CHAPTER – THREE

Knowledge Based Intelligent Intrusion Detection Multi-Agent System Design

Title

3.1 Introduction
3.2 Knowledge based Systems
3.3 Proposed IDPS Model
   3.3.1 Architecture of IDPS
   3.3.2 IDP model as Ruled-based Expert system
   3.3.3 Multi-agent IDPS Architecture
3.4 Expert System shell Used
   3.4.1 Java Expert System Shell (JESS)
   3.4.2 Architecture of a Java Expert system Shell (JESS)
   3.4.3 Integration of Java and Jess
3.5 Steps to Implementation of Model

Concluding Remark
PREVIEW

This chapter is divided into four primary sections. The first section provides an overview of issues of biometric system and intrusion detection fundamentals. The second section describes the architecture of rule based intrusion detection and prevention model. The third section provides how Jess is used as expert system shell to develop this model. Fourth section describes the steps to implementation of this model.

3.1 Introduction

Even though there are various advantages of biometric process, it is vulnerable to attacks, which can decline its security. The intrusion detection is a necessary supplement of traditional security protection measures such as firewalls, data encryption, because it can provide real protection against internal attacks, external attacks and abuse.

Intrusion detection system aim at detecting attacks against computer system and networks or in general against information systems. Indeed it is difficult to probably provide secure information systems and to maintain them in such secure state during their lifetime and utilization.

Knowledge based intrusion detection mechanism applies the knowledge accumulated about specific attacks and system vulnerabilities. The Intrusion
detection system contains information about these vulnerabilities and looks for attempts to exploit them. When such an attempt is detected, an alarm is raised. Therefore the accuracy of knowledge based intrusion detection systems is considered good. However, their completeness depends on the regular update of knowledge about attacks.

Advantage of this approach is to make it easier for security administrator using this intrusion detection system to understand the problem and to take preventive or corrective actions. But the difficulty to gather the relevant information about the known attacks and keeping it up to date with new vulnerabilities are the drawbacks of this approach.

3.2 Intrusion Detection and Prevention System

We can incrementally improve security through the use of tools such as Intrusion Detection Systems (IDS). The IDS approach to security is based on the assumption that a system will not be secure, but that violations of security policy (intrusions) can be detected by monitoring and analysing system behaviour.

Some specific examples of intrusions that concern system administrators include (Matt, 2003):

- Unauthorized modifications of system files so as to facilitate illegal access to either system or user information.
- Unauthorized access or modification of user files or information.
- Unauthorized modifications of tables or other system information in network components
- Unauthorized use of computing resources (perhaps through the creation of unauthorized accounts or perhaps through the unauthorized use of existing accounts).

Intrusion detection involves determining that some entity, an intruder, has attempted to gain, or worse, has gained unauthorized access to the system. None of the automated detection approaches of which we are aware seeks to identify an intruder before that intruder initiates interaction with the system. Intrusion detection systems are used in addition to such preventative measures. It is also assumed that intrusion detection is not a problem that can be solved once; continual vigilance is required. Anomaly detection is the general category of Intrusion Detection, which works by
identifying activities which vary from established patterns for users, or groups of users. Anomaly detection defines the normal behaviour of system. Behavior change from normal is considered as intrusion. Anomaly detection typically involves the creation of knowledge bases which contain the profiles of the monitored activities. Anomaly detection mechanisms are usually dependent on input from an operating system’s audit record. This analysis of the audit trail imposes potentially significant overhead requirements on the system because of the increased amount of processing power which must be utilized by the anomaly detector. Depending on the size of the audit trail and the processing ability of the system, the review of audit data could result in the loss of a real-time analysis capability.

Misuse Detection is a second approach to Intrusion Detection. This technique involves the comparison of a user’s activities with the known behaviours of attackers attempting to penetrate a system. Misuse Detection also utilizes a knowledge base of information. Misuse detection implements on the basis of pattern matching signature.

Like anomaly detection techniques, misuse detection systems suffer from the potential performance degradation which results from a dependency on audit trails for input. This disadvantage can be mitigated by improved system performance and reduced audit record sets.

