stylesheet is discussed below, and can be found in its entirety in Appendix D:

```
<xsl:template match="/">
<xsl:apply-templates select="LarkML-
Ruleset/Rules/Rule" />
<xsl:apply-templates select="LarkML-
Ruleset/Facts/Fact" />
</xsl:template>
```

The root template “/” is applied, which in turn contains the application of the Rule and Fact templates.

```
<xsl:template match="Rule">
<xsl:value-of select="@Name" />: if <xsl:apply-
templates select="LHS" />
then<xsl:apply-templates select="RHS" />.
</xsl:template>
```

The value of the Name parameter in the Rule tag is output, followed by “: if” denoting the start of an M.1 rule. The LHS template is applied, followed by “then” (denoting the start of the RHS of the M.1 rule), and finally the RHS template is applied. This construction is followed by a period, signaling the end of an M.1 rule.

```
<xsl:template match="LHS">
<xsl:apply-templates select="Pattern" />
<xsl:apply-templates select=" ../RHS/Action"
mode="Print" />
</xsl:template>
```

The LHS template applies the Pattern template, followed by a special construct. Because M.1 print statements are found in the LHS of the rule structure while LarkML print commands are in the RHS of the rule, a relative reference to the ../RHS/Action template must be made, with the mode parameter set to Print. This applies a special Action template, defined as having a mode of Print. This is shown later in this section.

```
<xsl:template match="Pattern">
```

```

<xsl:choose>
<xsl:when test="@Type='Bind'"></xsl:when>
<xsl:otherwise>
<xsl:value-of select="@Bool"/>
<xsl:if test="@Negate = 'true'"> not(</xsl:if>
<xsl:text disable-output-escaping="yes"> </xsl:text>
<xsl:apply-templates select="Atom" /></xsl:otherwise>
</xsl:choose>
<xsl:apply-templates select="Equals" />
<xsl:if test="@Negate = 'true'"></xsl:if>
<xsl:text disable-output-escaping="yes"> </xsl:text>
</xsl:template>

```

As discussed earlier, ruleset translation is often a compromise when languages have non-equivalent feature sets. Because M.1 does not have the ability to bind a fact to a variable in the same fashion as CLIPS, the Bind type pattern tags are not rendered, hence an empty `xsl:when` command for `Type='Bind'`. Following this, the value of the Bool parameter is output as M.1 LHS patterns are joined with Boolean operators. M.1 pattern tags allow for the use of the Negate parameter, which wraps the pattern in a Boolean negation function. This is followed by the application of the Atom template and finally the Equals template.

```

<xsl:template match="Atom">
<xsl:if test="position() > 1">-</xsl:if>
<xsl:choose>
<xsl:when test="@Type = 'Var'">
<xsl:analyze-string select="." regex="[A-Z_]+.*">
<xsl:matching-substring><xsl:value-of select="." />
</xsl:matching-substring>
<xsl:non-matching-substring>X<xsl:value-of select="." />
</xsl:non-matching-substring>
</xsl:analyze-string>
</xsl:when>
<xsl:when test="@Type = 'String'">
"<xsl:value-of select="." />" </xsl:when>
<xsl:otherwise>
<xsl:value-of select="." />
</xsl:otherwise>
</xsl:choose>
</xsl:template>

```

Because M.1 expressions may only have one symbol or symbols bound by the hyphen operator, this Atom template appends a hyphen to each Atom following the first. M.1 variables in are declared via the use of either a capital letter or an underscore as the first character in the variable name. The `<xsl:analyze-string>` tag declares a REGEX that tests to determine if the leading character of the variable name meets these

requirements and then appends a capital X if the REGEX criteria is not met. This is necessary because LarkML Var type Atoms have no constraint requiring them to use a capital letter. String type Atoms are enclosed in quotations, and Atoms with no type parameter are rendered normally.

The RHS template consists of the application of the Action template:

```
<xsl:template match="RHS">
  <xsl:apply-templates select="Action" />
</xsl:template>
```

The Action template follows:

```
<xsl:template match="Action">
  <xsl:choose>
    <xsl:when test="@Type = 'assert'">
      <xsl:text disable-output-escaping="yes"> </xsl:text>
      <xsl:value-of select="@Bool" />
      <xsl:text disable-output-escaping="yes"> </xsl:text>
      <xsl:apply-templates select="Atom" />
    </xsl:when>
    <xsl:when test="@Type = 'retract'">
      <xsl:text disable-output-escaping="yes"> </xsl:text>
      <xsl:value-of select="@Bool" />
      <xsl:text disable-output-escaping="yes"> </xsl:text>
      <xsl:apply-templates select="Atom" />
    </xsl:when>
    <xsl:when test="@Type = 'Print'">
      </xsl:when>
    <xsl:otherwise>
      <xsl:if test="@Bool != ''">
        <xsl:text disable-output-escaping="yes"> </xsl:text>
      </xsl:if>
      <xsl:value-of select="@Bool" />
      <xsl:text disable-output-escaping="yes"> </xsl:text>
      <xsl:apply-templates select="Atom" />
    </xsl:otherwise>
  </xsl:choose>
  <xsl:apply-templates select="Equals" />
</xsl:template>
```

Assert and retract actions represent a special case of the LarkML language for handling these functions in CLIPS. Because M.1 has no strictly equivalent functions, these action types are not rendered in M.1. The Atom template is applied, and that is all. The value of the Bool parameter follows the processing of both the assert and retract statements and precedes the application of the Atom template. The Print type Action is not rendered in the RHS side of M.1 rules, so the output for Print type

Actions is an empty string. These are handled in the LHS template. Actions with no Type parameter declarations are rendered normally. The Action template is finished with the application of the Equals template.

The special-case processing for Print type Actions follows:

```
<xsl:template match="Action" mode="Print">
<xsl:if test="@Type = 'Print'">
and display([<xsl:for-each select="Atom">
<xsl:choose>
<xsl:when test="@Type = 'Var'">
<xsl:analyze-string select="." regex="[A-Z_]+.*">
<xsl:matching-substring>
<xsl:value-of select="." /></xsl:matching-substring>
<xsl:non-matching-substring>
X<xsl:value-of select="." />
</xsl:non-matching-substring>
</xsl:analyze-string>
</xsl:when>
<xsl:when test="@Type = 'String'"> "<xsl:value-of
select="." />"
</xsl:when>
<xsl:otherwise>
<xsl:value-of select="." />
</xsl:otherwise>
</xsl:choose>
<xsl:if test="position() != last()">, </xsl:if>
</xsl:for-each>) </xsl:if>
</xsl:template>
```

The Action template with mode Print, is applied in the LHS of the M.1 rule being rendered. The Mode modifier allows for the creation of an alternative definition for the Action template, which was referenced in the LHS template, using mode="Print". The display() command is the mechanism in M.1 used for writing messages to the console, and is populated here with the contents of the each Atom value. Once again we use a REGEX to process variables, testing the leading character for conformance with the rule requiring the use of a capital letter or an underscore. Each Atom tag except for the last one processed is followed by a comma.

The Equals template consists of the equal character, followed by the application of the Atom template, the cf M.1 keyword and finally the value of the CF parameter:

```
<xsl:template match="Equals"> = <xsl:apply-templates
select="Atom" />
<xsl:if test="@CF"> cf <xsl:value-of select="@CF" />
</xsl:if>
</xsl:template>
```

The `Fact` template consists simply of writing the `Fact Name` parameter, followed by a colon, then applying the `FactPattern` template, followed by a period:

```
<xsl:template match="Fact">
<xsl:value-of select="@Name" />: <xsl:apply-templates
select="FactPattern" />.
</xsl:template>
```

The `FactPattern` template appends each `FactPattern` value to the previous value using the hyphen operator:

```
<xsl:template match="FactPattern">
<xsl:if test="position() > 1"></xsl:if><xsl:value-of
select="." />
</xsl:template>
```

Because of the significant differences between M.1 and CLIPS syntax, some notion of equivalence has to be decided upon and implemented in the XSL stylesheet used to render each language. True portability between M.1 and CLIPS is not achievable using just language translation, the rulesets themselves must be analyzed to ensure that any ruleset created in one language and transformed to the other is not only syntactically valid, but logically equivalent. The LARK Engine produces valid M.1 and CLIPS syntax, but verifying the logical equivalence of rulesets is outside the scope of this thesis. This must be done using another process.

4.8.3 Translating LarkML to Lark Natural Language

To allow LARK (or any engine that must parse a language) to create natural language rules, a strict definition of a natural language must be used. Lark Natural Language is a simple expression and refactoring of LarkML, and should not be considered an effort equivalent or even analogous to the Natural Rule Language (NRL) described in section 2.4. NRL has been specified as part of a SourceForge effort to maintain an open specification “language for constraining models that attempts a high-wire task: to be machine parseable by a grammar-based automaton, to maintain human readability and to stay as close to English as possible.” (Nentwich and James) Lark Natural Language has a similar, however much simpler goal, with a much finer scope – an easily readable version of the LarkML standard.

The current implementation of LNL, as described in Appendix B, is an XSL stylesheet used to transform the abstract syntax of LarkML into a more readable format. The transformation used is of trivial complexity in comparison to the translations used to convert LarkML to CLIPS and M.1 syntax, thus no explanation is included.

4.8.4 CLIPS Rule Parsing and Conversion to LarkML

The LARK Engine demonstrates bidirectional ruleset conversion between LarkML and CLIPS rulesets. The conversion from CLIPS to LarkML is technically more


```

defrule [rulename]
  (...) *
  =>
  [(...) | (... (...))] *
)

```

The “*” character denotes between 0..N repetitions of the preceding pattern. This matches our example rule, and adds it to the CLIPSRules variable:

```

(defrule findparents
  (find-parents-of ?child )
  (parent-of ?myvar1 )
  ?mybind <- (pattern2 )
  =>
  (retract ?mybind ))

```

2) Next the rule name is captured, and the beginning of the <Rule> tag is built:

```

//Build Rule header
const string RuleHeaderPattern = @"\s*(defrule\s\b(?:<RuleName>\w+)" ;
const string RuleHeaderReplacement = @"<Rule Name=""${RuleName}"">" +
"\n" + @"<LHS>";

CLIPSRules = Regex.Replace(CLIPSRules, RuleHeaderPattern,
RuleHeaderReplacement, RegexOptions.Singleline);

```

This regex matches the rule name to the RuleName variable and then populates the Name attribute of the rule tag. Here is our example rule:

```

<Rule Name="findparents">
<LHS>
  (find-parents-of ?child )
  (parent-of ?myvar1 )
  ?mybind <- (pattern2 )
  =>
  (retract ?mybind ))

```

3) Now the rule is split into LHS and RHS, using the => symbol as the pivot point:

```

//Split Rule in RHS and LHS
const string SplitPattern = @"=>";
const string SplitReplacement = @"</LHS>" + "\n" + @"<RHS>";
CLIPSRules = Regex.Replace(CLIPSRules, SplitPattern, SplitReplacement,
RegexOptions.Singleline);

```

Our example rule:

```

<Rule Name="findparents">
<LHS>
(find-parents-of ?child )
(parent-of ?myvar1 )
?mybind <- (pattern2 )
</LHS>
<RHS>
(retract ?mybind )

```

4) The <Rule> tag is finished next:

```

//Finish Rule tag
const string RuleTagPattern =
@"<RHS>([^(]*)\([^(]*\)[^(]*|([^(]*)\([^(]*\([^(]*\)[^(]*\)[^(]*)*
*\)";
const string RuleTagReplacement = @"$&</RHS>" + "\n" + @"</Rule>";
CLIPSRules = Regex.Replace(CLIPSRules, RuleTagPattern,
RuleTagReplacement, RegexOptions.Singleline);

```

This regex is similar to the initial regex used to recognize the CLIPS rule pattern. This pattern serves to identify the last half of the rule, everything on the RHS of the rule.

