XESS

The XML Expert System Shell

Acknowledgements

There are many people that I would like to thank for their patience and support over the last three years as I worked to complete this thesis.

First, I would like to thank Professor Leon Reznik for agreeing to serve as my thesis committee chair, for helping me to shape this project, for his patience and understanding through the sometimes many months of drought in between snippets of productivity, and for helping me to see London.

I would also like to thank my friend and mentor, Naveen Sharma, who has been a guiding influence in my professional growth for the last seven years, who graciously agreed to sit on my thesis committee in the role of reader, and who encouraged and supported me throughout the duration of this very long project (even when it distracted me from my work).

I would like to thank Professor Hans-Peter Bischof for going against his better judgment and agreeing to act as the observer on my committee. I am sad to say that over the past three years I was the embodiment of everything that he warned his classes against; the student that completes his coursework and heads off to industry, allowing the years to roll by without ever completing the last step. I am also hopeful, however, that with his help I will overcome that last hurdle and complete my studies. I am thankful to him for giving me an opportunity to do so.

I would like to thank my wife, Kimberly, for putting up with the frustration of a husband that works full time, and who also has to spend many hours away from home to attend class or to work on homework. She has been wonderful, and supportive, and I could not have done this without her.

And finally, I would like to thank my father, Robert J. St. Jacques, Sr. During the winter of my senior year in high school, my dad drove me the 8 hours from New Hampshire to Rochester, New York so that I could qualify for a grant, without which I never would have been able to attend RIT. He repeated that same trip for every break and holiday so that I could go home to spend my free time with my family. He worked two jobs, sometimes 70 hours a week or more, so that he could afford to help me with my tuition and other college expenses. But most importantly, he has been the example to me as one of the finest, most hard working and honorable people I have ever known. Without him none of this would have been possible, and I would not be the person that I am today.

1 Introduction

An Expert System, also often referred to as a knowledge-based system, is a computer simulation that emulates the decision-making ability of a human expert[1]. An Expert System Shell is the computer software on which an expert system runs; the knowledge encoded in the expert system is separate from the software program itself, as opposed to being hard-coded into the program[2]. This allows the same software to run multiple expert systems without modifications, the caveat being that each expert system shell has a unique language used to express expert systems that is generally incompatible will other shells.

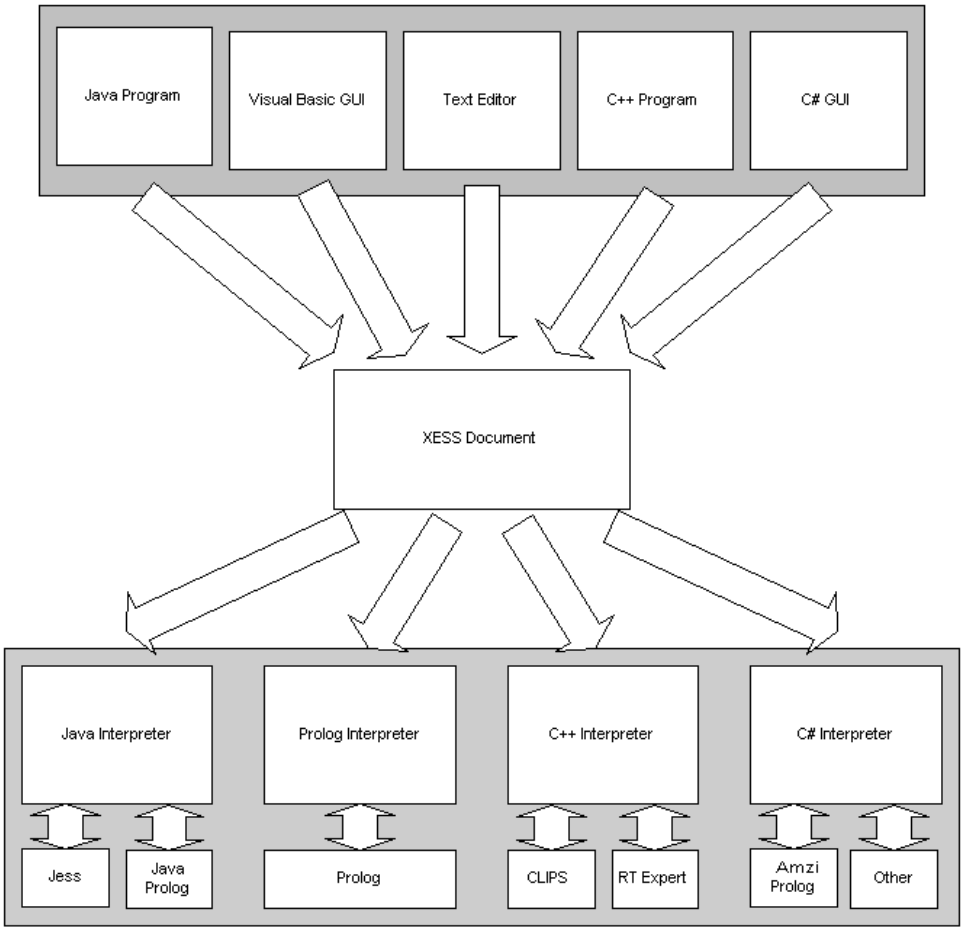
The Extensible Markup Language (XML) [3]is a simplified subset of SGML that offers powerful and extensible data modeling capabilities. An *XML Document* is a collection of data represented in XML. An *XML Schema* is a grammar that describes the structure of an XML document[4]. The purpose of XML is to express data in a common format that can easily be interpreted and understood by disparate systems. XML is an almost universally accepted mechanism for information interchange that is supported by virtually all modern programming languages[5]. An XML document may be created programmatically, though a graphical user interface, or, in many cases, by typing the XML directly into a text editor. Once a valid XML document has been generated it can be transmitted through the file system, over a network through protocols such as SMTP or HTTP, or even via hard copy that is printed from the source and then scanned at the destination using optical character recognition (OCR) software. Because of the ubiquitous nature of XML parsers, the XML document can then be parsed and; in fact, many programming environments such as the Java SDK and .Net include XML parsers and generators "out of the box". The majority of modern programmers how to read and write XML documents, and familiarity with XML parser technologies (such as SAX and DOM) is common.

The *XML Expert System Shell* or *XESS* (pronounced "excess") is a specification for an XML-based language and interpreter designed to bring the portability of XML into the realm of Expert System Shells. XESS does not try to reinvent the wheel by providing a new implementation of the Rete algorithm[6], a new breakthrough in backward chaining, or any other improvements in rule handling or execution. Instead, the ultimate goal of the language is to allow knowledge engineers to express expert

[1] John F. Gilmore, "Knowledge Base Systems in Computer Aided Technology"
[2] Du Zhang, Doan Nguyen, "PREPARE: A Tool for Knowledge Base Verification"
[3] Ronald Logsdon, "XML White Paper"
[4] J. Roy, A. Ramanujan, "XML Schema Language; Taking XML to the Next Level"
[5] Stephen Kirkham, "XML – A Disruptive Influence on Programming Languages and Methodologies"
[6] Charles L. Forgy, "Rete: A Fast Algorithm for Many Pattern/Many Object Pattern Match Problems"

systems in a flexible, simple, human readable language that uses terms familiar to expert systems developers *without depending on a specific implementation*. This is an extremely powerful concept as knowledge engineers frequently choose the expert system based not on the ease of use or the ability to quickly express complicated facts or rules in the native language of the system, but for other reasons such as speed, efficiency, or functionality[7]. Furthermore, once a knowledge engineer becomes very familiar with a single expert system they may default to that system, even when it is not the best choice as a solution.

The XESS language attempts to break the dependency between the often obscure languages used to express expert systems and the power of the underlying interpreters and engines. The XESS language and interpreters place a layer of abstraction between the knowledge engineer and the underlying implementation. An additional benefit is that the same expert system can be executed on multiple interpreters, on multiple platforms to compare performance or results *without modification* through the use of plug-ins.



[7] Arne Bultman, Joris Kuipers, and Frank van Harmelen, "Maintenance of KBS's by Domain Experts"

Figure 1.1: This diagram shows a subset of the potentially many different paths that an expert system written using XESS may take from generation to execution.

The typical lifecycle of an XESS document begins with generation of the facts and rules in the expert system. The document may be generated using XML tools available in a wide array of languages, custom graphical user interfaces that allow systems to be built using GUI widgets such as trees, or even by typing the human readable XML directly into a text editor.

Once the XESS document has been generated, it can be executed on a compliant XESS interpreter written in any language for which an XML parser is available. The interpreter parses the XESS document into a collection of objects designed to represent the facts and rules of the system. The collection is then passed to *plug-ins* that translate the XESS objects into the native language of specific expert systems. The plug-ins are then responsible for providing input to the specific expert systems, and for collecting feedback and making it available to the user through the XESS interpreter API (see Figure 1.1).

Later chapters of this document describe the XESS schema and interpreter APIs as well as provide detailed references and examples of expert systems written entirely in XESS.

1.1 Reader Level

The intended audience for this document is as follows:

- Students of Artificial Intelligence
- Knowledge Engineers/Architects
- Expert Systems Programmers and Developers
- Interpreter Developers
- Plug-In Developers

Where appropriate, each of these roles is explained in more detail in later chapters.

The remaining chapters of this document assume that the reader is familiar with high-level concepts of expert systems including facts, rules, and terms such as *knowledge engineer*. It is also assumed that the reader is familiar with some of the more common expert system shells, such as Prolog. The user must also have a grasp of some of the basic fundamentals of boolean logic, such as DeMorgan's law[8].

The sections concerning the XESS schema, including the document format and detailed examples, require a fairly deep understanding of XML. This

[8] S.P. Bali, "2000 Solved Problems in Digital Electronics," pp. 80-81

section pertains mostly to knowledge engineers that will be writing expert systems using the XESS language and the interpreter developers that will be writing the XML parsers that translate XESS documents into objects.

The sections concerning the XESS interpreters and plug-in API require a deep and detailed knowledge of object oriented design and object oriented programming. It is also very beneficial to have a detailed knowledge of Java as many of the programming examples, as well as the style of the API documentation, are taken from Java standards[9].

[9] Sun Microsystems, "Java Code Conventions"

2 State of the Industry

There are currently dozens, if not hundreds, of viable expert systems shells available for the development of expert systems on virtually every conceivable platform. The available options range from LISP interpreters that have changed very little since the introduction of the language in 1958 to large, enterprise-scale systems used by credit processing institutions capable of handling tens of thousands of transactions per minute.

But not all expert system shells are equal. Each has its strengths and weaknesses, and some are ideal for certain projects, and completely unfit for others. The role of the knowledge engineer on an expert system project is to determine which expert system shell is best suited for the specific requirements of the project, and this answer may be different every time.

In this section four popular languages will be discussed, including Prolog, CLIPS, JESS, and Blaze Advisor. The discussion will include detailed information on five important factors to consider when evaluating an expert system shell for a specific product:

- **Learning Curve** – a measure of how difficult it is for novice developers unfamiliar with the shell to learn how to use it to create non-trivial expert systems.
- **Portability** – Many development environments and target production platforms are heterogeneous; portability is a measure of how easy it is to install and run the same expert system on multiple, disparate platforms.
- **Security** – a measure of how secure expert systems implemented using the shell may be, during all phases of development and production.
- **Richness of Features** – a measure of how many features, beyond the basic ability to create facts and rules, that an expert system shell provides, and how useful those features may be to the target audience.
- **Resource Consumption** – a measure of the hardware and software required to execute the expert system shell.

The languages discussed herein are popular for a reason; each has strengths that make it ideal for projects of a certain scale, with certain requirements. Some languages are more difficult to learn, but contain more features. Others can be used to quickly develop simple systems suitable for small hardware. No one language is a perfect solution for every problem that may arise.

2.1 Prolog

The first preliminary version of the language that would come to be known as Prolog was released in 1971, with subsequent versions released in 1972 and 1973. Prolog, which gets its name from the French "*Programmation en Logique"* (meaning "programming in logic"), was the result of an attempt to create a programming language that used natural language to communicate with a computer. The first version of the language was written by Philippe Roussel in support of an application called *the man machine communication system*[10], which was being developed by Alain Colmerauer; the communication system allowed structured, natural language statements to be used to specify facts, which could then be queried using natural language questions. In 1973, the final version of Prolog was created, independent of any application, and released as a full fledged, standalone programming language.

While Prolog is a generic logic programming language that can be used for many programming tasks, one of its most common applications is in expert systems. Prolog provides a shell that allows programmers to quickly specify facts, or relationships, a set of rules, and then to submit queries. Prolog responds "yes" if the queries hold based upon the facts and rules specified, or "no" if the query is explicitly false or if it cannot be proven to hold based on the information provided.

Despite (or perhaps because of) its obvious simplicity, Prolog is an incredibly popular and powerful programming language still commonly used over 30 years after its invention.

2.1.1 Language Learning Curve

Prolog supports only one data type: the term. A term may be one of a handful of primitive types (atoms, numbers, or variables), or may be a compound term, which is defined by a name (*functor*) and a number of arguments (*arity*) which are themselves terms. Programming statements in Prolog consist of clauses, which are made up of a head and a tail. Clauses with a head but no tail are facts, which simply define a relationship between objects. Clauses that specify both a head and a tail are rules; the head, or *consequent*, holds if the all of the elements in the tail, or *antecedent*, hold. The basis for this logic is the Horn clause, an example of which can be seen here:

$X: Y_1, Y_2, \ldots, Y_n$

[10] Leon Sterling and Ehud Shapiro, "The Art of Prolog, 2nd Edition"

In this case the *consequent* (X) is true if every element of the *antecedent* (Y_1, Y_2, …, Y_n) is true.

The programmer defines relationships, rules that can be used to make inferences about those relationships, and then may specify queries to which Prolog will respond "yes" or "no". Prolog also includes support for arrays, lists, and strings as well as a powerful syntax for manipulating arrays and lists. That, in a nutshell, is all that there is to Prolog. It is an incredibly simple, yet powerful, language. The basic constructs of a Prolog program can be learned very quickly, which is why Prolog is a common language used to introduce students of computer science to logic programming.

While no language can be truly mastered in a very short amount of time, Prolog is a language with which students can begin programming within minutes of seeing their first examples of a few facts and rules. There is very little complexity to find beyond the first and simplest examples. The learning curve for Prolog is very shallow.

2.1.2 Portability

The ubiquitousness of Prolog interpreters is both a blessing, and a curse in regards to the portability of Prolog programs. Prolog is a relatively old language, and there are more than two dozen variations of Prolog in popular use today, including Win Prolog, Open Prolog, GNU Prolog, and Common Prolog, and each implementation can include slight variations on all of the others. A Prolog shell is an executable program like any other; it is typically written in a programming language like Ada or C and compiled for a specific hardware/software platform configuration.

Unfortunately, many of the available implementations have been written against different Prolog "dialects"; while simple Prolog programs consisting of basic clauses are likely to run on most (if not all) Prolog interpreters, each dialect introduces inconsistencies that can make portability difficult.

Additionally, very few operating systems include a Prolog interpreter by default, and so an expert system developer may have to find and install one of the available variations. Because no one version of Prolog is available for all platforms, and in some cases there are many different versions of Prolog that will run on a single platform, insuring that a Prolog program will run on every possible platform configuration may be difficult.

Overall, though, the portability of Prolog is very good because of the availability of interpreters for most modern platforms, and the fact that the

similarities between the various implementations far outweigh the differences.

2.1.3 Security

Prolog interpreters vary from platform to platform, but the majority offer no security features beyond validation of programs written in the language. There is no support for authorization, authentication, encryption, certificates, or any other security mechanisms above and beyond the platform on which the specific Prolog interpreter has been installed.

2.1.4 Richness of Features

The set of features offered by Prolog is relatively small. The language consists of a few basic constructs, including atoms, numbers, variables, terms, arrays, lists, strings, and clauses. Prolog also contains a sophisticated syntax for manipulating arrays and lists. The Prolog interpreter also allows programmers to enter Prolog statements and see the results in real time, or to load text files containing complete Prolog programs. Prolog does not offer much beyond this small set of features, but its simplicity is part of its attraction.

Prolog is quick to learn, easy to use, and very powerful for creating rapid prototypes. Prolog may even be suitable for production applications provided that the intended number of concurrent users per installation is small, and real time results are not essential; Prolog can be very slow depending on the size of the knowledge base, the number of rules and the nature of the queries submitted. A large number of extraneous features would actually detract from the simple nature of Prolog, the very thing that makes it ideally suited for speedy development of relatively small projects.

Still, compared to some of the other languages examined here, the small number of features offered by Prolog will preclude it from use in more complex projects; anything more complicated that putting a graphical user interface (GUI) front end on a basic Prolog program would require more features than Prolog has to offer. Some efforts have been made to integrate Prolog with high level programming languages, like the Prolog Café project for Java, but Prolog itself offers a very limited set of features. Still, for some developers, that is exactly what makes Prolog so attractive for so many projects; anything more than the essentials that prolog offers would be overhead and overkill.

2.1.5 Resource Consumption

As stated previously, there are over two dozen popular variations on the basic Prolog language, available for a wide variety of platforms. Some run from a command line while others offer GUI versions of the interactive interpreter. Others provide hooks into heavier weight programming languages such as Java or C++. It is difficult to specify exactly what kind of footprint would be required by a Prolog application without specifics regarding the requirements of the exact implementation.

In general, however, Prolog is a very small footprint application, particularly the flavors of Prolog that are executed as a command line shell. Such interpreters have been running on available hardware for over 30 years, and are capable of running with very small amounts of hardware, and very little processing power. Basic Prolog is an incredibly lightweight shell, with a very small footprint, though it is often slow and may not make use of more hardware if it is available.

2.2 CLIPS

The "List Processing" language, or LISP, invented at MIT by John McCarthy in 1958, is one of the oldest procedural languages still in use today. Though modern variations of LISP have changed greatly over the decades, LISP dialects such as Common LISP and Scheme still remain in wide use today[11]. LISP was one of the first high level languages to provide the tools necessary to create expert systems for artificial intelligence projects. Beginning in the 1970s many commercial vendors began providing expert systems tools based on offshoots of the popular language.

In the mid 1980's the Artificial Intelligence Section of NASA determined that LISP based tools had three insurmountable drawbacks: LISP tools were not widely available on conventional hardware; LISP software and hardware were prohibitively expensive; and finally, LISP was not well integrated with conventional languages. The Artificial Intelligence Section determined that an expert system application developed in a conventional language, such as C, would be better suited to the requirements at NASA.

A prototype of the C Language Integrated Production System (CLIPS) was developed in 1985 based on the syntax and of ART, an expert system tool developed by Inference Corporation. As time went on, CLIPS moved beyond its original scope as a tool that would provide a foundation for a replacement for the existing commercially available expert system tools, and became first a training tool, and eventually a viable expert system tool that is widely used today in industry and government. The current release

[11] Jim Veitch, "A History and Description of CLOS"

of CLIPS, version 6.2, offers a very robust and powerful expert systems tool supported by an open development community. [12]

2.2.1 Language Learning Curve

Like LISP, the CLIPS expert system shell makes extensive use of *s-expressions*, which are fairly intuitive but can be difficult to truly master. Fortunately, most programmers are familiar with *s-expressions* through exposure to LISP or one of its dialects (e.g. Scheme) as these languages are still commonly taught as a core part of many computer science programs. The CLIPS language should be very familiar, and the essentials easy to grasp, for any programmer that has previously used a fully-parenthesized language. The programming guide for the current version of the CLIPS language is broken into two pieces: a *basic* programming guide and an *advanced* programming guide, the current versions of which together offer a total of about 700 pages of documented language features. This speaks more to the richness of features offered by the CLIPS language than it does about the complexity of the initial learning curve. In fact, some of the most basic and powerful features of the CLIPS language are fairly easy to pick up, and are thoroughly documented in the basic guide.

Apart from the *s-expression*-based language that CLIPS uses to express the facts and rules that are provided as input to its interpreter, the CLIPS expert system shell can be difficult to use. Because of the platform-dependent nature of C-language binaries; CLIPS can only be distributed as source code. The CLIPS expert system shell must be compiled using a C/C++ compiler that is capable of interpreting ANSI C (e.g. the GNU C Compiler (GCC)). This requires an in depth knowledge of an ANSI C development environment for the specific platform on which CLIPS is to be used, and the expert systems developer must be fluent enough in the C language to debug any problems that may arise. To make matters more complicated, the CLIPS library must be recompiled for each platform on which it is to run, meaning that an expert systems developer that uses one operating system for development and another operating system for production or testing must compile and debug the binaries on each platform independently. Fortunately, the CLIPS language is time tested and extremely stable; the difficulties encountered should therefore be minimized. Furthermore, the *s-expression*-based CLIPS language is seamlessly portable from one platform to another once the CLIPS binaries have been compiled, meaning that the facts and rules created on the development platform can be executed on the production or test platform without modification.

[12] The History of CLIPS, *CLIPS Reference Guide*

The CLIPS language can also be extended beyond its base functionality through add-ons programmed in the ANSI C language, but this requires an in-depth knowledge of the C programming language (a skill that is becoming less and less prevalent with the advent of more recent languages such as Java and C#). In conclusion, an expert systems developer would potentially need to learn not only the *s-expression* based language used by CLIPS, but would need to become well versed in the C-programming language as well as the C-development tools required to build the CLIPS binaries on at least one development platform. For programmers lacking proficiency in *s-expressions* or the C language, the CLIPS language learning curve can be very steep.

2.2.2 Portability

Like the other expert system shells represented in this paper, CLIPS is an interpreted language in which the expert system facts and rules are expressed using a text-based language that is fed into an interpreter. Unlike the other expert system shells, however, in many cases the CLIPS interpreter itself can be compiled on the target platform, thus improving its portability compared to languages that require the availability of an existing interpreter for the target platform.

The CLIPS expert system is distributed as source code, and binaries can be generated by compiling the CLIPS source on the target platform (i.e. a specific operating system/processor combination). Additionally, the CLIPS source code is written in the common subset of C++ and ANSI C, which means that it can be compiled on any platform that has a C/C++ compiler that is capable of interpreting ANSI C (e.g. the widely available GNU C Compiler (GCC)). Once binaries have been generated on a specific platform, those binaries are generally not portable to other, dissimilar platforms, but the CLIPS interpreter can be compiled independently on each platform on which it is intended for use. This can potentially pose a significant hurdle for an expert systems developer, particularly if the developer is not well versed in C/C++ development and does not have the required tools installed on the target platforms; if a C/C++ compiler is not present on the target platform the expert system developer must locate one and learn how to install, configure, and use it, but is still an improvement over the other expert system shells that require a native executable be installed on the target platform. Because the source code compiled on the disparate platforms is common, the feature set and functionality of CLIPS is consistent from platform to platform, regardless of the configuration.

Whereas the other expert system shells require that an interpreter be downloaded and executed, CLIPS only requires that the target environment be configured to compile the interpreter, meaning that CLIPS

can potentially be installed on any platform configuration. This task must be repeated for each disparate target platform, but ANSI C compatible compilers are widely available for the vast majority of hardware/operating system combinations. Because of this, and the fact that facts and rules written in the *s-expression*-based CLIPS language are portable from one CLIPS interpreter to another, CLIPS has excellent portability.

2.2.3 Security

The CLIPS interpreter provides verification and validation features that essentially determine whether or not that the rules and facts written in the s-expression-based CLIPS language are syntactically valid. This does little more than provide the user with message indicating errors such as syntax violations, similar to output from a programming language compiler.

The CLIPS interpreter does not provide any native support for standard security mechanisms such as encryption, certificates, or digital signatures. Unlike other expert system language interpreters, however, CLIPS is extensible through add-ons written in C/C++, and it is feasible that a developer could add support for decrypting rule sets before they are passed to the CLIPS interpreter. Because the rule sets are typically generated outside of the scope of CLIPS, through the use of an IDE or a text editor, the encrypting or signing rule sets would need to take place outside of CLIPS using a separate tool. Still, CLIPS does provide the potential for security because of its extensibility, even though it has very little security "out of the box". Because of this, CLIPS is *potentially* the most secure of the expert system shells evaluated here.

2.2.4 Richness of Features

The CLIPS feature set is almost staggering when compared to a simple, bare bones language like Prolog. Where Prolog was designed to be a pure logical programming language, with features both powerful and limited, CLIPS is designed to provide as broad an appeal as possible.

The basic CLIPS feature list includes support for programming expert systems using rules-based (heuristic) programming, object oriented programming, or procedural programming. In addition, CLIPS provides libraries that can be linked and embedded directly in procedural code or called as a subroutine from within existing C/C++ programs. CLIPS also features a GUI shell that allows users to interact directly with the CLIPS engine by entering CLIPS language commands into the shell and observing the results in real time. CLIPS also supports verification and validation that can help detect problems or inconsistencies in rule sets that may result in runtime errors.

CLIPS has been in development for over 20 years, and its feature set is robust and stable, with few features added from release to release. Compared to simpler languages like Prolog or even LISP, the feature set offered by CLIPS is incredibly rich.

2.2.5 Resource Consumption

CLIPS requires more resources than Prolog, due to its significantly larger feature set, but most CLIPS applications will run on a small scale processor in under 640 kilobytes of RAM. This is one of the major advantages of compiling the CLIPS source code directly into a native executable; very little overhead is required. Of course, if a developer should choose to use the GUI interpreter provided with CLIPS to interact with the shell, the resource requirements are a bit steeper, but at runtime the GUI is not required.

It is possible to write very large, complex systems that may cause CLIPS to consume more memory, and it is possible to configure CLIPS to consume a larger amount of RAM. In general, however, CLIPS has fairly low resource requirements at runtime and slightly heavier resource requirements if the GUI shell is used.

2.3 JESS

The Java Expert System Shell (JESS) while still in its infancy was a child of CLIPS, meant to be little more than a translation of the popular C-language based expert system language/interpreter in Sun's popular Java programming language. But, since its initial release in late 1995, the author, Ernest Friedman-Hill, has added many new features that separate it from its predecessor. The most recent version, JESS 7.0p1, was released in December of 2006 with work on version 7.1 beginning immediately thereafter. In the same time, development on the CLIPS language has also continued (though at a slower pace due to its stability and relative completeness of features), further separating the two languages. Today, JESS is related to CLIPS in much the same way that CLIPS is related to LISP; an independent language with strong roots in its parent.

Like CLIPS, JESS uses a fully parenthesized language to create *s-expressions*, linked lists used to describe the facts and rules of an expert system. At its core, JESS uses the *Rete* algorithm to efficiently process the rules described in an expert system. The *Rete* algorithm, first defined in 1974 by Dr. Charles L. Forgy working at Carnegie Mellon University, is an extremely efficient pattern matching algorithm that is used in many rules processing systems. The JESS implementation of the algorithm is written entirely in Java.

Like many other expert system shells, including Prolog, LISP, and CLIPS, JESS includes support for an interactive "shell" that can be accessed through a GUI or from a command line interface; the shell allows users to type JESS language commands directly into the interpreter, or to load files that contain entire JESS programs, and to observe the results in real time. In addition to this, however, JESS provides a robust Java API that allows developers to manipulate the JESS environment programmatically, without ever having to deal directly with the *s-expression* based language parsed by the JESS interpreter. JESS also fully leverages Java's reflection API to allow for virtually limitless extensibility, even providing support for loading add-ons in the syntax of the text-based JESS language itself. For example, the FuzzyJ toolkit adds fuzzy logic to JESS through seamless plug-ins loaded automatically at runtime.

2.3.1 Language Learning Curve

The learning curve for JESS depends on the angle from which a developer chooses to attack the language, and the developer's own experience level. JESS offers many entry points, each with its own difficulties and intricacies, perhaps the simplest of which is learning through experimentation; developers are free to interact with JESS through its interpreter by typing JESS language commands directly into a GUI that provides immediate, real time feedback. For students of computer science that have worked with similar shells provided by LISP or Prolog implementations, this mechanism can be an intuitive and informative mechanism for learning the language, especially when combined with the tutorials that are packed with the JESS installer.

Developers may also choose to use the JESS platform more like a traditional compiler, by composing complete expert systems made up of any number of facts and rules in a text-file written in the JESS *s-expression* based language. These files can then be fed into the JESS interpreter for debugging and testing. This approach may be more natural for developers that are familiar with fully parenthesized languages like JESS, and who are willing to learn the basics of the language as described in the JESS manual before ever attempting to write a program.