Hybrid or Combined Anomaly/Misuse Detection is a third approach, which combine the Anomaly Detection approach and the Misuse Detection approach. The combined approach permits a single Intrusion Detection System to monitor for indications of external and internal attacks. Intrusion detection system which includes functions like monitoring and analysing both user and system activities and detect suspicious pattern (Pervez, et al., 2006).

While a significant advantage over the singular use of either method separately, the use of a combined anomaly/misuse detection mechanism does possess some disadvantages. The use of two knowledge bases for the intrusion detection system will increase the amount of system resources which must be dedicated to the system. Additional disk space will be required for the storage of the profiles, and increased memory requirements will be encountered as the mechanism compares user activities with information in the dual knowledge bases. In addition, the technique will share the disadvantage of either method individually in it's inability to detect collaborative or extended attack scenarios.
An intrusion-detection system can be described at a very macroscopic level as a detector that processes information coming from the system to be protected shown in figure 3.2.1. This detector can also launch probes to trigger the audit process. It uses three kinds of information: long-term information related to the technique used to detect intrusions (a knowledge base of attacks, for example), configuration information about the current state of the system, and audit information describing the events that are happening on the system.

All current Intrusion Detection Systems make four assumptions about the systems that they are designed to protect:

1. Activities taken by users, either authorized or unauthorized, can be monitored.
2. It is possible to identify those actions, which are indications of an attack on a system.
3. Information obtained from the Intrusion Detection System can be utilized to enhance the overall security of the network.
4. The system is able to make analysis of an attack in real-time.

A Rule-based is most of the widely used approach for intrusion detection systems. Such systems are built on a number of conditional if-then rules for their detection techniques. Rules are developed by analyzing attacks or misuses by experts and
then transferring them into conditional rules which are later used by inference modules of IDSs to compare against logs (monitoring data) to detect any misuse.

### 3.3 Knowledge based System

The knowledge based systems are systems based on the methods and techniques of Artificial Intelligence. Their core components are the knowledge base and the inference mechanisms. The scientific goal of Artificial intelligence is to understand intelligence by building computer programs that exhibit intelligent behavior. It is concerned with the concepts and methods of symbolic inference, or reasoning, by a computer, and how the knowledge is used to make those inferences will be represented inside the machine.

AI programs that achieve expert level competence in solving problems in task areas by bringing to bear a body of knowledge about specific tasks are called Knowledge based or expert systems. Often the term expert systems are reserved for programs whose knowledge base contains the knowledge used by human experts. More often, the two terms expert systems (ES) and knowledge based systems (KBS) are used alternatively.

The Knowledge based system consists of two principal parts: the knowledge base; and the reasoning or inference engine. The knowledge base of this type of systems contains both factual knowledge and heuristics knowledge. Factual knowledge is that knowledge of the task domain that is widely shared, typically found in books, journals, conference proceedings, reports etc. and commonly agreed upon by those knowledgeable in the particular field.

The development of a Knowledge based system is the most time consuming and painstaking job. The study depends on source of knowledge; hence structuring the more relevant knowledge to a particular problem is a distinguishing characteristic of an expert system. Hence task of structuring knowledge or constructing a knowledge base or in short knowledge engineering entertains knowledge acquisition, knowledge representation and application of knowledge.

- **Knowledge acquisition**: It can be defined as the transparent and transformation of potential problem-solving expertise from some knowledge source to a program.

- **Knowledge Representation**: The way in which information might be stored in human brain and the way it can be formally described for the purpose of
computation of knowledge are logical adequacy, heuristic power and notational convenience.

- **Application of Knowledge:** This sub field relates to the issue of planning and control in the field of problem solving expert systems design and involves paying close attention to the details of how knowledge is accessed and applied during the search for a solution.

When all the knowledge has been collected, the task of programming the system begins. The best first step is to examine the knowledge and design data structures that will make it easy to implement the rules clearly and directly. From the literature review it is clear that the Knowledge Base is the most integral part of an expert system since it addresses a specific problem with the specific set of rules. Hence developing an expert system mainly entails creation of Knowledge Base or Rule Base. Rule Base is also referred to as Knowledge Base.