Our example rule:

```

<Rule Name="findparents">
<LHS>
(find-parents-of ?child )
(parent-of ?myvar1 )
?mybind <- (pattern2 )
</LHS>
<RHS>
(retract ?mybind )
</RHS>
</Rule>

```

5) As can be seen in our above example rule there is a trailing parenthesis at the end of the rule that needs to be trimmed:

```

//Trim trailing parenthesis
const string ParenthesisPattern = @"\)</RHS>";
const string ParenthesisReplacement = "\n" + @"</RHS>";
CLIPSRules = Regex.Replace(CLIPSRules, ParenthesisPattern,
ParenthesisReplacement, RegexOptions.Singleline);

```

Our cleaned up rule:

```

<Rule Name="findparents">
<LHS>
(find-parents-of ?child )
(parent-of ?myvar1 )
?mybind <- (pattern2 )
</LHS>
<RHS>
(retract ?mybind
</RHS>
</Rule>

```

6) At this point we're ready to identify each LHS pattern and each RHS action. This requires an iterative process that is facilitated with the use of a "breadcrumb." This breadcrumb is a period, set at the end of each closing pattern tag (</Pattern.>) that makes the identification of the last pattern (or action) tag possible. We set the breadcrumbs in this step, and we iterate through all the pattern and action tags in the next step. Each iteration removes the previous breadcrumb and places it in the following tag.

```

//Create first <Pattern></Pattern> tag, starting "." breadcrumb
//Breadcrumb allows sequential <Pattern> processing
const string PatternPattern =
@"<LHS>[^()<>]*(?<Pattern>\([^()]*\)|\?[^()]*\([^()]*\))";
const string PatternReplacement = @"<LHS>" + "\n" +
@"<Pattern>${Pattern}</Pattern.>";
CLIPSRules = Regex.Replace(CLIPSRules, PatternPattern,
PatternReplacement, RegexOptions.Singleline);

//Create first <Action></Action> tag, starting "." breadcrumb
//Breadcrumb allows sequential <Action> processing
const string ActionPattern =
@"<RHS>[^()<>]*(?<Action>\([^()]*\)|\([^()]*\([^()]*\)[^()]*\))";
const string ActionReplacement = @"<RHS>" + "\n" +
@"<Action>${Action}</Action.>";
CLIPSRules = Regex.Replace(CLIPSRules, ActionPattern,
ActionReplacement, RegexOptions.Singleline);

```

Example Rule:

```

<Rule Name="findparents">
<LHS>
<Pattern>(find-parents-of ?child )</Pattern.>
(parent-of ?myvar1 )
?mybind <- (pattern2 )
</LHS>
<RHS>
<Action>(retract ?mybind </Action.>
</RHS>
</Rule>

```

7) Now that the first breadcrumb is set, each pattern and action is processed in a loop, and enclosed in tags. Every time a new tag is added, the breadcrumb is removed from the previous tag and put in the new one. Finally the last regex replacement cleans up the breadcrumb:

```
//Create each additional <Pattern></Pattern.> tag using the "."
//to follow the trail
const string PatternTagPattern =
@"</Pattern.>[^()?<>]*(?<Pattern>\([^()]*\)|\?[^()]*\([^()]*\))";
const string PatternTagReplacement = @"</Pattern>" + "\n" + @"<Pattern
Bool="and">${Pattern}</Pattern.>";
while (Regex.Replace(CLIPSRules, PatternTagPattern,
PatternTagReplacement, RegexOptions.Singleline) != CLIPSRules)
{
    CLIPSRules = Regex.Replace(CLIPSRules, PatternTagPattern,
PatternTagReplacement, RegexOptions.Singleline);
}

//Create each additional <Action></Action.> tag using the "."
//to follow the trail
const string ActionTagPattern =
@"</Action.>[^()?<>]*(?<Action>\([^()]*\)|\([^()]*\([^()]*\)[^()]*\))"
;
const string ActionTagReplacement = @"</Action>" + "\n" +
"<Action>${Action}</Action.>";
while (Regex.Replace(CLIPSRules, ActionTagPattern,
ActionTagReplacement, RegexOptions.Singleline) != CLIPSRules)
{
    CLIPSRules = Regex.Replace(CLIPSRules, ActionTagPattern,
ActionTagReplacement, RegexOptions.Singleline);
}

//Cleanup "." in </Action.> AND </Pattern.> tags
// it is no longer needed
const string CrumbCleanupPattern = @"</(?:<TagName>[^\.<>]*)\.>";
const string CrumbCleanupReplacement = @"</${TagName}>";
CLIPSRules = Regex.Replace(CLIPSRules, CrumbCleanupPattern,
CrumbCleanupReplacement, RegexOptions.Singleline);
```

Example Rule:

```
<Rule Name="findparents">
<LHS>
<Pattern>(find-parents-of ?child )</Pattern>
<Pattern Bool="and">(parent-of ?myvar1 )</Pattern>
<Pattern Bool="and">?mybind <- (pattern2 )</Pattern>
</LHS>
<RHS>
<Action>(retract ?mybind</Action>
</RHS>
</Rule>
```

Each pattern and action has been recognized, and is now wrapped in the proper tag. The very last `</Pattern>` tag contained the remaining breadcrumb, which is removed in the last statement.

8) Now Bind type patterns are processed. These are patterns that are assigned to variables, and this regex targets the following line in our example rule:

```
<Pattern Bool="and">?mybind <- (pattern2 )</Pattern>
```

This regex is built to recognize the opening of the pattern tag, followed by a question mark. The variable name following this sequence of characters is captured as is the pattern following this variable.

```
//Process "bind" type patterns
const string BindPattern =
@"<Pattern(?<PatternOptions>[>]*)>[<?()]*\?(?<BindName>\w+)\s*<-\";
const string BindReplacement = @"<Pattern ${PatternOptions}
Type="Bind" Name=" "${BindName}" ">";
CLIPSRules = Regex.Replace(CLIPSRules, BindPattern, BindReplacement,
RegexOptions.Singleline);
```

This refactors the example rule into:

```
<Rule Name="findparents">
<LHS>
<Pattern>(find-parents-of ?child )</Pattern>
<Pattern Bool="and">(parent-of ?myvar1 )</Pattern>
<Pattern Bool="and" Type="Bind" Name="mybind"> (pattern2
)</Pattern>
</LHS>
<RHS>
<Action>(retract ?mybind </Action>
</RHS>
</Rule>
```

9) The next step is to clean up the parentheses found in the action and pattern tags:

```
//Remove parentheses in Action/Pattern tags
const string ParenTagPattern =
@"<(?(?<TagType>Pattern|Action)(?<Options>[>]*)>\s*\(((?<Atoms>[<()>]*)
\)</\1>";
const string ParenTagReplacement =
@"<${TagType}${Options}>${Atoms}</${TagType}>";
CLIPSRules = Regex.Replace(CLIPSRules, ParenTagPattern,
ParenTagReplacement, RegexOptions.Singleline);
```

The regex in this step recognizes the opening of both pattern and action tags. The tag type is bound to a variable `TagType`, pattern options (like the `Bind` parameter for example) are bound to the `Options` variable, and the contents of the pattern are bound

to the Atoms variable. Finally the Replace() function refactors the rule to filter out the parentheses.

Example rule, sans parentheses:

```
<Rule Name="findparents">
<LHS>
<Pattern>find-parents-of ?child </Pattern>
<Pattern Bool="and">parent-of ?myvar1 </Pattern>
<Pattern Bool="and" Type="Bind" Name="mybind">pattern2
</Pattern>
</LHS>
<RHS>
<Action>retract ?mybind </Action>
</RHS>
</Rule>
```

10) The next two steps process assert, retract, and print actions. Assert and retract tags are specific to CLIPS, and have no analog in M.1.

```
//Process "Assert" and "Retract" actions
const string FactActionPattern =
@"<Action(?<PatternOptions>[>]*)>\s*(\s*)(?<ActionType>assert|retract)\s*(\s*)(?<Atoms>[^(<>)]*\s*(\s*))\s*(\s*)\s*</Action>";
const string FactActionReplacement = @"<Action${PatternOptions}
Type=""${ActionType}"">${Atoms}</Action>";
CLIPSRules = Regex.Replace(CLIPSRules, FactActionPattern,
FactActionReplacement, RegexOptions.Singleline);

//Process print commands
const string PrintPattern =
@"<Action(?<TagOptions>[>]*)>\s*printout\s*t\s*(?<Innards>[<]*?)\s*(
crLf|)\s*</Action>";
const string PrintReplacement = @"<Action${TagOptions} Type=""Print""
Destination=""Console"">${Innards}</Action>";
CLIPSRules = Regex.Replace(CLIPSRules, PrintPattern, PrintReplacement,
RegexOptions.Singleline);
```

Our example rule does not contain a print command, but we can see an example print tag:

```
<Action Type="Print" Destination="Console">Print
this.</Action>
```

Assert and retract are Type options in the Action tag, shown in the example rule:

```
<Rule Name="findparents">
<LHS>
<Pattern>find-parents-of ?child </Pattern>
<Pattern Bool="and">parent-of ?myvar1 </Pattern>
```

```

<Pattern Bool="and" Type="Bind" Name="mybind">pattern2
</Pattern>
</LHS>
<RHS>
<Action Type="retract">?mybind </Action>
</RHS>
</Rule>