Finally, developers fluent in the Java programming language may be most comfortable approaching JESS through its robust application programming interface (API). The API allows developers to build complete expert systems, from the ground-up, programmatically, constructing facts and rules by calling methods directly on the JESS classes. The API is fully documented using JavaDoc™, and the JESS platform even includes plug-ins and support for the popular Eclipse IDE.

The flexibility that JESS provides to developers can be daunting at first, but it essentially allows developers to tailor the learning experience to their own abilities. Because of this, the initial learning curve for JESS can be fairly gradual, but like many languages with as much complexity and as many features as JESS, fully mastering the platform can be difficult. In support of this, JESS does provide a wealth of documentation and example programs.

2.3.2 Portability

As stated previously, JESS has been fully implemented in pure Java, meaning that the compiled byte-codes can be executed on any platform that supports a Java Virtual Machine (JVM) compliant with the Java Standard Edition version 1.4 or later. The current version of JESS does not yet support some of the features introduced in version 1.5 of the Java language, particularly generics and enumerations, which may cause some warnings when attempting to compile the JESS source code. The JESS binaries, however, can be executed unmodified.

It should be noted that some small footprint java implementations, particularly those based on the J2ME specifications, do not support many features required by JESS; such JVMs typically only support a small subset of the language. It is possible, however, to find alternate vendors who supply non-standard JVMs that are both small footprint, and that support the Java APIs required by JESS.

Because JESS is fully portable (assuming that a compliant JVM is available on the target platform), there is no need for different implementations of the basic shell for different platform configurations. This eliminates the chance that different features are supported on different platforms (as often happens with languages, such as Prolog, that are essentially rewritten from one platform to the next, often resulting in the creation if disparate and incompatible ‘dialects’), and guarantees that the execution of the interpreter will be exactly the same, regardless of the underlying hardware or operating system. Needless to say, expert systems written in the JESS language can be executed without modification on any platform on which the JESS interpreter itself can be executed. JESS is the most portable of the expert system shells discussed here.

2.3.3 Security

Java provides a rich set of security features that are built into the language, but out of the box JESS does not leverage any of these features. In fact, the word “security” is mentioned exactly once in the entire 200 page manual for JESS version 7.0p1 in reference to a bug fix for Applet security issues; the words “secure”, “encrypt”, “encryption”, and

"signature" do not appear at all. JESS relies primarily on the operating system on which it runs to provide security to developers and users.

It is possible for advanced Java programmers to create a security profile that can be deployed along with JESS and any expert systems intended to run in the secure environment. Most JVMs support a standard Java security profile that can restrict access to certain features of the system based on the roles assigned to programs or users, but JESS does nothing to facilitate or enhance the security offered by Java.

JESS does not offer any native support for encryption, which would help protect and secure expert system components written in the JESS language in transit or during storage. Additionally, JESS does not offer native support for digital signatures, which identify whether or not the creator of the expert systems written in the JESS language is a trusted source, nor does it provide any mechanism to prevent or detect unauthorized modification of unsigned components.

Despite the lack of security features built into the JESS language or the interpreter, JESS developers do have access to the source code, and JESS provides a rich, reflection-based mechanism for extending the language (as discussed previously). Assuming that any security mechanisms grafted onto the language would be written in Java, such features would allow developers to write platform independent, fully portable, Java-based plug-ins that could add security to the otherwise insecure language. Such independently implemented security add-ons would be non-standard, however, and would hurt the portability of the expert systems components developed to make use of them. JESS security is weak, but on par with (or slightly better than) that offered by CLIPS and Prolog.

2.3.4 Richness of Features

JESS began in 1995 as a Java implementation of the CLIPS expert system shell, and as such copied many of the features available in the CLIPS shell at the time. In the time since then, the languages have evolved independently, with features being added to both as newer versions become available.

Like CLIPS, JESS offers a GUI shell for interacting directly with the language, allowing developers to experiment by typing commands directly into the shell to see the results in real-time. JESS also provides a development plug-in for the popular open-source IDE Eclipse, for programmers that prefer to develop against the JESS Java APIs.

The most obvious divergence from the original CLIPS functionality, however, is that JESS has been designed to be extensible using the

reflection APIs in Java. The JESS s-expression language includes a syntax for automatically loading libraries external to JESS than enhance its features. One of the most popular examples is the FuzzyJ add-on, part of the FuzzyJ Toolkit by Bob Orchard. The add-on allows users to load the FuzzyJ Toolkit fuzzy logic libraries into JESS at runtime by using JESS language syntax to specify the Java packages to load using reflection.

Recent versions of JESS have also made significant additions to the JESS feature set, including support for backward chaining, and translation between the fully parenthesized JESS language and XML. Overall the JESS feature set is very rich.

2.3.5 Resource Consumption

JESS is an inexpensive, feature rich shell, but compared to other shells like Prolog and CLIPS the resource requirements are steep before even a single fact is declared because JESS is written in Java. Java is an interpreted language that typically runs within an on top of an executable written in some other high level language, such as C++. This means that, typically, Java programs run slower than programs written in a language that compiles to native executables; the Java byte codes must be interpreted and executed at runtime, which results in a delay.

Java programs also require much more memory than native applications with similar functionality. Even small-footprint versions of the Java Virtual Machine (JVM) include relatively large libraries that must be loaded even for In order for even the most basic Java programs to function. Some vendors offer a “micro” edition of Java that requires far fewer resources than a full JVM, but such implementations do not support the features required to execute JESS. A typical JVM running on Windows requires 6-8 megabytes of RAM simply to run a “Hello World!” program; smaller footprint variations can require 1-2 megabytes of RAM for the same purpose. The JESS shell adds several megabytes to that footprint, and can consume a large amount of CPU time when evaluating complex rule sets with many partial rule matches. The Rete engine on which JESS runs is efficient, but still may be resource intensive depending on the rule set and facts provided.

While it may be possible to force JESS to run small, simple rule sets on a small footprint JVM with just a few megabytes of RAM and a small amount of CPU time, such applications are far better suited for a language designed around those requirements, like Prolog. JESS is at its best with a good amount of available resources, and expert systems that make full use of its entire feature set. For such systems, the resources required by JESS are comparatively high.

2.4 Blaze Advisor

Bill Fair, an engineer, and Earl Isaac, a mathematician, founded Fair, Isaac, and Company in 1956. The corporation developed the first credit scoring systems in 1958. Over the next several decades, as such systems became more and more ubiquitous, larger and more complex hardware and software solutions were required to process the millions of transactions required by banks throughout the world. In response to this demand, the renamed Fair Isaac Corporation created Blaze Advisor, a large and powerful rules engine designed to handle thousands of simultaneous transactions.

In 2005 Fair Isaac Corporation bought the rights to the Rete III algorithm, a modified version of the original Rete algorithm for use in Blaze Advisor. In the past several years Blaze Advisor has been retooled and rewritten from the ground up to make use of the latest Java technologies, including full web services support and integration. In addition, Blaze Advisor now stores facts, rules, and workflows created by developers as XML documents.

Though Blaze Advisor was designed to handle the raw credit processing needs of large, multinational banks and other lending institutions, it is a general purpose, scalable rules engine suitable for any large scale expert system. Because Blaze Advisor is designed for very large systems processing thousands transactions per second, it does not lend itself to smaller applications, and designed to run on large, powerful servers; Blaze Advisor is also a closed-source application only available commercially, with licensing fees that are far too expensive for all but the largest projects.

2.4.1 Language Learning Curve

Blaze Advisor is a hugely complex, server-based product requiring teams of developers and administrators to design, implement, and maintain the large scale expert systems that it is designed to run. The learning curve is immense, far steeper than any of the other expert systems described here. Fair Isaac Corporation offers a series of three, week long courses (totaling about 120 hours of class time) to train developers and administrators on the use of the product.

In addition, Fair Isaac Corporation sells consulting services and support contracts designed to solve problems encountered even by those that have completed the full set of courses. Like many other large companies, Fair Isaac Corporation creates large, and powerful, yet difficult to use products and then makes additional profit by helping users unlock the mysteries that they may encounter. Unlike the other expert systems examined here,

Blaze Advisor does not have a healthy, open, active, independent development community that provides advice, documentation, tutorials, and training for free. Tackling Blaze Advisor is a task that requires a great amount of time, effort, and in many cases, expense.

That being said, the primary mechanism for interacting with Blaze Advisor during the development process is a feature rich graphical user interface (GUI) that allows developers to create and share repositories of facts and rules, create rule flows, and write rules using drag-and-drop tools. Additionally, developers can example the XML associated with each of these items and edit the XML directly to tweak settings or functionality (though this is not recommended, particularly for novice developers; the tools for viewing and editing the raw XML are provided in most cases for informational purposes only, and to give developers insight into what the output of the GUI tool looks like).

Blaze Advisor also features a proprietary language with a Java-like syntax that allows developers to manipulate the entities in the system programmatically instead of (or combined with) the drag-and-drop interface. The creators of the language tried to use "real world" terms and phrasing to create a high level language as close to English as possible. For example, where a Java programmer might create a class that looks like this that shown in *Figure 2.4.1.1*.

```
public class Policy {
  Date effectiveDate;
  Date createdDate;
  Date expirationDate;
  long duration;

  public Policy() {
    createdDate = new Date();

    Calendar c = Calendar.getInstance();

    // January (0) 1, 2005
    c.set( 2005, 0, 1 );
    effectiveDate = c.getTime();

    // March (2) 31, 2006
    c.set( 2006, 2, 31 );
    expirationDate = c.getTime();
  }
}
```

Figure 2.4.1.1: An example of a simple Java class.

While a Blaze Advisor developer would create a similar object like that in *Figure 2.4.1.2*.

```
a Policy is an object with {
```

```
  an effectiveDate : a date;
  a createdTimestamp : a timestamp;
  an expirationDate : a date;
  an effectiveDuration : a duration;
  } initially {
    createdTimestamp = calendar().currentTimestamp(true);
    effecitveDate = '1/01/2005';
    expirationDate = '3/31/2006';
  }
}
```

Figure 2.4.1.1: An example of a simple Blaze Advisor object.

It is difficult to tell what the aim of the unique syntax of this language is; Blaze Advisor, while extremely powerful and versatile, is a dauntingly large, complex, and difficult to use product. It seems logical that only qualified and experienced developers would be likely to develop the expert systems intended for use with Blaze Advisor. Creating a high-level language with an English-like syntax may seem to make sense when catering to novice users, but such users do not seem to be the likely (or logical) target audience for Blaze Advisor. Instead, such a language uses syntax that is strange and unfamiliar for experienced programmers, and may actually create a hurdle where otherwise there would be none. Being that Blaze Advisor uses a combination of Java, XML, and GUI drag-and-drop programming, a proprietary language introduced into the mix only adds complexity to an already complex package.

Blaze Advisor inarguably has the steepest learning curve of all the expert systems examined here; it is not only difficult to master, as the other languages are, but the learning curve is very steep from the very beginning.

2.4.2 Portability

Blaze Advisor is written in Java, arguably the most portable programming language available today and the reason that the JESS interpreter enjoys such high portability. Unlike JESS, however, Blaze Advisor is designed for large enterprise-scale expert systems, and is meant to process thousands of concurrent transactions. Blaze Advisor is a rules engine running behind a fully featured web server, and as such it requires server-scale hardware. This would include one or more servers capable of the raw processing and memory requirements required to handle thousands of concurrent transactions. While it is possible to run Blaze Advisor on smaller hardware, the overhead required simply to run the web server, rules engine, and repositories is overkill for anything but the largest applications. Unlike the other languages that are more bound by software requirements than hardware, Blaze Advisor simply was not intended to be portable or deployed on small footprint platforms.

The financial cost of a Blaze Advisor license must also be examined; not only is the hardware required to run Blaze Advisor likely to be expensive, but the software itself comes at a steep price; a single production license and a handful of development licenses can cost $100,000 or more, and gets even more expensive when combined with a support contract and the training required to learn the software. This puts Blaze Advisor far out of the reach of anything but the largest development projects, whereas the other languages examined here are typically free, especially for academic uses (though JESS does require a licensing fee for commercial uses).

That being said, Blaze Advisor is a Java application, and can be deployed on many server configurations including myriad Linux/UNIX, or Windows based server configurations. It is not bound to a specific operating system, despite its hardware requirements. Still, despite the potential portability of the software itself, the number of possible deployment options for Blaze Advisor is very low; Blaze Advisor has the lowest portability of any of the languages discussed here.

2.4.3 Security

The hardware on which Blaze Advisor is executed will typically be secure in many ways. The location of the expensive server hardware will not be accessible to most malicious users. In addition, the server operating system software is likely to be more secure than that running on a desktop, laptop, or other development machine. The repositories in which the Blaze Advisor knowledge bases are stored are likely to be similarly secure. But, like the other expert system shells discussed here, the security that Blaze Advisor *adds* to that provided by the hardware and software on which it is executed is not extensive.

As has been mentioned previously, Blaze Advisor is software meant to run on a server; specifically, it is a web application meant to be accessible to clients over a network; the primary mechanism for interacting with Blaze Advisor is over the network, through secure web service calls. In this way, it is unique among the expert system shells examined here as the other shells are meant to be interacted with directly by a user or developer. Blaze Advisor, on the other hand, is meant to be executed on a server and accessed remotely. Blaze Advisor applications can be triggered by external events, can generate events that are sent to subscribers, and can be called through standard web services interfaces. It supports encrypted communications (via HTTPS), and standard web services security that requires authentication and authorization. Additionally, Blaze Advisor can be configured to require authentication and authorization to access the repositories that contain the knowledge bases that are executed on the rules engine (which may be stored on a separate server), though Blaze Advisor does not encrypt the XML-based documents containing the

knowledge base, nor does it support digital signatures to help verify the authenticity of the documents once they have been retrieved from the repository.

Still, by restricting access to the application itself via remote means, and leveraging SSL communications (which can be configured to require certificates that establish trust between the client and the server), Blaze Advisor does offer a level of security well above and beyond that offered by the other shells.

2.4.4 Richness of Features

Blaze Advisor offers a wealth of features not available in the other languages examined here. Somewhat like JESS and CLIPS, Blaze Advisor offers a GUI-based environment to developers wishing to create knowledge bases. Unlike the other GUI shells examined here, though, the Blaze Advisor GUI is an incredibly heavy weight, dense piece of software with thousands of options, and it does not provide an interactive "shell" that can be used to directly input commands and see the real-time results of executing those commands. Instead, it supports full drag-and-drop programming, allowing developers to create complex rule workflows, facts, rules, and entire knowledge bases almost entirely with the mouse. But more than that, developers can dive into each widget and add code (similar to adding scripts to GUI components in Visual Basic), and even examine the XML documents used to persistently store each and every item in the system.

Blaze Advisor allows developers to create and access rules repositories that act as a sort of built-in source control system, making it easy for many developers to share rules, and to co-develop large applications. The rules repositories are used to store both development and production versions of knowledge bases, that can eventually be used in deployed applications.

Blaze Advisor features its own, unique development language as well, allowing developers to program reasoning directly into the system. The creators of Blaze Advisor have attempted to craft a language with an English-like syntax allowing developers to specify that an entity "is an Object" or that a primitive declared inside an object is "a date" or "a timestamp". The language is fully object oriented, and supports nine primitive types, enumerations, arrays, and a host of advanced, high level language features designed to make programming as easy as possible. The stated goal is to make the language less intimidating to novice developers, though it is unlikely that novice developers will be creating the large enterprise scale applications for which Blaze Advisor is most suited.

Blaze Advisor also allows developers to create or import custom Java objects, databases, and XML schemas directly in the development GUI that can then be used in conjunction with the application. Blaze Advisor also features wizards for handling almost anything, from creating a project to organizing the facts and rules into a branching flow, all using drag-and-drop GUI programming.

It would be difficult to fully summarize the feature set of Blaze Advisor as it far outstrips the features available in most other expert system shells, and certainly dwarfs the available features in any other shell examined here. For the majority of applications not intended for the large enterprise, and perhaps even many of those that *are* intended for large scale production uses, the number of features may frequently be overkill. Blaze Advisor tries to provide a feature to do almost everything, and this unfortunately comes at a price; the application is large, dense, often confusing, and requires weeks of training to use effectively. There are also many known bugs, and frequent issues with stability (at least as of version 6.0); though the company actively releases patches and bug fixes for known issues, they also offer lucrative support contracts to their customers that offer assistance in solving the many issues that may arise.

2.4.5 Resource Consumption

It would be difficult to discuss Blaze Advisor without repeatedly stressing that it is a rules engine that has been designed to run on very large scale hardware; Blaze Advisor is meant to process large enterprise scale expert systems on an array of servers. Blaze Advisor is an incredibly heavy weight, feature rich application that demands a huge amount of resources, including a large CPU with a significant amount of memory. The application was never intended to run on small footprint hardware or even typical desktop systems for any purpose other than development and cursory testing.

Blaze Advisor has extremely high resource requirements.

2.5 Results of the Comparison

Each of the languages evaluated in this section is still in popular use today for very good reasons; each has its strengths and weaknesses, but each is best suited for different problems. Some languages, like Prolog, are fairly easy to pick up, available on many platforms, and can accomplish most of the tasks required for rapid prototyping and even some small scale production applications. Others, like Blaze Advisor, while difficult to learn, and incredibly resource intensive, are very well suited for large

scale applications meant to handle large volumes of transactions that would cripple the other languages.

Because of this it's difficult to rank the languages from "best" to "worst"; depending on the requirements the "best" language may (and often does) differ from application to application. Factors like the target hardware, operating system, availability of software, user requirements, user load, and the types of problems being solved vary greatly, and so does the appropriate solution. This is why knowledge engineers are trusted to choose the right system for the task on a case-by-case basis, and do not simply pick a single solution and try to apply it to every application.

The information presented in *Table 2.5.1* is meant to demonstrate the differences between the languages at a glance, and to highlight the fact that no single language is suited for all problems.

	Learning Curve	**Portability**	**Security**	**Features**	**Resource Consumption**
Prolog	Shallow	Medium	Low	Medium	Low
CLIPS	Steep	Medium	Low	High	Medium
JESS	Medium	High	Low	High	High
Blaze Advisor	Extremely Steep	Low	Medium	Extremely High	Extremely High

Table 2.5.1: An "at a glance" comparison of the expert system shells and languages examined in this section.

Each language is given rankings based on the criteria examined in much greater detail earlier in this chapter, including:

- **Learning Curve** – Rated on a scale of *shallow*, meaning fairly quick and easy to pick up and learn, to *extremely steep*, meaning extremely difficult and time consuming to learn.
- **Portability** – Rated on a scale of *low*, meaning that in many cases it is difficult to transport either the knowledge base and programs or the interpreter itself between disparate platforms, to *high*, meaning that in the majority of cases it is very easy to move the knowledge base, programs, and the interpreter between disparate platforms.
- **Security** – Rated on a scale of *low*, meaning that little or no security is offered beyond that provided by the operating system on which the interpreter runs, to *high*, meaning that many security options are available to developers and users.
- **Features** – Rated on a scale of *low*, meaning that there are very few features beyond basic functionality, to *extremely high*, meaning that many robust features are included with the expert system shell.

- **Resource Consumption** – Rated on a scale of *low*, indicating very low resource consumption suitable for small footprint platforms, to *extremely high*, indicating that the resources required to run the interpreter is much higher than the average application (e.g. server hardware).

Each shell has been rated based on the detailed information given earlier in this section, and it is clear from looking at the data presented in *Table 2.5.1* that no one language is clearly superior to all others for any application. For example, the Prolog language may be best suited for rapid prototyping or small scale applications due to its low resource requirements and ease of use. The CLIPS shell, on the other hand, offers a much richer set of features in exchange for an increase in the required resources (largely due to its GUI components) and a steeper learning curve. JESS provides many of the same features as CLIPS, but adds some Java specific features like reflection, extensibility, and high portability, meaning that applications with unique features intended for deployment on multiple platforms may work best in the JESS environment. Finally, Blaze Advisor is the only application discussed here capable of handling large enterprise scale applications supporting thousands of concurrent transactions, but because of its incredibly steep learning curve and high resource requirements it is not a suitable option for applications of a smaller scale.

This is precisely where XESS may help knowledge engineers who may want to begin development and testing of prototype expert systems applications when the requirements are not clear, or the best solution is not immediately obvious. XESS allows expert systems developers to create knowledge bases, including facts and complex rules, in a simple XML language that uses common, immediately recognizable, language-independent terms. Such systems can then be executed on multiple shells to test performance, reliability, resource usage, and any other aspects of the expert system that may be important. Once a decision has been made, the expert system can be executed through XESS plug-ins, or translated into the native language of the target shell using the same plug-ins. This frees the development team from the task of delaying development until an appropriate shell is chosen, or from needing to rewrite a knowledge base already in development if the decision is made to switch shells in mid development.

3 XESS Schema

3.1 Introduction

The purpose of an XML schema is to provide a detailed description of an XML language so that documents written in the language can be validated[13]. In effect, the schema defines the grammar of the language, which allows parsers to interpret the contents of the document. The XESS schema provides detailed information on every valid element of an XESS document, including the sections of the document that must be used to define facts, rules, and comments.

The later sections of this chapter examine each of the possible elements in detail, providing context sensitive definitions for tags, attributes, and possible values. The schema language makes it difficult to clearly express rigid requirements, such as valid value ranges for attributes, and so wherever possible the detailed descriptions examples included in this chapter will elaborate on aspects of the schema that may be unclear.

The XESS schema has been designed to be extensible, so that other developers can extend existing templates to create new facts, rules, or entirely novel elements of custom XESS documents. Such extensions are considered custom and are not likely to be supported by individually developed interpreters and plug-ins.

The XESS schema is presented in its entirety in Appendix A for reference purposes.

3.2 Entities

Figure 3.2.1 on the following page provides an overview of the major entities in an XESS document as defined in the XESS schema. It shows the relationship between the different system entities. This diagram is for illustrative purposes only.

[13] Jian Bing Li, James Miller, "Testing the Semantics of W3C XML Schema"

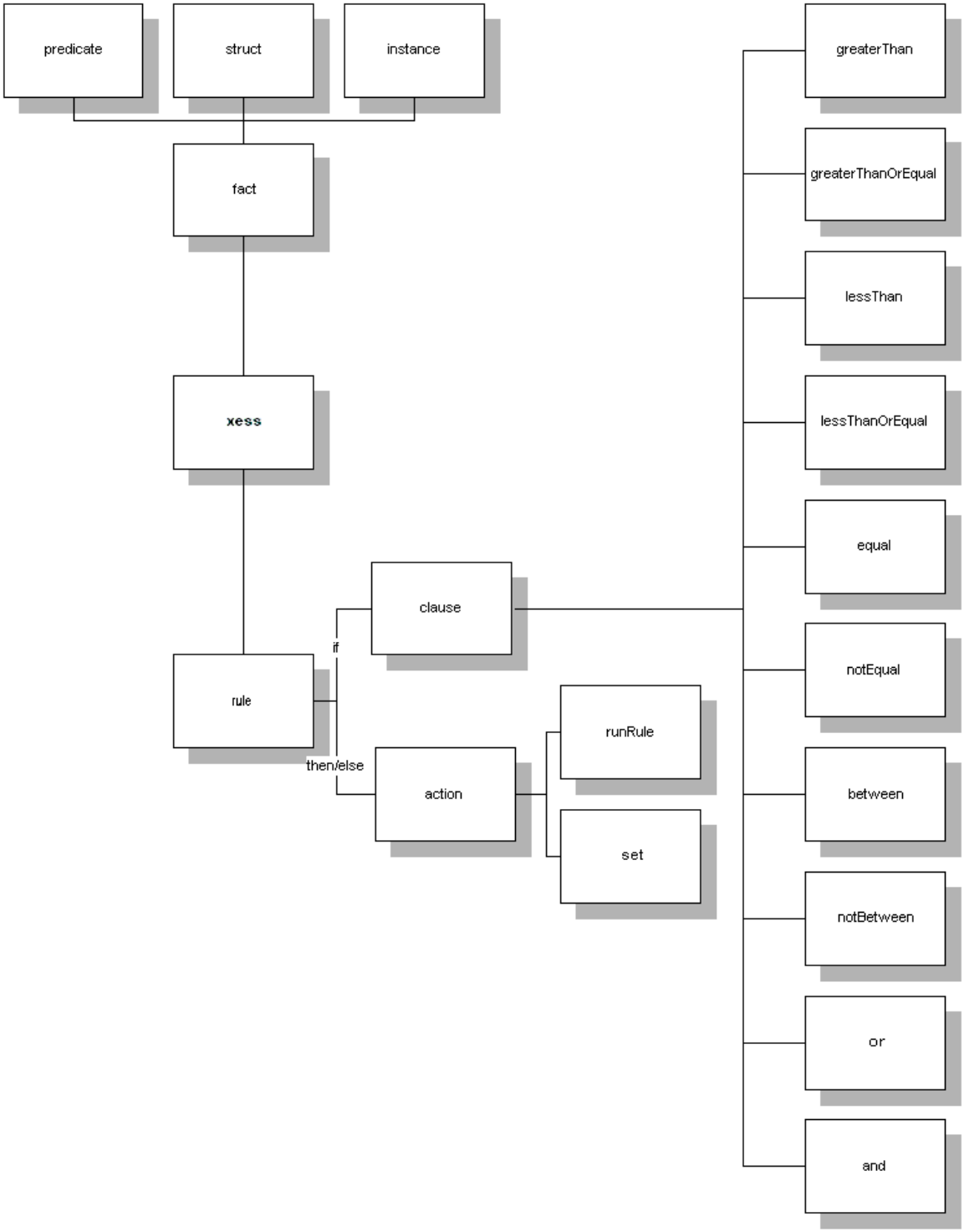


Figure 3.2.1: The major entities within an XESS document as defined by the XESS schema

3.3 Facts

The XESS scheme defines an abstract fact type that is the parent of all of the supported facts. The *fact* entity enforces the rule that all facts must be named by including a *name* attribute. Fact names must be unique, though it is impossible to enforce this rule via the schema, and therefore name checking must be performed at runtime by the XESS interpreter (this is discussed in later chapters). Figure 3.3.1 shows the abstract *fact* entity within the XESS schema.

```
<xsd:element name="fact" type="factType"/>

<xsd:complexType name="factType" abstract="true">
  <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>
```

Figure 3.3.1: The XESS schema definition for the abstract fact entity.

Facts come in several forms, but essentially facts are name/value pairs; in same cases (such as *structs* or *instances)* a fact is a set of name/value pairs that are associated within a larger structure. Facts can be passed as arguments into rules, or referenced globally, and facts may be updated with new values at runtime as a result of the execution of rules. These properties of facts are discussed in later chapters.

3.3.1 Fact Value Types

In general values within the XESS language are not strongly typed. All values are specified as strings, and types are only interpreted as necessary. Individual XESS plug-ins may assign strong types based on initial values, and then enforce those types at runtime. XESS neither requires, nor enforces this behavior.

In the cases where types are inferred, the following rules are applied:

- Strings beginning with a “+”, “-“, or digit and containing only digits are assumed to be integer values.
- Strings beginning with a “+”, “-“, or a digit and containing numerical characters in addition to exactly one decimal character (“.”) are assumed to be floating point values.
- Strings with the value “true” or “false” (without respect to case) are assumed to be boolean values.
- Strings that do not meet the requirements of any of the above types that contain exactly *one* character are considered character values.
- Strings beginning with the “@” (commercial at) character are considered references to variables by name; e.g. “@somePredicate” refers to a fact within the current scope that has the name “somePredicate”.
- All other strings are considered “string” values.

In the majority of cases types are handled by the underlying engines, and thusly the responsibility of translating types rests with the *plug-ins* responsible for translating the data between the XESS interpreter and the expert system used at runtime.