To build the knowledge based systems two ways are available. One way is they can be built from scratch and another way is they can be built using a piece of development software known as a ‘tool’ or a ‘shell’. Building knowledge based systems by using shells offers significant advantages. A system can be built to perform a unique task by entering into a shell all the necessary knowledge about the task domain. The inference engine that applies the knowledge to the task at hand is built into the shell. Here authors have used Java Expert system Shell (JESS) to build the proposed knowledge based system.

### 3.4 Proposed Intrusion Detection and Prevention (IDP) Model

#### 3.4.1 Architecture of IDPS

The intrusion detection technology is the process of identifying network activity that can lead to compromise of security policy. IDS must analyse and correlate a large volume of data collected from different critical network access points. This task requires an IDS to be able to characterise distributed patterns and to detect situation where a sequence of multiple events occurs in System. Distributed IDS architecture based on agent paradigm. Based on multiple independent entities called autonomous agent for intrusion detection framework where the data is collected from different sources (Benattou & K.Tamine, 2005).
All current intrusion detection systems make four assumptions about the systems that they are designed to protect:

- Activities taken by system users, either authorized or unauthorized, can be monitored.
- It is possible to identify those actions which are indications of an attack on a system.
- Information obtained from the intrusion detection system can be utilized to enhance the overall security of the network.
- A fourth element which is desirable from any intrusion detection mechanism is the ability of the system to make an analysis of an attack in real-time.

To design robust security system, it performs the objectives of security like authenticity, confidentiality, integrity, availability and non-repudiation. IDPS (Intrusion detection and Prevention System) contains modules to detect intrusion, filtering intrusion, trace back of intrusion origin, and prevention mechanism for theses intrusions.

This security system needs the robust automated auditing and intelligent reporting mechanism and robust prevention techniques. The authors suggest security system using intelligent models for biometric protection approach.

This system is divided into 3 processes those are

- Intrusion detection
- Backtracking of intrusion source
- Prevention techniques

The Rule based intelligent intrusion detection and prevention model for biometric system contains detectors to detect normal or abnormal activity by comparing activity database. If activity is normal then standard alarming and reporting would be executed. If abnormal activity is found then the rule engine checks the rule to detect intrusion point and type of intrusion. The model also contains an expert system to detect source of intrusion and suggests best possible prevention technique and suitable controls for different intrusions.

With the help of Knowledge Base the inference engine reports the solution to the user along with the reasoning. The stored expertise about a problem area can be represented as a rule set or rule base.
A rule-based program can have hundreds or even thousands of rules. Often the rules represent the heuristic knowledge of a human expert in some domain, and the knowledge base represents the state of an evolving situation. In this proposed model authors collected knowledge which is available in literature like journal papers, Conference proceedings, Technical reports, books etc.

An expert system consists of a set of rules which encode the knowledge of a human "expert". These rules are used by the system to make conclusions about the data which is returned by the intrusion detection system. Expert systems permit the incorporation of an extensive amount of human experience into a computer application which utilizes that information to identify activities which matched the defined characteristics of misuse and attack. Unfortunately, expert systems require frequent updates by a system administrator to remain current. While expert systems offer an enhanced ability to review audit data, the frequently required updates may be ignored or performed infrequently by the administrator. At a minimum, this leads to an expert system with reduced capabilities. At worst, this lack of maintenance will degrade the security of the entire system by causing the system's users to be mislead into believing that the system is secure, even as one of the key components becomes increasingly ineffective over time. The components of the intrusion detection system are shown in figure 3.4.1.1.

![Figure 3.4.1.1: Components of Intrusion Detection Process](image_url)

This model also uses security audit as well as alarming and reporting mechanisms. The malicious activity database is stored for future intrusion detection. To detect the
source by tracking, backward chaining approach is used. The rules are defined and are stored in the Rule engine of the system. Intrusion points and type is passed to expert system shown in figure 3.4.1.2.

![Diagram](image)

**Figure 3.4.1.2 : Components of backtracking of intrusion source**

Expert system evaluates that data with known malicious activity database and detects the source using backward chaining approach. Backward-chaining systems employ the reverse strategy; starting from a proposed hypothesis they proceed to collect supportive evidence. Backward chaining systems are typically applied to problems of diagnosis.

### 3.4.2. IDP model as Ruled-based Expert system

Expert systems are the most common form of AI applied today in intrusion detection system. Expert systems are rule-based computer programs that capture the knowledge of human experts in their own fields of expertise, were a success story for artificial intelligence research in the 1970s and 1980s.