```

11) Atom tags are processed next. Atoms are alphanumeric character sequences separated by whitespace. These are processed in a similar fashion as the pattern and action tags, first by adding a breadcrumb, then by iterating through each atom moving the breadcrumb forward, and finally removing the breadcrumb from the last atom tag.

```

//Process first Atom adding "." breadcrumb
const string AtomPattern =
@"<(?(?<TagType>Pattern|Action)(?<TagOptions>[ ^>]*)>\s*(?<Var>[ ^\s]*)(<?<
Remainder>[ ^(<>]*)</\1>">";
const string AtomReplacement = @"<${TagType}${TagOptions}>" + "\n" +
@"<Atom>${Var}</Atom.>${Remainder}" + "\n" + @"</${TagType}>";
CLIPSRules = Regex.Replace(CLIPSRules, AtomPattern, AtomReplacement,
RegexOptions.Singleline);

//Process remaining atoms
const string AtomCrumbPattern =
@"</Atom.>\s*(?<Var>\\"[ ^\" ]*\b\\"| [ ^\s ]*\b)(?<Remainder>[ ^< ]*)\s*</(?
<TagType>Pattern|Action)>";
const string AtomCrumbReplacement =
@"</Atom.><Atom>${Var}</Atom.>${Remainder}</${TagType}>";
while (Regex.Replace(CLIPSRules, AtomCrumbPattern,
AtomCrumbReplacement, RegexOptions.Singleline) != CLIPSRules)
{
    CLIPSRules = Regex.Replace(CLIPSRules, AtomCrumbPattern,
AtomCrumbReplacement, RegexOptions.Singleline);
}

//Breadcrumb cleanup, remove "." from <Atom.> tag
const string AtomCleanPattern = @"</Atom\.">";
const string AtomCleanReplacement = @"</Atom>";
CLIPSRules = Regex.Replace(CLIPSRules, AtomCleanPattern,
AtomCleanReplacement, RegexOptions.Singleline);

```

Our example rule patterns now contain the proper atom tags:

```

<Rule Name="findparents">
<LHS>
<Pattern>
<Atom>find-parents-of</Atom><Atom>?child</Atom>
</Pattern>
<Pattern Bool="and">
<Atom>parent-of</Atom><Atom>?myvar1</Atom>
</Pattern>
<Pattern Bool="and" Type="Bind" Name="mybind">

```

```

<Atom>pattern2</Atom>
</Pattern>
</LHS>
<RHS>
<Action Type="retract">
<Atom>?mybind</Atom>
</Action>
</RHS>
</Rule>

```

12) Finally the atoms themselves are processed. CLIPS variables are identified by recognizing question marks followed by alphanumeric strings. Strings are identified as character sequences that are enclosed in quotations, and the tilde “~” operator denotes a negated atom:

```

//Process "var" atoms
const string VarPattern = @"<Atom>\s*\?(?<Innards>[^\<]*)\s*</Atom>";
const string VarReplacement = @"<Atom Type=""Var"">${Innards}</Atom>";
CLIPSRules = Regex.Replace(CLIPSRules, VarPattern, VarReplacement,
RegexOptions.Singleline);

//Process "string" atoms
const string StringPattern =
@"<Atom>\s*\""(?<Innards>[^\"]*)\""\s*</Atom>";
const string StringReplacement = @"<Atom
Type=""String"">${Innards}</Atom>";
CLIPSRules = Regex.Replace(CLIPSRules, StringPattern,
StringReplacement, RegexOptions.Singleline);

//Process negated atoms
const string NegatePattern =
@"<Atom(?<AtomOptions>[^\>]*)>\s*~(?<Innards>[^\<]*)\s*</Atom>";
const string NegateReplacement = @"<Atom${AtomOptions}
Negate=""true"">${Innards}</Atom>";
CLIPSRules = Regex.Replace(CLIPSRules, NegatePattern,
NegateReplacement, RegexOptions.Singleline);

```

Both variable and string type atoms are specified in the Type parameter of the Atom tag.

```

<Rule Name="findparents">
<LHS>
<Pattern>
<Atom>find-parents-of</Atom><Atom Type="Var">child</Atom>
</Pattern>
<Pattern Bool="and">
<Atom>parent-of</Atom><Atom Type="Var">myvar1</Atom>
</Pattern>
<Pattern Bool="and" Type="Bind" Name="mybind">
<Atom>pattern2</Atom>
</Pattern>
</LHS>

```

```

<RHS>
<Action Type="retract">
<Atom Type="Var">mybind</Atom>
</Action>
</RHS>
</Rule>

```

It should be noted that this parsing algorithm only serves to process certain CLIPS rule-types, and that more complex rule patterns and features exist that may not be recognized by this process. Any unmatched rule patterns are ignored in step 1. Any unrecognized rule features are processed according to the regular expressions described above. CLIPS rules that match step 1 that contain features not specifically matched in the following steps are still processed.

The LARK Engine serves to create a baseline CLIPS parsing engine that can later be extended, as appropriate.

4.8.5 CLIPS Fact Parsing and Conversion to LarkML

The current version of the LARK Engine parses CLIPS assertion statements in lieu of processing deffacts constructs. Because of the inherent complexity of deffacts constructs in addition to the abstraction between the deffacts definition and declaration, the decision was made to process assertion statements and to allow deffacts parsing and processing to be included in a later version of the LARK Engine. This section details the parsing of CLIPS assertion statements.

An initial string is established to put our facts into, CLIPSFacts:

```
string CLIPSFacts = ""; //String to add CLIPS facts to
```

To discover the fact assertions in our ruleset, first we take the entire ruleset, then subtract all the rules. Because the regex used to match fact assertions will also match portions of rules, we need to first remove all rules from the ruleset before we search for fact assertions to eliminate any false positives:

```

//Set FactMatches equal to the source ruleset, after CLIPS
//rules have been removed
const string RulePattern =
@"\((defrule([\^()]*\([\^()]*\)[\^()]*=>([\^()]*\([\^()]*\)[\^()]*|[\^()]*\([\^()]*\([\^()]*\)[\^()]*\)[\^()]*\)[\^()]*\)*";
const string RuleReplacement = "";
string FactMatches = Regex.Replace(GetSourceRules(), RulePattern,
RuleReplacement, RegexOptions.Singleline);

```

Now we use our fact matching regex to discover the fact assertions in what remains of our ruleset:

```

//Append fact assertions to CLIPSFacts using while loop
//to match assertions
const string FactPattern = @"\s*assert\s*\([\^()]*\)\s*";
Regex FactRegex = new Regex(FactPattern, RegexOptions.Singleline);
//Facts REGEX matches CLIPS assertion format

```

```

foreach (Match item in FactRegex.Matches(FactMatches))
{
    CLIPSFacts += item.Value + "\n\n";
}

```

To provide an example of how the assertions are processed, we use example fact assertions, that match the above loop and are appended to CLIPSFacts:

```

(assert (ingredients "mustard" "garlic" "cayenne"))
(assert (numbers 3 5 2 8))
(assert (friends Maynard Reznor Hendrix))

```

We begin processing our facts by wrapping them in a <Fact> tag, and creating a parameter to populate the fact name:

```

//Process assertions
const string AssertPattern =
@"\(\s*assert\s*\(((?<Innards>[^\s]*)\)\)\s*\)";
const string AssertReplacement = @"<Fact Name="">${Innards}</Fact>";
CLIPSFacts = Regex.Replace(CLIPSFacts, AssertPattern,
AssertReplacement, RegexOptions.Singleline);

```

Example facts:

```

<Fact Name="">ingredients "mustard" "garlic"
"cayenne"</Fact>
<Fact Name="">numbers 3 5 2 8</Fact>
<Fact Name="">friends Maynard Reznor Hendrix</Fact>

```

Because CLIPS does not allow for the naming of fact assertions, and there is value in providing a name for translation to other languages (specifically M.1), we assign an arbitrary, unique name to each fact:

```

//Name facts - create fact name with integers in temporaryFacts
string NamedFacts = "";
//Named facts string
FactRegex = new Regex(@"(?<Begin><Fact\s*Name="">)",
RegexOptions.Singleline); //Match on facts
int factNumber = 0;
//Integer used to name facts
foreach (Match item in FactRegex.Matches(CLIPSFacts))
//Use foreach loop to name facts
{
    NamedFacts += item.Value + "fact" + (++factNumber).ToString() +
"\n\n";
}

//Add NamedFacts to CLIPSFacts after processing into LarkML
factNumber = 0;
//Set factNumber to 0
FactRegex = new Regex(@"(?<End>"">[^\s]*</Fact>)",
RegexOptions.Singleline); //Match on <Facts>

```

```

foreach (Match item in FactRegex.Matches(CLIPSFacts))
//Process each fact
{
    //Match on each fact, using factNumber in the
    //REGEX to name match, then append the fact value
    NamedFacts = Regex.Replace(NamedFacts,
@"(?<Begin><Fact\s*Name="fact" + (++factNumber).ToString() + ")",
@"${Begin}" + item.Value, RegexOptions.Singleline);
}
CLIPSFacts = NamedFacts; //Set CLIPSFacts to NamedFacts

```

Example facts:

```

<Fact Name="fact1">ingredients "mustard" "garlic"
"caiyenne"</Fact>
<Fact Name="fact2">numbers 3 5 2 8</Fact>
<Fact Name="fact3">friends Maynard Reznor Hendrix</Fact>

```

Our facts must now be divided into fact pattern tags and processed using the breadcrumb methodology, as was done with the CLIPS rules processing in the previous section. After breadcrumb processing is complete, the final breadcrumb is removed:

```

//Process first FactPattern adding "." breadcrumb,
//then process remaining atoms
const string FactCrumbPattern =
@"(?<Tag><Fact\s*[^>]*>)\s*(?<Var>[\s]*)(?<Remainder>[^(<>)]*)</Fact>";
const string FactCrumbReplacement = @"${Tag}" + "\n" +
@"<FactPattern>${Var}</FactPattern.>" + "\n" + @"${Remainder}</Fact>";
CLIPSFacts = Regex.Replace(CLIPSFacts, FactCrumbPattern,
FactCrumbReplacement, RegexOptions.Singleline);

//Using "." breadcrumb, process each <FactPattern> tag
const string CrumbPattern =
@"</FactPattern.>\s*(?<Var>\"\"[^\"]*\b\"\"|[\s]*\b)(?<Remainder>[^<]*)\s*</Fact>";
const string CrumbReplacement = @"</FactPattern>" + "\n" +
@"<FactPattern>${Var}</FactPattern.>" + "\n" + @"${Remainder}</Fact>";
while (Regex.Replace(CLIPSFacts, CrumbPattern, CrumbReplacement,
RegexOptions.Singleline) != CLIPSFacts)
{
    //Set CLIPSFacts equal to temporaryFacts as long as they
    //are different, thus processing each element
    CLIPSFacts = Regex.Replace(CLIPSFacts, CrumbPattern,
CrumbReplacement, RegexOptions.Singleline);
}

//Breadcrumb cleanup of <FactPattern> tag
const string CrumbCleanPattern = @"</FactPattern.>";
const string CrumbCleanReplacement = @"</FactPattern>";
CLIPSFacts = Regex.Replace(CLIPSFacts, CrumbCleanPattern,
CrumbCleanReplacement, RegexOptions.Singleline);

```

Example facts:

```
<Fact Name="fact1">
<FactPattern>ingredients</FactPattern>
<FactPattern>"mustard"</FactPattern>
<FactPattern>"garlic"</FactPattern>
<FactPattern>"cayenne"</FactPattern>
</Fact>
```

```
<Fact Name="fact2">
<FactPattern>numbers</FactPattern>
<FactPattern>3</FactPattern>
<FactPattern>5</FactPattern>
<FactPattern>2</FactPattern>
<FactPattern>8</FactPattern>
</Fact>
```

```
<Fact Name="fact3">
<FactPattern>friends</FactPattern>
<FactPattern>Maynard</FactPattern>
<FactPattern>Reznor</FactPattern>
<FactPattern>Hendrix</FactPattern>
</Fact>
```

Finally fact patterns are processed, accounting for string matches:

```
//Process "string" FactPatterns
const string StringPattern =
@"<FactPattern>\s*\\"(?<Innards>[^\"]*)\\"\\s*</FactPattern>";
const string StringReplacement = @"<FactPattern
Type="String">${Innards}</FactPattern>";
CLIPSFacts = Regex.Replace(CLIPSFacts, StringPattern,
StringReplacement, RegexOptions.Singleline);
```

Example facts:

```
<Fact Name="fact1">
<FactPattern>ingredients</FactPattern>
<FactPattern Type="String">mustard</FactPattern>
<FactPattern Type="String">garlic</FactPattern>
<FactPattern Type="String">cayenne</FactPattern>
</Fact>
```

```
<Fact Name="fact2">
<FactPattern>numbers</FactPattern>
<FactPattern>3</FactPattern>
<FactPattern>5</FactPattern>
<FactPattern>2</FactPattern>
<FactPattern>8</FactPattern>
```

```

</Fact>

<Fact Name="fact3">
<FactPattern>friends</FactPattern>
<FactPattern>Maynard</FactPattern>
<FactPattern>Reznor</FactPattern>
<FactPattern>Hendrix</FactPattern>
</Fact>

```

The CLIPS fact processing facility provided by the LARK Engine is an initial effort with a logical step forward: the processing of deffacts data structures. Future versions of the LARK Engine will take this evolutionary improvement under consideration.