3.3.2 The Predicate Fact

The *predicate* defined in the XESS schema provides the syntax for the simplest kind of fact. Sometimes referred to as an assertion or an assignment, the *predicate* fact type simply associates a name with a value. Predicates can be used to create simple, named variables with global scope that can be modified and accessed by all of the rules within the system.

```
<xsd:element name="predicate" type="predicateType"
   substitutionGroup="fact"/>

<xsd:complexType name="predicateType">
  <xsd:complexContent>
    <xsd:extension base="factType">
      <xsd:attribute name="value" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.3.2.1: The XESS schema definition for the Predicate fact type.

The *predicate* fact type uses the tag name *predicate* and contains two attributes: the *name* attribute is inherited from the abstract parent and must be unique within the global scope as two predicates with exactly the same names may cause errors or unexpected behavior (a strict XESS interpreter may throw an exception, while more lenient interpreters may simply replace the old predicate with the most recently parsed predicate with the same name); and the *value* attribute which is used to specify the value of the *predicate* as a string.

Predicates are not strongly typed, and the value of each predicate is specified as a string. The interpreter or plug-in may attempt to intelligently determine predicate types whenever a comparison between two predicates is required, but otherwise predicate types are not considered (see the previous section for more detail).

Figure 3.3.2.2 is an example of a *predicate* that creates a simple assignment between a *name* and a *value*. In this case if the value must be typed, it will be interpreted as a string (see the previous sections for rules on typing).

```
<predicate name=”examplePredicate” value=”a value”/>
```

Figure 3.3.2.2: An example of a Predicate fact expressed in XML that creates a simple string assignment.

3.3.3 The Structure Fact

A *structure* is a collection of facts that are associated within a parent element, similar to a struct in the C/C++ language or a class in Java that contains fields but no methods. These *fields* are represented as child elements within the *structure*.

```
<xsd:element name="struct" type="structType"
   substitutionGroup="fact"/>

<xsd:complexType name="structType">
  <xsd:complexContent>
    <xsd:extension base="factType">
      <xsd:sequence>
        <xsd:element ref="comment" minOccurs="0"
           maxOccurs="1"/>
        <xsd:element name="field" minOccurs="1"
           maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="name" type="xsd:string"/>
            <xsd:attribute name="type" type="xsd:string"
               use="optional"/>
            <xsd:attribute name="initialValue"
               type="xsd:string" use="optional"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.3.3.1: The XESS schema definition for the Structure fact type.

The *structure* fact type uses the tag *struct* and inherits the *name* attribute from the abstract parent *fact* entity; because *structures* are facts, they must be uniquely named with respect to the other facts in the system. The child *field* entities use the tag name field and are very similar to the *predicate* fact type mentioned in the previous section. Each *field* contains two attributes, the *name* and the optional *initialValue*. The *name* attribute specifies the name of the field within the scope of the parent *structure*; because of this, the *name* need only be unique within the *structure*. Different *structures* may contain fields with the same name or with names the same as other *facts* within the system without creating collisions. The *initialValue* attribute is optional, and is used to specify a default value for the *field*. Each *structure* may also contain an optional *comment* that describes the purpose and contents of the *structure*.

Figure 3.3.3.2 is an example of a simple *structure* that contains three named fields. Two of the fields have initial values, which will be used as default values in the event that specific values for those fields are not supplied in an *instance* of the *structure* (see the section immediately following this for more information on *instances*).

```
<struct name=”ExampleStruct”>
  <comment>This is an example struct.</comment>
  <field name=”Field1” initialValue=”default1”/>
  <field name=”Field2”/>
  <field name=”Field3” initialValue=”default3”/>
</struct>
```

Figure 3.3.3.2: An example of a Structure containing 3 fields, 2 of which have default values.

For clarification, Figure 3.3.3.3 is an example of what the same *structure* might look like if written as a class in the Java language. Note that the *Field2* field is initialized as null, while the other two fields are given default values. All fields are generic objects, which allow any Java object to be assigned to those values. Types are largely ignored until they are needed, for example when the fields must be compared to some other values for equality.

```
public class ExampleStruct {
  public Object Field1 = new String( “default1” );
  public Object Field2 = null;
  public Object Field3 = new String( “default3” );
}
```

Figure 3.3.3.3: An example of the same Structure written in the Java language.

A *structure* may also contain one or more *fields* that contain pointers to instances of other *structures*; in these cases the *field* type specifies the name of the *structure* of which the *field* is an instance. Figure 3.3.3.4 is an example of such a relationship.

```
<struct name="Person">
      <field name="first-name"/>
      <field name="middle-initial"/>
      <field name="last-name"/>
      <field name="sex"/>
      <field name="age"/>
</struct>

<struct name="Parents">
      <field name="parent1" type="Person"/>
      <field name="parent2" type="Person"/>
      <field name="child" type="Person"/>
</struct>
```

Figure 3.5.5.4: An example of a structure that contains fields that are instances of other structures.

In this example, the “Person” *structure* defines a simple relationship between several *fields* with primitive types; a “Person” has a *first name*, a *middle initial*, a *last name*, a *sex* (presumably “male” or “female”), and an *age*. The “Parents” *structure*, however, defines a relationship between

three *fields* that are each instances of a *strucure*; two parents and a child, each of which is itself an instance of the "Person" *structure*.

Like classes in Java and other high-level, object-oriented languages a *structure* is simply a description of a concrete entity within the system; references to *structures* cannot be passed as arguments to rules. Instead, *structures* are meant to be instantiated in the form of *instances*, which are described in more detail in the following section.

3.3.4 The Instance Fact

A *structure* fact is a template that defines a relationship between *fields* that are collected within a larger envelope. As specified in the previous section, *structures* are not concrete entities, and provide only default values for the child *fields*. Concrete representations of a *structure* within the system are referred to as *instances* of the *structure*. The *instance* fact type provides real values for the fields within a *structure*, and the values of the fields within each *instance* can be modified without changing the default values of the parent *structure*.

```
<xsd:element name="instance" type="instanceType"
   substitutionGroup="fact"/>

<xsd:complexType name="instanceType">
  <xsd:complexContent>
    <xsd:extension base="factType">
      <xsd:sequence>
        <xsd:element ref="comment" minOccurs="0"
           maxOccurs="1"/>
        <xsd:element name="field" minOccurs="0"
           maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="name" type="xsd:string"/>
            <xsd:attribute name="value" type="xsd:string"
               use="optional"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="type" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.3.4.1: The XESS schema definition for the Instance fact type.

The *instance* fact type uses the tag name *instance* and contains two attributes, *name* and *type*. Like all other facts within an XESS document, the *name* is inherited from the abstract parent fact entity and must be unique within the system. The *type* attribute associates the *instance* with the *structure* of which it is a concrete implementation. Each *instance* contains zero or more *field* elements that each contains two attributes; a *name* and a *value*. The *name* field must correspond to one of the names in

the parent *structure*. The *value* field is used to specify a value for the field, or to override the default value inherited from the parent *structure*.

Figure 3.3.4.2 shows an example of an *instance* of a *structure* (the same structure featured in Figure 3.3.3.2 in the previous section). In this case the *instance* provides overriding values for two of the three fields defined in the structure; the third field is omitted and therefore inherits its default value from the parent *structure*.

```
<struct name=″ExampleStruct″>
  <comment>This is an example struct.</comment>
  <field name=″Field1″ initialValue=″default1″/>
  <field name=″Field2″/>
  <field name=″Field3″ initialValue=″default3″/>
</struct>

<instance name=″ExampleInstance1″ type=″ExampleStruct″>
  <comment>This is an example instance.</comment>
  <field name=″Field1″ value=″assigned value 1″/>
  <field name=″Field2″ value=″assigned value 2″/>
</struct>
```

Figure 3.3.4.2: An example Instance of the Structure in Figure 3.3.3.2.

In the above example the final values of the three fields would be as follows: *Field1* has the value “assigned value 1”, *Field2* has the value “assigned value 2”, and *Field3* has the value “default3”, which is the default value inherited from the parent *structure*.

Figure 3.3.4.3 is an example of a second *instance* of the same *structure*. In this example all three *fields* within the *instance* are assigned values. These values are wholly separate from both the parent *structure* and all other instances of the same *structure*. Furthermore, *fields* within each instance can be modified at runtime without affecting the initial values within the parent *structure* or other *instances*.

```
<instance name=″ExampleInstance2″ type=″ExampleStruct″>
  <comment>This is another example instance.</comment>
  <field name=″Field1″ value=″tom″/>
  <field name=″Field2″ value=″dick″/>
  <field name=″Field3″ value=″harry″/>
</struct>
```

Figure 3.3.4.3: A second example Instance of the Structure in Figure 3.3.3.2.

For clarification, Figure 3.3.4.4 is an example of what the same instances might look like in Java. In this case the *main* method is used to create two instances of the class *ExampleSruct* (that was originally defined in Figure 3.3.3.3 in the previous chapter, but is repeated here for convenience). The first instance, *exampleInstance1*, inherits the default value assigned to *Field3*, but programmatically overrides the values of the other two fields.

The second instance, *exampleinstance1*, provides non-default values for all three fields.

```
public class ExampleStruct {
  public Object Field1 = new String( “default1” );
  public Object Field2 = null;
  public Object Field3 = new String( “default3” );

  public static void main( String[] argv ) {
    ExampleStruct exampleInstance1 = new ExampleStruct();
    exampleInstance1.Field1 = “assigned value 1”;
    exampleInstance1.Field2 = “assigned value 2”;

    ExampleStruct exampleInstance2 = new ExampleStruct();
    exampleInstance2.Field1 = “tom”;
    exampleInstance2.Field2 = “dick”;
    exampleInstance2.Field3 = “harry”;
  }
}
```

Figure 3.3.4.4: A example of the same Instances written in the Java language.

Clearly, modifying the fields in *exampleInstance1* affects neither the default values assigned within the *ExampleStruct* class, nor the programmatically assigned values of *exampleInstance2*.

3.4 Clauses

The XESS schema defines an abstract clause type that is the parent of all of the supported clauses. The *clause* entity serves essentially as a marking interface for all other clauses within an XESS document and is used to indicate appropriate locations for clauses within other XESS entities (such as rules or other clauses). A clause generally defines an operation that evaluates to either true of false based on some input. This section of the document briefly describes the parent *clause* entity defined in the schema; the remaining subsections of this chapter define concrete extensions of the abstract clause entity. Figure 3.4.1 shows the excerpt of the XESS schema that contains the abstract *clause* entity definition.

```
<xsd:element name="clause" type="clauseType"/>

<xsd:complexType name="clauseType" abstract="true"/>
```

Figure 3.4.1: The XESS schema definition for the abstract clause entity.

3.4.1 The Greater Than Clause

The *greater than* clause defined in the XESS schema provides the syntax for a clause that compares two values and evaluates to true if and only if the first value is explicitly greater than the second value.

```
<xsd:element name="greaterThan" type="greaterThanType"
   substitutionGroup="clause"/>
```

```
<xsd:complexType name="greaterThanType">
  <xsd:complexContent>
    <xsd:extension base="clauseType">
      <xsd:attribute name="value1" type="xsd:string"/>
      <xsd:attribute name="value2" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.4.1.1: The XESS schema definition for the Greater Than clause.

The *greater than* clause uses the tag name *greaterThan* and contains two attributes; the *value1*attribute is used to specify the first value to be used in the comparison while *value2* is used to specify the second value. Each attribute may be used to refer to a scoped variable by name, or a literal value. Like all other attributes in the XESS schema the values are represented as strings in the XML, but may be interpreted as other value types (such as integers) at runtime.

Figure 3.4.1.2 is an example of a *greater than* clause that compares two integer values and should evaluate to false; the first value, 10, is clearly not greater than the second value, 100. This example also clearly demonstrates that primitive types such as integers are represented as string attributes in XML.

```
<greaterThan value1=”10” value2=”100”/>
```

Figure 3.4.1.2: An example of a Greater Than clause expressed in XML that compares two integer values and evaluates to false.

Figure 3.4.1.3 is an example of a *greater than* clause that compares two string values. The result of string comparison is largely dependent on the implementation of the underlying system, but in many cases strings are compared lexographically. Such a comparison assigns a numeric value to each character in the string, often the ASCII or Unicode value of the character, and compares those numeric values character by character until an inequality is found. Whichever character in the inequality has the larger value represents the lexographically larger string. In this example, the string “def” would be the larger string as the ASCII value of the character “d” is numerically greater than the ASCII value of the character “a”. Therefore the example clause could be expected to evaluate to true.

```
<greaterThan value1=”def” value2=”abc”/>
```

Figure 3.4.1.3: An example of a Greater Than clause expressed in XML that compares two string values and evaluates to true.

Finally, Figure 3.4.1.4 is an example of a *greater than* clause that evaluates to false because *value1* is equivalent, but not greater than, *value2*. This fails to satisfy the conditions of the clause, which demands that *value1* is explicitly greater than *value2.*

```
<greaterThan value1=”25” value2=”25”/>
```
Figure 3.4.14: An example of a Greater Than clause expressed in XML that compares two equivalent integer values and evaluates to false.

3.4.2 The Greater Than or Equal Clause

The *greater than or equal* clause defined in the XESS schema provides the syntax for a clause that compares two values and evaluates to true if and only if the first value is greater than or equivalent to the second value.

```
<xsd:element name="greaterThanOrEqual"
   type="greaterThanOrEqualType"
   substitutionGroup="clause"/>

<xsd:complexType name="greaterThanOrEqualType">
  <xsd:complexContent>
    <xsd:extension base="clauseType">
      <xsd:attribute name="value1" type="xsd:string"/>
      <xsd:attribute name="value2" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Figure 3.4.2.1: The XESS schema definition for the Greater Than Or Equal clause.

The *greater than or equal* clause uses the tag name *greaterThanOrEqual* and contains two attributes; the *value1*attribute is used to specify the first value to be used in the comparison while *value2* is used to specify the second value. Each attribute may be used to refer to a scoped variable by name, or a literal value. Like all other attributes in the XESS schema the values are represented as strings in the XML, but may be interpreted as other value types (such as integers) at runtime.

Figure 3.4.2.2 is an example of a *greater than or equal* clause that compares two integer values and should evaluate to true; the first value, 100, is clearly greater than the second value, 90, and thus satisfies the clause.

```
<greaterThanOrEqual value1=”100” value2=”90”/>
```
Figure 3.4.2.2: An example of a Greater Than Or Equal clause expressed in XML that compares two integer values and evaluates to true.

Figure 3.4.2.3 is an example of a *greater than or equal* clause that compares two string values, similar to the example in the previous section. In this example the strings are lexographically equivalent, which satisfies the condition of the *greater than or equal* clause that checks for equality between the two values.

```
<greaterThanOrEqual value1=”abc” value2=”abc”/>
```
Figure 3.4.2.3: An example of a Greater Than Or Equal clause expressed in XML that compares two string values and evaluates to true.

Finally, Figure 3.4.2.4 is an example of a *greater than or equal* clause that evaluates to false. The first integer value is clearly less than the second integer value, and therefore fails to satisfy the conditions of the clause.

```
<greaterThanOrEqual value1=”25” value2=”50”/>
```

Figure 3.4.2.4: An example of a Greater Than Or Equal clause expressed in XML that compares two integer values and evaluates to false.

3.4.3 The Less Than Clause

The *less than* clause defined in the XESS schema provides the syntax for a clause that compares two values and evaluates to true if and only if the first value is explicitly less than the second value.

```
<xsd:element name="lessThan" type="lessThanType"
   substitutionGroup="clause"/>

<xsd:complexType name="lessThanType">
  <xsd:complexContent>
    <xsd:extension base="clauseType">
      <xsd:attribute name="value1" type="xsd:string"/>
      <xsd:attribute name="value2" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.4.3.1: The XESS schema definition for the Less Than clause.

The *less than* clause uses the tag name *lessThan* and contains two attributes; the *value1*attribute is used to specify the first value to be used in the comparison while *value2* is used to specify the second value. Each attribute may be used to refer to a scoped variable by name, or a literal value. Like all other attributes in the XESS schema the values are represented as strings in the XML, but may be interpreted as other value types (such as integers) at runtime.

Figure 3.4.3.2 is an example of a *less than* clause that compares two integer values and should evaluate to false; the first value, 50, is clearly greater than the second value, 40, and thus fails to satisfy the clause.

```
<lessThan value1=”50” value2=”40”/>
```

Figure 3.4.3.2: An example of a Less Than clause expressed in XML that compares two integer values and evaluates to false.

Figure 3.4.3.3 is an example of a *less than* clause that compares two string values, similar to the examples in the previous sections. In this example the strings are lexographically equivalent, which fails to satisfy the condition that the first value be explicitly less than the second value, and therefore the clause should evaluate to false.

```
<lessThan value1=”def” value2=”def”/>
```
Figure 3.4.3.3: An example of a Less Than clause expressed in XML that compares two string values and evaluates to false.

Finally, Figure 3.4.3.4 is an example of a *less than* clause that evaluates to true. The first integer value, 25, is clearly less than the second integer value, 50, and therefore satisfies the condition of the clause.

```
<lessThan value1=”25” value2=”50”/>
```
Figure 3.4.3.4: An example of a Less Than clause expressed in XML that compares two integer values and evaluates to true.

3.4.4 The Less Than or Equal Clause

The *less than or equal* clause defined in the XESS schema provides the syntax for a clause that compares two values and evaluates to true if and only if the first value is less than or equivalent to the second value.

```
<xsd:element name="lessThanOrEqual"
   type="lessThanOrEqualType"
   substitutionGroup="clause"/>

<xsd:complexType name="lessThanOrEqualType">
  <xsd:complexContent>
    <xsd:extension base="clauseType">
      <xsd:attribute name="value1" type="xsd:string"/>
      <xsd:attribute name="value2" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Figure 3.4.4.1: The XESS schema definition for the Less Than Or Equal clause.

The *less than or equal* clause uses the tag name *lessThanOrEqual* and contains two attributes; the *value1*attribute is used to specify the first value to be used in the comparison while *value2* is used to specify the second value. Each attribute may be used to refer to a scoped variable by name, or a literal value. Like all other attributes in the XESS schema the values are represented as strings in the XML, but may be interpreted as other value types (such as integers) at runtime.

Figure 3.4.4.2 is an example of a *less than or equal* clause that compares two integer values and should evaluate to true; the first value, 100, is clearly less than the second value, 200, and thus satisfies the clause.

```
<lessThanOrEqual value1=”100” value2=”200”/>
```
Figure 3.4.4.2: An example of a Less Than Or Equal clause expressed in XML that compares two integer values and evaluates to true.

Figure 3.4.4.3 is an example of a *less than or equal* clause that compares two string values, similar to the examples in the previous sections. In this example the strings are lexographically equivalent, which satisfies the

condition of the *less than or equal* clause that checks for equality between the two values.

```
<lessThanOrEqual value1=″abc″ value2=″abc″/>
```

Figure 3.4.4.3: An example of a Less Than Or Equal clause expressed in XML that compares two string values and evaluates to true.

Finally, Figure 3.4.4.4 is an example of a *less than or equal* clause that evaluates to false. The first integer value is clearly greater than the second integer value, and therefore fails to satisfy the conditions of the clause.

```
<lessThanOrEqual value1=″125″ value2=″75″/>
```

Figure 3.4.4.4: An example of a Less Than Or Equal clause expressed in XML that compares two integer values and evaluates to false.

3.4.5 The Equal Clause

The *equal* clause defined in the XESS schema provides the syntax for a clause that compares two values and evaluates to true if and only if the first value is exactly equivalent to the second value.

```
<xsd:element name="equal" type="equalType"
   substitutionGroup="clause"/>

<xsd:complexType name="equalType">
  <xsd:complexContent>
    <xsd:extension base="clauseType">
      <xsd:attribute name="value1" type="xsd:string"/>
      <xsd:attribute name="value2" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.4.5.1: The XESS schema definition for the Equal clause.

The *equal* clause uses the tag name *equal* and contains two attributes; the *value1*attribute is used to specify the first value to be used in the comparison while *value2* is used to specify the second value. Each attribute may be used to refer to a scoped variable by name, or a literal value. Like all other attributes in the XESS schema the values are represented as strings in the XML, but may be interpreted as other value types (such as integers) at runtime.

Figure 3.4.5.2 is an example of an *equal* clause that compares two integer values and should evaluate to false; the first value, 30, is clearly not equivalent to the second value, 31, and thus fails to satisfy the clause.

```
<equal value1=″30″ value2=″31″/>
```

Figure 3.4.5.2: An example of an Equal clause expressed in XML that compares two integer values and evaluates to false.

Finally, Figure 3.4.5.3 is an example of an *equal* clause that compares two string values, similar to the examples in the previous sections. In this example the strings are lexographically equivalent, which satisfies the condition that the first value be exactly equivalent to second value, and therefore the clause should evaluate to true.

```
<lessThan value1=”def” value2=”def”/>
```

Figure 3.4.5.3: An example of an Equal clause expressed in XML that compares two string values and evaluates to true.

3.4.6 The Not Equal Clause

The *not equal* clause defined in the XESS schema provides the syntax for a clause that compares two values and evaluates to true if and only if the first value is not equivalent to the second value.

```
<xsd:element name="notEqual" type="notEqualType"
   substitutionGroup="clause"/>

<xsd:complexType name="notEqualType">
  <xsd:complexContent>
    <xsd:extension base="clauseType">
      <xsd:attribute name="value1" type="xsd:string"/>
      <xsd:attribute name="value2" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.4.6.1: The XESS schema definition for the Not Equal clause.

The *not equal* clause uses the tag name *notEqual* and contains two attributes; the *value1*attribute is used to specify the first value to be used in the comparison while *value2* is used to specify the second value. Each attribute may be used to refer to a scoped variable by name, or a literal value. Like all other attributes in the XESS schema the values are represented as strings in the XML, but may be interpreted as other value types (such as integers) at runtime.

Figure 3.4.6.2 is an example of a *notEqual* clause that compares two integer values and should evaluate to true; the first value, 50, is clearly not equivalent to the second value, 49, and thus satisfies the clause.

```
<notEqual value1=”50” value2=”49”/>
```

Figure 3.4.6.2: An example of a Not Equal clause expressed in XML that compares two integer values and evaluates to true.

Finally, Figure 3.4.6.3 is an example of a *notEqual* clause that compares two string values, similar to the examples in the previous sections. In this example the strings are lexographically equivalent, which fails to satisfy the condition that the first value cannot be exactly equivalent to second value, and therefore the clause should evaluate to false.

```
<notEqual value1=”efg” value2=”efg”/>
```
Figure 3.4.6.3: An example of a Not Equal clause expressed in XML that compares two string values and evaluates to false.

3.4.7 The Between Clause

The *between* clause in the XESS schema defines the syntax for a clause that compares an input value to minimum and maximum boundary values, and evaluates to true if the input value is greater than or equal to the minimum *and* less than or equal to the maximum.

```
<xsd:element name="between" type="betweenType"
  substitutionGroup="clause"/>

<xsd:complexType name="betweenType">
  <xsd:complexContent>
    <xsd:extension base="clauseType">
      <xsd:attribute name="value" type="xsd:string"/>
      <xsd:attribute name="min" type="xsd:string"/>
      <xsd:attribute name="max" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Figure 3.4.7.1: The XESS schema definition for the Between clause.

The *between* clause uses the tag name *between*, and contains three attributes: the *value* attribute specifies the input value; the *min* attribute specifies the minimum boundary value; and the *max* attribute specifies the maximum boundary value. Each attribute may be used to refer to a scoped variable by name, or a literal value. Like all other attributes in the XESS schema the values are represented as strings in the XML, but may be interpreted as other value types (such as integers) at runtime.

Figure 3.4.7.2 is an example of a *between* clause that compares an integer input value to an integer minimum and an integer maximum and should evaluate to true; the first input value, 36, is clearly between the minimum value, 30, and the maximum value, 40.

```
<between value=”36” min=”30” max=”40/>
```
Figure 3.4.7.2: An example of a Between clause that compares an integer input value to integer minimum and maximum boundaries and evaluates to true.

Figure 3.4.7.3 is an example of a *between* clause that compares a string value to string minimum and maximum boundary values and evaluates to false. A lexographic comparison of all three strings places the input string outside the maximum boundary and therefore fails to satisfy the maximum condition of the clause.

```
<between value=”zzz” min=”aaa” max=”yyy”/>
```

Figure 3.4.7.3: An example of a Between clause that compares a string input value to string minimum and maximum boundaries and evaluates to false.

Finally, Figure 3.4.7.4 is an example of a *between* clause that compares an integer input value to integer minimum and maximum boundary values and evaluates to true. In this case the input value, 35, is equivalent to the minimum value and less than the maximum value of 55. This still satisfies the conditions that the input value be greater than *or equal to* the minimum boundary and less than or equal to the maximum boundary.

```
<between value=″35″ min=″35″ max=″55″/>
```

Figure 3.4.7.4: An example of a Between clause that compares an integer input value to integer minimum and maximum boundaries and evaluates to true.

3.4.8 The Not Between Clause

The *not between* clause in the XESS schema defines the syntax for a clause that compares an input value to minimum and maximum boundary values, and evaluates to true if and only if the input value is explicitly less than the minimum *or* explicitly greater than the maximum.

```
<xsd:element name="notBetween" type="betweenType"
   substitutionGroup="clause"/>

<xsd:complexType name="notBetweenType">
  <xsd:complexContent>
    <xsd:extension base="clauseType">
      <xsd:attribute name="value" type="xsd:string"/>
      <xsd:attribute name="min" type="xsd:string"/>
      <xsd:attribute name="max" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.4.8.1: The XESS schema definition for the Not Between clause.

The *not between* clause uses the tag name *notBetween*, and contains three attributes: the *value* attribute specifies the input value; the *min* attribute specifies the minimum boundary value; and the *max* attribute specifies the maximum boundary value. Each attribute may be used to refer to a scoped variable by name, or a literal value. Like all other attributes in the XESS schema the values are represented as strings in the XML, but may be interpreted as other value types (such as integers) at runtime.

Figure 3.4.8.2 is an example of a *notBetween* clause that compares an integer input value to an integer minimum and an integer maximum and should evaluate to false; the first input value, 47, is clearly between the minimum value, 40, and the maximum value, 60.

```
<notBetween value=″36″ min=″30″ max=″40/>
```

Figure 3.4.8.2: An example of a Not Between clause that compares an integer input value to integer minimum and maximum boundaries and evaluates to false.

Figure 3.4.8.3 is an example of a *not between* clause that compares a string value to string minimum and maximum boundary values and evaluates to true. A lexographic comparison of all three strings places the input string outside the minimum boundary and therefore satisfies the condition of the clause that allows the input value to be less than or equal to the minimum boundary.

```
<notBetween value="fff" min="bbb" max="aaa"/>
```

Figure 3.4.8.3: An example of a Not Between clause that compares a string input value to string minimum and maximum boundaries and evaluates to true.

Finally, Figure 3.4.8.4 is an example of a *not between* clause that compares an integer input value to integer minimum and maximum boundary values and evaluates to false. In this case the input value, 75, is equivalent to the maximum value. This does not satisfy the conditions that the input value is explicitly *greater than* maximum boundary or explicitly less than maximum boundary.

```
<notBetween value="75" min="45" max="75"/>
```

Figure 3.4.8.4: An example of a Not Between clause that compares an integer input value to integer minimum and maximum boundaries and evaluates to false.

3.4.9 The And Clause

The *and* clause in the XESS schema defines the syntax for a clause that contains two or more sub-clauses and evaluates to true if and only if *all* of the sub-clauses evaluates to true.

```
<xsd:element name="and" type="andType"
   substitutionGroup="clause"/>

<xsd:complexType name="andType">
  <xsd:complexContent>
    <xsd:extension base="clauseType">
      <xsd:sequence>
        <xsd:element ref="clause" minOccurs="2"
           maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.4.9.1: The XESS schema definition for the And clause.

The *and* clause uses the tag name *and* and contains no attributes. Instead, the *and* clause is the parent entity for two or more sub-clauses; the clauses may be of any valid clause type, including the *and* and *or* clauses meaning that an *and* clause may potentially become very deeply nested, though this harms the readability of an XESS document; XML can quickly become deeply nested and inscrutable. One of the goals of the XESS language is

to maintain human readability so that the expert system can be understood quickly and easily by both authors and readers.