Rule-Based Systems rely on sets of predefined rules which are provided by an administrator, automatically created by the system, or both. Each rule is mapped to a specific operation in the system. The rules serve as operational preconditions which are continuously checked in the audit record by the intrusion detection mechanism. If the required conditions of a rule are satisfied by user activity the specified operation is executed (J.Page, Heane, Adkins, & Dolsen, 1989).
Rule-based expert systems have played an important role in modern intelligent systems and their applications in fault monitoring, diagnosis and so on. Conventional rule-based expert systems use human expert knowledge to solve real-world problems that normally would require human intelligence. Expert knowledge is often represented in the form of rules, or as data within the computer. Knowledge representation in expert systems may be rule-based or encapsulated in objects. The rule-based approach uses IF-THEN type rules and it is the method currently used in constructing expert systems. IF-THEN rules take the following form:

\[ \text{IF there is a flame THEN there is a fire} \]

The modern rule-based expert systems are based on the Newel and Simon model of human problem solving in terms of long-term memory (rules), short-term memory (working memory) and cognitive processor (inference engine). A knowledge-based system may be dependent on the knowledge commonly available; a true ‘expert’ system will be based on unwritten expertise, acquired from a human expert. In the conditions where no algorithm is available to solve a particular problem, a reasonable solution is the best we can expect from an expert (system or human).

The expert system will infer a solution from the facts provided by the user and the rules in the knowledge base. Therefore, it should be able to explain the reasoning employed to achieve the solution.

These rules are used by the system to make conclusions about the security-related data from the intrusion detection system. Expert system permits the incorporation of an extensive amount of human experience into a computer application and then utilizes that knowledge to identify activities that match the defined characteristics of misuse and attack. Expert system detects intrusions by encoding intrusion scenarios as a set of rules. These rules replicate the partially ordered sequence of actions that include the intrusion scenario. Some rules may be applicable to more than one intrusion scenario.

Rule-based programming is one of the most commonly used techniques for developing expert systems. Rule based analysis relies on sets of predefined rules that can be repeatedly applied to a collection of facts and that are provided by an administrator, automatically created by the system or both. Facts represent conditions that describe a certain situation in the audit records or directly from system activity monitoring and rules represent heuristics that define a set of actions to be
executed in a given situation and describe known intrusion scenario(s) or generic techniques. The rule then fires. It may cause an alert to be raised for a system administrator.

Alternatively, some automated response, such as terminating that user's session, block user's account will be taken. Normally, a rule firing will result in additional assertions being added to the fact base. They in turn, may lead to additional rule-fact bindings. This process continues until there are no more rules to be fired.

Consider the intrusion scenario in which two or more unsuccessful authentication attempts are made in a period of time shorter than it would take a human to present biometric info in the login information at biometric sensor. If the rule or rules for this scenario fire, then suspicion level of specific user can get increased. The system may raise an alarm or report ‘freeze action’ to the named user's account. Account freeze would be entered into the fact database.

Expert system module categorizes the audit data by fact base component initially and then uses relevant detection technique for different audit data. Rules can be defined using JESS. Jess is a clone of the popular expert system shell CLIPS, rewritten entirely in Java.

If biometric template is stored in central database, alteration and deletion of biometric template is not allowed to any user except root or system or super user for database administration purpose. The attacker modifies or deletes the biometric template. The rule which is to be checked for unauthorized modification of biometric template is:

```
If ((user is "root" || "superuser" || "system")&&(transaction_type is "Modification")&&
(not (time_stamp is normaltime_stamp)))
Then (Alert: “Unauthorized Modification”)
```

**Table 3.4.2.1. Sample rule to detect unauthorized Modification**

The rule which is to be checked for unauthorized deletion of biometric template is:

```
If ((user is "root" || "superuser" || "system")&&(transaction_type is "Deletion")&&
(not (time_stamp is normaltime_stamp)))
Then (Alert: “Unauthorized Deletion”)
```

**Table 3.4.2.2. Sample rule to detect unauthorized Deletion**
The imposter steals the biometric template of an authorized user from template storage or from other biometric system. The rule which is to be checked for illegal copy of biometric template is:

```
If (Transcation_type is “Copy”)
Then (Alert: “Access Denied”)
```

### Table 3.4.2.3. Sample rule to detect unauthorized copy

A security attack or intrusion can be defined like any action or set of actions which can violate the security of a system and tries to compromise the confidentiality.