4.8.6 M.1 Rule Parsing and Conversion to LarkML

M.1 rules are parsed in a fashion similar to the CLIPS rule parsing described in section 4.8.4. The M.1 rule format is discussed in section 2.3.3, which shows the following general rule format:

```

[rule label]:
  if
    [premise clause 1]...[premise clause N]
  then
    [conclusion clause 1]...[conclusion clause N]

```

The rules begin with a label, followed by the `if` keyword, and then 1 to N premise clauses. The rule LHS and RHS portions are divided via the `then` keyword, which is then followed by 1 to N conclusion clauses. The strategy for LarkML conversion consists of identifying and converting key portions of the M.1 rule syntax, such as the premise clauses and variables, and then extracting the relevant portions and refactoring them into LarkML. This section details the regex mechanisms used for this refactoring process and demonstrates rule conversion using an example rule.

1) Read incoming M.1 ruleset and identify rules via the regex match function. All rules are appended to string `M1Rules`:

```

//Create strings
string M1Rules = "";

//Rules REGEX defines a general rule regular expression
//used for matching with M1 rules
const string RulesPattern =
@"(?<RuleName>[^\s]*)\s*:\s*if(?:<LHS>.*?)then(?:<RHS>[^\.]*)\.";
Regex RulesRegex = new Regex(RulesPattern, RegexOptions.Singleline);

foreach (Match item in RulesRegex.Matches(GetSourceRules()))

```

```
//Analyzes source rules for M1 rules meeting the Rules REGEX format
{
    M1Rules += item.Value + "\n\n";
}
```

The regex `RuleRegex` matches on the general M.1 rule format discussed above, then as each rule is matched, it is appended to the `M1Rules` string. The regex matches the two following example rules:

```
rule-3:  if not(tastiness = average)
         and display(['Tastiness not average'])
         then best-body = light cf 30.

v-rule-3: if best-X is unknown and
          preferred-X is unknown and
          default-X = V
          then recommended-X = V.
```

2) The rules contained within `M1Rules` are now divided into their respective LHS and RHS portions, while the rule names are captured:

```
//Process Split rules into LHS and RHS
const string SplitPattern =
@"(?<RuleName>[^\s]*)\s*:\s*if\s*(?<LHS>.*?)\s+then\s+(?<RHS>[^\s]*?)\s*\.";
const string SplitReplacement = @"<Rule Name=""${RuleName}"">" + "\n"
+ "<LHS>${LHS}</LHS>" + "\n" + "<RHS>${RHS}</RHS>" + "\n" + "</Rule>";
M1Rules = Regex.Replace(M1Rules, SplitPattern, SplitReplacement,
RegexOptions.Singleline);
```

The example rules are now:

```
<Rule Name="rule-3">
<LHS>not(tastiness = average)
    and display(['Tastiness not average'])</LHS>
<RHS>best-body = light cf 30</RHS>
</Rule>

<Rule Name="v-rule-3">
<LHS>best-X is unknown and
    preferred-X is unknown and
    default-X = V</LHS>
<RHS>recommended-X = V</RHS>
</Rule>
```

3) The next two regex refactorings identify the first of the LHS patterns and RHS actions, and set the breadcrumb “.” used for iterative processing, as first described in section 4.8.4:

```

//Process first <Pattern><Atom>${Pattern}</Atom></Pattern.>, setting
breadcrumb
const string PatternCrumbPattern =
@"<LHS>(?(?<Pattern>(?:(?:\['^']*\\')|(?:[^<]*?))*)(?<Ending>\b(and|or)\b
[^\<]*</LHS>|</LHS>)" ;
const string PatternCrumbReplacement = @"<LHS>" + "\n" + @"<Pattern>"
+ "\n" + @"<Atom>${Pattern}</Atom>" + "\n" + @"</Pattern.>${Ending}";
MlRules = Regex.Replace(MlRules, PatternCrumbPattern,
PatternCrumbReplacement, RegexOptions.Singleline);

//Process first <Action><Atom>${Pattern}</Atom></Action.>, setting
breadcrumb
const string AtomCrumbPattern =
@"<RHS>(?(?<Action>(?:(?:\['^']*\\')|(?:[^<]*?))*)(?<Ending>\b(and|or)\b
[^\<]*</RHS>|</RHS>)" ;
const string AtomCrumbReplacement = @"<RHS>" + "\n" + @"<Action>" +
"\n" + @"<Atom>${Action}</Atom>" + "\n" + @"</Action.>${Ending}";
MlRules = Regex.Replace(MlRules, AtomCrumbPattern,
AtomCrumbReplacement, RegexOptions.Singleline);

```

Once these breadcrumbs are set, the example rules read as such:

```

<Rule Name="rule-3">
<LHS>
<Pattern>
<Atom>not(tastiness = average)</Atom>
</Pattern.>and display(['Tastiness not average'])</LHS>
<RHS>
<Action>
<Atom>best-body = light cf 30</Atom>
</Action.></RHS>
</Rule>

<Rule Name="v-rule-3">
<LHS>
<Pattern>
<Atom>best-X is unknown </Atom>
</Pattern.>and preferred-X is unknown and default-X =
V</LHS>
<RHS><Action>
<Atom>recommended-X = V</Atom>
</Action.></RHS>
</Rule>

```

4) Now that the breadcrumbs are set, each LHS pattern and RHS action tag can be created by using the breadcrumb as a reference point, and then creating new tags upon the discovery of additional syntax beyond the last breadcrumb:

```

//Loop through creating <Pattern><Atom> Pairs, using
//breadcrumb (.) and populating our Bool parameters

```

```

const string PatternAtomPattern =
@"</Pattern\.\>\s*(?<Bool>and|or)\s*(?<Pattern>\w(?:\:(?:\['^']*\\')|(?:[^<
<]*?))*)\s*(?<Ending>\b(and|or)\b[^\<]*</LHS>|</LHS>)" ;
const string PatternAtomReplacement = @"</Pattern>" + "\n" +
@"<Pattern Bool=""${Bool}"">" + "\n" + @"<Atom>${Pattern}</Atom>" +
"\n" + @"</Pattern.\>${Ending}";
while (Regex.Replace(M1Rules, PatternAtomPattern,
PatternAtomReplacement, RegexOptions.Singleline) != M1Rules)
{
    M1Rules = Regex.Replace(M1Rules, PatternAtomPattern,
PatternAtomReplacement, RegexOptions.Singleline);
}

//Loop through creating <Action><Atom> Pairs, using
//breadcrumb (.) and populating our Bool parameters
const string ActionAtomPattern =
@"</Action\.\>\s*(?<Bool>and|or)\s*(?<Innards>\w(?:\:(?:\['^']*\\')|(?:[^<
<]*?))*)\s*(?<Ending>\b(and|or)\b[^\<]*</RHS>|</RHS>)" ;
const string ActionAtomReplacement = @"</Action>" + "\n" + @"<Action
Bool=""${Bool}"">" + "\n" + @"<Atom>${Innards}</Atom>" + "\n" +
@"</Action.\>${Ending}";
while (Regex.Replace(M1Rules, ActionAtomPattern,
ActionAtomReplacement, RegexOptions.Singleline) != M1Rules)
{
    M1Rules = Regex.Replace(M1Rules, ActionAtomPattern,
ActionAtomReplacement, RegexOptions.Singleline);
}

```

Our example rules now contain LHS pattern and RHS action tags:

```

<Rule Name="rule-3">
<LHS>
<Pattern>
<Atom>not(tastiness = average)</Atom>
</Pattern>
<Pattern Bool="and">
<Atom>display(['Tastiness not average'])</Atom>
</Pattern.\>
</LHS>
<RHS>
<Action>
<Atom>best-body = light cf 30</Atom>
</Action.\></RHS>
</Rule>

<Rule Name="v-rule-3">
<LHS>
<Pattern>
<Atom>best-X is unknown </Atom>
</Pattern>
<Pattern Bool="and">
<Atom>preferred-X is unknown</Atom>

```

```

</Pattern>
<Pattern Bool="and">
<Atom>default-X = V</Atom>
</Pattern.>
</LHS>
<RHS>
<Action>
<Atom>recommended-X = V</Atom>
</Action.>
</RHS>
</Rule>

```

5) The breadcrumbs are now removed. For rules containing the not Boolean operator, the Negate parameter is created. M.1 expressions containing the equals sign with and without a confidence factor modifier are processed as well:

```

//Cleanup breadcrumbs
const string CrumbPattern =
@"</(?:<Tag>Pattern|Action)\.></(?:<ClosingTag>RHS|LHS)>";
const string CrumbReplacement = @"</${Tag}>" + "\n" +
@"</${ClosingTag}>";
MlRules = Regex.Replace(MlRules, CrumbPattern, CrumbReplacement,
RegexOptions.Singleline);

//Process "not" boolean operator, insert Negate parameter
const string NegatePattern =
@"<Pattern(?:<PatternOptions>[^\>]*)>\s*<Atom(?:<AtomOptions>[^\>]*)>\s*no
t\s*\(((?:<Innards>[^\>]*)\s*\)\s*</Atom>";
const string NegateReplacement = @"<Pattern${PatternOptions}
Negate="true"><Atom${AtomOptions}>${Innards}</Atom>";
MlRules = Regex.Replace(MlRules, NegatePattern, NegateReplacement,
RegexOptions.Singleline);

//Process Equals tags with CF modifier
const string CFPattern =
@"<Atom(?:<TagOptions>[^\>]*)>(?:<Pattern>[^\<]*?)\s*=\s*(?:<Value>[^\<]*?)\
s*cf\s*(?:<CF>-?[0-9]*)\s*</Atom>";
const string CFReplacement =
@"<Atom${TagOptions}>${Pattern}</Atom><Equals
CF="${CF}"><Atom>${Value}</Atom></Equals>";
MlRules = Regex.Replace(MlRules, CFPattern, CFReplacement,
RegexOptions.Singleline);

//Process Equals tags without CF modifier
const string EqualsPattern =
@"<Atom(?:<TagOptions>[^\>]*)>(?:<Pattern>[^\<]*?)\s*(=|\s|\s)\s*(?:<Value
>[^\<]*?)\s*</Atom>";
const string EqualsReplacement =
@"<Atom${TagOptions}>${Pattern}</Atom><Equals><Atom>${Value}</Atom></E
quals>";
MlRules = Regex.Replace(MlRules, EqualsPattern, EqualsReplacement,
RegexOptions.Singleline);