Figure 3.4.9.2 is an example of an *and* clause that contains two sub-clauses and evaluates to true. The first clause, a *greater than or equal*, evaluates to true as the first value, 65, is clearly greater than the second value, 45. The second clause, a *less than or equal*, evaluates to true as the first value, 65, is clearly less than the second value, 75.

```
<and>
  <greaterThanOrEqual value1="65" value2="45"/>
  <lessThanOrEqual value1="65" value2="75"/>
</and>
```

Figure 3.4.9.2: An example of an And clause that evaluates two sub-clauses and evaluates to true.

Note that the above example could be rewritten as shown in Figure 3.4.9.3; this is just one example of simplifying an XESS document to improve readability and to prevent unnecessarily deeply nested XESS documents.

```
<between value="65" min="45" max="75"/>
```

Figure 3.4.9.3: An example of a Between clause used to simplify the example And clause presented in Figure 3.4.9.2.

Figure 3.4.9.4 is an example of an *and* clause that contains three sub-clauses and evaluates to true. The first clause, a *between* evaluates to true as the input value, 33, is clearly between the min of 30 and the max of 40. The second clause, a *greater than*, also evaluates to true as the first value of 22 is clearly greater than the second value of 21. The third clause, an *equal*, evaluates to false because the first value, "abc", is clearly not equivalent to the second value, "def".

```
<and>
  <between value="33" min="30" max="40"/>
  <greaterThan value1="22" value2="21"/>
  <equal value1="abc" value2="def"/>
</and>
```

Figure 3.4.9.4: An example of an And clause that evaluates three sub-clauses and evaluates to true.

Because an *and* clause requires that *each* sub-clause evaluates to true, this example evaluates to false; even though the first two clauses hold, the third and final clause does not.

Finally, Figure 3.4.9.5 is an example of a deeply nested *and* clause that evaluates to false because one of the most deeply nested clauses evaluates to false. This example is very difficult to read, and undermines the

reader's ability to determine the intent of the clause. Such clauses should be refactored whenever possible.

```
<and>
  <lessThan value1=”12” value2=”15”/>
  <and>
    <and>
      <equal value1=”22” value2=”22”/>
      <between value=”10” min=”9” max=”11”/>
      <and>
        <greaterThan value1=”12” value2=”10”/>
        <and>
          <greaterThan value1=”22” value2=”20”/>
          <equal value1=”abc” value2=”def”/>
        </and>
        <and>
          <notBetween value=”12” min=”10” max=”15”/>
          <lessThan value1=”22” value2=”32”/>
        </and>
      </and>
      <greaterThanOrEqual value1=”10” value2=”10”/>
    </and>
    <between value=5” min=”0” max=”10”/>
</and>
```

Figure 3.4.9.5: An example of a deeply-nested And clause that evaluates to false.

3.4.10 The Or Clause

The *or* clause in the XESS schema defines the syntax for a clause that contains two or more sub-clauses and evaluates to true if and only if *at least* one of the sub-clauses evaluates to true.

```
<xsd:element name="or" type="orType"
   substitutionGroup="clause"/>

<xsd:complexType name="andType">
  <xsd:complexContent>
    <xsd:extension base="clauseType">
      <xsd:sequence>
        <xsd:element ref="clause" minOccurs="2"
           maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.4.10.1: The XESS schema definition for the Or clause.

The *or* clause uses the tag name *or* and contains no attributes. Instead, the *or* clause is the parent entity for two or more sub-clauses; the clauses may be of any valid clause type, including the *and* and *or* clauses meaning that an *or* clause may potentially become very deeply nested in the same ways described in the previous section.

Figure 3.4.10.2 is an example of an *or* clause that contains two sub-clauses and evaluates to true. The first clause, a *between*, evaluates to true as the first value, 55, is clearly between the minimum of 54 and the maximum of 65. The second clause, a *not equal* evaluates to true as the first value, "zzz", is clearly not equal to the second value, "aaa". Because each of the sub-clauses evaluates to true, clearly the *or* evaluates to true as well.

```
<or>
  <between value="55" min="45" max="65"/>
  <notEqual value1="zzz" value2="aaa"/>
</or>
```

Figure 3.4.10.2: An example of an Or clause that evaluates two sub-clauses and evaluates to true.

The next example, shown in Figure 3.4.10.3, is an or clause that also evaluates to true despite the fact that one of the two sub-clauses evaluates to false. The first clause, a *greater than*, evaluates to true as the first value, 33, is clearly greater than the second value, 30. The second clause, a *not between*, evaluates to false as the input value, 22, is between the minimum of 20 and the maximum of 30.

```
<or>
  <greaterThan value1="33" value2="30"/>
  <notBetween value="22" min="20" max="30"/>
</or>
```

Figure 3.4.10.3: An example of an Or clause that evaluates two sub-clauses and evaluates to true.

Finally, Figure 3.4.10.4 is an example of an *or* clause that contains three sub-clauses and evaluates to false. The first clause, a *less than*, evaluates to false because the first value, 12, is equal to the second value. The second clause, a *greater than*, evaluates to false because the first value, 14, is less than the second value, 15. The third and final clause, an *equal*, evaluates to false because the first value, 10, is not equal to the second value, 11.

```
<or>
  <lessThan value1="12" value2="12"/>
  <greaterThan value1="14" value2="15"/>
  <equal value1="10" value2="11"/>
</or>
```

Figure 3.4.10.4: An example of an Or clause that evaluates three sub-clauses and evaluates to true.

Because the *or* clause requires *at least* one of its sub-clauses to evaluate to true this final evaluation of this example is false.

3.4.11 The Absence of a Not Clause

The XESS schema definition does not contain an explicit *not* clause. While it would be possible to include a *not* tag such as that shown in Figure 3.4.11.1 this would only lead to more deeply nested (and therefore more difficult to read) XESS documents, and make the intent of the clauses more difficult to discern.

```
<not>
  <and>
    <lessThanOrEqual value1="8" value2="10"/>
    <equal value1="5" value2="4"/>
    <lessThan value1="15" value2="20"/>
  </and>
</not>
```

Figure 3.4.11.1: An example of what a Not clause might look like.

Each of the standard clauses in the XESS language has a converse clause that can be used for negation. In some cases, such as the *equal* and *between* clauses the negating clauses are explicit. In other cases, such as the *and* and *or* clauses basic theorems of boolean logic must be applied during the negation. In the majority of cases, however, negation is achieved simply by passing the same arguments to the converse clause. Figure 3.4.11.2 is an example of two statements that are boolean opposites; in each case the same arguments are applied to the clauses but the opposite boolean results are achieved.

```
<lessThan value1="X" value2="Y"/>

<greaterThanOrEqual value1="X" value2="Y"/>
```

Figure 3.4.11.2: An example of negation through the use of converse clauses.

Whenever the *less than* clause evaluates to true, the *greater than or equal* clause must evaluate to false; the value for X cannot be *less than* Y while simultaneously being *greater than or equal* to Y. Conversely, whenever the *greater than or equal* clause evaluates to true, the *less than* clause must evaluate to false. Table 4.4.11.1 lists each standard clause and the converse clause required to negate it. Only in the case of the *and* and *or* clauses must DeMorgan's Law also be applied to achieve the opposite boolean result.

Clause	**Converse Clause**
Equal	Not Equal
Not Equal	Equal
Less Than	Greater Than Or Equal
Less Than Or Equal	Greater Than
Greater Than	Less Than or Equal
Greater Than Or Equal	Less Than

Between	Not Between
Not Between	Between
And	DeMorgan's Law: '(A*B) = 'A+'B
Or	DeMorgan's Law: '(A+B) = 'A*'B

Table 3.4.11.1: The list of supported clauses and the converse clauses used for negation.

In the case of the *and* clause negation is achieved by negating each of the two or more sub-clauses and changing the *and* to an *or* (this is according to DeMorgan's Law). Figure 3.4.11.3 is an example of the negation of an *and* clause when the example is read top-to-bottom.

```
<and>
  <lessThanOrEqual value1="X" value2="Y"/>
  <between value="A" min="B" max="C"/>
  <greaterThanOrEqual value1="P" value2="Q"/>
</and>

<or>
  <greaterThan value1="X" value2="Y"/>
  <notBetween value="A" min="B" max="C"/>
  <lessThan value1="P" value2="Q"/>
</or>
```

Figure 3.4.11.3: An example of negating the And clause.

In the above example each sub-clause is first negated using the converse clauses in Table 3.4.11.1: the *less than or equal* sub-clause is negated by converting it to a *greater than* clause; the *between* clause is negated by converting it to a *not between* clause; and finally, the *greater than or equal* clause is negated by converting it into a *less than* clause. The arguments to each clause remain unchanged. Once each of the sub-clauses is negated, the enclosing *and* clause is converted to an *or* clause, thus achieving the boolean opposite of the original clause.

Similarly, the *or* clause is negated by first negating each of the two or more sub-clauses and then converting the enclosing *or* clause to an *and*. If this logic is applied to the final *or* statement in Figure 3.4.11.3 the original *and* statement results. First each of the sub-clauses is negated using the converse clauses in Table 3.4.11.1: the *greater than* is negated by converting it into a *less than or equal*; the *not between* is negated by converting it into a *between*; and finally the *less than* is negated by converting it into a *greater than or equal*. Finally the *or* clause is converted to an *and* clause, thus achieving the opposite boolean result.

3.5 Actions

The XESS schema defines an abstract *action* type that is the parent of all of the supported actions. The *action* entity serves essentially as a marking interface for all other actions within an XESS document and is used to indicate appropriate locations for actions within other XESS entities (such

as the *then* or *else* parts of a *rule*). An *action* generally defines an operation that is taken based upon the execution of a rule. Every *rule* that is evaluated may result in the execution of zero or more *actions*; separate *actions* may be specified depending on whether the *rule* evaluates to true or false. This section of the document briefly describes the parent *action* entity defined in the schema; the remaining subsections of this chapter define concrete extensions of the abstract *action* entity. Figure 3.5.1 shows the excerpt of the XESS schema that contains the abstract *action* entity definition.

```
<xsd:element name="action" type="actionType"/>

<xsd:complexType name="actionType" abstract="true"/>
```

Figure 3.5.1: The XESS schema definition for the abstract action entity.

3.5.1 The Set Action

The *set* action in the XESS schema defines the syntax for an action that updates the current value of a fact in the system. The *set* action can be used to update the value of *predicates*, specific *fields* within an *instance*, or even the value of arguments passed to a *rule*.

```
<xsd:element name="set" type="setType"
   substitutionGroup="action"/>

<xsd:complexType name="setType">
  <xsd:complexContent>
    <xsd:extension base="actionType">
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="value" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.5.1.1: The XESS schema definition for the Set action.

The *set* action uses the tag name *set* and contains two attributes; *name* and *value*. The *name* attribute specifies the name of the fact to be modified; this may refer to a globally accessible fact, or a parameter name within the scope of the current *rule*. If a parameter name is used, the set action should dereference the parameter and modify the *fact* to which it points. The *value* attribute contains the new value for the *fact* to be updated. This value is specified as a string.

Figure 3.5.1.2 is an example of a *set* action that, if executed, updates a *predicate* with the name “myPredicate” with the value “newValue”. The value of the *predicate* is only updated if the action is executed, which in turn depends on the results of the evaluation of the *rule* enclosing the action (more on this in subsequent chapters).

```
<set name=”myPredicate” value=”newValue”/>
```

Figure 3.5.1.2: An example of a Set action that updates a predicate value.

The scope of the *predicate* being updated depends on the context of the *set* action. If one of the *parameters* of the *rule* in which the *set* action is contained is named “myPredicate”, the *predicate* to which the parameter points will be updated. Otherwise, “myPredicate” is assumed to be a globally accessible *predicate*. If no such *predicate* exists, an error will be generated; the *set* action will not create a new *predicate* and assign it an initial value.

Figure 3.5.1.3 is an example of a *set* action that, if executed, updates a specific *field* within an *instance.* Just like any other *set* action, the *name* specified may refer to a variable with local scope, or a globally accessible *instance*. The dot-notation is used to specify both the *instance* name and the *field* name.

```
<set name=”myInstance.myField” value=”newValue”/>
```

Figure 3.5.1.3: An example of a Set action that updates a Field within an Instance.

3.5.2 The Run Rule Action

The *run rule* action in the XESS schema defines the syntax for an action that executes another *rule* within the system. The *run rule* action can be used to chain *rules* together so that execution of one *rule* logically leads to the execution of another *rule* based whether or not the initial *rule* evaluates to true of false.

```
<xsd:element name="runRule" type="runRuleType"
   substitutionGroup="action"/>

<xsd:complexType name="runRuleType">
  <xsd:complexContent>
    <xsd:extension base="actionType">
      <xsd:sequence>
        <xsd:element name="argument" minOccurs="0"
           maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="value" type="xsd:string"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3.5.2.1: The XESS schema definition for the Run Rule action.

The *run rule* action uses the tag name *runRule* and contains a single attribute *name*. The *name* attribute is used to specify the unique name of the *rule* that is to be executed. Optionally, the *runRule* call may specify a set of *arguments* to the *rule*. The *arguments* are specified as child entity,

and each *argument* has a single attribute; the *value* attribute specifies the value of the argument and must conform to the rules governing *fact* value types (as explained earlier in this document). The order of the *arguments* must exactly match the order of the *parameters* specified in the *rule* definition.

Figure 3.5.2.2 is an example of a *run rule* action that executes a *rule* with the name "myRule". The *rule* has no parameters, and so the optional child *arguments* need not be specified in this case.

```
<runRule name=″myRule″/>
```

Figure 3.5.2.2: An example of a Run Rule action that executes a rule with no parameters.

Just as with all other *actions*, the *run rule* action is only executed based on the evaluation result of the *rule* that encloses the *action*. Figure 3.5.2.3 is an example of a more complex *run rule* that calls a *rule* that accepts multiple parameters.

```
<runRule name=″myOtherRule″>
  <argument value=″100″/>
  <argument value=″abcd″/>
  <argument value=″John Smith″/>
</runRule>
```

Figure 3.5.2.3: An example of a Run Rule action that executes a rule with three parameters.

In the above example the *run rule* action passes three *arguments* into a *rule* with the unique name *"myOtherRule"*. The number and order of the *arguments* must match the number and order of the *parameters* specified in the definition of the *rule* that is being invoked. As with most other values in an XESS system, type checking is handled later.

3.6 Rules

The *rule* entity in the XESS schema defines the syntax for *rules* within an XESS document. The *rule* entity is the most complex entity in the system as it ties all of the other elements together, including *facts*, *clauses*, and *actions* to define the behavior of the system during execution.

```
<xsd:element name="rule" minOccurs="0"
   maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="comment" minOccurs="0"
         maxOccurs="1"/>
      <xsd:element name="parameter" minOccurs="0"
         maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:attribute name="name" type="xsd:string"/>
          <xsd:attribute name="type" type="xsd:string"/>
        </xsd:complexType>
```

```
      </xsd:element>
      <xsd:element name="if">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="clause"/>
          </xsd:sequence>
         </xsd:complexType>
      </xsd:element>
      <xsd:element name="then">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="action"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="else" minOccurs="0" maxOccurs="1">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="action"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"
       use="optional"/>
  </xsd:complexType>
</xsd:element>
```

Figure 3.6.1: The XESS schema definition for the Run entity.

The *rule* entity uses the tag name *rule* and contains a single attribute, *name*. The *name* of the *rule* must be unique within the XESS document; if two rules specify the same name an XESS interpreter may generate an error, or simply overwrite the older *rule* with the newer *rule* causing unexpected results during execution.

The *rule* entity contains several child elements that can be separated into four main categories. The first is an optional set of *parameters*. Each *parameter* has a single attribute, *name*. The *name* is used to refer to the parameter within the scope of the rule. A *rule* may define zero or more *parameters*.

The next child entity is the *if*. The *if* encloses a single *clause* that is used to evaluate the *rule*. If a compound *clause*, such as an *and* or an *or* is used, the *rule* may test several conditions when executed.

Following the *if* child entity is the *then* entity. The *then* is used to enclose a set of zero or more *actions* to be taken if the *clause* specified in the *if* part of the *rule* evaluates to true. The *then* entity may specify any number of *actions*, but the *actions* are executed synchronously in the order in which they are specified.

Finally, an optional *else* child may be specified to enclose a separate list of zero or more *actions* that are executed in the event that the *clause* specified in the *if* part of the *rule* evaluates to false. Like the *then* part of the *rule*, the *actions* specified in the *else* part are executed synchronously in the order in which they are specified.

Figure 3.6.2 is an example of a basic *rule* that contains a single, non-compound *clause* and does not specify any *parameters* or an *else*. This is the simplest form of *rule*.

```
<rule name=”ExampleRule”>
  <if>
    <equals value=”1” value2=”1”/>
  </if>
  <then>
    <set name=”ExampleRuleResult” value=”true”/>
  </then>
</rule>
```

Figure 3.6.2: A basic Rule written in the XESS language.

The above example *rule* simple compares two hard-coded values for equality; the result of the comparison is obviously true, which causes the *action* in the *then* section to be executed. The solitary *set* specified in the *then* simply updates a global *predicate* with the name *ExampleRuleResult* to true. Figure 3.6.3 is an example of the same *rule* as it would look written in the Java programming language.

```
if( 1 == 1 ) {
  ExampleRuleResult = true;
}
```

Figure 3.6.3: The Java implementation of the Rule presented in Figure 3.6.2.

4 XESS Interpreter

4.1 The XESS API

The XESS abstract programming interface (API). Specification Version 1.0.

4.1.1 Summary

This section of the document details the interfaces and classes that should be implemented by an XESS interpreter or shell, regardless of the language used to implement the shell. While the definitions use object-oriented terminology that is common to most modern object-oriented languages, any programming examples required will use the Java language syntax.

Most of the interfaces, classes, fields, and methods defined in this section are object representation of the entities defined in the XESS XML Schema. When an XESS document is parsed by an interpreter, it is translated into implementations of the classes defined here. These implementations allow the XML entities to be examined, manipulated, modified, and executed without editing the original XML. These implementations can also be provided as input the plug-ins that will execute the facts and rules defined therein, and in many cases can be used to generate XML output that is a snapshot of the system before, during, or after it has been evaluated.

Because of variability between languages it may not be possible or desirable to completely comply with the specification, but the specification does provide programmers with details about the expected behavior of object-oriented language representations of the XESS entities; it is recommended that any concrete implementation stay as close to the specification as possible.

4.1.2 Interface XmlConstants

XmlConstants is an interface that provides its implementing classes with access to a complete set of useful constants for parsing or generating valid XESS documents, but does not directly represent any of the entities defined in the XESS XML Schema. The majority of classes and interfaces in the XESS API implement or extend *XmlConstants*. Many of the constants are useful for parsing or generating XESS documents.

4.1.2.1 static String AND = “and”

The tag used to create an *and* clause within the body of the **if** in a RULE; an *and* clause evaluates two or more sub-clauses; if every clause evaluates to true, the *and* evaluates to true, otherwise the *and* evaluates to false. For example:

```
<and>
  <equal value1="arg1.field1" value2="arg2.field1"/>
  <notEqual value1="arg1.field2" value2="arg2.field2"/>
</and>
```

Figure 4.1.2.1: A basic example of an AND tag in an XESS document.

4.1.2.2 static String ARGUMENT = "argument"

The tag used to specify an *argument* in a RUN_RULE action. See the RUN_RULE constant for more information.

4.1.2.3 static String BETWEEN = "between"

The tag used to create a *between* clause within the body of the **if** in a RULE; a *between* clause uses minimum and maximum limits to determine if a single value is greater than or equal to the minimum and less than or equal to the maximum. For example:

```
<between value="arg1.field2" min="100" max="200"/>
```

Figure 4.1.2.2: A basic example of a BETWEEN tag in an XESS document.

4.1.2.4 static String COMMENT = "comment"

The tag used wherever *comments* are allowed within an XESS document. For example:

```
<comment>This is a comment.</comment>
```

Figure 4.1.2.3: A basic example of a COMMENT tag in an XESS document.

4.1.2.5 static String DESCRIPTION = "description"

The attribute used wherever *descriptions* are allowed within an XESS document.

4.1.2.6 static String ELSE = "else"

The tag used to create the *else* within a RULE. See the RULE constant for more information.

4.1.2.7 static String EQUAL = "equal"

The tag used to create an *equal* clause within the body of the **if** in a RULE; an equal clause compares two values for equality. For example:

```
<equal value1="arg1.field1" value2="arg2.field1"/>
```
Figure 4.1.2.4: A basic example of an EQUAL tag in an XESS document.

4.1.2.8 static String FIELD = "field"

The tag used to specify a *field* within a STRUCT or an INSTANCE in an XESS document.

4.1.2.9 static String GREATER_THAN = "greaterThan"

The tag used to create a *greater-than* clause within the body of the **if** in a RULE; a *greater-than* clause compares two values to determine if the first is larger than the second. For example:

```
<greaterThan value1="arg1.field1" value2="arg2.field1"/>
```
Figure 4.1.2.X: A basic example of a GREATER THAN tag in an XESS document.

4.1.2.10 static String GREATER_THAN_OR_EQUAL = "greaterThanOrEqual"

The tag used to create a *grater-than-or-equal* clause within the body of the **if** in a RULE; a *greater-than-or-equal* clause compares two values to determine if the first is larger than or equal to the second. For example:

```
<greaterThanOrEqual value1="arg1.field1"
        value2="arg2.field1"/>
```
Figure 4.1.2.X: A basic example of a GREATER THAN OR EQUAL tag in an XESS document.

4.1.2.11 static String IF = "if"

The tag used to create the **if** within a RULE.

4.1.2.12 static String INITIAL_VALUE = "initialValue"

The attribute used to specify the *initial* (or default) *value* of a FIELD within a STRUCT. The *initial value* is inherited by the fields of any INSTANCE of the STRUCT.

4.1.2.13 static String INSTANCE = "instance"

The tag used to create an *instance* element within an XESS document; each *instance* must be associated with a STRUCT with the same basic structure. For example:

```
<instance name="MyInstance" type="ExampleStruct">
  <field name="field1" value="new value 1"/>
  <field name="field1" value="new value 2"/>
</instance>
```
Figure 4.1.2.X: A basic example of an INSTANCE tag in an XESS document.

4.1.2.14 static String LESS_THAN = “lessThan”

The tag used to create a *less-than* clause within the body of the **if** in a RULE; a *less-than* clause compares two values to determine if the first is less than the second. For example:

```
<lessThan value1="arg1.field1" value2="arg2.field1"/>
```

Figure 4.1.2.X: A basic example of a LESS THAN tag in an XESS document.

4.1.2.15 static String LESS_THAN_OR_EQUAL = “lessThanOrEqual”

The tag used to create a *less-than-or-equal* clause within the body of the **if** in a RULE; a *less-than-or-equal* clause compares two values to determine if the first is less than or equal to the second. For example:

```
<lessThanOrEqual value1="arg1.field1"
        value2="arg2.field1"/>
```

Figure 4.1.2.X: A basic example of a LESS THAN OR EQUAL tag in an XESS document.

4.1.2.16 static String MAX = “max”

The attribute used to indicate the maximum value of an element within an XESS document; for example, this may be used to indicate the upper of two values used in BETWEEN or NOT BETWEEN elements.

4.1.2.17 static String MIN = “min”

The attribute used to indicate the minimum value of an element within an XESS document; for example, this may be used to indicate the lower of two values used in BETWEEN or NOT BETWEEN elements.

4.1.2.19 static String NAME = “name”

The attribute used to specify the name of an element within an XESS document; for example, this may be used to specify the name of an XESS RULE.

4.1.2.20 static String NOT_BETWEEN = “notBetween”

The tag used to create a *not-between* clause within the body of the **if** in a RULE; a *not-between* clause uses minimum and maximum limits to determine if a single value is greater than the maximum or less than the minimum. For example:

```
<notBetween value="arg1.field2" min="100" max="200"/>
```

Figure 4.1.2.X: A basic example of a NOT BETWEEN tag in an XESS document.

4.1.2.21 static String NOT_EQUAL = “notEqual”

The tag used to create a *not-equal* clause within the body of the **if** in a RULE; a *not-equal* clause compares two values for inequality. For example:

```
<notEqual value1="arg1.field1" value2="arg2.field1"/>
```

Figure 4.1.2.X: A basic example of a NOT EQUAL tag in an XESS document.

4.1.2.22 static String OR = "or"

The tag used to create an *or* clause within the body of the **if** in a RULE; an *or* clause evaluates two or more sub-clauses; if at least one clause evaluates to true, the *or* evaluates to true, otherwise the *or* evaluates to false. For example:

```
<or>
  <equal value1="arg1.field1" value2="arg2.field1"/>
  <notEqual value1="arg1.field2" value2="arg2.field2"/>
</or>
```

Figure 4.1.2.2: A basic example of an OR tag in an XESS document.

4.1.2.23 static String PARAMETER = "parameter"

The tag used to define a *parameter* element within a RULE.

4.1.2.24 static String PREDICATE = "predicate"

The tag used to create a *predicate* element within an XESS document. A *predicate* is the simplest form of FACT in an XESS document. For example:

```
<predicate name="example-predicate" value="example-value"/>
```

Figure 4.1.2.X: A basic example of a PREDICATE tag in an XESS document.

4.1.2.25 static String RULE = "rule"

The tag used to define a rule within an XESS document. For example:

```
<rule name="example-rule">
  <parameter name="arg1" type="ExampleStruct"/>
  <parameter name="arg2" type="ExampleStruct"/>
  <if>
    <!-- body of the if goes here -->
  </if>
  <then>
    <!-- body of the then goes here -->
  </then>
  <else>
    <!-- body of the else goes here -->
  </else>
</rule>
```

Figure 4.1.2.X: A basic example of a RULE tag in an XESS document.

4.1.2.26 static String RUN_RULE = “runRule”

One of the possible actions taken in the **then** or **else** elements within a RULE. The *run rule* action is used to execute a rule within the system. For example:

```
<runRule name="example-rule">
  <argument name="arg1" value="arg1"/>
  <argument name="arg2" value="arg2"/>
</runRule>
```

Figure 4.1.2.X: A basic example of a RUN RULE tag in an XESS document.

4.1.2.27 static String SET = “set”

One of the possible actions taken in the **then** or **else** elements within a RULE. The *set* action is used to modify the value of a variable within the system, which may include predicates, instance fields, or parameters to the RULE from which the *set* action is invoked. For example:

```
<set name="arg1.field1" value="updated value 1"/>
```

Figure 4.1.2.X: A basic example of a SET tag in an XESS document.

4.1.2.28 static String SET_INSTANCE = “setInstance”

One of the possible actions taken in the **then** or **else** elements within a RULE. The *set instance* action is used to create or modify an instance. For example:

```
<setInstance type=”ExampleStruct”>
  <field name=”field1” value=”new-value1”/>
  <field name=”field2” value=”new-value2”/>
</setInstance>
```

Figure 4.1.2.X: A basic example of a SET INSTANCE tag in an XESS document.

4.1.2.29 static String SET_PREDICATE = “setPredicate”

One of the possible actions taken in the **then** or **else** elements within a RULE. The *set predicate* action is used to create or modify a predicate.