#### 3.4.3. Multi-agent IDPS Architecture

The concept of Distributed Artificial intelligence (DAI) was defined, at the beginning of the Seventies, to find solutions to specific AI problems. Traditional AI concept deliberates intelligence within a single system. This involves some difficulties because of the need for integrating, within a same base of knowledge, expertise, competencies and knowledge of different individuals who, in reality, communicate and collaborate in the realization of a common goal. The purpose of DAI is to extend the AI field in order to distribute the intelligence among several agents not subject to a centralized control.

The agent is a program module that functions continuously in a particular environment. It is able to carry out activities in a flexible and intelligent manner that is responsive to change in the environment (real or virtual). An agent is able to learn from its experiences. The agent is autonomous. It takes action based on its built-in knowledge and its past experiences.

Based on Franklin and Graesser framework (Franklin & Graesser, 1996) the main agent’s properties are as follows:

- **Reactive**: able to act and respond at time to changes in its environment.
- **Autonomous**: proactive and can take decision and exercises control over its own actions.
- **Temporally continuous**: persistence of identity and is a continuously running process.
- **Communicative**: able to communicate with other agents.
- **Cooperative**: able to collaborate with other agents to achieve a common goal.
- **Adaptive**: able to learn and changes its behaviour with experience.
- **Mobile**: able to migrate from one machine to another.

The multi-agent system is a system that consists of multiple agents that can interact together to learn or to exchange experiences jointly to take actions or to solve problems (Hegazy, Al-Arif, Fayed, & Faheem, 2003).

A multi-agent system (MAS) can be defined as a computational system composed by more than one intelligent software agent. An intelligent software agent (ISA) uses Artificial Intelligence (AI) modern approach to interact with the environment, perceiving and acting autonomously over it to achieve defined goals. Thus, a multi agent system is a system, where many agents interact with the environment in a cooperative or competitive way to achieve individual or group objectives (d'Inverno & Luck., 2004).

A multi-agent system is designed and implemented as several interacting agents. Multi-agent systems are ideally suited to representing problems that have multiple problem solving methods and multiple perspectives. Intelligent agents take initiative where appropriate, and socially interact, where appropriate, with other artificial agents and humans in order to complete their own problem solving and to help others with their activities.

Artificial Intelligence techniques enhance agent capabilities. Intelligent agents and multi-agent systems are among the most rapidly growing areas of research and development.

Agents can be classified into five categories as, simple reflex agents, model-based reflex agents, goal-based agents, utility-based agents, and learning agents. The authors consider the simple reflex agent, which distinguishes the input from its environment. The simple reflex agent function is based on the condition-action rule: if condition then action.

Agents trace intruders and collect input data that is related only to the intrusion along the intrusion route. Building the IDS using the agent technology has several advantages. As agents are running separately; those can be added or removed from the system without altering other agents. The agents can be reconfigured or upgraded to newer versions without disturbing other agents.
Different Agents deployed on different locations are shown in 3.4.3.1.

![Diagram of different agents deployed on different locations]

**Figure 3.4.3.1: Different Agents deployed on different locations**

The authors design Multi-agent Intrusion Detection model which contains three agents. Implementation details of those agents are as follows:

- **Agent 1**
  This intelligent Agent can be implemented on biometric template database. Here we consider biometric template database store in central repository system. It performs intrusion detection using Operating System’s audit trail, and RDBMS audit trail. The system consists of a user interface module, an inference engine, a knowledgebase of illegal transactions and audit trail of ORACLE database.

- **Agent 2**
  This intelligent agent can be deployed on biometric System where Feature Extraction and Matching (Decision) modules are stored. Plenty of IDS/IPS are already available to detect the computer system and network attacks. Few examples are TripWire, Snort (open source and rule based), Symantec Network Security SecureNet, iPolic, eTrust Intrusion Detection, Cisco IPS

- **Agent 3**
  This intelligent agent can be located on the Biometric device. It performs intrusion detection using Operating System’s audit trail and device manager information. The system consists of a user interface module, an inference...
engine, a knowledgebase of illegal transactions and certified biometric devices and status of liveness detection.