```

After the refactorings above, the example rules read as such:

```
<Rule Name="rule-3">
  <LHS>
    <Pattern Negate="true">
      <Atom>tastiness</Atom>
      <Equals><Atom>average</Atom></Equals>
    </Pattern>
    <Pattern Bool="and">
      <Atom>display(['Tastiness not average'])</Atom>
    </Pattern>
  </LHS>
  <RHS>
    <Action>
      <Atom>best-body</Atom>
      <Equals CF="30"><Atom>light</Atom></Equals>
    </Action>
  </RHS>
</Rule>
```

```
<Rule Name="v-rule-3">
  <LHS>
    <Pattern>
      <Atom>best-X</Atom>
      <Equals><Atom>unknown</Atom></Equals>
    </Pattern>
    <Pattern Bool="and">
      <Atom>preferred-X</Atom>
      <Equals><Atom>unknown</Atom></Equals>
    </Pattern>
    <Pattern Bool="and">
      <Atom>default-X</Atom>
      <Equals><Atom>V</Atom></Equals>
    </Pattern>
  </LHS>
  <RHS>
    <Action>
      <Atom>recommended-X</Atom>
      <Equals><Atom>V</Atom></Equals>
    </Action>
  </RHS>
</Rule>
```

6) M.1 display statements are converted to Print type actions, then each portion of the statement that is separated by the comma character is split into individual atoms. In M.1, the “-” operator is used to allow wildcards (variables) to bind to specific parts of an

expression. To properly express M.1 variables in LarkML, each portion of the expression delimited by a hyphen must be separated and housed in a separate tag:

```
//Create Display when there is at least one action
const string DisplayActionPattern =
@"\s*<Pattern[^>]*>\s*<Atom>\s*display\s*\(\s*\[\s*(?<PrintMe>.*?)\s*\]
\s*\)\s*</Atom>\s*</Pattern>\s*(?<MoveMe>.*?<RHS>)\s*<Action>";
const string DisplayActionReplacement = @"${MoveMe}" + "\n" +
@"<Action Type=""Print""
Destination=""Console""><Atom>${PrintMe}</Atom></Action><Action>";
M1Rules = Regex.Replace(M1Rules, DisplayActionPattern,
DisplayActionReplacement, RegexOptions.Singleline);

//Create Display when there are no actions
const string DisplayPattern =
@"\s*<Pattern[^>]*>\s*<Atom>\s*display\s*\(\s*\[\s*(?<PrintMe>.*?)\s*\]
\s*\)\s*</Atom>\s*</Pattern>\s*(?<MoveMe>.*?<RHS>)\s*</RHS>";
const string DisplayReplacement = @"${MoveMe}" + "\n" + @"<Action
Type=""Print""
Destination=""Console""><Atom>${PrintMe}</Atom></Action></RHS>";
M1Rules = Regex.Replace(M1Rules, DisplayPattern, DisplayReplacement,
RegexOptions.Singleline);

//Separate atoms hinging on the "-" operator
const string HyphenPattern = @"<Atom(?<TagOptions>[^>]*)>(?!<Atom>[^-
<]*)-(?!<Remainder>[^<]*)</Atom>";
const string HyphenReplacement =
@"<Atom${TagOptions}>${Atom}</Atom><Atom>${Remainder}</Atom>";
while (Regex.Replace(M1Rules, HyphenPattern, HyphenReplacement,
RegexOptions.Singleline) != M1Rules)
{
    M1Rules = Regex.Replace(M1Rules, HyphenPattern, HyphenReplacement,
RegexOptions.Singleline);
}

//Separate print statements hinging on the "," operator
const string CommaPattern =
@"<Atom(?<AtomOptions>[^>]*)>(?!<Innards>(?!'[^<]*'|'[^<]*')),(?<Remain
der>[^<]*)</Atom>";
const string CommaReplacement =
@"<Atom${AtomOptions}>${Innards}</Atom><Atom>${Remainder}</Atom>";
while (Regex.Replace(M1Rules, CommaPattern, CommaReplacement,
RegexOptions.Singleline) != M1Rules)
{
    M1Rules = Regex.Replace(M1Rules, CommaPattern, CommaReplacement,
RegexOptions.Singleline);
}
```

After processing the display statements and separating expressions with the “-” operator, the example rules are as follows:

```
<Rules>
<Rule Name="rule-3">
```

```

<LHS>
<Pattern Negate="true">
<Atom>tastiness</Atom>
<Equals><Atom>average</Atom></Equals>
</Pattern></LHS>
<RHS>
<Action Type="Print" Destination="Console">
<Atom>'Tastiness not average'</Atom>
</Action>
<Action>
<Atom>best</Atom>
<Atom>body</Atom>
<Equals CF="30"><Atom>light</Atom></Equals>
</Action>
</RHS>
</Rule>

<Rule Name="v-rule-3">
<LHS>
<Pattern>
<Atom>best</Atom><Atom>X</Atom>
<Equals><Atom>unknown</Atom></Equals>
</Pattern>
<Pattern Bool="and">
<Atom>preferred</Atom>
<Atom>X</Atom>
<Equals><Atom>unknown</Atom></Equals>
</Pattern>
<Pattern Bool="and">
<Atom>default</Atom>
<Atom>X</Atom>
<Equals><Atom>V</Atom></Equals>
</Pattern>
</LHS>
<RHS>
<Action>
<Atom>recommended</Atom>
<Atom>X</Atom>
<Equals><Atom>V</Atom></Equals>
</Action>
</RHS>
</Rule>

```

7) The last steps in M.1 rule processing consist of identifying variables (denoted by an expression term beginning with either a capital letter or an underscore), processing strings, and finally wrapping the entire ruleset in <Rules></Rules> tags:

```

//Assign variables
const string VarPattern = @"<Atom(?:<TagOptions>[^\>]*)>\s*(?:<Var>[A-Z_]+[^\<]*)\s*</Atom>";
const string VarReplacement = @"<Atom${TagOptions}
Type=""Var"">${Var}</Atom>";
MlRules = Regex.Replace(MlRules, VarPattern, VarReplacement,
RegexOptions.Singleline);

//Process ' ' strings
const string ApostrophePattern=
@"<Atom(?:<Options>[^\>]*)>\s*'(?<String>[^\'])*'\s*</Atom>";
const string ApostropheReplacement = @"<Atom${Options}
Type=""String"">${String}</Atom>";
MlRules = Regex.Replace(MlRules, ApostrophePattern,
ApostropheReplacement, RegexOptions.Singleline);

//Process "" strings
const string QuotePattern =
@"<Atom(?:<Options>[^\>]*)>\s*""(?<String>[^\"])*""\s*</Atom>";
const string QuoteReplacement = @"<Atom${Options}
Type=""String"">${String}</Atom>";
MlRules = Regex.Replace(MlRules, QuotePattern, QuoteReplacement,
RegexOptions.Singleline);

//Return MlRules inside of <Rules> tags
return "<Rules>" + "\n" + MlRules + "</Rules>" + "\n";

```

Example rules:

```

<Rules>
<Rule Name="rule-3">
<LHS>
<Pattern Negate="true">
<Atom>tastiness</Atom>
<Equals><Atom>average</Atom></Equals>
</Pattern></LHS>
<RHS>
<Action Type="Print" Destination="Console">
<Atom Type="String">Tastiness not average</Atom>
</Action>
<Action><Atom>best</Atom><Atom>body</Atom>
<Equals CF="30"><Atom>light</Atom></Equals>
</Action>
</RHS>
</Rule>

<Rule Name="v-rule-3">
<LHS>
<Pattern>
<Atom>best</Atom>
<Atom Type="Var">X</Atom>

```

```

<Equals><Atom>unknown</Atom></Equals>
</Pattern>
<Pattern Bool="and">
<Atom>preferred</Atom>
<Atom Type="Var">X</Atom>
<Equals><Atom>unknown</Atom></Equals>
</Pattern>
<Pattern Bool="and">
<Atom>default</Atom>
<Atom Type="Var">X</Atom>
<Equals><Atom Type="Var">V</Atom></Equals>
</Pattern>
</LHS>
<RHS>
<Action>
<Atom>recommended</Atom>
<Atom Type="Var">X</Atom>
<Equals><Atom Type="Var">V</Atom></Equals>
</Action>
</RHS>
</Rule>
</Rules>

```

Once the M.1 rules are parsed and refactored into LarkML, the next step is to parse and refactor the M.1 facts, described in the following section.

4.8.7 M.1 Fact Parsing and Conversion to LarkML

M.1 facts processing is mostly a subset of the regex replace functions used in the M.1 rule processing function, therefore it would be superfluous to include example facts. M.1 fact conventions are discussed at length in section 2.3.1. The process described below starts with the initialization of a string variable to store the facts found by searching the imported M.1 ruleset for the following general pattern:

```
[label] : [expression] = [value] cf [integer value of cf]
```

M.1 fact parsing begins with removing M.1 rules from the imported ruleset, and then appending facts matching the M.1 fact regex to the `M1Facts` string:

```

string M1Facts = "";           //String to add M1 facts

//Set MatchFacts equal to the source ruleset, after M1 rules
//have been removed
const string RulePattern =
@"(?<RuleName>[^\s]*)\s*:\s*if(?<LHS>.*?)then(?<RHS>[^\.]*)\.";
const string RuleReplacement = "";

```

```

string MatchFacts = Regex.Replace(GetSourceRules(), RulePattern,
RuleReplacement, RegexOptions.Singleline);

//Append fact assertions to M1Facts using while loop
//to match assertions
const string FactPattern =
@"(?<FactName>[^\s]*)\s*:\s*(?<Fact>[^\s]*?)\s*\.";
Regex FactRegex = new Regex(FactPattern, RegexOptions.Singleline);
//Facts REGEX matches M1 fact format
foreach (Match item in FactRegex.Matches(MatchFacts))
{
    M1Facts += "<Fact>" + item.Value + "</Fact>" + "\n\n";
}