```
<setPredicate name=”example-predicate” value=”value”/>
```

Figure 4.1.2.X: A basic example of a SET PREDICATE tag in an XESS document.

4.1.2.30 static String STRUCT = “struct”

The tag used to create a *struct* element within an XESS document. For example:

```
<struct name="ExampleStruct">
```

```
  <field name="field1" initialValue="value1"/>
  <field name="field2" initialValue="value2"/>
</struct>
```

Figure 4.1.2.X: A basic example of a STRUCT tag in an XESS document.

4.1.2.31 static String THEN = “then”

The tag used to create the **then** within a RULE.

4.1.2.32 static String TYPE = “type”

The attribute used to indicate the type of an element within an XESS document; for example, this may be used to indicate the INSTANCE type of a parameter in a RULE.

4.1.2.33 static String VALUE = “value”

The attribute used to indicate the string value of an element within an XESS document; for example, this may be used to indicate the initial value of a FIELD within an XESS STRUCT.

4.1.2.34 static String VALUE1 = “value1”

The attribute used to indicate the first string value of an element within an XESS document that has multiple values; for example, this may be used to indicate the first of two values in a GREATER THAN element.

4.1.2.35 static String VALUE2 = “value2”

The attribute used to indicate the second string value of an element within an XESS document that has multiple values; for example, this may be used to indicate the second of two values in a GREATER THAN element.

4.1.2.36 static String XESS = “xess”

The top-level tag in an XESS document. For example:

```
<xess>
  <!-- the body of the document goes here -->
</xess>
```

Figure 4.1.2.X: A basic example of an XESS tag in an XESS document.

4.1.2.37 static String XML_VERSION = “<?xml version=\”1.0\”?>”

The XML version tag.

4.1.3 Interface XmlElement extends XmlConstants

The *XmlElement* provides a simple interface for objects that can transform themselves into XML-formatted strings. The *XmlElement* interface extends *XmlConstants* to provide any implementing classes with direct access to the pre-defined XESS tags and attributes, but does not directly represent any of the entities defined in the XESS XML Schema.

4.1.3.1 public String toXml()

This method returns the *XmlElement* as an XML-formatted string that is compliant with the XESS XML Schema. The specific format of the string is determined by the implementing classes.

Return – The *XmlElement* as an XML-formatted string.

4.1.4 Abstract Class Clause implements XmlElement

The abstract parent of all clauses; a *clause* is a conditional statement within a rule that evaluates to true or false. The *clause* class represents the *clause* entity defined in the XESS XML Schema.

4.1.4.1 public Clause()

Creates a new *Clause* with a null description.

4.1.4.2 public Clause(String description)

description – The description of the clause.

Creates a new *Clause* with the specified description.

4.1.4.3 public String getDescription()

Returns the description of the *Clause*. This value may be null as descriptions are optional.

Returns – A string describing the *Clause*.

4.1.4.4 public abstract String getName()

Returns the name of the *Clause*. This abstract method must be implemented by a child class, and should return the name of the *Clause* as it is defined in the XESS language specification. For example, an *and*

clause should return the string *"and"*, while a *not between* clause should return the string *"notBetween"*.

Returns – A the name of the *Clause* as it is defined in the XESS specification.

4.1.4.5 public void setDescription(String description)

description – A string describing the *Clause*.

Sets the description of the *Clause* to the specified value. This value may be null as descriptions are optional.

4.1.5 Abstract Class SimpleClause extends Clause

A *Simple Clause* represents a basic comparison between two values; whether or not the *Clause* is satisfied is based on the results of the comparison defined by a sub-class.

4.1.5.1 public String getValue1()

This method returns the first of the two values to be used in the comparison.

Returns – The first of two values.

4.1.5.2 public String getValue2()

This method returns the second of the two values to be used in the comparison.

Returns – The second of two values.

4.1.5.3 public void setValue1(String v)

v – The value to which *value1* should be set.

This method sets the first of the two values to be used in the comparison to the specified value *v*.

4.1.5.4 public void setValue2(String v)

v – The value to which *value2* should be set.

This method sets the second of the two values to be used in the comparison to the specified value *v*.

4.1.5.5 public String toXml()

This method provides a common implementation that returns sub-classes of the *Simple Clause* as an XML-formatted string. The method generates a string in the following format:

```
<{name} value1="{value 1}" value2="{value 2}"/>
```

Where the value for *{name}* is obtained by calling the *getName()* method on the sub-class, the value for *{value 1}* is obtained by invoking the *getValue1()* method, and the value for *{value 2}* is obtained by invoking the *getValue2()* method.

Returns – The *Simple Clause* as an XML-formatted string.

4.1.6 Abstract Class ClauseList extends Clause

A *Clause List* is a *Clause* that contains a collection of sub-clauses, some combination of which must be satisfied in order to satisfy the *Clause List*. Examples of a *Clause List* include the *And* and *Or* clauses.

4.1.6.1 public void addClause(Clause c)

c – The *Clause* to be added to the *Clause List*.

This method adds the specified *Clause* to the collection of sub-clauses that make up the *Clause List*.

4.1.6.2 public Clause[] getClauses()

This method returns the collection of sub-clauses that make up the *Clause List.*

Returns – The sub-clauses of the *Clause List* as a collection or an array.

4.1.6.3 public void removeClause(Clause c)

c – The *Clause* that should be removed from the *Clause List*.

This method removes the specified *Clause* from the collection of sub-clauses that make up the *Clause List*.

4.1.6.4 public String toXml()

This method provides a common implementation that returns sub-classes of the *Clause List* as an XML-formatted string. The method generates a string in the format:

```
<{name} description=\"{description}\">
     {clause list}
</{name}>
```

Where the value for *{name}* is obtained by calling the *getName()* method on the sub-class, the value for *{description}* is obtained by invoking the *getDescription()* method, and the value for *{clause list}* is obtained by iterating over the collection of sub-clauses and calling the *toXml()* method on each *Clause* in the collection.

Returns – The *Clause List* as an XML-formatted string.

4.1.7 Class Equal extends SimpleClause

An *Equal* is a *Clause* that is satisfied when the first value of the *Clause* is exactly equal to the second value of the *Clause*.

4.1.7.1 public Equal(String v1, String v2)

v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *Equal* with the specified values and a null description. The values of the *Equal* must be equal in order to satisfy the *Clause*.

4.1.7.2 public Equal(String d, String v1, String v2)

d – The description of the *Simple Clause*.
v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *Equal* with the specified values and description. The values of the *Equal* must be equal in order to satisfy the *Clause*.

4.1.7.3 public String getName()

This method returns the name of the *Clause*; in this case, the string "equal".

Returns – A string with the value “equal”.

4.1.8 Class GreaterThan extends SimpleClause

A *Greater Than* is a *Clause* that is satisfied when the first value of the *Simple Clause* is greater than the second value of the *Simple Clause*.

4.1.7.1 public GreaterThan(String v1, String v2)

v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *Equal* with the specified values and a null description. The values of the *Equal* must be equal in order to satisfy the *Clause*.

4.1.7.2 public GreaterThan(String d, String v1, String v2)

d – The description of the *Simple Clause*.
v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *Equal* with the specified values and description. The values of the *Equal* must be equal in order to satisfy the *Clause*.

4.1.8.1 public String getName()

This method returns the name of the *Clause*; in this case, the string “greaterThan”.

Returns – A string with the value “greaterThan”.

4.1.9 Class GreaterThanOrEqual extends SimpleClause

A *Greater Than Or Equal* is a *Clause* that is satisfied when the first value of the *Simple Clause* is greater than or equal to the second value of the *Simple Clause*.

4.1.7.1 public GreaterThanOrEqual(String v1, String v2)

v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *Equal* with the specified values and a null description. The values of the *Equal* must be equal in order to satisfy the *Clause*.

4.1.7.2 public GreaterThanOrEqual (String d, String v1, String v2)

d – The description of the *Simple Clause*.
v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *Equal* with the specified values and description. The values of the *Equal* must be equal in order to satisfy the *Clause*.

4.1.9.1 public String getName()

This method returns the name of the *Clause*; in this case, the string "greaterThanOrEqual".

Returns – A string with the value "greaterThanOrEqual".

4.1.10 Class LessThan extends SimpleClause

A *Less Than* is a *Clause* that is satisfied when the first value of the *Simple Clause* is less than the second value of the *Simple Clause*.

4.1.7.1 public LessThan(String v1, String v2)

v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *Equal* with the specified values and a null description. The values of the *Equal* must be equal in order to satisfy the *Clause*.

4.1.7.2 public LessThan (String d, String v1, String v2)

d – The description of the *Simple Clause*.
v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *Equal* with the specified values and description. The values of the *Equal* must be equal in order to satisfy the *Clause*.

4.1.10.1 public String getName()

This method returns the name of the *Clause*; in this case, the string "lessThan".

Returns – A string with the value "lessThan".

4.1.11 Class LessThanOrEqual extends SimpleClause

A *Less Than Or Equal* is a *Clause* that is satisfied when the first value of the *Simple Clause* is less than or equal to the second value of the *Simple Clause*.

4.1.11.1 public LessThanOrEqual(String v1, String v2)

v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *Equal* with the specified values and a null description. The values of the *Equal* must be equal in order to satisfy the *Clause*.

4.1.11.2 public LessThanOrEqual (String d, String v1, String v2)

d – The description of the *Simple Clause*.
v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *Equal* with the specified values and description. The values of the *Equal* must be equal in order to satisfy the *Clause*.

4.1.11.3 public String getName()

This method returns the name of the *Clause*; in this case, the string "lessThanOrEqual".

Returns – A string with the value "lessThanOrEqual".

4.1.12 Class NotEqual extends SimpleClause

A *Not Equal* negates an *Equal*; it is a *Clause* that is satisfied when the first value of the *Simple* Clause is not equal to the second value of the *Simple Clause*.

4.1.12.1 public NotEqual(String v1, String v2)

v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *NotEqual* with the specified values and a null description. The values of the *NotEqual* must be equal in order to satisfy the *Clause*.

4.1.12.2 public NotEqual (String d, String v1, String v2)

d – The description of the *Simple Clause*.
v1 – The first value of the *Simple Clause*.
v2 – The second value of the *Simple Clause*.

This constructor creates a new *Equal* with the specified values and description. The values of the *Equal* must be equal in order to satisfy the *Clause*.

4.1.12.3 public String getName()

This method returns the name of the *Clause*; in this case, the string "notEqual".

Returns – A string with the value "notEqual".

4.1.13 Class And extends ClauseList

An *And* is a *Clause* that is only satisfied if and only if every sub-clause is satisfied.

4.1.13.1 public And(Clause[] clauses)

clauses – The collection of sub-clauses that must be satisfied in order for the *And* to evaluate to true.

This constructor creates a new *And* with the specified set of sub-clauses, all of which must evaluate to true in order for the *And* to be satisfied.

4.1.13.2 public And(String desc, Clause[] clauses)

desc – A string describing the *And*.
clauses – The collection of sub-clauses that must be satisfied in order for the *And* to evaluate to true.

This constructor creates a new *And* with the specified description and set of sub-clauses, all of which must evaluate to true in order for the *And* to be satisfied.

4.1.13.3 public String getName()

This method returns the name of the *Clause*; in this case, the string “and”.

Returns – A string with the value “and”.

4.1.14 Class Or extends ClauseList

An *Or* is a *Clause* that is satisfied if one or more of its sub-clauses is satisfied.

4.1.13.1 public Or(Clause[] clauses)

clauses – The collection of sub-clauses, at least one of which must be satisfied in order for the *Or* to evaluate to true.

This constructor creates a new *Or* with the specified set of sub-clauses, at least one of which must evaluate to true in order for the *Or* to be satisfied.

4.1.13.2 public Or(String desc, Clause[] clauses)

desc – A string describing the *Or*.
clauses – The collection of sub-clauses, at least one of which must be satisfied in order for the *Or* to evaluate to true.

This constructor creates a new *Or* with the specified description and set of sub-clauses, at least one of which must evaluate to true in order for the *Or* to be satisfied.

4.1.14.1 public String getName()

This method returns the name of the *Clause*; in this case, the string “or”.

Returns – A string with the value “or”.

4.1.15 Class Between extends Clause

A *Between* is a *Clause* that is satisfied if the argument is greater than or equal to the *minimum* value and less than or equal to the *maximum* value.

4.1.16 Class NotBetween extends Between

A *NotBetween* is a *Clause* that negates a *Between*; it is only satisfied if the argument is less than the *minimum* value or greater than the *maximum* value.

4.1.17 Interface Action extends XmlElement

The *Action* interface provides a simple marking mechanism to identify objects as representations of one of the available *actions* defined in the XESS Schema. *Actions* are used in the *then* and *else* parts of a *rule* to determine what actions should be taken as a result of the successful execution of the *rule*. The concrete implementations of the *Action* interface provide more details about what these resulting *actions* may be.

The *Action* interface does not define any additional state or behavior beyond those defined in the parent interfaces.

4.1.18 Class RunRuleAction implements Action

Actions are used in the *then* and *else* parts of a *rule*. The *Run Rule Action* indicates that a *Rule* should be invoked as the result of the evaluation of another *Rule*. In some expert system implementations *rules* are automatically fired based on changes in the state of a system, and it is difficult or impossible to directly invoke a *rule*; in such cases the *Run Rule Action* may be ignored or otherwise omitted.

The name returned by the *Run Rule Action* determines the name of the *Rule* that should be invoked if the *action* is taken, and the arguments to the *Run Rule Action* specify the arguments that should be given as input to the specified *Rule* if and when it is invoked.

4.1.18.1 public RunRuleAction(String name)

This constructor creates a new *Run Rule Action* that, if taken, will attempt to invoke the *Rule* with the specified name.

name – The name of the *Rule* that should be invoked in the event that the *Run Rule Action* is taken.

4.1.18.2 public void setName(String name)

This method sets the name of the *Rule* that should be invoked in the event that the *Run Rule Action* is taken. This value must be the unique name of a *Rule* within the same XESS system as the *Run Rule Action*.

name – The name of the *Rule* that should be invoked in the event that the *Run Rule Action* is taken.

4.1.18.3 public String getName()

This method returns the name of the *Rule* that should be invoked in the event that the *Run Rule Action* is taken. This value must be the unique name of a *Rule* within the same XESS system as the *Run Rule Action*.

Returns – The name of the *Rule* that should be invoked in the event that the *Run Rule Action* is taken.

4.1.18.4 public void setArgument(String name, String value)

This method sets an argument with the specified *name* and *value* in the *Run Rule Action*. The argument will be passed to the *Rule* that is invoked by the action in the event that the action is taken. The name of the argument must correspond with one of the parameters defined by the specified *Rule*, and the value should conform with the XESS specification for values as discussed in section 4.3.1 of this document.

name – The name of the argument to be set on the *Run Rule Action*; this name must correspond with the name of one of the parameters on the *Rule* that is to be invoked.

value – The value of the argument; this must comply with the XESS specification for value types as discussed in section 4.3.1 of this document.

4.1.18.5 public String[] getArgumentNames()

This method returns a collection of zero or more strings, each representing the name of one of the arguments that has been set on the *Run Rule Action*. This collection can be used to iterate over the arguments in the action.

Returns – A collection of zero or more strings, each representing the name of one of the arguments that has been set on the *Run Rule Action*.

4.1.18.6 public String getArgument(String name)

This method returns the value of the argument with the specified name, if an argument with the specified name has been set on the *Run Rule Action*. The method otherwise returns null.

Return – The value of the argument with the specified name, or null if an argument with the specified name has not been set.

4.1.18.7 public String removeArgument(String name)

This method removes the argument with the specified name from the *Run Rule Action* if it exists, but has no effect if no such argument has been set.

name – The name of the argument that should be removed from the *Run Rule Action*.

4.1.18.8 public void clearArguments()

The method clears all of the arguments that have been set on the *Run Rule Action*.

4.1.18.9 public String toXml()

This method returns the *Run Rule Action* as an XML-formatted string compliant with the XESS schema definition for a *runRuleAction*; the string has the following format:

```
<runRuleAction name="{name}">
     {argument list}
</runRuleAction>
```

Where *{name}* is the name of the *Rule* returned by the *getName* method of the *Run Rule Action*. The *{argument list}* contains zero or more entries in the following format:

```
<argument name="{name}" value="{value}"/>
```

Where *{name}* is the name of one of the arguments returned by the *getArgumentNames()* method of the *Run Rule Action* class, and *{value}* is the value returned by the *getArgument()* method of the *Run Rule Action* when the *{name}* is specified as the input parameter.

4.1.19 Class SetAction implements Action

Actions are used in the *then* and *else* parts of a *Rule*. The *Set Action*, if taken as the result of the invocation of a corresponding *rule*, is used to create or modify the value of a fact, field, or parameter within the system.

The *name* used by the *Set Action* is used to determine the name of the fact, field, or parameter to be created or modified; this may refer to a *predicate*, a specific field within an *instance*, or the one of the arguments passed into the *rule* when it is invoked. If the name exactly matches one of the parameters to the *rule* in which the *Set Action* is contained, the entity that the argument to the rule represents will be modified; if the name exactly matches a *predicate* in the system, that *fact* will be modified; finally, if the name matches the pattern *{instance name}.{field name}* (an *instance* name and *field* name separated by a ".") the specified field within the specified instance will be modified.

Note that it is possible for the name used to match multiple entities within the system; in this case the normal rules of scope apply: first the parameters of the *rule* are evaluated, followed by the system-level *predicates* and *instances*. The value of the *Set Action* determines the new value of the fact or field, and must comply with the specification in section 4.3.1 of this document.

4.1.19.1 public SetAction(String name, String value)

This constructor creates a new *Set Action* that sets the *fact* or *field* with the specified name to the specified value. The name indicates which entity (a *fact*, field, or *parameter*) should be modified in the event that the *Set Action* is taken, and the value specified the *value* to which the entity should be set. The value must comply with the specification for fact values as indicated I section 4.3.1 of this document.

name – The name of the entity that should be modified if the *Set Action* is taken.

value – The value to which the entity should be set of the *Set Action* is taken.

4.1.19.2 public void setName(String name)

This method is used to indicate the name of the entity that should be created or modified if the *Set Action* is taken.

name – The name of the entity that should be created or modified in the event that the *Set Action* is taken.

4.1.19.3 public String getName()

This method is used to return the name of the entity that should be created or modified in the event that the *Set Action* is taken.

Returns – The name of the entity that should be created or modified in the event that the *Set Action* is taken.

4.1.19.4 public void setValue(String value)

This method sets the value used to create or modify the target of the *Set Action* in the event that the action is taken. The value must comply with the specification for XESS values as defined in section 4.3.1 of this document.

value – The value of the entity that should be created or modified in the event that the action is taken.

4.1.19.5 public String getValue()

This method returns the value used to create or modify the target of the *Set Action* in the event that the action is taken.

Returns – The value of the entity that should be created or modified in the event that the action is taken.

4.1.19.6 public String toXml()

This method returns the *Set Action* as an XML-formatted string compliant with the XESS schema definition for a *setAction*; the string has the following format:

```
<set name="{name}" value="{value}"/>
```

Where *{name}* is the name of the entity to be created or modified, and *{value}* is the value to which the entity should be set.

4.1.20 Class SetInstanceAction implements Action

Actions are used in the *then* and *else* parts of a *Rule*. The *Set Instance Action*, if taken as the result of the invocation of a corresponding *rule*, is used to create or modify the value of an *instance* within the system, allowing for multiple fields within the instance to be set at the same time. The *name* used by the *Set Instance Action* is used to determine the name of the *instance* to be created or modified; this may refer to a specific *instance*, or the one of the arguments passed into the *rule* when it is invoked.

Note that it is possible for the name used to match multiple entities within the system; in this case the normal rules of scope apply: first the parameters of the *rule* are evaluated, followed by the system-level

instances. The values of the *Set Instance Action* determine the new values of the *instance*, and must comply with the specification in section 4.3.1 of this document.

The *Set Instance Action* also contains a *type* field, which indicates the type of *instance* that should be created or modified. The specified type should match the name of one of the *structures* defined in the same XESS system.

Finally, each *Set Instance Action* also contains a collection of zero or more *fields* that are used to set or modify the corresponding *fields* within the *instance* with the specified name in the event that the action is taken.

4.1.20.1 public SetInstanceAction()

This constructor creates a new *Set Instance Action* with a null name, type, and an empty collection of *fields*.

4.1.20.2 public SetInstanceAction(String name, String type)

This constructor creates a new *Set Instance Action* with the specified *instance* name and type.

name – This is used to specify the name of the *instance* to be created or modified in the event that the *Set Instance Action* is taken. If an *instance* with the specified name does not already exist at the time that the action is taken, it will be created.

type – This is used to specify the type of *instance* to be created or modified; the type must correspond with one of the *structures* that have been defined in the same XESS system at the time that the action is taken.

4.1.20.3 public void addField(Field field)

Adds the specified *field* to the collection of *fields* in the *Set Instance Action*, or replaces the *field* with the same name if it already exists.

field – The *field* to be added to the *Set Instance Action*. Any *fields* added will be used to create or modify an *instance* in the event that the *Set Instance Action* is taken.

4.1.20.4 public Field getField(String name)

This method returns the *field* with the specified name if it exists in the collection of *fields* contained by the *Set Instance Action*.

name – The name of the desired *field*.

Returns – The *field* with the specified name, if it exists.

4.1.20.5 public void removeField(String name)

This method removes the *field* with the specified name from the *Set Instance Action*, if it exists. The specified *field* will be omitted from creation or modification in the event that the action is taken.

4.1.20.6 public Field[] getFields()

This method returns the collection of *fields* that have been added to the *Set Instance Action*. This collection may contain zero or more fields.

Returns – A collection of zero or more *fields* that have been added to the action.

4.1.20.7 public void clearFields()

This method clears all of the *fields* from the *Set Instance Action.*

4.1.20.8 public void setName(String name)

This method sets the name of the *instance* that should be created or modified in the event that the *Set Instance Action* is taken. If an *instance* with the specified name exists when the action is taken, it will be modified; if no such *instance* exists, a new one will be created with the specified name.

name – The name of the *instance* to be created or modified.

4.1.20.9 public String getName()

This method returns the name of the *instance* to be created or modified in the event that the *Set Instance Action* is taken.

Returns – The name of the *instance* to be created or modified.

4.1.20.10 public void setType(String type)

This method sets the type of *instance* that should be created or modified in the event that the *Set Instance Action* is taken. This *type* should correspond with the name of a *structure* that has been defined in the same XESS system at the time that the action is taken.

type – The type corresponding with a *structure* that has been defined in the system at the time that the action is taken.

4.1.20.11 public String getType()

This method returns the type of *instance* that should be created or modified in the event that the *Set Instance Action* is taken. This type should correspond with the name of a *structure* that has been defined in the same XESS system at the time that the action is taken.

Returns – The type of *instance* that should be created or modified in the event that the action is taken.

4.1.20.12 public String toXml()

This method returns the *Set Instance Action* as an XML-formatted string compliant with the XESS schema definition for a *setAction*; the string has the following format:

```
<setInstance name="{name}" type="{type}">
     {field list}
</setInstance>
```

Where *{name}* is the name of the *instance* to be created or modified, and *{type}* is the type of *instance* to be created or modified; the type should correspond with a *structure* that has been defined at the time that the action is taken.

The *{field list}* is a list of one or more *fields* in the following format:

```
<field name="{name}" type="{type}"
     value="{value}"/>
```

Where *{name}* is the name of the *field* as defined in the *structure*, the *{type}* is the *field* type, and the *{value}* is the value of the *field*; this value must comply with the specification for XESS fact values, as defined in section 4.3.1 of this document.

4.1.21 Class Parameter implements XmlElement

A *Parameter* is very similar to a *field* and represents the expected input to a *Rule*. Each *Rule* may contain zero or more *parameters*, and each *parameter* represents one argument that must be passed to the *Rule* in the event that it is invoked at run time.

4.1.21.1 public Parameter()

This constructor creates a new *Parameter* with no name or type.

4.1.21.2 public Parameter(String name, String type)

This constructor creates a new *Parameter* with the specified name and type. The name is the name of the *Parameter* within the scope of the enclosing *rule* and must be unique among the scope of the *parameters* within that *rule* but may share the same value as other entities in the system. The *type* may be used to specify the name of a *structure* defined in the same XESS system if the *Parameter* represents an *instance*, but may otherwise be null.

name – The name of the *Parameter*; this must be unique within the scope of *parameters* in the enclosing *rule*.

type – The *structure* type if the *parameter* represents an *instance*; otherwise this value may be null.

4.1.21.3 public void setName(String name)

This method sets the name of the *Parameter* to the specified value. The name should be unique within the scope of *parameters* enclosed by the same *rule* but may be shared with other entities in the system.

name – The new value for the name of the *Parameter*.

4.1.21.4 public String getName()

This method returns the current value for the name of the *Parameter*.

Returns – The current value for the name of the *Parameter*.

4.1.21.5 public String toXml()

This method returns the *Parameter* as an XML-formatted string compliant with the XESS schema definition for a *parameter*; the string has the following format:

```
<parameter name="{name}" type="{type}"/>
```

Where *{name}* is the name of the *parameter* within the scope of the *Rule*. The *{type}* may be used to specify an *instance* type in the event that the *parameter* is an *instance*; in this case the *type* should correspond with the name

of a *structure* that has been defined in the same XESS system, otherwise the entire *{type}* attribute may be omitted.

4.1.22 Class Rule implements XmlElement

An XESS system is composed of a knowledge base (which itself is composed of *facts*), and a set of *rules* that operate on those *facts*. The *Rule* class provides an object representation of the *rule* entity defined in the XESS XML Schema, and allows developers to interact with XESS *rules* programmatically.

```
public void exampleRule() {
  if( x = y ) {
    z = 4;
  }
  else {
    z = 2;
  }
}
```

Figure 4.1.21.1: A basic example of an if/then/else statement written in the Java programming language. XESS rules follow this same basic pattern.

Every *rule* essentially follows the same basic pattern as a Java if/then/else statement encapsulated within a uniquely named method, a simple example of which can be seen in Figure 4.1.21.1. In this example, the *rule* checks a condition, whether or not the *fact* with the name "x" has a value that is equal to the *fact* with the name "y", to determine whether or not it is true. In the event that the *facts* "x" and "y" are equal, then the *fact* "z" is set to a value of *4*; otherwise, the fact with the name "z" is set to a value of *2*.

In other words: *if* x is equal to y *then* set z to 4 *else* set z to 2.

Of course *rules* in XESS may be more complicated than that, just as Java methods may be more complicated. An XESS *rule* may specify *parameters* which must be passed to the rule for evaluation, and the *rule* may check multiple conditions, and take multiple actions whether it evaluates to true *or* false. Each *rule* must be uniquely named within the scope of all *rules* in the same XESS system, and *rules* may be invoked at any time, either automatically as each rule is systematically checked against the current knowledge base to determine whether or not it evaluates to true, or by other *rules* in the system.