These agents are developed using Java, Jess and integration of both. The user interface is developed in Java and rules are developed in Jess. These agents, equipped with a Jess rule engine and a knowledge base (KB) that contains environment knowledge, behaviour patterns and strategies/policies. By altering the knowledge base, the agent’s knowledge and/or behaviour is modified accordingly.

3.5 Expert System Shell Used

3.5.1 Java Expert System Shell (JESS)

Expert systems can be developed with Expert System Shells. An expert system shell is a software programming environment which enables the construction of expert or knowledge based systems.

Jess functions as an inference engine, a task that includes processing the products knowledge base, drawing conclusions, and preparing questions (query) for the user in response to user input.

Jess, the Java Expert System Shell is a general-purpose rule engine, developed at Sandia National Laboratories. Written in the Java programming language, Jess offers easy integration with other Java-based software. Jess is a rule-based language for specifying expert systems. The Jess engine can be invoked as an interactive interpreter, where Jess language strings can be typed into a shell and invoked in real-time, or in batch mode, where one or multiple files of Jess code can be executed at once. The Jess engine is implemented in Java, as well as the shell or interpreter mode, it can also be invoked from Java code at runtime. Jess code is able to call other Java code, or be executed in a Java object.

Jess has been used to develop a broad range of commercial software, including:

- Expert systems that evaluate insurance claims and mortgage applications
- Agents that predict stock prices and buy and sell securities
- Network intrusion detectors and security auditors
- Design assistants that help mechanical engineers
- Smart network switches for telecommunications
- Servers to execute business rules
- Intelligent e-commerce sites
Games

JESS was originally a clone of essential core of CLIPS. A CLIPS (C Language Integrated Production System) is an expert system tool developed by the Software Technology Branch (STB) at the NASA/ Lyndon B. Johnson Space centre. It was released in 1986 for the first time and has undergone continual refinement and improvement ever since.

3.5.2 Architecture of a Java Expert System Shell (JESS)

An expert system shell is just the inference engine and other functional parts of an expert system with all the domain-specific knowledge removed. Most modern rule engines can be seen as more or less specialized expert system shells, with features to support operation in specific environments or programming in specific domains. A typical rule engine contains:

- An inference engine
  - A pattern matcher
  - An agenda
  - An execution engine
- A rule base
- A working memory

These components are shown schematically in figure 3.5.2.1.

![Figure 3.5.2.1: JESS Architecture Diagram](image_url)

**Figure 3.5.2.1: JESS architecture Diagram**

- **The inference engine**
  The primary job of a rule engine is to apply rules to data. The inference engine is the central part of a rule engine. The inference engine controls the whole process of applying the rules to the working memory to obtain the
outputs of the system. Usually an inference engine works in discrete cycles that go something like this:

i. All the rules are compared to working memory (using the pattern matcher) to decide which ones should be activated during this cycle. This unordered list of activated rules, together with any other rules activated in previous cycles, is called the conflict set.

ii. The conflict set is ordered to form the agenda. The agenda is the list of rules whose right-hand sides will be executed, or fired. The process of ordering the agenda is called conflict resolution.

iii. To complete the cycle, the first rule on the agenda is fired (possibly changing the working memory) and the entire process is repeated. This repetition implies a large amount of redundant work, but many rule engines use sophisticated techniques to avoid most or all of the redundancy. In particular, results from the pattern matcher and from the agenda’s conflict resolver can be preserved across cycles, so that only the essential, new work needs to be done.

- **The pattern matcher**
  The inference engine has to decide what rules to fire, and when. The purpose of the pattern matcher is to decide which rules apply, given the current contents of the working memory. In general, this is a hard problem. If the working memory contains thousands of facts, and each rule has two or three premises, the pattern matcher might need to search through millions of combinations of facts to find those combinations that satisfy rules. Each conditional element from the list of conditions is matched against all possible facts, and if all elements of the list do have an assigned fact, then the rule is active.