```

Fact names are gleaned from the fact label, confidence factors and expressions are processed:

```

//Assign fact names
const string NamePattern =
@"<Fact>(?!<FactName>[^\s]*)\s*:\s*(?<Fact>[^\s]*?)\s*\.</Fact>";
const string NameReplacement = @"<Fact
Name=""${FactName}""><FactPattern>${Fact}</FactPattern></Fact>";
M1Facts = Regex.Replace(M1Facts, NamePattern, NameReplacement,
RegexOptions.Singleline);

//Process CF and Equals
const string CFPattern =
@"<FactPattern>(?!<Pattern>[^\s]*?)\s*=\s*(?<Value>[^\s]*?)\s*cf\s*(?<CF>
-?[0-9]*)\s*</FactPattern>";
const string CFReplacement =
@"<FactPattern>${Pattern}</FactPattern><Equals
CF=""${CF}""><Atom>${Value}</Atom></Equals>";
M1Facts = Regex.Replace(M1Facts, CFPattern, CFReplacement,
RegexOptions.Singleline);

//Process Equals
const string EqualsPattern =
@"<FactPattern>(?!<Pattern>[^\s]*?)\s*(=|\s|\s)\s*(?<Value>[^\s]*?)\s*</
FactPattern>";
const string EqualsReplacement =
@"<FactPattern>${Pattern}</FactPattern><Equals><Atom>${Value}</Atom></
Equals>";
M1Facts = Regex.Replace(M1Facts, EqualsPattern, EqualsReplacement,
RegexOptions.Singleline);

```

As discussed in section 4.8.6, expressions containing the “-” operator are split into individual atoms:

```

//Separate atoms hinging on the "-" operator
const string HyphenPattern = @"<Atom(?<TagOptions>[^\s]*)>(?!<Atom>[^\s
]*)-(?<Remainder>[^\s]*)</Atom>";
const string HyphenReplacement =
@"<Atom${TagOptions}>${Atom}</Atom><Atom>${Remainder}</Atom>";
while (Regex.Replace(M1Facts, HyphenPattern, HyphenReplacement,
RegexOptions.Singleline) != M1Facts)

```

```

{
    MlFacts = Regex.Replace(MlFacts, HyphenPattern, HyphenReplacement,
RegexOptions.Singleline);
}

```

Next variables (denoted by an expression term starting with an underscore or capital letter) are identified and processed:

```

//Assign variables
const string VarPattern = @"<Atom(?<Options>[^\s]*)>\s*(?<Var>[A-Z_]+[^\s]*)\s*</Atom>";
const string VarReplacement = @"<Atom${Options}
Type=" "Var" ">${Var}</Atom>";
MlFacts = Regex.Replace(MlFacts, VarPattern, VarReplacement,
RegexOptions.Singleline);

```

Finally strings are processed, and the MlFacts string is wrapped in <Facts></Facts> tags:

```

//Process ' ' strings
const string ApostrophePattern =
@"<Atom(?<Options>[^\s]*)>\s*'(?<String>[^\s]*)'\s*</Atom>";
const string ApostropheReplacement = @"<Atom${Options}
Type=" "String" ">${String}</Atom>";
MlFacts = Regex.Replace(MlFacts, ApostrophePattern,
ApostropheReplacement, RegexOptions.Singleline);

//Process " " strings
const string QuotePattern =
@"<Atom(?<Options>[^\s]*)>\s*" "(?<String>[^\s]*)" "\s*</Atom>";
const string QuoteReplacement = @"<Atom${Options}
Type=" "String" ">${String}</Atom>";
MlFacts = Regex.Replace(MlFacts, QuotePattern, QuoteReplacement,
RegexOptions.Singleline);

//Return MlFacts wrapped in <Facts> tags
return "<Facts>" + "\n" + MlFacts + "</Facts>" + "\n";

```

Once both M.1 rules and facts are processed, they are nested in the following structure, as discussed in section 4.7.1:

```

<?xml version="1.0" encoding="UTF-8"?>
<LarkML-Ruleset>
  <Rules>...</Rules>
  <Facts>...</Facts>
</LarkML-Ruleset>

```

4.8.8 LarkML Language Mapping Considerations and Exceptions

The LARK Engine combined with the LarkML language exists to allow the conversion of LarkML rules and facts to M.1, CLIPS, and LNL rulesets. It should be noted that additional considerations need to be made to allow for a functional knowledge

base for both M.1 and CLIPS. CLIPS knowledge bases often contain deftemplates and other syntax not currently supported by LarkML. Similarly the M.1 engine begins execution through the use of a **goal**, which is not currently specified in LarkML. The combination of the LARK engine and LarkML does not result in fully functionally, turn-key executable knowledge bases for either M.1 or CLIPS in most cases, however it does serve to create basic ruleset portability for both expert system engines.

LarkML was developed to allow for features from M.1 that would not necessarily be of use in CLIPS, and vice-versa. This section details exceptions in the parsing and translation between LarkML and CLIPS / M.1. Please refer to Appendix E for a language compatibility matrix.

It would not be appropriate to attempt to parse M.1 specific features in a CLIPS ruleset that do not exist in the CLIPS language. Because M.1 is based largely on the use of expressions, an `<Equals>` tag is allowed inside of the `<Fact-Pattern>`, `<LHS-Pattern>`, and `<RHS-Pattern>` tags. While an expression can be emulated in CLIPS, this would be a special case in a ruleset most likely developed for allowing a ruleset to be ported to M.1. In practice this is such a special case that it does not merit consideration for our parsing engine, and it would not be appropriate to attempt to parse for expressions. The `<Equals>` tag is not supported for CLIPS parsing, thus any tags found inside are therefore not supported, by inheritance, in the context of the `<Equals>` tag. It should be noted that CLIPS does contain a facility for testing and evaluating equations, but the implementation is not equivalent to that of the M.1 use of expressions.

The `<Pattern-Bool>` tag found in the `<LHS-Pattern>` and `<RHS-Pattern>` is also unsupported by the CLIPS parsing and transformation functions. LHS and RHS patterns are expressed as atomic units, separated from other patterns by a set of parentheses. All patterns in CLIPS are implicitly assumed to be connected by the “and” Boolean operator, there is no facility in place to allow the Boolean “or” function at the pattern level. CLIPS does allow for the use of connective constraints inside LHS patterns (such as the Boolean “not” operator “~”), used to modify pattern symbols. For this reason the `Negate` option of the `<LHS-Pattern>` is not supported for CLIPS, but is supported in the `<Atom>` tag. The opposite holds true for M.1. Because M.1 evaluates expressions as LHS patterns, only entire expressions can be negated, as opposed to the constituent atoms (M.1 atoms being equivalent to symbols in CLIPS). For this reason, the `Negate` option is supported in the `<LHS-Pattern>` tag, but not in the `<Atom>` tag for M.1 parsing and translation.

While CLIPS allows for a “Bind” type LHS pattern, whereby a pattern address is assigned to a variable, M.1 does not. This allows for CLIPS facts to be attached to variables and then manipulated in the RHS of the rule. Because M.1 does not support this feature, the `<Pattern-Name>` and `<Pattern-Type>` options of the `<LHS-Pattern>` tag are not processed by the parsing or transformation functions of the LARK Engine. The `<Pattern-Name>` is used for assigning names to the `Bind` variables, and the only current option for `<Pattern-Type>` is `Bind`, thus both are unnecessary for any M.1 processing.

Action types `Assert` and `Retract` are specific features for the CLIPS language, and there is no strictly equivalent function in M.1. It could be argued that setting the value of an expression to either “yes” or “no” is a relatively close equivalent,

but for the sake of the LARK Engine it is simply not close enough. The setting of an expression to “no” (the M.1 equivalent of the “false” Boolean qualifier) is not tantamount to the non-existence of said expression, thus it would not be appropriate to map an assertion or retraction statement to this operation.

The `<FactPattern-Type>` option allows the pattern to be set to a `String` type. This is not supported in M.1.

4.9 LARK Engine Class Implementation

This section contains a brief description of the most pertinent classes used in the development of the LARK Engine.

4.9.1 RuleSet Class

The `RuleSet` class provides a mechanism to store rulesets in text form along with their respective filenames:

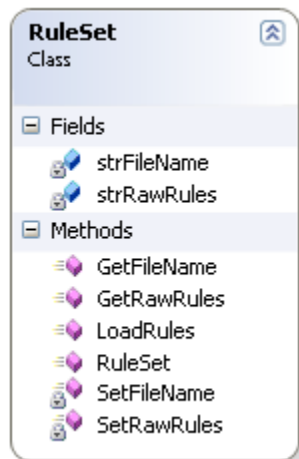


Figure 6: RuleSet Class Diagram

Variables `strFileName` and `strRawRules` contain the ruleset filename and the rules in raw text form, respectively. Methods `GetFileName()`, `GetRawRules()`, `SetFileName()`, and `SetRawRules()` act as getters and setters for the class variables `strFileName` and `strRawRules`. The `RuleSet` class is a simple tool that facilitates the XML transforms used in class `RuleTranslator` and the rule parsing and refactoring executed in the `RuleSmasher` class.

4.9.2 RuleTranslator Class

The `RuleTranslator` class was built to transform LarkML (XML) rulesets into M.1, CLIPS, and LNL. Transformations from the LarkML language to the goal language, for example CLIPS, are accomplished via XSL transforms. Each goal language has an XSL stylesheet defined in the `\transforms` subdirectory (with respect

to the LARK.exe executable) of the LARK Engine installation folder. When additional XSL stylesheets are added to this folder, the LARK Engine attempts to load them, so they must be syntactically valid.

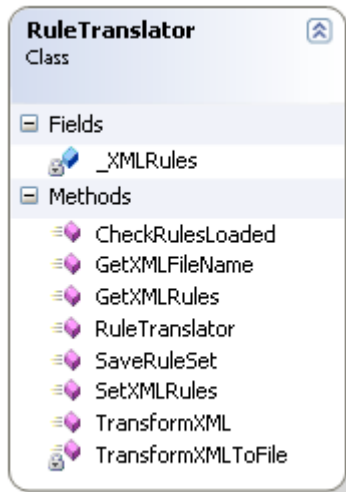


Figure 7: RuleTranslator Class Diagram

The `RuleTranslator` class contains a single private variable, the `_XMLRules` instance of the `RuleSet` class. The `CheckRulesLoaded()` method provides a check to ensure that a `RuleSet` has been loaded before a transformation is applied to the `_XMLRules` LarkML ruleset. The `GetXMLFileName()` and `GetXMLRules()` methods access the `GetFileName()` and `GetRawRules()` methods of the `RuleSet` class, respectively. `TransformXML()` applies the currently selected XSL transform in the “Output Format” dropdown to the LarkML ruleset shown in the “LarkML Ruleset” text box, and displays the output in the “Output Ruleset” text box. The `SaveRuleSet()` method allows users to export transform rulesets to a text file. The `SaveRuleSet()` method invokes the `TransformXMLToFile()` method to write the transform to file.

The transforms from LarkML to CLIPS, M.1, and LNL are described in sections 4.8.1, 4.8.2, and 4.8.3, respectively.

4.9.3 RuleSmasher Class

The `RuleSmasher` class was built to parse and refactor CLIPS and M.1 rules into LarkML syntax.