```
<rule name="Example-Rule">
  <if>
    <equal value1="x" value2="y"/>
  </if>
  <then>
    <set name="z" value="4"/>
```

```
  </then>
  <else>
    <set name="z" value="4"/>
  </else>
</rule>
```

Figure 4.1.21.2: A basic example of an if/then/else statement written in the XESS XML language.

The same if/then/else statement can be rewritten as an XESS *rule* using the XML shown in Figure 4.1.21.2.

4.1.22.1 public Rule()

This constructor creates a new *Rule* with no name, if clause, then actions, or else actions.

4.1.22.2 public Rule(String name)

This constructor creates a new *Rule* with the specified name. The name must be unique within the scope of the other *rules* in the same XESS system.

name – The name of the new *Rule*; this name must be unique within the scope of the *rules* in the same XESS system.

4.1.22.3 public void setName(String name)

This method sets the name of the *Rule* to the specified value. The name must be within the scope of the *rules* in the same XESS system.

name – The new value for the name of the *Rule*; this name must be unique within the scope of the *rules* in the same XESS system.

4.1.22.4 public String getName()

This method returns the current value for the name of the *Rule*.

Returns – The current value for the name of the *Rule*.

4.1.22.5 public void addParameter(Parameter parameter)

This method adds the specified *parameter* to the *Rule*. Each *parameter* specifies a name and type, that must correspond to an argument at run time whenever the *Rule* is invoked. The *name* must be unique within the scope of *parameters* within the *Rule* but may be the same as other entities in the system. The arguments to the Rule are passed by reference, meaning that

if the *parameter* is modified as a result of the invocation of the *Rule*, the entity to which the *parameter* points should also be modified as well.

Each *parameter* may specify a type, which may correspond with a *structure* that has been defined in the same XESS system. In this case the argument at run time should be an *instance* of the specified type.

parameter – The *parameter* to be added to the *Rule*. If another *parameter* with the same name already exists, the new *parameter* replaces it.

4.1.22.6 public Parameter getParameter(String name)

This method returns the *parameter* with the specified name, if it exists within the scope of the *Rule*.

Returns – The *parameter* with the specified name, if it exists within the scope of the *Rule*. Otherwise, a null value is returned.

4.1.22.7 public void removeParameter(String name)

This method removes the *parameter* with the specified name from the *Rule*, if it exists within the scope of the *Rule*.

name – The name of the *parameter* to be removed from the scope of the *Rule*.

4.1.22.8 public Parameter[] getParameters()

This method returns a collection of *parameters* that have been added to the *Rule*. This collection must contain all of the uniquely named *parameters* that have been added to the *Rule*, but may be empty if no *parameters* have been added.

Returns – A complete collection of the uniquely named *parameters* that have been added to the *Rule*.

4.1.22.9 public void clearParameters()

This method clears all of the *parameters* that have been added to the *Rule*.

4.1.22.10 public void setIfClause(Clause clause)

This method sets the if clause on the *Rule*. The if clause represents the condition(s) that the *Rule* tests when invoked. The if clause may contain references to *facts* outside the scope of the rule, or *parameters* inside the scope of the rule as well as constants. If during invocation of the *Rule*, the

if clause evaluates to true, the then *actions* are taken; otherwise the *else* actions are taken.

clause – The clause that is evaluated when the *Rule* is invoked.

4.1.22.11 public Clause getIfClause()

This method returns the current if clause for the *Rule*. The if clause represents the condition(s) that the *Rule* tests when invoked.

Returns – The current if clause for the *Rule*.

4.1.22.12 public void addThenAction(Action action)

This method adds an *action* to the collection of *actions* that are taken each time the *Rule* is invoked and the if clause for the *Rule* evaluates to true. It should not be assumed that the *actions* are executed in the order in which they are added to the *Rule*; depending on the implementation of the collection used to store the *actions*, they may be executed in an arbitrary order.

action – The *action* that is added to the collection of *actions* that are taken each time the *Rule* is invoked and the if clause for the *Rule* evaluates to true.

4.1.22.13 public void removeThenAction(Action action)

This method removes the specified *action* from the collection of *actions* that are taken when the *Rule* is invoked and the if clause evaluates to true.

action – The *action* that is removed from the collection of *actions* that are taken each time the *Rule* is invoked and the if clause for the *Rule* evaluates to true.

4.1.22.14 public Action[] getThenActions()

This method returns the collection of *actions* that are taken when the *Rule* is invoked and the if clause evaluates to true. This method may return zero or more *actions* in an arbitrary order, but should return all *actions* that have been added to the *Rule* as then *actions*.

Returns – The collection of *actions* that are taken when the *Rule* is invoked and the if clause evaluates to true.

4.1.22.15 public void clearThenActions()

This method clears all of the then *actions* that have been added to the *Rule*. If no other else *actions* are added to the *Rule*, no *action* will be taken in the event that the *Rule* is invoked and the if clause evaluates to true.

4.1.22.16 public void addElseAction(Action action)

This method adds an *action* to the collection of *actions* that are taken each time the *Rule* is invoked and the if clause for the *Rule* evaluates to false. It should not be assumed that the *actions* are executed in the order in which they are added to the *Rule*; depending on the implementation of the collection used to store the *actions*, they may be executed in an arbitrary order.

action – The *action* that is added to the collection of *actions* that are taken each time the *Rule* is invoked and the if clause for the *Rule* evaluates to false.

4.1.22.17 public void removeElseAction(Action action)

This method removes the specified *action* from the collection of *actions* that are taken when the *Rule* is invoked and the if clause evaluates to false.

action – The *action* that is removed from the collection of *actions* that are taken each time the *Rule* is invoked and the if clause for the *Rule* evaluates to false.

4.1.22.18 public Action[] getElseActions()

This method returns the collection of *actions* that are taken when the *Rule* is invoked and the if clause evaluates to false. This method may return zero or more *actions* in an arbitrary order, but should return all *actions* that have been added to the *Rule* as else *actions*.

Returns – The collection of *actions* that are taken when the *Rule* is invoked and the if clause evaluates to false.

4.1.22.19 public void clearElseActions()

This method clears all of the else *actions* that have been added to the *Rule*. If no other else *actions* are added to the *Rule*, no *action* will be taken in the event that the *Rule* is invoked and the if clause evaluates to false.

4.1.22.20 public String toXml()

This method returns the *Rule* as an XML-formatted string compliant with the XESS schema definition for a *rule*; the string has the following format:

```
<rule name=”{name}>
     {parameter list}
     <if>{clause}</if>
     <then>{then actions}</then>
     <else>{else actions}</else>
</rule>
```

Where *{name}* is the value returned by the *getName* method on the *Rule*, the value of which must be unique within the scope of *rules* within the same XESS system.

The *{parameter list}* contains the XML for the *parameters* to the *Rule*. See the specification for output to the *toXml* method on the *Parameter* class for more details.

The *{if clause}* contains the XML for the *clause* that is evaluated whenever the *Rule* is invoked. See the specification for the various *clause* types elsewhere in this document for more details.

The *{then actions}* contains the XML for one or more *actions* taken in the event that the *Rule* is invoked and the if *clause* evaluates to true. See the specification for the various *action* types elsewhere in this document for more details.

The *{else actions}* contains the XML for one or more *actions* taken in the event that the *Rule* is invoked and the if *clause* evaluates to false. See the specification for the various *action* types elsewhere in this document for more details.

4.1.23 Abstract Class Fact extends XmlElement

A *Fact* is an unconditional value in the system. *Facts* may represent simple equalities such as “`X = Y`” or be specified as more complicated relations that establish a relationship between an unbounded set of simpler facts. The abstract *Fact* class is the parent of the more specialized types of *Facts* that make up an expert system. In an XESS system, each *Fact* must have a unique name, but the same name may be reused with other XESS entity types (such as rules).

4.1.17.1 public Fact()

This constructor creates a new *Fact* with a null name.

4.1.17.2 public Fact(String n)

n – A string specifying the value for the name of the new *Fact*.

This constructor creates a new *Fact* with the specified name.

4.1.17.3 public void setName(String n)

n – A string specifying a new value for the name of the *Fact*.

This method sets the name of the *Fact* to the specified value.

4.1.17.4 public String getName()

This method returns the name of the *Fact*.

Returns – A string with the name of the *Fact*.

4.1.24 Class Predicate extends Fact

A *Predicate* is the simplest form of *Fact*; it simply associates a name with a value (e.g. X=Y).

4.1.15.1 public Predicate(String n, String v)

n – A string specifying the value for the name of the new *Predicate*.
v – A string specifying the value for the new *Predicate*.

This constructor creates a new *Predicate* with the specified name and value.

4.1.15.2 public void setValue(String v)

v – A string specifying a new value for the *Predicate*.

This method sets the value of the *Predicate* to the specified string.

4.1.15.3 public String getValue()

This method returns the current value of the *Predicate*.

Returns – A string with the current value of the *Predicate*.

4.1.25 Class Field implements XmlElement

The *Field* class represents a simple value within a more complex *Fact* in an XESS system. Similar to *Predicates*, *Facts* are essentially name/value

pairs, but *Fields* are used very differently depending on the context. A *Fact* may also specify a type; types may be used to indicate that the *Field* represents a custom type, for which a simple name and value are not sufficient. *Field* types are discussed in greater detail later in this section.

4.1.19.1 public Field()

This constructor creates a new *Field* with a null name, type, and value.

4.1.19.2 public Field(String n, String t, String v)

n – A string specifying the value for the name of the new *Field.* The name of the *Field* must be unique within its scope.
t – A string specifying the type of the *Field*; the usage of a *Field* type is discussed in greater detail later in this section.
v – A string specifying the value for the new *Field*.

This constructor creates a new *Field* with the specified name, type, and value.

4.1.19.3 public void setName(String n)

n – A string specifying a new value for the name of the *Field*.

This method sets the name of the *Field* to the specified value. In general, the name of a *Field* must be unique within its scope.

4.1.19.4 public String getName()

This method returns the name of the *Field*.

Returns – A string with the name of the *Field*.

4.1.19.5 public void setType(String t)

t – A string specifying a new type for the *Field*.

This method sets the type of the *Field* to the specified value; the usage of a *Field* type is discussed in greater detail later in this section.

4.1.19.6 public String getType()

This method returns the type of the *Field*; the usage of a *Field* type is discussed in greater detail later in this section.

Returns – A string specifying the current the type of the *Field*.

4.1.19.7 public void setValue(String v)

v – A string specifying a new value for the *Field*.

This method sets the value of the *Field* to the specified string.

4.1.19.8 public String getValue()

This method returns the current value of the *Field*.

Returns – A string specifying the current the value of the *Field*.

4.1.26 Class Struct extends Fact

The *Struct* class is an object representation of the *structure* entity defined in the XESS Schema. A *Struct* contains a name, inherited from the abstract *Fact* class, and a collection of *Fields*. The *Struct* must be uniquely named with respect to all other facts within the scope of a single XESS system (this includes other fact types such as *Predicates*). The *Fields* within the *Struct* must be uniquely named within the scope of the *Struct* itself, but may share the same name as entities outside of the scope of the *Struct* (such as other *facts* or *rules* within the same XESS system).

The *Struct* object is similar to type definitions supported in many current expert system shell languages (e.g. the JESS *template*) but may also be represented in languages without support for custom types as a collection of facts and/or rules that are associated by name; the specified implementation is at the discretion of the plug-in developer.

4.1.20.1 public Struct()

This constructor creates a new *Struct* with a blank name and an empty collection of *Fields*. As stated previously, the name of the *Struct* must be unique within the scope of the XESS system containing the *Struct*; while a *Struct* may be created with a blank name, the name should be set to a unique value using the *setName* method of the parent *Fact* class.

4.1.20.2 public Struct(String n)

n – The name of the new *Struct*.

This constructor creates a new *Struct* with the specified name and an empty collection of *Fields*. The name of the *Struct* must be unique within its scope, but validation is not performed at construction. The name may

be modified at any time using the *setName* method of the parent *Fact* class.

4.1.20.3 public void addField(Field f)

f – The *Field* to be added to the *Struct*.

This method adds the specified *Field* to the *Struct*. If another *Field* with the same name already exists in the collection of *Fields*, it will be overridden by the newly added *Field*; otherwise the new *Field* is simply added to the *Struct*. If the *Field* type specifies the name of another *Struct* within the system (i.e. it is used to specify a non-standard type), validation must occur to insure that such a *Struct* exists.

4.1.20.4 public Field getField(String n)

n – The name of the *Field* to be retrieved.

This method returns the *Field* with the specified name, if such a *Field* currently exists within the scope of the *Struct*.

Returns – The *Field* with the specified name, if it exists.

4.1.20.5 public Field removeField(String n)

n – The name of the *Field* to be removed.

This method returns the *Field* with the specified name, if such a *Field* currently exists within the scope of the *Struct*, and removes that *Field* from the *Struct*.

Returns – The *Field* with the specified name, if it exists.

4.1.20.6 public void setFields(Field[] f)

f – The collection of *Fields* to be set on the *Struct*.

This method replaces the current set of *Fields* contained by the *Struct* with the specified collection of *Fields*. All *Fields* currently contained within the *Struct* are removed before the new collection of *Fields* is added.

4.1.20.7 public Field[] getFields()

This method returns the entire collection of *Fields* contained within the Struct.

Returns – The collection of *Fields* contained within the *Struct*.

4.1.20.8 public void clearFields()

This method clears the current collection of *Fields* contained by the *Struct*. The *Struct* will no longer contain any *Field* definitions after this method is called.

4.1.20.9 public Instance newInstance()

This method creates a new *Instance* of the *Struct* that inherits all of the current *Fields* of the *Struct*, including their default values (if specified). The *Instance* class is discussed in more detail in the next section.

Returns – A new *Instance* of the *Struct*.

4.1.20.10 public String toXml()

This method returns the *Struct* as an XML-formatted string compliant with the XESS schema definition for a *structure*; the string has the following format:

```
<struct name="{name}">
     {field list}
</struct>
```

Where *{name}* is the name of the *Struct* returned by the *getName* method of the parent *Fact* class. The *{field list}* contains zero or more entries in the following format:

```
<field name="{name}" type="{type}"
     initialValue="{value}">
```

Where *{name}* is the name of the *Field* returned by the *getName()* method of the *Field* class, *{type}* is the type of the *Field* returned by the *getType()* method of the *Field* class (this attribute is omitted if the type is null), and *{value}* is the value of the *Field* returned by the *getValue()* method of the *Field* class (this attribute is omitted if the value is null).

4.1.27 Class Instance extends Fact

The *Instance* class is an object representation of the *instance* entity defined in the XESS Schema. A *Struct* defines a custom type within the XESS system by creating a named relationship between a collection of fields; an *Instance* is a concrete representation of the same type, where the *fields* have been assigned meaningful values. The name of each *instance*

must be unique within the scope of the *facts* within a single XESS system, but may share the same name with other entities (such as *fields* or *rules*).

Each instance must be associated with a parent *Struct*, and default values for each field are inherited from the *Struct* whenever a default value has been specified; an *Instance* may override the default value for any field simply by setting a field with the same name and a different value. Because a *Struct* is an abstract definition of a custom type, *Structs* may never be passed as arguments to rules. Though rules (and other *Structs*) may specify a *Struct* as the type for one of its parameters or fields, a concrete *Instance* must be used at runtime.

The example in *Figure 4.1.21.1* shows a *Struct* that defines the custom type "person"; each person has fields representing full name, sex, and birthday. The example also includes example *Instances* that each represent a concrete "person".

```
<struct name=″Person″>
  <field name=″full-name″/>
  <field name=″sex″/>
  <field name=″birth-month″/>
  <field name=″birth-day″/>
  <field name=″birth-year″/>
</struct>

<instance name=″first-person″ type=″Person″>
  <field name=″full-name″ value=″john j. doe″/>
  <field name=″sex″ value=″male″/>
  <field name=″birth-month″ value=″3″/>
  <field name=″birth-day″ value=″30″/>
  <field name=″birth-year″ value=″1975″/>
</instance>

<instance name=″second-person″ type=″Person″>
  <field name=″full-name″ value=″jane k. doe″/>
  <field name=″sex″ value=″female″/>
  <field name=″birth-month″ value=″4″/>
  <field name=″birth-day″ value=″29″/>
  <field name=″birth-year″ value=″1975″/>
</instance>
```

Figure 4.1.21.1: An example of a Struct with two Instances

In this example, each *instance* defines a value for every *field* in the parent *struct*; default values do not make sense because there is no sensible default for any of the fields specified (although one could argue that a default value for the "sex" field would be correct about 50% of the time, and therefore may be appropriate).

4.1.27.1 public Instance(Struct type)

This constructor creates a new *Instance* with the specified parent *Struct*. The new instance inherits any *fields* of the specified *struct*, including any relevant default values. The new *instance* will have a null name.

type – The parent *structure* that defines the fields of the new *instance*.

4.1.27.2 public Instance(String name, Struct type)

This constructor creates a new *instance* with the specified name and parent *structure*. The new *instance* inherits any *fields* of the specified *structure*, including any relevant default values.

name – The name of the new *Instance* which must be unique within the scope of the *facts* of a single XESS system.
type – The parent *structure* that defines the *fields* of the new *instance*.

4.1.27.3 public void setField(Field f)

This method sets the specified *field* on the *instance*.

f – The *field* to set on the *instance*.

Exceptions – This method should throw an exception if a *field* with the specified name does not exist in the parent *structure*.

4.1.27.4 public Field getField(String name)

This method returns the *field* with the specified name from the *instance*. If a *field* with the same name has not been set on the *instance*, the default value of the *field* as specified in the parent *structure* is returned instead (which may be null).

name – The name of the desired *field*.

Exceptions – This method should thrown an exception if a *field* with the specified name has not been defined in the parent *structure*.

4.1.21.5 public Field[] getFields()

This method returns the entire collection of *Fields* contained within the *instance*.

Returns – The collection of *Fields* contained within the *instance*.

4.1.21.6 public void clearFields()

This method clears the current collection of *Fields* contained by the *instance*. This method will not clear the *field* definitions from the parent *structure*.

4.1.21.7 public Struct getType()

This method returns the parent *structure* for the *instance*.

Returns – The parent *structure* for the instance.

4.1.21.8 public void toXml()

This method returns the *instance* as an XML-formatted string compliant with the XESS schema definition for an *instance*; the string has the following format:

```
<instance name="{name}" type="{type}">
     {field list}
</instance>
```

Where *{name}* is the name of the *instance* returned by the *getName* method of the parent *Fact* class; and *{type}* is the name of the parent *structure*. The *{field list}* contains zero or more entries in the following format:

```
<field name="{name}" type="{type}"
     value="{value}">
```

Where *{name}* is the name of the *Field* returned by the *getName()* method of the *Field* class, *{type}* is the type of the *Field* returned by the *getType()* method of the *Field* class (this attribute is omitted if the type is null), and *{value}* is the value of the *Field* returned by the *getValue()* method of the *Field* class (if the value is null, the default value from the parent *structure* is used instead).

4.1.28 Class Xess implements XmlElement

The *Xess* class provides an object representation of the *xess* entity defined in the XESS XML Schema. Each instance of the *Xess* class represents an expert system; it contains a knowledge base that is composed of *facts*, and a set of *rules* as well as methods for manipulating both. It is important to note that an *Xess* instance is not an executable class; it simply represents a snapshot of an expert system. The XESS API requires one or more plug-ins to interpret and execute an *Xess* on a rules engine; the XESS API simply provides a layer of abstract between the expert system and the shell upon which it is executed.

The *Xess* class provides a convenient wrapper to contain the disparate entities of a potentially complex set of *facts* and *rules* that allows them to be passed around, interpreted, executed, and updated as a whole. The same *Xess* instance can be executed on multiple rules engines through different plug-ins, and different *Xess* instances can share information by moving facts and rules back and forth between them. In its simplest form, however, the *Xess* class is a direct translation of an XESS document.

4.1.28.1 public Xess()

This constructor creates a new, empty *Xess* instance. The newly created *Xess* will not contain any *facts*, *rules*, or any trace information.

4.1.28.2 public void addRule(Rule rule)

This method adds the specified *rule* to the set of *rules* contained by the *Xess* instance. The name of the *rule* must be unique among the scope of the *rules* contained by the *Xess*; if another *rule* with the same name already exists, it will be replaced.

rule – The *rule* that is added to the collection of *rules* contained by the *Xess* instance.

4.1.28.3 public Rule getRule(String name)

This method returns the *rule* with the specified name, if it exists within the collection of *rules* contained by the *Xess* instance. If no such *rule* exists, a null value is returned instead.

Returns – The *rule* with the specified name, if it exists within the collection of *rules* contained by the *Xess* instance; otherwise returns null.

4.1.28.4 public void removeRule(String name)

This method removes the *rule* with the specified name from the collection of *rules* contained by the *Xess* instance.

name – The name of the *rule* to be removed from the *Xess* instance.

4.1.28.5 public String[] getRuleNames()

This method returns a collection of zero or more strings, each of which represents the unique name of a *rule* that has been added to the *Xess* instance. The order of the collection of strings should match the order in which the *rules* were added to the *Xess* instance.

Returns – An ordered collection of zero or more strings, each of which represents the name of a *rule* that has been added to the *Xess* instance in the order in which the *rules* were added.

4.1.28.6 public Rule[] getRules()

This method returns an ordered collection of zero or more *rules* that have been added to the *Xess* instance in the order in which they were added. The collection must contain all of the *rules* that have been added to the *Xess* instance (other than those that have been removed, cleared, or replaced).

Returns – An ordered collection of zero or more *rules* that have been added to the *Xess* instance in the order in which they were added.

4.1.28.7 public void clearRules()

This method removes all of the *rules* that have been added to the *Xess* instance.

4.1.28.8 public void addFact(Fact fact)

This method adds the specified *fact* to the set of *facts* contained by the *Xess* instance. The name of the *fact* must be unique among the scope of the *facts* contained by the *Xess*; if another *fact* with the same name already exists, it will be replaced.

fact – The *fact* that is added to the collection of *facts* contained by the *Xess* instance.

4.1.28.9 public Fact getFact(String name)

This method returns the *fact* with the specified name, if it exists within the collection of *facts* contained by the *Xess* instance. If no such *fact* exists, a null value is returned instead.

Returns – The *fact* with the specified name, if it exists within the collection of *facts* contained by the *Xess* instance; otherwise returns null.

4.1.28.10 public void removeFact(String name)

This method removes the *fact* with the specified name from the collection of *facts* contained by the *Xess* instance.

name – The name of the *fact* to be removed from the *Xess* instance.

4.1.28.11 public String[] getFactNames()

This method returns a collection of zero or more strings, each of which represents the unique name of a *fact* that has been added to the *Xess* instance. The order of the collection of strings should match the order in which the *facts* were added to the *Xess* instance.

Returns – An ordered collection of zero or more strings, each of which represents the name of a *fact* that has been added to the *Xess* instance in the order in which the *facts* were added.

4.1.28.12 public Fact[] getFacts()

This method returns an ordered collection of zero or more *facts* that have been added to the *Xess* instance in the order in which they were added. The collection must contain all of the *facts* that have been added to the *Xess* instance (other than those that have been removed, cleared, or replaced).

Returns – An ordered collection of zero or more *facts* that have been added to the *Xess* instance in the order in which they were added.

4.1.28.13 public void clearFacts()

This method removes all of the *facts* that have been added to the *Xess* instance.

4.1.28.14 public void trace(String source, String trace)

One of the most important features of an expert system is not only the decisions to which it arrives, but the ability to determine *how* it arrived at those decisions. The *Xess* class provides a simple tracing mechanism that allows every action taken by a rule engine to be traced, so that the path to that decision can be later examined and verified. This method allows an external entity to add a single line of information to the trace. Each time this method is called, a string with the format *{source}:{trace}* is added to the trace stack.

source – A descriptive string identifying the source of the trace message.

trace – A detailed trace message.

4.1.28.15 public String[] getTrace()

This method returns an ordered collection of strings, each element of which contains a single trace message in the format *{source}:{trace}*. The

order of the strings in the collection must be the same as the order in which the strings were traced, and the collection must contain all trace messages that have been added (other than those that were cleared).

Returns – An ordered collection of strings, each of which is a trace message in the format *{source}:{trace}*.

4.1.28.16 public void clearTrace()

This method clears all of the trace messages currently stored by the *Xess* instance.

4.1.29 Interface XessParser extends XessConstants

The *Xess Parser* interface provides a very simple, basic interface that must be implemented by a parser than can translate from XESS Schema compliant XML documents to an instance of the *Xess* class. This interface provides a layer of abstraction between the *Xess* components and specific XML parsers, allowing developers to hot-swap between them before or during runtime based on specific requirements (e.g. runtime validation, memory footprint, personal preference, etc.).

4.1.29.1 public Xess parseXess(String filename)

This method parses the XESS Schema compliant XML document in the specified file and returns an instance of the *Xess* class that has been populated with the *facts* and *rules* specified in the document.

filename – The path to the file containing the XESS Schema compliant XML document to be parsed.

Returns – The *Xess* instance that has been parsed from the specified document. This instance must contain all of the *facts* and *rules* specified in the original XESS document in the order in which they were specified.

Exceptions – This method throws an exception in the event that the file does not exist, cannot be opened or read, or does not contain a valid XESS Schema compliant document.

4.1.30 Interface XessPlugin

The XESS API does not contain an explicit rules engine that is capable of interpreting *facts* and *rules* and producing results. That is not the problem that XESS attempts to solve. The purpose of XESS is to provide a layer of abstraction between the expert systems developer and the rules engine

implementation, and allows the expert systems developer to execute the same rules on different engines without modification.

As demonstrated in section 2 of this document there are many disparate expert systems shells that are popular and in use today, each of which has different strengths and weaknesses when compared to the other (though all of those examined are fairly weak on security).

Without the layer of abstraction that XESS provides, an expert systems developer that wishes to run the same rule set on multiple expert system shell implementations would need to completely rewrite the facts and rules for the system in the native language of each expert system shell, and execute them individually to compare the results. The XESS API attempts to provide that flexibility without requiring that the rules be written more than once. This allows developers to evaluate the same rule set on several different systems, and to potentially determine which system is the best fit for a particular rule set. The developer may then choose to run the rule set on that specific system through the XESS API, or to translate the set using XESS into the native language of the specific system to be run without the overhead that XESS requires.

This is accomplished by writing the *facts* and *rules* in the XESS XML language, and using the XESS API to parse the language into objects that can then be translated and executed on one or more rules engines without modification. This translation and execution is handled through the use of XESS *plug-ins*.

The *Xess Plug-In* interface provides a very simple, but powerful, interface for executing an *Xess* instance containing the *facts* and *rules* that are to be translated and executed. The plug-in itself has a very simple set of methods that are used to make a blocking call to a rules engine; the *Xess* instance is submitted and the method returns once evaluation has completed.

4.1.30.1 public String getName()

This method returns the name of the plug-in; this name must be unique within the scope of plug-ins registered in the same runtime environment.

Returns – The name of the plug-in; this name must be unique within the scope of plug-ins registered in the same runtime system, and should be used to identify the plug-in as the *source* of any trace messages generated by the plug-in.

4.1.30.2 public void execute(Xess xess)

This method evaluates the rule set contained in the specified *Xess* instance and blocks until evaluation is complete. Trace messages must be generated (through the use of the trace methods on the *Xess* class) for every action taken, particularly if any *facts* or *rules* are created or modified as a result of the evaluation.

The mechanics of the evaluation depend on the rules engine through which the plug-in performs evaluation. The primary responsibility of the plug-in is to translate from the XESS API classes and interfaces into the native language of the rules engine, and back again. This may be accomplished in any one of a number of ways, such as calling methods directly on another rules engine API, or by translating the XESS objects into a text document in the native language of the rules engine that is then executed. Specifics are up to the plug-in developer, but plug-ins *must* be generic; they should be able to handle translation to and from any combination of XESS facts and rules.

4.1.31 Interface XessPluginDriver

The *Xess Plug-in Driver* is the interface for classes responsible for creating and managing different instances of plug-ins for the same rules engine. Because *Xess Plug-ins* may be stateful with respect to the *Xess* rule sets that they have executed, it may be necessary (or at least desirable) to create more than one instance of the same plug-in for use in executing different *Xess* rule sets within the same runtime environment.

The *Xess Plug-in Driver* is a simple interface that is used to create and return instances of plug-ins for the same rules engine. The driver may return the same instance each time, different instances each time, or it may cycle through several instances based upon some criteria (such as whether or not the instance is currently in use). This is up to the discretion of the plug-in developer and the requirements of the specific rules engine through which the plug-in evaluates *Xess* rule sets. This driver also insulates the XESS user from any detailed information regarding the setup for the specific rules engine, which otherwise may be necessary if the user were to construct plug-ins directly.

4.1.31.1 public String getName()

This method returns the name of the plug-in; this name must be unique within the scope of plug-ins (and drivers) registered in the same runtime environment.

Returns – The name of the plug-in; this name must be unique within the scope of plug-ins (and drivers) registered in the same runtime system, and should be used to identify the plug-in as the *source* of any trace messages generated by the plug-in.

4.1.31.2 public XessPlugin getPlugin()

This method returns an instance of the *Xess Plug-in* interface that can be used to interpret, and execute instances of the *Xess* class that contain an XESS rule set. The specific implementation of this method is up to the plug-in developer and may differ depending on the constraints of the rules engine through which the plug-in interprets and executes the XESS rule sets.

Returns – An implementation of the *Xess Plug-in* interface that can be used to interpret and execute instances of the *Xess* class that contain an XESS rule set.

4.1.32 Class XessPluginManager

The *Xess Plug-in Manager* is a singleton class that provides a simple API for registering and managing the *Xess Plug-in Drivers* that are used to create and return plug-ins. The *Xess Plug-in Manager* is the single point of contact for the system when retrieving plug-ins that can be used to interpret and execute instances of the *Xess* class that contain XESS rule sets.

4.1.32.1 public void registerDriver(XessPluginDriver driver)

This method registers the specified *Xess Plugin Driver* with the driver manager. The driver must be uniquely named, and if another driver with the same name has been registered previously it will be replaced with the newly registered driver.

Good *Xess Plugin Drivers* should register automatically when the driver is constructed or when the driver class is loaded; this method should therefore never need to be explicitly called by the XESS API user.

driver – The *Xess Plugin Driver* that should be registered using the name returned by the *getName* method on the driver. The driver should register itself (passing itself as an argument) upon being constructed or loaded.

4.1.32.2 public void deregisterDriver(String name)

This method deregisters the specified *Xess Plugin Driver* with the specified name from driver manager. From this point forward the driver may not be used via the driver manager to create or manage plug-ins.

name – The name of the *Xess Plug-in Driver* to deregister. The driver may still be used, but will not be accessible through the *Xess Driver Manager* API.

4.1.32.3 public XessPlugin getPlugin(String name)

This method looks up the *Xess Plug-in Driver* with the specified name, and uses it to create and return a corresponding plug-in that can then be used to interpret and execute the rule set in an instance of the *Xess* class. If no such driver exists, a null value is returned instead.

name – The name of the *Xess Plug-in Driver* that should be used to create and return an instance of the *Xess Plug-in* interface. Once returned this plug-in can then be used to interpret and execute an instance of the *Xess* class containing an XESS rule set. Whether or not the plug-in returned is unique to this call is dependent on the specific *Xess Plug-in Driver* implementation.

Returns – An *Xess Plug-in* created by the *Xess Plug-in Driver* with the specified name, if it exists and is currently registered with the *Xess Plug-in Driver Manager*. If no such driver exists, a null value is returned instead.

4.2 A Java Implementation of the XESS API

For the purposes of testing the practical applications of the XESS API described in the first section of this chapter, an implementation of the entire API was created using the Java 5 SDK. The specification for the XESS API as outlined here is written in a form that should be generically applicable to any modern object oriented programming language with minimal modifications. The changes made to the Java 5 version of the prototype implementation are outlined here.

- **Packaging:** All class an interface definitions for the entities described in the first section of this chapter are included in the *xess* package, or a sub-package; e.g. *xess.XessPluginManager*.

- **KXML Parser:** An implementation of the *XessParser* was written using the KXML Parser 2.0[14]. The KXML Parser is an extremely light weight, small footprint parser[15] that does not implement SAX[16] or DOM[17] parsing as is common for most XML parsers. Instead, KXML implements the XML Pull Parser (XMLPP)[18] standard, which is designed for small footprint, embedded code. This implementation is included in the *xess.xmlpp* package.

[14] The home of kXML at kObjects.net, http://kobjects.org/kxml/
[15] D.S. Kochnev, A.A. Terekhov, "Surviving Java for Mobiles"
[16] Simple API for XML Parsing (SAX), http://www.saxproject.org/
[17] Document Object Model (DOM), http://www.w3.org/DOM/
[18] XML Pull Parsing, http://xmlpull.org/

- **Jess Plug-in:** An implementation of the XESS Plug-in APIs was created for the Java Expert System Shell[19]. The implementation is included in the *xess.jess* package.

- **FuzzyJ Plug-in:** An implementation of the XESS Plug-in APIs was created for the FuzzyJ Toolkit[20]. The implementation is included in the *xess.fuzzyj* package.

- **Fuzzy Plug-in:** An implementation of the XESS Plug-in APIs was created for the Open Source Fuzzy Engine[21] by Edward Sazonov. The implementation is included in the *xess.fuzzy* package.

- **Java 5 Collections:** Finally, the Java 5 implementation of the XESS APIs makes use of the Java 5 collections APIs, which includes support for "generics," or strongly typed collections. In any place where the XESS API definition calls for a strongly typed array, an appropriate Java 5 collection class is used instead. For example, the XESS API definition for the *ClauseList* class requires a *getClauses* method that returns an array of *Clause* objects contained in the clause list. The Java 5 implementation of the *ClausList* object instead returns a *java.util.Collection<Clause>*, which is a strongly typed collection of *Clause* objects. The advantages of using collections over vanilla arrays are numerous and include efficient searching, sorting, and iteration.

[19] "Jess, the Rule Engine for the Java Platform," Sandia National Laboratories, http://herzberg.ca.sandia.gov/
[20] "The FuzzyJ Toolkit," National Research Council Canada, http://www.iit.nrc.ca/IR_public/fuzzy/fuzzyJToolkit.html
[21] "Open Source Fuzzy Inference Engine for Java," Edward Sazonov, http://people.clarkson.edu/~esazonov/FuzzyEngine.htm

5 The Expert System for Security Assessment

The XML Expert System Shell was designed to have several advantages over specific expert system shell implementations. In brief, these advantages are:

- Language Independence – Because and XESS expert system is written in XML, it is independent from any specific programming languages, and can be interpreted by any language capable of parsing XML.
- Human Readability – XESS uses a text-based XML language and terms that are composed of fully formed English language words. The XESS schema has been designed to prevent the deeply-nested trees that can often occur in XML documents.
- Shallow Learning Curve - Every expert system shell language has a learning curve, and though many languages are iterations upon the same basic themes (like LISP and Scheme) each is unique and requires students to learn the associated terms and syntax. XESS seeks to create a shallow learning curve by using terms common in describing expert systems to students of artificial intelligence, and by using the XML syntax, which is familiar to most programmers.
- Expert System Independence – The XESS interpreter inserts a layer of abstraction between the expert system and the rules engine on which the system is interpreted. This theoretically allows the same system to be executed on multiple rules engines without modification, and in some cases would allow a system to be broken into pieces that are executed in parallel on different engines for the best result.

To effectively demonstrate that the XESS language meets the above stated goals, a non-trivial expert system must be expressed entirely using the XESS language. This system should contain a significant number of non-trivial rules that are representative of those found in many expert systems. For the purposes of this demonstration, an Expert System for Security Assessment (ESSA) was developed based on the security assessment recommendations of the National Institute of Standards and Technology (NIST). The complete XESS code for the system can be seen in Appendix B.

The ESSA is rules based system that can provide an evaluation for the overall security level of a complex computer system by evaluating and combining individual scores in areas such as security controls, policies, and procedures. The rules used in the system are based on a subset of the NIST standards as shown in Figure 5.1.

1. **Risk Management**
    a. How many times per year is a risk assessment performed? (enter 0 if risk assessments are not performed.)
    b. What percentage of systems are assessed and documented as of this time?
2. **Security controls**
    a. What percentage of systems have been tested for security controls in the past year?
    b. How many weaknesses were discovered?
3. **Authorize Processing (Certification and Accreditation)**
    a. How many systems have been certified and accredited?
4. **System Security Plan - target 100%**
    a. Is there a documented system security plan?
    b. How many systems follow it?
5. **Personnel Security - target near 100%**
    a. How many systems divide sensitive functions among different individuals?
    b. How many of your users have undergone background screening?
6. **Physical Protection**
    a. Are deposits/withdrawals of physical data(tapes) logged?
    b. Is physical access to data lines protected?
        i. By lock?
        ii. By keypad?
        iii. By biometrics?
        iv. By keycard?
    c. Are mobile systems protected?
        i. Is encryption software installed on laptops?
7. **Contingency planning**
    a. How many critical systems are there?
    b. How many have backup systems established?
    c. How many systems have a contingency plan?
8. **Hardware and systems software maintenance**
    a. How many systems have restrictions on who performs maintenance/repairs?
    b. How many systems log maintenance activity?
    c. Are the engineers that perform maintenance internal, external, or remote?
    d. Are software changes documented and approved?
    e. How many systems were scanned for vulnerabilities in the past year?
    f. How many systems had to be patched?
9. **Data Integrity**
    a. Is there automated anti virus protection?
    b. How many systems use anti virus protection?
    c. Is auto update for antivirus enabled?
    d. How many systems are password protected?
    e. How many times a year are passwords required to change?
10. **Identification and authentication**
    a. Are users identified by passwords, tokens, or biometrics?
    b. On how many systems are the default vendor passwords being used?
    c. How many user ID's exist?
    d. How many are unique?
11. **Password system, password verification and security**

Figure 5.1: The NIST standards on which the ESSA rule set is based.

Input for the ESSA may be collected in a number of ways including questionnaires, documents, and practical experiments. The input may be provided via a text file, or through a user interface that collects user input directly. The implementation of the ESSA used to test the XESS implementation used the latter method; a command-line program that prompts the user for input. The ESSA could be further enhanced by collecting data through a graphical user interface, or a web application as well; XESS provides a layer of abstraction between the input collection mechanism and the rules engine, keeping it independent and flexible.

From the NIST guidelines[22], nine control fields were identified:

- The Risk Management control field is evaluated by determining the percentage of systems on which risk assessment is performed

[22] M. Swanson, et al, "Security Metrics Guide for Information Technology Systems"

each year, and the frequency per year that risk assessment is performed on those systems.

- The Security controls control field is evaluated based on the percentage of systems have been tested for security for security controls in the previous year, and what percentage of those machines tested were found to have at least one weakness.
- The Certification & Planning control field is evaluated based on the percentage of the total systems that have been certified and accredited as well as the number of systems that follow a documented security plan.
- The Personnel Security control field is evaluated based on the percentage of systems that provide access to sensitive functions to more than one individual, and the proportion of the total user base that has undergone background screening.
- The Physical Protection control field is evaluated based on whether or not access to physical backups is logged (and how often), what measures are taken to limit physical access to data lines (e.g. locks, keypads, biometrics), and the percentage of mobile systems on which encryption software has been installed.
- The Contingency Planning control field is evaluated based on what percentage of those systems that have been identified as "critical" are backed up, and the percentage of those same systems for which a contingency plan has been established for use in the event of an emergency.
- The Maintenance control field is evaluated based on a relatively large number of factors including what percentage of systems have restrictions on who performs maintenance & repairs, what percentage of maintenance is logged, the percentage of systems scanned for vulnerabilities, and the percentage of systems with vulnerabilities that were patched.
- The Data Integrity control field is evaluated based on the percentage of systems on which virus protection software has been installed, the percentage of virus protected systems that use automated scans & updates, the percentage of systems that use password protection, and the frequency per year that the passwords on those systems are required to be changed.
- Finally, the Identification & Authentication control field is evaluated based on the percentage of systems that use default (vendor specified) passwords, the percentage of users that share a common password, and the password requirements (to prevent easy guessing).

Each of the above control fields is assessed by collecting data in two or more sub-categories and calculating a weighted average based on the individual categorical scores.

The ESSA was implemented in XESS as a set of predicates and rules that operate using those predicates as input; the system knowledge base implements NIST standards such as[23].

Each predicate in the knowledge base represents the rating for a sub-category or an overall rating for one of the nine security controls in the ESSA. The example in Figure 5.2 shows the predicate for the "risk assessment" sub-category, as well as the predicate for the overall "risk" security control rating.

```
<!-- How many times per year is risk assessment performed
  (0-52)? -->
<predicate name="risk_assessment" value="0.0"/>

<!-- ratings: 0 (worst) - 100 (best) -->
<predicate name="risk" value="0.0"/>
```

Figure 5.1: Examples of the predicates used in the XESS implementation of the ESSA.

The initial value for each of the predicates in the system is assumed to be 0.0, though due to variance in the individual ratings this is sometimes an optimal score (e.g. given that 100% of systems were scanned for vulnerabilities, a score of 0.0 for systems found to have a security control weakness would be optimal). In other cases a score of 0.0 is the worst case (e.g. the number of systems on which anti-virus software has been installed). Because of this, it may be desirable to adjust each default score to be either the worst case, or the best case for consistency. In the example implementation input is collected from the user for every rating, and therefore the default of 0.0 is sufficient as the default value will be overwritten.

Each of the rules in the XESS implementation of the ESSA operates on one or more of the sub-categorical scores, and uses the value of each score to adjust the relevant security control rating. The rules shown in Figure 5.3 are used to determine the overall rating for the risk management security control, which is based on the number of times in the last three years that risk assessment was performed, and the percentage of machines covered under the assessment.

```
<rule name="risk_assessment_rule_001">
  <if>
    <and>
      <greaterThan value1="@risk_assessment" value2="0.0"/>
      <lessThan value1="@risk_assessment" value2="6.0"/>
    </and>
```

[23] Marianne Swanson, "Security Self-Assessment Guide for Information Technology Systems"

```
  </if>
  <then>
    <set name="risk" value="@risk+25.0"/>
  </then>
</rule>
<rule name="risk_assessment_rule_002">
  <if>
    <greaterThanOrEqual name="risk_assessment" value="6.0"/>
  </if>
  <then>
    <set name="risk" value="@risk+50.0"/>
  </then>
</rule>
<rule name="risk_assessment_rule_003">
  <if>
    <and>
      <greaterThanOrEqual name="risk_system_coverage"
                          value="50.0"/>
      <lessThan name="risk_system_coverage" value="75.0"/>
    </and>
  </if>
  <then>
    <set name="risk" value="@risk+25.0"/>
  </then>
</rule>
<rule name="risk_assessment_rule_004">
  <if>
    <greaterThanOrEqual name="risk_system_coverage"
                        value="75.0"/>
  </if>
  <then>
    <set name="risk" value="@risk+50.0"/>
  </then>
</rule>
```

Figure 5.3: An example of the ESSA rule to determine the risk management security control rating, as implemented in XESS.

The above rules compare the sub-categories for the risk management security control against certain thresholds, and adjusts the overall risk score accordingly, starting with a base score of 0.0 and adjusting up from there as warranted. The simple system used here can only produce ratings of 0, 25.0, 50.0, 75.0, or 100.0. This may be sufficient for many systems, but it is possible that a fuzzy logic system would be more suited for determining individual security control ratings with more precision; this will be discussed in the next section, which demonstrates the ESSA as implemented using the Fuzzy Logic extensions for XESS.

The rules are designed to function in a hierarchical fashion. First evaluating the sub-categories such as the number of physical controls used to prevent access to sensitive systems, the percentage of systems for which automated updates have been available, or password strength.

For example, when evaluating password strength (as a sub-category of the Identification & Authentication security control rating), the NIST standard requires a good password to be at least twelve characters, and that the

password contain both upper and lower case characters, as well as a combination of numbers, letters, and symbols. Such passwords are determined to be sufficiently hard to crack. Input values for each of these requirements are used to determine the score for overall password strength in the system being assessed. Using these rules a user of the system may see how subtle changes in individual properties affect categorical ratings; programmers may use these observed results to further adjust the variables used as input, giving more or less weight to individual properties as needed.

Each sub-category is evaluated, and its contribution to the relevant security control is determined. Once all of the security control ratings have been fully adjusted based on all of the input, the overall security assessment rating is determined by computing a weighted average of all of the individual security control scores. In the example implementation each security control rating is given equal weight, but it would be possible to adjust the weight for each rating based on the importance to the customer for whom the assessment is being performed.

One last rule provides an overall security assessment rating by accepting the categorical assessment ratings as input and providing a "score" from 0.0 (very weak) to 100.0 (very strong). This final rule was designed in such a way that a single, weak categorical rating may produce a relatively weak overall security rating; a system need only have one known weakness to be penetrated.

The system is modeled in the XESS high-level language, and because of this it can be parsed and translated by any XESS interpreter. The language-independent XML bindings are easily translated into objects at runtime that are then executed on a specific rules engine through the use of the generic plug-in API. To illustrate the strengths of the XESS language, the system was tested on the three different engines mentioned previously (Jess, FuzzyJ Toolkit, and the Open Source Fuzzy Inference Engine). The nature of the XESS language allows some variables and rules to be interpreted by one engine, while others are interpreted by another, to the point where some rules can be targeted at a specific engine that produces the best result. The final analysis can be computed using any combination of rules engines for which there is an XESS plug-in available, breaking the entire knowledge base into pieces that are computed separately and reassembled for analysis.

One interesting, though untested, application of this idea is to divide the knowledge base into two pieces: those facts and rules suited to a traditional "crisp" expert system, and those better suited to a fuzzy logic inference engine. The flexibility of XESS allows the two sets of rules to be interpreted simultaneously, within the same XESS interpreter, on two

different plug-ins; one interpreting crisp rules on a crisp rules engine, and second plug-in interpreting fuzzy rules using a fuzzy inference engine. The results can then be combined, and re-interpreted on either engine ad infinitum, until the desired result is achieved.

The system, as it stands, provides only the individual categorical ratings along with a final, overall security rating; a weighted average of the original ratings on the same 0-100 scale. From these results, any number of recommendations can be inferred and traced back to the original input; in fact, an application could easily be built on top of the XESS infrastructure to display this useful information. The XESS API provides a full trace mechanism, so that the process with which the categorical ratings were calculated can easily be examined.

Conceivably the knowledge base could be further extended to include specific recommendations based on the overall score, and the individual categorical ratings that affected the score (either positively or negatively). Because each category is rated based on a relatively small number of inputs, both the weak points and the strong points are easy to identify. For example, questions regarding passwords are grouped together because the recommendation is the same for all negative answers: require that passwords be changed in a timely fashion (at least several times a year), and require a mix of upper case letters, lower case letters, numbers, and symbols.

Finally, the ESSA effectively demonstrates that the initial version of XESS described in this document effectively meets the stated goals for the language.

- Language Independence – The entire system was written in XML and is compliant with the XESS Schema described in earlier chapters. Though the interpreter used for this test case was implemented in Java, as discussed earlier XML is not tied to any specific programming language and parsers already exist for most modern high level languages.
- Human Readability, Shallow Learning Curve – Unfortunately, these criteria are highly subjective, and each individual must determine for themselves whether or not XESS is easily human readable and understandable. This case study does prove that a non-trivial rule set can be expressed using English Language words (e.g. “if”, “greater than”, “less than or equal”) that are recognizable to programmers and laymen alike. The example also exhibits several of the properties of good language design such as simplicity and orthogonality[24], and overcoming indentation[25].

[24] Leslie B. Wilson, Robert G. Clark, “Comparative Programming Languages”

- Expert System Independence – Finally, the system was tested on three different inference engines through plug-ins, including JESS[26], The FuzzyJ Toolkit[27], and Fuzzy Engine[28] demonstrating that the same XESS rule set could be executed unmodified using a single interpreter. In all three cases the ESSA rule set was parsed into XESS Objects structured as a tree that mirrored the structure of the original XML document. The XESS objects were then executed on the different inference engines through generic plug-ins that translated the XESS Objects into facts and rules at runtime.

---

[25] Christopher Seiwald, "Seven Pillars of Pretty Code"
[26] "Jess, the Rule Engine for the Java Platform," Sandia National Laboratories, http://herzberg.ca.sandia.gov/
[27] "The FuzzyJ Toolkit," National Research Council Canada, http://www.iit.nrc.ca/IR_public/fuzzy/fuzzyJToolkit.html
[28] "Open Source Fuzzy Inference Engine for Java," Edward Sazonov, http://people.clarkson.edu/~esazonov/FuzzyEngine.htm

6 Conclusion

Many of the entry level artificial intelligence courses at schools of computer science begin the same way: the student is introduced to a handful of rules engines that demonstrate the many and varied approaches to artificial intelligence. A typical semester course may introduce the student to Prolog, Lisp and/or Scheme, CLIPS, or Jess or any number of other languages.

Each language has its loyal followers, and with good reason: each has strengths and weaknesses that elevate it above the others for certain applications; most popular rules engines are very good, general purpose reasoning systems, but each excels in some specific areas while it may be weak in others. This can be a frustrating exercise for the student who is learning artificial intelligence concepts at the same time that he or she struggles with many different languages, each of which uses its own syntax and terms which may or may not have much in common with the universal definitions taught in the classroom.

The primary inspiration behind the development of the XML Expert System Shell was to allow for the expression of facts and rules using terms common in the instruction of artificial intelligence such as “fact,” “rule,” “predicate,” and “universe of discourse” to make the transition from learning the concepts of artificial intelligence, and implementing those concepts in a programming language more fluid.

Additionally, XESS seeks to provide a sandbox in which a knowledge base can be defined, and then executed on several different languages to compare the strengths and weaknesses of those languages without rewriting. Instead of focusing on syntactic differences between languages, students can compare the same rule system in different environments. How well does Prolog handle simple mathematical operations? Does a forward chaining inference engine handle specific kinds of logic more efficiently than a backward chaining system? How does the complexity of an XESS plug-in relate to the complexity of the inference engine for which it performs translation?

The potential for XESS goes beyond educational purposes as well; there are some intriguing industrial applications. Used as a modeling language, XESS potentially allows knowledge engineers to fully express the knowledge base well before the inference engine is selected. Furthermore, if no single inference engine is a clear choice, the XESS interpreter allows the development team to experimentally execute the rule set on multiple engines to guide in the decision-making process without needing to rewrite the entire knowledge base for each engine. Finally, if the performance overhead of the interpreter does not cause the final program

to run outside of the runtime efficiency requirements, the XESS version of the knowledge base may be used in the final product. The experimental version of XESS written for this paper was implemented on extremely small footprint technologies for Java, and provides performance suitable for a non-real time embedded system. Otherwise, the XESS plug-in for the inference engine ultimately chosen can be used to translate the knowledge base into the native language for that engine. The original XESS-version of the knowledge base can be kept and used as a base for translation should another inference engine be chosen at a later time (as opposed to scrapping the existing rule set and rewriting from scratch).

The XESS language is still in the experimental stages of its infancy. The initial version supports only the definition of a few forms of basic facts (predicates, structures, and instances) as well as fairly complex rules, but there are many features supported in various knowledge based systems that are not reflected in this version of the language. Still, XESS has room to grow. The XML schema definition is open for extension (as is intended by schemas)[29]; in fact, as of the writing of this paper the base schema has already been adapted for fuzzy logic including the definition of fuzzy variables and fuzzy rules; Figure 7.1 shows a subset of the ESSA knowledge base rewritten using the XESS Fuzzy Logic extensions. Additionally, the prototype XESS interpreter, written in Java, can be released as open source, inviting other contributors to modify and extend the API to handle newer versions of the XESS schema.

```
<xess>
  <!-- How many times per year is risk assessment performed? -->
  <fuzzyVariable name="risk-assessment" units="times per year">
    <universeOfDiscourse min="0.0" max="52.0"/>
    <fuzzyterm name="none">
      <point x="0.0" y="1.0"/>
      <point x="1.0 y="0.0"/>
    </fuzzyTerm>
    <fuzzyTerm name="few">
      <point x="1.0" y="1.0"/>
      <point x="6.0" y="0.0"/>
    </fuzzyTerm>
    <fuzzyTerm name="many">
      <point x="3.0" y="0.0"/>
      <point x="12.0" y="1.0"/>
    </fuzzyTerm>
  </fuzzyVariable>

  ...

  <!-- rules -->
  <compoundFuzzyRule name="risk-assessment-rule">
    <fuzzyRule>
      <if>
        <is name="risk-assessment" value="none"/>
      </if>
      <then>
        <set name="risk" value="very low"/>
      </then>
```

[29] J. Roy, A. Ramanujan, XML Schema Language; Taking XML to the Next Level

```
    </fuzzyRule>
    <fuzzyRule>
      <if>
        <is name="risk-assessment" value="few"/>
      </if>
      <then>
        <set name="risk" value="low"/>
      </then>
    </fuzzyRule>
    <fuzzyRule>
      <if>
        <is name="risk-assessment" value="many"/>
      </if>
      <then>
        <set name="risk" value="high"/>
      </then>
    </fuzzyRule>
  </compoundFuzzyRule>

  ...

</xess>
```

Figure 1: A subset of the ESSA knowledge base rewritten using the XESS fuzzy logic extensions.

The initial versions of XESS have been highly successful, as discussed in previous chapters. The merits of XESS and the fuzzy logic extensions have been recognized by the IEEE in the form of a peer-reviewed paper that was accepted for presentation at the 2007 IEEE International Conference on Fuzzy Systems in London, UK[30]. The implementation of the Expert System for Security Assessment shows that even in its infancy the system can capture fairly complex rules and generate conclusions and recommendations.

In conclusion, the XESS system shows that there is much potential in placing a layer of abstraction between the knowledge base and the expert system engine. The very first expert system shells were created precisely in support of this: they provided interpreted languages separate from the rules processing engine that allowed developers to express facts and rules without requiring them to embed such rules in the code of the engine itself, but the resulting languages were far to specific to a single engine to allow the rules to execute on any other engine. XESS takes the idea of separating the knowledge base from the rules engine one step farther, by allowing the same knowledge base to execute unmodified on any rules engine. This functionality has been demonstrated successfully, and the merits have been recognized by experts in the field.

[30] L. Reznik, R. St. Jacques, "Fuzzy Expert System Development with Computer Security Assessment Application"

Appendix A: The XESS Schema

This section contains the XESS schema in its entirety for reference purposes only. Earlier sections of this chapter contain detailed descriptions and examples of each of the entities defined in the XESS schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <!-- comments -->
  <xsd:element name="comment" type="xsd:string"/>

  <!-- clause elements -->
  <xsd:element name="clause" type="clauseType"/>
  <xsd:element name="greaterThan" type="greaterThanType" substitutionGroup="clause"/>
  <xsd:element name="greaterThanOrEqual" type="greaterThanOrEqualType"
     substitutionGroup="clause"/>
  <xsd:element name="lessThan" type="lessThanType" substitutionGroup="clause"/>
  <xsd:element name="lessThanOrEqual" type="lessThanOrEqualType"
     substitutionGroup="clause"/>
  <xsd:element name="equal" type="equalType" substitutionGroup="clause"/>
  <xsd:element name="notEqual" type="notEqualType" substitutionGroup="clause"/>
  <xsd:element name="between" type="betweenType" substitutionGroup="clause"/>
  <xsd:element name="notBetween" type="betweenType" substitutionGroup="clause"/>
  <xsd:element name="or" type="orType" substitutionGroup="clause"/>
  <xsd:element name="and" type="andType" substitutionGroup="clause"/>

  <xsd:complexType name="clauseType" abstract="true"/>

  <xsd:complexType name="greaterThanType">
    <xsd:complexContent>
      <xsd:extension base="clauseType">
        <xsd:attribute name="value1" type="xsd:string"/>
        <xsd:attribute name="value2" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="greaterThanOrEqualType">
    <xsd:complexContent>
      <xsd:extension base="clauseType">
        <xsd:attribute name="value1" type="xsd:string"/>
        <xsd:attribute name="value2" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="lessThanType">
    <xsd:complexContent>
      <xsd:extension base="clauseType">
        <xsd:attribute name="value1" type="xsd:string"/>
        <xsd:attribute name="value2" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="lessThanOrEqualType">
    <xsd:complexContent>
      <xsd:extension base="clauseType">
        <xsd:attribute name="value1" type="xsd:string"/>
        <xsd:attribute name="value2" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="equalType">
```

```
    <xsd:complexContent>
      <xsd:extension base="clauseType">
        <xsd:attribute name="value1" type="xsd:string"/>
        <xsd:attribute name="value2" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="notEqualType">
    <xsd:complexContent>
      <xsd:extension base="clauseType">
        <xsd:attribute name="value1" type="xsd:string"/>
        <xsd:attribute name="value2" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="betweenType">
    <xsd:complexContent>
      <xsd:extension base="clauseType">
        <xsd:attribute name="value" type="xsd:string"/>
        <xsd:attribute name="min" type="xsd:string"/>
        <xsd:attribute name="max" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="notBetweenType">
    <xsd:complexContent>
      <xsd:extension base="clauseType">
        <xsd:attribute name="value" type="xsd:string"/>
        <xsd:attribute name="min" type="xsd:string"/>
        <xsd:attribute name="max" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="orType">
    <xsd:complexContent>
      <xsd:extension base="clauseType">
        <xsd:sequence>
          <xsd:element ref="clause" minOccurs="2" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="andType">
    <xsd:complexContent>
      <xsd:extension base="clauseType">
        <xsd:sequence>
          <xsd:element ref="clause" minOccurs="2" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- fact types -->

  <xsd:element name="fact" type="factType"/>
  <xsd:element name="predicate" type="predicateType" substitutionGroup="fact"/>
  <xsd:element name="struct" type="structType" substitutionGroup="fact"/>
  <xsd:element name="instance" type="instanceType" substitutionGroup="fact"/>

  <xsd:complexType name="factType" abstract="true">
    <xsd:attribute name="name" type="xsd:string"/>
  </xsd:complexType>

  <xsd:complexType name="predicateType">
    <xsd:complexContent>
```

```
      <xsd:extension base="factType">
        <xsd:attribute name="value" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="structType">
    <xsd:complexContent>
      <xsd:extension base="factType">
        <xsd:sequence>
          <xsd:element ref="comment" minOccurs="0" maxOccurs="1"/>
          <xsd:element name="field" minOccurs="1" maxOccurs="unbounded">
            <xsd:complexType>
              <xsd:attribute name="name" type="xsd:string"/>
              <xsd:attribute name="type" type="xsd:string" use="optional"/>
              <xsd:attribute name="initialValue" type="xsd:string" use="optional"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="instanceType">
    <xsd:complexContent>
      <xsd:extension base="factType">
        <xsd:sequence>
          <xsd:element ref="comment" minOccurs="0" maxOccurs="1"/>
          <xsd:element name="field" minOccurs="0" maxOccurs="unbounded">
            <xsd:complexType>
              <xsd:attribute name="name" type="xsd:string"/>
              <xsd:attribute name="value" type="xsd:string" use="optional"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="type" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- main elements -->

  <xsd:element name="action" type="actionType"/>
  <xsd:element name="set" type="setType" substitutionGroup="action"/>
  <xsd:element name="runRule" type="runRuleType" substitutionGroup="action"/>

  <xsd:complexType name="actionType" abstract="true"/>

  <xsd:complexType name="setType">
    <xsd:complexContent>
      <xsd:extension base="actionType">
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="value" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="runRuleType">
    <xsd:complexContent>
      <xsd:extension base="actionType">
        <xsd:sequence>
          <xsd:element name="argument" minOccurs="0"
             maxOccurs="unbounded">
            <xsd:complexType>
              <xsd:attribute name="name" type="xsd:string"/>
              <xsd:attribute name="value" type="xsd:string" use="optional"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string"/>
```

```
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:element name="xess" type="xessType"/>

  <xsd:complexType name="xessType">
    <xsd:sequence>
      <xsd:element ref="comment" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="fact" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="rule" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="comment" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="parameter" minOccurs="0"
               maxOccurs="unbounded">
                 <xsd:complexType>
                       <xsd:attribute name="name" type="xsd:string"/>
                       <xsd:attribute name="type" type="xsd:string"/>
                 </xsd:complexType>
            </xsd:element>
            <xsd:element name="if">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element ref="clause"/>
                </xsd:sequence>
               </xsd:complexType>
            </xsd:element>
            <xsd:element name="then">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element ref="action"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
            <xsd:element name="else" minOccurs="0" maxOccurs="1">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element ref="action"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
          <xsd:attribute name="name" type="xsd:string" use="optional"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Appendix B: Expert System for Security Assessment

This section contains the example Security Evaluation Expert System written in the XESS language based and as described in chapter 5.