Figure 8: RuleSmasher Class Diagram

The `_RuleType` string variable is used as a reference for the current type of ruleset loaded for parsing - either M.1 or CLIPS. The `_SourceRules` private class variable is a `RuleSet` that contains the current loaded ruleset.

The `CheckRulesLoaded()` method is used to ensure a ruleset is loaded before parsing is executed. The `GetSourceFileName()` and `GetSourceRules()` functions return the filename of the current ruleset and the source rules in raw text format, respectively. The `SetInputRules()` function loads the input rules into the `_SourceRules` ruleset. `SetRuleType()` is invoked whenever a ruleset is loaded into the LARK Engine and allows the engine to know the current ruleset type (either M.1 or CLIPS). `WriteToFile()` writes the converted LarkML ruleset shown in the “LarkML Ruleset” text box to a file.

The remaining functions are used for converting a source ruleset into LarkML. According to the current string defined in `_RuleType`, when a rule conversion begins the `ConvertToLarkML()` function invokes either `CLIPSConversion()` or `M1Conversion()`. To convert to CLIPS, the `CLIPSConversion()` function is used which in turn calls the `GetCLIPSFacts()` and `GetCLIPSRules()` functions. For M.1 conversion, the `M1Conversion()` function is invoked, which calls `GetM1Rules()` and then `GetM1Facts()` to generate the converted LarkML ruleset. The processes for parsing and refactoring CLIPS rules, CLIPS facts, M.1 rules, and M.1 facts are defined in sections 4.8.4, 4.8.5, 4.8.6, and 4.8.7, respectively.

5. CONCLUSIONS AND FUTURE DIRECTION

5.1 Conclusions

The initial goal of this thesis was to provide a much greater degree of portability, indeed I was hoping to be able to take a functional CLIPS ruleset and translate it directly into a functional M.1 ruleset, and vice-versa. This proved to be too ambitious as the effort required to author such a program is beyond the scope of this thesis. I have however created a general parser for basic M.1 and CLIPS rules and facts, which is a stride towards a general ruleset translator. Additionally the extensible definition of the LarkML language provides an open, portable language that can be easily transformed into any production system language, beyond just M.1 and CLIPS.

All functional and non-functional requirements detailed in sections 4.5 and 4.6, respectively, were met.

5.2 Future Direction

The LARK Engine provides a start for a general expert system language parser. Additional rule complexities may be accounted for in future work, as well as CLIPS mathematical functions, and the Deftemplate fact template construction. M.1 specific features should be added including the use of goals, additional meta-facts, and mathematical functions. Because of the use of XSL stylesheet transformations, LarkML is easily formed into other languages, and additional production systems may be added to future versions of the LARK Engine. The effort to parse and refactor additional expert system languages into LarkML poses a more formidable task, but is a worthy future goal as well.

APPENDIX A

LARKML LANGUAGE DEFINITION

Appendix A specifies LarkML rules and facts in Extended Backus-Naur Form (EBNF). EBNF is defined in International Standard ISO/IEC 14977:1996, Information technology – Syntactic metalanguage – Extended BNF.

```
<LarkML Ruleset> ::= '<LarkML-Ruleset>', <Rules>*,
                    <Facts>*, '</LarkML-Ruleset>'

<Rules> ::= '<Rules>', [<Rule>]*, '</Rules>'

<Facts> ::= '<Facts>', [<Fact>]*, '</Facts>'

<alpha> ::= Lower or uppercase alphabetic character

<alphanumeric> ::= <alpha> | numeric digit

<atomic> ::= <alpha>, [<alphanumeric>]*

<Fact> ::= '<Fact ', <Fact-Name>, '>', <FactPattern>*,
           [<Equals>], '</Fact>'

<Fact-Name> ::= 'Name="' , <atomic>, '"'

<FactPattern> ::= '<FactPattern ', [<FactPattern-Type>],
                  '>', <atomic>, '</FactPattern>'

<FactPattern-Type> ::= 'Type="' , 'String', '"'

<cf-numeric> ::= numeric digit [-100 to 100]

<Equals> ::= '<Equals [<Equals-CF>]>', <Atom>*, '<Equals>'

<Equals-CF> ::= 'CF="' , <cf-numeric>, '"'

<Atom> ::= '<Atom [<Atom-Type>] [<Negate>] >', <atomic>,
           '</Atom>'

<Atom-Type> ::= 'Type="' , 'Var' | 'String', '"'

<Negate> ::= 'Negate="true"'

<Rule> ::= <RuleHeader>, <LHS>, <RHS>, <RuleTrailer>
```

```

<RuleHeader> ::= '<Rule Name="' , <atomic> , '"'>'
<LHS> ::= '<LHS>' , [<LHS-Pattern>]* , '</LHS>'
<LHS-Pattern> ::= '<Pattern ` , [<Pattern-Name>] [<Pattern-
Type>] [<Pattern-Bool>] [<Negate>], `]'>' , <Atom>*,
[<Equals>], '</Pattern>'
<Pattern-Name> ::= 'Name="' , <atomic> , '"'
<Pattern-Type> ::= 'Type="' , 'Bind' , '"'
<Pattern-Bool> ::= 'Bool="' , 'and' | 'or' , '"'
<RHS> ::= '<RHS>' , <RHS-Pattern>*, '</RHS>'
<RHS-Pattern> ::= '<Action ` , [<Action-Type>], [<Pattern-
Bool>], `]'>' , <Atom>*, [<Equals>] '</Action>'
<Action-Type> ::= 'Type="' , 'Assert' | 'Retract' | 'Print' ,
'"'
<RuleTrailer> ::= '</Rule>'

```

APPENDIX B

LARK NATURAL LANGUAGE XSL

For the sake of simplicity, Lark Natural Language (LNL) has been referred to as its own language throughout this document; however there is no formal EBNF definition. LNL is simply a transform performed upon LarkML and is defined in an XSL format below:

```
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0" >
<xsl:output method="text" />

<xsl:template match="/">
<xsl:apply-templates select="LarkML-Ruleset/Rules/Rule" />
<xsl:apply-templates select="LarkML-Ruleset/Facts/Fact" />
</xsl:template>

<xsl:template match="Rule">
Rule <xsl:if test="@Name">Name:<xsl:value-of select="@Name"
/></xsl:if>
<xsl:apply-templates select="LHS" />
<xsl:apply-templates select="RHS" />
</xsl:template>

<xsl:template match="LHS">
LHS Patterns:
<xsl:apply-templates select="Pattern" />
</xsl:template>

<xsl:template match="Pattern"> Pattern:
<xsl:if test="@Name">      Name = "<xsl:value-of
select="@Name" />"
</xsl:if>
<xsl:if test="@Type">      Type = "<xsl:value-of
select="@Type" />"
</xsl:if>
<xsl:if test="@Bool">      Bool = "<xsl:value-of
select="@Bool" />"
</xsl:if>
<xsl:if test="@Negate">      Negate = "<xsl:value-of
select="@Negate" />"</xsl:if>
<xsl:apply-templates select="Atom" />
<xsl:apply-templates select="Equals" />
</xsl:template>
```

```

<xsl:template match="Atom">      Atom:
<xsl:if test="@Negate">          Negate="<xsl:value-of
select="@Negate" />"
</xsl:if>
<xsl:if test="@Type">           Type="<xsl:value-of
select="@Type" />"
</xsl:if>
<xsl:if test=".">               Value="<xsl:value-of select="."
/>"
</xsl:if>
</xsl:template>

```

```

<xsl:template match="RHS">RHS Patterns:
<xsl:apply-templates select="Action" />
</xsl:template>

```

```

<xsl:template match="Action"> Action:
<xsl:if test="@Bool">      Bool Value="<xsl:value-of
select="@Bool" />"
</xsl:if>
<xsl:if test="@Type">     Type Value="<xsl:value-of
select="@Type" />"
</xsl:if>
<xsl:apply-templates select="Atom" />
<xsl:apply-templates select="Equals" />
</xsl:template>

```

```

<xsl:template match="Equals">      Equals:
<xsl:text disable-output-escaping="yes">
</xsl:text><xsl:apply-templates select="Atom" />
<xsl:if test="@CF">          CF = "<xsl:value-of select="@CF"
/>"
</xsl:if>
</xsl:template>

```

```

<xsl:template match="Fact">
Fact:
<xsl:if test="@Name">      Name = "<xsl:value-of
select="@Name" />"</xsl:if>
<xsl:text disable-output-escaping="yes">
</xsl:text><xsl:apply-templates select="FactPattern" />
<xsl:apply-templates select="Equals" />
</xsl:template>

```

```

<xsl:template match="FactPattern">
<xsl:text disable-output-escaping="yes">

```

```
    </xsl:text>Fact Pattern:
<xsl:if test="@Type">      Type = "<xsl:value-of select="."
/>"
</xsl:if>
<xsl:if test=".">      Value = "<xsl:value-of select="." />"
</xsl:if>
</xsl:template>
</xsl:stylesheet>
```

APPENDIX C

CLIPS XSL

The XSL stylesheet used to transform LarkML rulesets to CLIPS is shown below, note that minor formatting changes were made to increase readability:

```
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0" >
<xsl:output method="text" />

<xsl:template match="/">
<xsl:apply-templates select="LarkML-Ruleset/Rules/Rule" />
<xsl:apply-templates select="LarkML-Ruleset/Facts/Fact" />
</xsl:template>

<xsl:template match="Rule">
(defrule <xsl:value-of select="@Name" />
<xsl:apply-templates select="LHS" />
=<xsl:text disable-output-escaping="yes">&gt;</xsl:text>
<xsl:apply-templates select="RHS" />)
</xsl:template>

<xsl:template match="LHS">
<xsl:apply-templates select="Pattern" />
</xsl:template>

<xsl:template match="Pattern">
<xsl:text disable-output-escaping="yes">
</xsl:text>
<xsl:if test="@Type='Bind'">?<xsl:value-of select="@Name"
/><xsl:text disable-output-escaping="yes"> &lt;-
</xsl:text>
</xsl:if>
(<xsl:apply-templates select="Atom" />)</xsl:template>

<xsl:template match="Atom">
<xsl:if test="@Negate = 'true'">~</xsl:if>
<xsl:choose>
<xsl:when test="@Type = 'Var'">?<xsl:value-of select="."
/></xsl:when>
<xsl:when test="@Type = 'String'">"<xsl:value-of select="."
/>"</xsl:when>
<xsl:otherwise>
<xsl:value-of select="." />

```

```

</xsl:otherwise>
</xsl:choose>
<xsl:text disable-output-escaping="yes"> </xsl:text>
</xsl:template>

<xsl:template match="RHS">
<xsl:apply-templates select="Action" />
</xsl:template>

<xsl:template match="Action">
<xsl:text disable-output-escaping="yes">
</xsl:text>
<xsl:choose>
<xsl:when test="@Type = 'assert'">(assert (<xsl:apply-
templates select="Atom" />))</xsl:when>
<xsl:when test="@Type = 'retract'">(retract <xsl:apply-
templates select="Atom" />)</xsl:when>
<xsl:when test="@Type = 'Print'">(printout <xsl:if
test="@Destination='Console'"> t </xsl:if> <xsl:apply-
templates select="Atom" /><xsl:if
test="@Destination='Console'"> crlf</xsl:if>)</xsl:when>
<xsl:otherwise>(<xsl:apply-templates select="Atom"
/>)</xsl:otherwise>
</xsl:choose>
</xsl:template>

<xsl:template match="Fact">
(assert (<xsl:apply-templates select="FactPattern" />))
</xsl:template>

<xsl:template match="FactPattern">
<xsl:choose>
<xsl:when test="@Type='String'"> "<xsl:value-of select="."
/>"</xsl:when>
<xsl:otherwise><xsl:value-of select="." />
</xsl:otherwise>
</xsl:choose>
<xsl:text disable-output-escaping="yes"> </xsl:text>
</xsl:template>

</xsl:stylesheet>

```

APPENDIX D

M.1 XSL

The XSL stylesheet used to transform LarkML rulesets to M.1 is shown below, note that minor formatting changes were made to increase readability. It should be noted that this stylesheet uses XSLT 2.0. XSLT 2.0 supports the use of some advanced functionality including the `<xsl:analyze-string>` tag that allows for the use of REGEX string matching.

```
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="2.0" >
<xsl:output method="text" />

<xsl:template match="/">
<xsl:apply-templates select="LarkML-Ruleset/Rules/Rule" />
<xsl:apply-templates select="LarkML-Ruleset/Facts/Fact" />
</xsl:template>

<xsl:template match="Rule">
<xsl:value-of select="@Name" />: if
<xsl:apply-templates select="LHS" />then
<xsl:apply-templates select="RHS" />.

</xsl:template>

<xsl:template match="LHS">
<xsl:apply-templates select="Pattern" />
<xsl:apply-templates select="../RHS/Action" mode="Print"/>
</xsl:template>

<xsl:template match="Pattern">
<xsl:choose>
<xsl:when test="@Type='Bind'"></xsl:when>
<xsl:otherwise>
<xsl:value-of select="@Bool"/>
<xsl:if test="@Negate = 'true'"> not(</xsl:if>
<xsl:text disable-output-escaping="yes"> </xsl:text>
<xsl:apply-templates select="Atom" />
</xsl:otherwise>
</xsl:choose>
<xsl:apply-templates select="Equals" />
<xsl:if test="@Negate = 'true'"></xsl:if>
<xsl:text disable-output-escaping="yes">
</xsl:text>
```

```

</xsl:template>

<xsl:template match="Atom">
<xsl:if test="position() > 1"></xsl:if>
<xsl:choose>
<xsl:when test="@Type = 'Var'">
<xsl:analyze-string select="." regex="[A-Z_]+.*">
<xsl:matching-substring>
<xsl:value-of select="." />
</xsl:matching-substring>
<xsl:non-matching-substring>X<xsl:value-of select="."
/></xsl:non-matching-substring>
</xsl:analyze-string>
</xsl:when>
<xsl:when test="@Type = 'String'"> "<xsl:value-of
select="." />" </xsl:when>
<xsl:otherwise>
<xsl:value-of select="." />
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<xsl:template match="RHS">
<xsl:apply-templates select="Action" />
</xsl:template>

<xsl:template match="Action">
<xsl:choose>
<xsl:when test="@Type = 'assert'">
<xsl:text disable-output-escaping="yes"> </xsl:text>
<xsl:value-of select="@Bool" />
<xsl:text disable-output-escaping="yes"> </xsl:text>
<xsl:apply-templates select="Atom" /></xsl:when>
<xsl:when test="@Type = 'retract'">
<xsl:text disable-output-escaping="yes"> </xsl:text>
<xsl:value-of select="@Bool" />
<xsl:text disable-output-escaping="yes"> </xsl:text>
<xsl:apply-templates select="Atom" />
</xsl:when>
<xsl:when test="@Type = 'Print'"></xsl:when>
<xsl:otherwise><xsl:if test="@Bool != ''">
<xsl:text disable-output-escaping="yes">
</xsl:text></xsl:if>
<xsl:value-of select="@Bool" />
<xsl:text disable-output-escaping="yes"> </xsl:text>
<xsl:apply-templates select="Atom" /></xsl:otherwise>
</xsl:choose>

```

```

<xsl:apply-templates select="Equals" />
</xsl:template>

<xsl:template match="Action" mode="Print">
<xsl:if test="@Type = 'Print'">
and display([<xsl:for-each select="Atom">
<xsl:choose>
<xsl:when test="@Type = 'Var'">
<xsl:analyze-string select="." regex="[A-Z_]+.*">
<xsl:matching-substring>
<xsl:value-of select="." /></xsl:matching-substring>
<xsl:non-matching-substring>X<xsl:value-of select="."
/></xsl:non-matching-substring>
</xsl:analyze-string>
</xsl:when>
<xsl:when test="@Type = 'String'"> "<xsl:value-of
select="." />"</xsl:when>
<xsl:otherwise>
<xsl:value-of select="." />
</xsl:otherwise>
</xsl:choose>
<xsl:if test="position() != last()">, </xsl:if>
</xsl:for-each>) </xsl:if>
</xsl:template>

<xsl:template match="Equals"> = <xsl:apply-templates
select="Atom" />
<xsl:if test="@CF"> cf <xsl:value-of select="@CF" />
</xsl:if>

</xsl:template>

<xsl:template match="Fact">
<xsl:value-of select="@Name" />: <xsl:apply-templates
select="FactPattern" />
<xsl:apply-templates select="Equals" />.
</xsl:template>

<xsl:template match="FactPattern">
<xsl:if test="position() > 1">-</xsl:if>
<xsl:value-of select="." /></xsl:template>

</xsl:stylesheet>

```

APPENDIX E

LARKML COMPATIBILITY

Specific features of LarkML are compatible with CLIPS and M.1 according to which of these features the respective expert systems shells use. It should be noted that tag support can be context specific. For example in CLIPS the `<Atom>` tag is supported inside of the `<LHS-Pattern>` tag, but not inside of an `<Equals>` tag. The context of each LarkML tag is denoted by nesting. In cases where all the sub-options of a tag are not shown, it is implied that the support or lack there-of for the parent option is inherited by all the sub-options.

Table 10: LarkML Compatibility Matrix

LarkML Grammar	CLIPS	M.1	Legend
<code><atomic></code>	+	+	+: supported
<code><Fact></code>	+	+	-: unsupported
<code><Fact-Name></code>	+	+	
<code><FactPattern></code>	+	+	
<code><FactPattern-Type></code>	+	-	
<code>"String"</code>	+	-	
<code><atomic></code>	+	+	
<code><Equals></code>	-	+	
<code><Equals-CF></code>	-	+	
<code><Atom></code>	-	+	
<code><Atom-Type></code>	-	+	
<code>"Var"</code>	-	+	
<code>"String"</code>	-	+	
<code><Negate></code>	-	-	
<code>"true"</code>	-	-	
<code><Rule></code>	+	+	
<code><RuleHeader></code>	+	+	
<code>Name= " "</code>	+	+	
<code><LHS></code>	+	+	
<code><LHS-Pattern></code>	+	+	
<code><Pattern-Name></code>	+	-	
<code><Pattern-Type></code>	+	-	
<code>"Bind"</code>	+	-	
<code><Pattern-Bool></code>	-	+	
<code>"and"</code>	-	+	
<code>"or"</code>	-	+	
<code><Negate></code>	-	+	
<code>"true"</code>	-	+	
<code><Atom></code>	+	+	
<code><Atom-Type></code>	+	+	
<code>"Var"</code>	+	+	
<code>"String"</code>	+	+	

Table 11: LarkML Compatibility Matrix - Continued

LarkML Grammar	CLIPS	M.1	Legend
<Negate>	+	-	+: supported
"true"	+	-	-: unsupported
<Equals>	-	+	
<RHS-Pattern>	+	+	
<Action-Type>	+	+	
"Assert"	+	-	
"Retract"	+	-	
"Print"	+	+	
<Pattern-Bool>	-	+	
<Atom>	+	+	
<Equals>	-	+	
<Equals-CF>	-	+	

REFERENCES

- Boley, Harold. "RuleML Initial Steps." RuleML The Rule Markup Initiative. 22 Aug. 2002. The RuleML Initiative. 1 July 2008 <<http://www.ruleml.org/insteps.html>>
- . "Schema Specification of RuleML 0.91." RuleML The Rule Markup Initiative. 24 Aug. 2006. The RuleML Initiative. 15 July 2008 <http://www.ruleml.org/0.91/>
- Boley, Harold, et al. "Functional RuleML." RuleML The Rule Markup Initiative. 11 Aug. 2006. The RuleML Initiative. 7 July 2008 <<http://www.ruleml.org/fun/>>.
- CLIPS Architecture Manual, Version 5.1. 1992 <CD-ROM arch5-1.pdf>.
- CLIPS Basic Programming Guide, Version 6.22. 2004 <CD-ROM bpg.pdf>.
- "Expert Systems." PC AI Artificial Intelligence. 2002. PC AI. 1 June 2008 <http://www.pcai.com/web/ai_info/expert_systems.html>
- Forgy, Charles L. On the efficient implementation of production systems. Ph.D. Thesis – Carnegie-Mellon University, 1979. University Microfilms International.
- Giarratano, Joseph C. CLIPS User's Guide, Version 6.20. 2002 <CD-ROM usrguide.pdf>.
- Giarratano, Joseph C., and Gary D. Riley. Expert Systems. Canada: Thomson Course Technology, 2005.
- Hawke, Sandro. "Rule Interchange Format Working Group Charter." W3C World Wide Web Consortium. 11 Nov. 2005. W3C. 7 June 2008 <<http://www.w3.org/2005/rules/wg/charter.html>>.
- Hilyard, Jay, and Stephen Teilhet. C# Cookbook, Second Edition. United States of America: O'Reilly Media Inc, 2006.
- ISO/IEC. "International Standard ISO/IEC 14977:1996, Information technology -- Syntactic metalanguage -- Extended BNF." 15 Dec. 1996.
- Nentwich, Christian, and Rob James. "Natural Rule Language (NRL) Version 1.0." The Natural Rule Language (NRL). 24 Apr. 2006. SourceForge. 1 July 2008 <<http://nrl.sourceforge.net/spec/>>.
- Taylor, James, Christian de Sainte Marie, and Sridhar Iyengar. OMG document bmi/2007-03-05 Production Rule Representation. 2007. Object Management Group (OMG). 15 June 2008 <<http://www.omg.org/docs/bmi/07-03-05.pdf>>.
- Teknowledge, Inc. M.1 Reference Manual. Delaware: Teknowledge, 1986.

BIOGRAPHICAL SKETCH

Kenneth Lloyd Ayers III

Staff Computer Scientist – Applied Research Associates, Inc. (2001-Present)

Graduated 2007, Florida State University, B.S. Computer Science – Software Engineering, Cum Laude.

Graduated 2002, Gulf Coast Community College, A.A. Magna Cum Laude.