```
<xess>
  <!—
    facts: individual scores
  -->
  <!-- How many times per year is risk assessment performed (0-52)? -->
  <predicate name="risk_assessment" value="0.0"/>
  <!-- What percentage of systems are assessed and documented as of this time
    (0-100%)? -->
  <predicate name="risk_system_coverage" value="0.0"/>
  <!-- What percentage of systems have been tested for security controls in
    the past year (0-100%)? -->
  <predicate name="control_system_coverage" value="0.0"/>
  <!-- What percentage of systems were found to have at least one
    security control weakness (0-100%)? -->
  <predicate name="control_weaknesses" value="0.0"/>
  <!-- What percentage of systems are certified and accredited (0-100%)? -->
  <predicate name="certified_systems" value="0.0"/>
  <!-- What percentage of systems follow a documented security plan
    (0-100%)? -->
  <predicate name="documented_plan_systems" value="0.0"/>
  <!-- What percentage of systems divide sensitive functions among
    different individuals? -->
  <predicate name="divided_sensitive_functions" value="0.0"/>
  <!-- What percentage of users have undergone background screening
    (0-100%)? -->
  <predicate name="screened_users" value="0.0"/>
  <!-- What percentage of deposits and withdrawls of physical data tapes
    are logged (0-100%)? -->
  <predicate name="data_tapes_logged" value="0.0"/>
  <!-- How many of the following measures are taken to limit
    physical access to data lines (0-5)?
    (lock, keypad, biometrics, keycard, other) -->
  <predicate name="physical_access_restrictions" value="0.0"/>
  <!-- On what percentage of portable systems has encryption
    software been installed (0-100%)? -->
  <predicate name="portable_systems_encrypted" value="0.0"/>
  <!-- What percentage of critical systems have backup
    systems established (0-100%)? -->
  <predicate name="critical_systems_backup" value="0.0"/>
  <!-- What percentage of critical systems have a
    contingency plan (in case of failure) (0-100%)? -->
  <predicate name="critical_systems_contingency" value="0.0"/>
  <!-- What percentage of systems have restrictions on who
    performs maintenance/repairs (0-100%)? -->
  <predicate name="system_maintenance_restrictions" value="0.0"/>
  <!-- For what percentage of systems is maintenance activity
    logged (0-100%)? -->
  <predicate name="system_maintenance_logs" value="0.0"/>
  <!-- What percentage of software changes are documented
    and approved (0-100%)? -->
  <predicate name="software_changes_approved" value="0.0"/>
  <!-- What percentage systems were scanned for vulnerabilities
    last year (0-100%)? -->
  <predicate name="systems_scanned_for_vulnerabilities" value="0.0"/>
  <!-- What percentage of scanned systems needed to be patched (0-100%)? -->
  <predicate name="vulnerable_systems_patched" value="0.0"/>
  <!-- What percentage of systems use virus protection (0-100%)? -->
  <predicate name="systems_virus_protection" value="0.0"/>
  <!-- What percentage of virus protected systems use automated
    virus protection (0-100%)? -->
  <predicate name="systems_automated_virus_protection" value="0.0"/>
```

```
<!-- What percentage of virus protected systems are configured to
  update virus protection software automatically (0-100%)? -->
<predicate name="systems_automated_updates" value="0.0"/>
<!-- What percentage of systems are password protected (0-100%)? -->
<predicate name="systems_password_protected" value="0.0"/>
<!-- How many times per year are passwords required
  to be changed (0-365)? -->
<predicate name="passwords_changed" value="0.0"/>
<!-- What percentage of systems use default vendor passwords (0-100%)? -->
<predicate name="vendor_passwords_used" value="0.0"/>
<!-- What percentage of user IDs are shared by more than one person
  (0-100%)? -->
<predicate name="shared_user_ids" value="0.0"/>
<!-- How many of the following are required for passwords (0-5)?
  (upper case letters, lower case letters, numbers, symbols, other) -->
<predicate name="password_requirements" value="0.0"/>

<!—
  facts: categorical ratings: 0 (worst) - 100 (best)
-->
<predicate name="risk" value="0.0"/>
<predicate name="security_controls" value="0.0"/>
<predicate name="certification_planning" value="0.0"/>
<predicate name="personnel_security" value="0.0"/>
<predicate name="physical_protection" value="0.0"/>
<predicate name="contingency_planning" value="0.0"/>
<predicate name="maintenance" value="0.0"/>
<predicate name="data_integrity" value="0.0"/>
<predicate name="identification_authentication" value="0.0"/>
<predicate name="overall_security" value="0.0"/>

<!—
  rules: risk rating
-->
<rule name="risk_rule_001">
  <if>
    <and>
       <greaterThan name="risk_assessment" value="0.0"/>
       <lessThan name="risk_assessment" value="6.0"/>
     </and>
  </if>
  <then>
     <set name="risk" value="@risk+25.0"/>
  </then>
</rule>
<rule name="risk_rule_002">
  <if>
     <greaterThanOrEqual name="risk_assessment" value="6.0"/>
  </if>
  <then>
    <set name="risk" value="@risk+50.0"/>
  </then>
</rule>
<rule name="risk_rule_003">
  <if>
    <and>
       <greaterThanOrEqual name="risk_system_coverage" value="50.0"/>
       <lessThan name="risk_system_coverage" value="75.0"/>
    </and>
  </if>
  <then>
     <set name="risk" value="@risk+25.0"/>
  </then>
</rule>
<rule name="risk_rule_004">
  <if>
    <greaterThanOrEqual name="risk_system_coverage" value="75.0"/>
  </if>
  <then>
    <set name="risk" value="@risk+50.0"/>
```

```
    </then>
  </rule>

  <!–
    rules: control assessment
  -->
  <rule name="security_controls_rule_001">
    <if>
      <and>
        <greaterThanOrEqual name="control_system_coverage" value="50.0"/>
        <lessThan name="control_system_coverage" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="security_controls" value="@security_controls+25.0"/>
    </then>
  </rule>
  <rule name="control_assessment_rule_002">
    <if>
      <greaterThanOrEqual name="control_system_coverage" value="75.0"/>
    </if>
    <then>
      <set name="security_controls" value="@security_controls+50.0"/>
    </then>
  </rule>
  <rule name="control_assessment_rule_003">
    <if>
      <equal name="control_weaknesses" value="0.0"/>
    </if>
    <then>
      <set name="security_controls" value="@security_controls+50.0"/>
    </then>
  </rule>
  <rule name="control_assessment_rule_004">
    <if>
      <and>
        <greaterThan name="control_weaknesses" value="0.0"/>
        <lessThan name="control_weaknesses" value="3.0"/>
      </and>
    </if>
    <then>
      <set name="security_controls" value="@security_controls+25.0"/>
    </then>
  </rule>

  <!–
    rules: certification planning rating
  -->
  <rule name="certification_planning_rule_001">
    <if>
      <and>
        <greaterThan name="certified_systems" value="50.0"/>
        <lessThan name="certified_systems" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="certification_planning" value="@certification_planning+25.0"/>
    </then>
  </rule>
  <rule name="certification_planning_rule_002">
    <if>
      <greaterThanOrEqual name="certified_systems" value="75.0"/>
    </if>
    <then>
      <set name="certification_planning" value="@certification_planning+50.0"/>
    </then>
  </rule>
  <rule name="certification_planning_rule_003">
    <if>
      <and>
```

```
        <greaterThan name="documented_plan_systems" value="50.0"/>
        <lessThan name="documented_plan_systems" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="certification_planning" value="@certification_planning+25.0"/>
    </then>
  </rule>
  <rule name="certification_planning_rule_004">
    <if>
      <greaterThanOrEqual name="documented_plan_systems" value="75.0"/>
    </if>
    <then>
      <set name="certification_planning" value="@certification_planning+50.0"/>
    </then>
  </rule>

  <!—
    rules: personnel security rating
  -->
  <rule name="personnel_security_rule_001">
    <if>
      <and>
        <greaterThanOrEqual name="divided_sensitive_functions" value="0.0"/>
        <lessThan name="divided_sensitive_functions" value="40.0"/>
      </and>
    </if>
    <then>
      <set name="personnel_security" value="@personnel_security+50.0"/>
    </then>
  </rule>
  <rule name="personnel_security_rule_002">
    <if>
      <and>
        <greaterThanOrEqual name="divided_sensitive_functions" value="40.0"/>
        <lessThanOrEqual name="divided_sensitive_functions" value="80.0"/>
      </and>
    </if>
    <then>
      <set name="personnel_security" value="@personnel_security+25.0"/>
    </then>
  </rule>
  <rule name="personnel_security_rule_003">
    <if>
      <and>
        <greaterThanOrEqual name="screened_users" value="50.0"/>
        <lessThan name="screened_users" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="personnel_security" value="@personnel_security+25.0"/>
    </then>
  </rule>
  <rule name="personnel_security_rule_004">
    <if>
      <greaterThanOrEqual name="screened_users" value="75.0"/>
    </if>
    <then>
      <set name="personnel_security" value="@personnel_security+50.0"/>
    </then>
  </rule>

  <!—
    rules: physical protection rating
  -->
  <rule name="physical_protection_rule_001">
    <if>
      <and>
        <greaterThanOrEqual name="data_tapes_logged" value="50.0"/>
        <lessThan name="data_tapes_logged" value="75.0"/>
```

```
      </and>
    </if>
    <then>
      <set name="physical_protection" value="@physical_protection+16.5"/>
    </then>
  </rule>
  <rule name="physical_protection_rule_002">
    <if>
      <greaterThanOrEqual name="data_tapes_logged" value="75.0"/>
    </if>
    <then>
      <set name="physical_protection" value="@physical_protection+33"/>
    </then>
  </rule>
  <rule name="physical_protection_rule_003">
    <if>
      <and>
        <greaterThan name="physical_access_restrictions" value="0.0"/>
        <lessThan name="physical_access_restrictions" value="4.0"/>
      </and>
    </if>
    <then>
      <set name="physical_protection" value="@physical_protection+16.5"/>
    </then>
  </rule>
  <rule name="physical_protection_rule_004">
    <if>
      <greaterThanOrEqual name="physical_access_restrictions" value="4.0"/>
    </if>
    <then>
      <set name="physical_protection" value="@physical_protection+33"/>
    </then>
  </rule>
  <rule name="physical_protection_rule_005">
    <if>
      <and>
        <greaterThanOrEqual name="portable_systems_encrypted" value="50.0"/>
        <lessThan name="portable_systems_encrypted" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="physical_protection" value="@physical_protection+16.5"/>
    </then>
  </rule>
  <rule name="physical_protection_rule_006">
    <if>
      <greaterThanOrEqual name="portable_systems_encrypted" value="75.0"/>
    </if>
    <then>
      <set name="physical_protection" value="@physical_protection+33"/>
    </then>
  </rule>

  <!—
    rules: contingency planning rating
  -->
  <rule name="contingency_planning_rule_001">
    <if>
      <and>
        <greaterThanOrEqual name="critical_systems_backup" value="50.0"/>
        <lessThan name="critical_systems_backup" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="contingency_planning" value="@contingency_planning+25.0"/>
    </then>
  </rule>
  <rule name="contingency_planning_rule_002">
    <if>
      <greaterThanOrEqual name="critical_systems_backup" value="75.0"/>
```

```
    </if>
    <then>
      <set name="contingency_planning" value="@contingency_planning+50.0"/>
    </then>
  </rule>
  <rule name="contingency_planning_rule_003">
    <if>
      <and>
        <greaterThanOrEqual name="critical_systems_contingency" value="50.0"/>
        <lessThan name="critical_systems_contingency" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="contingency_planning" value="@contingency_planning+25.0"/>
    </then>
  </rule>
  <rule name="contingency_planning_rule_004">
    <if>
      <greaterThanOrEqual name="critical_systems_contingency" value="75.0"/>
    </if>
    <then>
      <set name="contingency_planning" value="@contingency_planning+50.0"/>
    </then>
  </rule>

  <!—
    rules: maintenance rating
  -->
  <rule name="maintenance_rule_001">
    <if>
      <and>
        <greaterThanOrEqual name="system_maintenance_restrictions"
                            value="50.0"/>
        <lessThan name="system_maintenance_restrictions" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="maintenance" value="@maintenance+10.0"/>
    </then>
  </rule>
  <rule name="maintenance_rule_002">
    <if>
      <greaterThanOrEqual name="system_maintenance_restrictions" value="75.0"/>
    </if>
    <then>
      <set name="maintenance" value="@maintenance+20.0"/>
    </then>
  </rule>
  <rule name="maintenance_rule_003">
    <if>
      <and>
        <greaterThanOrEqual name="system_maintenance_logs" value="50.0"/>
        <lessThan name="system_maintenance_logs" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="maintenance" value="@maintenance+10.0"/>
    </then>
  </rule>
  <rule name="maintenance_rule_004">
    <if>
      <greaterThanOrEqual name="system_maintenance_logs" value="75.0"/>
    </if>
    <then>
      <set name="maintenance" value="@maintenance+20.0"/>
    </then>
  </rule>
  <rule name="maintenance_rule_005">
    <if>
      <and>
```

```
        <greaterThanOrEqual name="software_changes_approved" value="50.0"/>
        <lessThan name="software_changes_approved" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="maintenance" value="@maintenance+10.0"/>
    </then>
  </rule>
  <rule name="maintenance_rule_006">
    <if>
      <greaterThanOrEqual name="software_changes_approved" value="75.0"/>
    </if>
    <then>
      <set name="maintenance" value="@maintenance+20.0"/>
    </then>
  </rule>
  <rule name="maintenance_rule_007">
    <if>
      <and>
        <greaterThanOrEqual name="systems_scanned_for_vulnerabilities"
                            value="50.0"/>
        <lessThan name="systems_scanned_for_vulnerabilities" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="maintenance" value="@maintenance+10.0"/>
    </then>
  </rule>
  <rule name="maintenance_rule_008">
    <if>
      <greaterThanOrEqual name="systems_scanned_for_vulnerabilities"
                          value="75.0"/>
    </if>
    <then>
      <set name="maintenance" value="@maintenance+20.0"/>
    </then>
  </rule>
  <rule name="maintenance_rule_009">
    <if>
      <and>
        <greaterThanOrEqual name="vulnerable_systems_patched" value="25.0"/>
        <lessThan name="vulnerable_systems_patched" value="50.0"/>
      </and>
    </if>
    <then>
      <set name="maintenance" value="@maintenance+10.0"/>
    </then>
  </rule>
  <rule name="maintenance_rule_010">
    <if>
      <lessThan name="vulnerable_systems_patched" value="25.0"/>
    </if>
    <then>
      <set name="maintenance" value="@maintenance+20.0"/>
    </then>
  </rule>

  <!—
    rules: data integrity rating
  -->
  <rule name="data_integrity_rule_001">
    <if>
      <and>
        <greaterThanOrEqual name="systems_virus_protection" value="50.0"/>
        <lessThan name="systems_virus_protection" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="data_integrity" value="@data_integrity+10.0"/>
    </then>
```

```
  </rule>
  <rule name="data_integrity_rule_002">
    <if>
      <greaterThanOrEqual name="systems_virus_protection" value="75.0"/>
    </if>
    <then>
      <set name="data_integrity" value="@data_integrity+20.0"/>
    </then>
  </rule>
  <rule name="data_integrity_rule_003">
    <if>
      <and>
        <greaterThanOrEqual name="systems_automated_virus_protection"
                            value="50.0"/>
        <lessThan name="systems_automated_virus_protection" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="data_integrity" value="@data_integrity+10.0"/>
    </then>
  </rule>
  <rule name="data_integrity_rule_004">
    <if>
      <greaterThanOrEqual name="systems_automated_virus_protection"
                          value="75.0"/>
    </if>
    <then>
      <set name="data_integrity" value="@data_integrity+20.0"/>
    </then>
  </rule>
  <rule name="data_integrity_rule_005">
    <if>
      <and>
        <greaterThanOrEqual name="systems_automated_updates" value="50.0"/>
        <lessThan name="systems_automated_updates" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="data_integrity" value="@data_integrity+10.0"/>
    </then>
  </rule>
  <rule name="data_integrity_rule_006">
    <if>
      <greaterThanOrEqual name="systems_automated_updates" value="75.0"/>
    </if>
    <then>
      <set name="data_integrity" value="@data_integrity+20.0"/>
    </then>
  </rule>
  <rule name="data_integrity_rule_007">
    <if>
      <and>
        <greaterThanOrEqual name="systems_password_protected" value="50.0"/>
        <lessThan name="systems_password_protected" value="75.0"/>
      </and>
    </if>
    <then>
      <set name="data_integrity" value="@data_integrity+10.0"/>
    </then>
  </rule>
  <rule name="data_integrity_rule_008">
    <if>
      <greaterThanOrEqual name="systems_password_protected" value="75.0"/>
    </if>
    <then>
      <set name="data_integrity" value="@data_integrity+20.0"/>
    </then>
  </rule>
  <rule name="data_integrity_rule_009">
    <if>
```

```
      <and>
        <greaterThan name="passwords_changed" value="0.0"/>
        <lessThanOrEqual name="passwords_changed" value="3.0"/>
      </and>
    </if>
    <then>
      <set name="data_integrity" value="@data_integrity+10.0"/>
    </then>
  </rule>
  <rule name="data_integrity_rule_010">
    <if>
      <and>
        <greaterThan name="passwords_changed" value="3.0"/>
        <lessThanOrEqual name="passwords_changed" value="6.0"/>
      </and>
    </if>
    <then>
      <set name="data_integrity" value="@data_integrity+20.0"/>
    </then>
  </rule>
  <rule name="data_integrity_rule_011">
    <if>
      <and>
        <greaterThan name="passwords_changed" value="6.0"/>
        <lessThanOrEqual name="passwords_changed" value="12.0"/>
      </and>
    </if>
    <then>
      <set name="data_integrity" value="@data_integrity+10.0"/>
    </then>
  </rule>

  <!—
    rules: identification/authentication rating
  -->
  <rule name="identification_authentication_rule_001">
    <if>
      <equal name="vendor_passwords_used" value="0.0"/>
    </if>
    <then>
      <set name="identification_authentication"
           value="@identification_authentication+33.0"/>
    </then>
  </rule>
  <rule name="identification_authentication_rule_002">
    <if>
      <and>
        <greaterThan name="vendor_passwords_used" value="0.0"/>
        <lessThan name="vendor_passwords_used" value="5.0"/>
      </and>
    </if>
    <then>
      <set name="identification_authentication"
           value="@identification_authentication+16.5"/>
    </then>
  </rule>
  <rule name="identification_authentication_rule_003">
    <if>
      <equal name="shared_user_ids" value="0.0"/>
    </if>
    <then>
      <set name="identification_authentication"
        value="@identification_authentication+33.0"/>
    </then>
  </rule>
  <rule name="identification_authentication_rule_004">
    <if>
      <and>
        <greaterThan name="shared_user_ids" value="0.0"/>
        <lessThan name="shared_user_ids" value="15.0"/>
```

```
      </and>
    </if>
    <then>
      <set name="identification_authentication"
           value="@identification_authentication+16.5"/>
    </then>
  </rule>
  <rule name="identification_authentication_rule_005">
    <if>
      <and>
        <greaterThan name="password_requirements" value="1.0"/>
        <lessThan name="password_requirements" value="4.0"/>
      </and>
    </if>
    <then>
      <set name="identification_authentication"
           value="@identification_authentication+33.0"/>
    </then>
  </rule>
  <rule name="identification_authentication_rule_006">
    <if>
      <between name="password_requirements" min="4.0" max="5.0"/>
    </if>
    <then>
      <set name="identification_authentication"
        value="@identification_authentication+16.5"/>
    </then>
  </rule>
</xess>
```

Bibliography

- Giarratano, Joseph C. and Riley, Gary D. Expert Systems: Principles and Programming. 5th Ed. Canada: Thomson Course Technology, 2005

- Sterling, Leon and Shapiro, Ehud. The Art of Prolog. 2nd Ed. Cambridge, Massachusetts: MIT Press, 1994

- Russell, Stuart and Norvig, Peter. Artificial Intelligence: A Modern Approach. Upper Saddle River, New Jersey: Pearson Education, Inc. 2003

- John F. Gilmore, Knowledge Base Systems in Computer Aided Technology, The 23rd IEEE Conference on Decision and Control pp. 586-590 vol. 23 part 1, 1984

- Du Zhang, Doan Nguyen, PREPARE: A Tool for Knowledge Base Verification, IEEE Transactions on Knowledge and Data Engineering, Volume 6, Issue 6, pp. 983-989, 1994

- Ronald Logsdon, XML White Paper, XML Workgroup, 2000

- J. Roy, A. Ramanujan, XML Schema Language; Taking XML to the Next Level, IT Professional, Volume 3, Issue 2 pp. 37-40, 2001

- Stephen Kirkham, XML – A Disruptive Influence on Programming Languages and Methodologies, XML Conference & Exposition, 2001

- Charles L. Forgy, Rete: A Fast Algorithm for Many Pattern/Many Object Pattern Match Problems, Expert Systems: A Software Methodology for Modern Applications, IEEE Computer Society Press, 1991

- Arne Bultman, Joris Kuipers, and Frank van Harmelen, Maintenance of KBS's by Domain Experts: The Holy Grail in Practice, 2005

- Jian Bing Li, James Miller, Testing the Semantics of W3C XML Schema, IEEE Computer Software and Applications Conference, pp. 443-448 vol. 2, 2005

- S.P. Bali, 2000 Solved Problems in Digital Electronics, Tata McGraw-Hill, 2005

- Sun Microsystems Inc., Java Code Conventions, Sun Microsystems Inc. Mountain View, CA, 1997

- Jim Veitch, "A History and Description of CLOS", Handbook of Programming Languages, Volume IV: Functional and Logic Programming Languages, Macmillan Computer Publishing 1998

- D.S. Kochnev, A.A. Terekhov, Surviving Java for Mobiles, IEEE Pervasive Computing, Volume 2, Issue 2, pp. 90-95, 2003

- Leon Reznik, Robert St. Jacques, Fuzzy Expert System Shell Development with Computer Security Assessment Application, IEEE International Conference on Fuzzy Systems, 2007

- Friedman-Hill, Ernest. JESS: The Rule Engine for the Java Platform. Albuquerque, New Mexico: Sandia National Laboratories. 2006

- Leslie B. Wilson, Robert G. Clark, Comparative Programming Languages: Third Edition, Addison Wesley, 2001

- Christopher Seiwald, Whitepaper: Seven Pillars of Pretty Code, Perforce Software

- Blaze Fundamentals: Blaze Advisor 6.0
  Minneapolis, Minnesota: Fair Isaac Corporation. November, 2005

- Advisor 6.0 Laboratory: Training Exercises
  Minneapolis, Minnesota: Fair Isaac Corporation. November, 2005

- Marianne Swanson, et al, Security Metrics Guide for Information Technology Systems, National Institute of Standards and Technology (NIST SP 800-55)

- Marianne Swanson, Security Self-Assessment Guide for Information Technology Systems, National Institute of Standards and Technology (NIST SP 800-26)

Internet References

- Wikipedia, Various Contributors. Prolog. January, 2007
  <http://www.wikipedia.org/wiki/Prolog>

- Riley, Gary. What is CLIPS? May 22, 2004
  <http://clipsrules.sourceforge.net/WhatIsCLIPS.html>

- Giarratano, Joseph, Ph.D. CLIPS User's Guide v 6.30. March, 2002
  <http://clipsrules.sourceforge.net/documentation/v630/ug.htm>

- CLIPS Home Page. CLIPS Reference Manual, Volume I: Basic Programming Guide, v 6.30. June, 2006
  <http://clipsrules.sourceforge.net/documentation/v630/bpg.htm>

- CLIPS Home Page. CLIPS Reference Manual, Volume II: Advanced Programming Guide, v 6.30. June, 2006
  <http://clipsrules.sourceforge.net/documentation/v630/apg.htm>

- Wikipedia, Various Contributors. CLIPS Programming Language. January, 2007 <http://www.wikipedia.org/wiki/CLIPS_programming_language>

- Friedman-Hill, Ernest. The JESS FAQ. December, 2006 <http://www.jessrules.com/jess/FAQ.shtml>

- National Research Council Canada, NRC FuzzyJ Toolkit, <http://www.iit.nrc.ca/IR_public/fuzzy/fuzzyJToolkit2.html>

- Edward Sazonov, Open Source Fuzzy Inference Engine for Java, <http://people.clarkson.edu/~esazonov/FuzzyEngine.htm>

- Fallside, David C. and Walmsley, Priscilla. XML Schema Part 0: Primer Second Edition. World Wide Web Consortium (W3C). October, 2004 <http://www.w3.org/TR/xmlschema-0>

- Thomson, Henry S., et al. XML Schema Part 1: Structures Second Edition. World Wide Web Consortium (W3C). October, 2004 <http://www.w3.org/TR/xmlschema-1>

- Biron, Paul V. and Malhotra, Ashok. XML Schema Part 2: Datatypes Second Edition. World Wide Web Consortium (W3C). October, 2004 <http://www.w3.org/TR/xmlschema-2>

- Bray, Tim, et al. Extensible Markup Language (XML) 1.0 (Fourth Edition). World Wide Web Consortium (W3C). August, 2006 <http://www.w3.org/TR/2006/REC-xml-20060816>

- Reagle, Joseph. XML Encryption Requirements. World Wide Web Consortium (W3C). March, 2002 <http://www.w3.org/TR/xml-encryption-req>

- Imamura, Takeshi, et al. XML Encryption Syntax and Processing. World Wide Web Consortium (W3C). December, 2002 <http://www.w3.org/TR/xmlenc-core>

- Reagle, Joseph. XML-Signature Requirements. World Wide Web Consortium (W3C). October, 1999 <http://www.w3.org/TR/xmldsig-requirements>

- Bartel, Mark, et al. XML-Signature Syntax and Processing. World Wide Web Consortium (W3C). February, 2002 <http://www.w3.org/TR/xmldsig-core>