- **The agenda**
  Once the inference engine figures out which rules should be fired, it still must decide which rule to fire first. The list of rules that could potentially fire is stored on the agenda. The agenda is responsible for using the conflict strategy to decide which of the rules, out of all those that apply, have the highest priority and should be fired first. Again, this is potentially a hard problem, and each rule engine has its own approach. Commonly, the conflict strategy might take into account the specificity or complexity of each rule and the relative age of the premises in the working memory. Rules may also have
specific priorities attached to them, so that certain rules are more important and always fire first.

- **The execution engine**
  Finally, once the rule engine decides what rule to fire, it has to execute that rule’s action part. The execution engine is the component of a rule engine that fires the rules. The engine will choose one of the active rules and process its actions. This can (and usually will) change the defined set of facts, causing other rules to be activated or deactivated. If there are no more active rules, the engine may halt automatically or wait for the set of facts to change by external means.

- **The rule base**
  The rule engine will obviously need to store rules somewhere. A rule is a mapping from one list of conditions to one list of actions. The rule base contains all the rules the system knows. They may simply be stored as strings of text, but most often a rule compiler processes them into some form that the inference engine can work with more efficiently. Jess’s rule compiler builds a complex, indexed data structure called a Rete network. A Rete network is a data structure that makes rule processing fast.

- **The working memory**
  It needs to store the data which rule engine will operate on. In a typical rule engine, it is the working memory, sometimes called the fact base. A fact is much like a database record; it consists of a number of named slots, which would be stored in the columns of a table. The working memory can hold both the premises and the conclusions of the rules. Typically, the rule engine maintains one or more indexes, similar to those used in relational databases, to make searching the working memory a very fast operation.

The pattern matcher applies the rules in the rule-base to the facts in working memory to construct the agenda. The execution engine fires the rules from the agenda, which changes the contents of working memory and restarts the cycle.

A rule-based system maintains a collection of knowledge nuggets called **facts**. This collection is known as the **knowledge base**. It is somewhat akin to a relational database, especially in that the facts must have a specific structure. In Jess, there are three kinds of facts: **ordered facts**, **unordered facts**, and **definstance facts**. You can add ordered facts to the knowledge base using the assert function. You can see
a list of all the facts in the knowledge base using the facts command. You can completely clear Jess of all facts and other data using the clear command.

Ordered facts are useful, but they are unstructured. Most of the time you need a bit more organization. In object-oriented languages, objects have named fields in which data appears.

Unordered facts offer this capability (although the fields are traditionally called slots). Before you can create unordered facts, you have to define the slots they have using the deftemplate syntax as is follows:

```
(deftemplate <deftemplate-name> [extends <classname>] [<doc-comment>] [[(slot <slot-name> [(default | default-dynamic <value>))] [(type <typespec>))]*]
```

**Table 3.5.2.1: Syntax of Deftemplate**

The `<deftemplate-name>` is the head of the facts that will be created using this template. There may be an arbitrary number of slots. Each `<slot-name>` must be an atom. The default slot qualifier states that the default value of a slot in a new fact is given by `<value>`; the default is the atom nil. The ‘default-dynamic’ version will evaluate the given value each time a new fact using this template is asserted. The ‘type’ slot qualifier is accepted but not currently enforced by Jess; it specifies what data type the slot is allowed to hold. Acceptable values are ANY, INTEGER, FLOAT, NUMBER, ATOM, STRING, LEXEME, and OBJECT.

Now that we’ve learned how to develop a knowledge base, we can answer the obvious question: what is it good for? The answer is that queries can search it to find relationships between facts, and rules can take actions based on the contents of one or more facts. As soon as we enter the run command, the activated rule fires. A rule will be activated only once for a given set of facts; once it has fired, that rule will not fire again for the same list of facts.

All Jess rules are defined using the defrule construct. Syntax of Defrule is as follows:
Table 3.5.2.2: Syntax of Defrule

Some rule-based systems, notably Prolog and its derivatives, support **backward chaining**. In a backward chaining system, rules are still if... then statements, but the engine seeks steps to activate rules whose preconditions are not met. This behaviour is often called "goal seeking". Jess supports both forward and backward chaining, but Jess’s version of backward chaining is not transparent to the programmer.

You have to declare which kinds of facts can serve as backward-chaining triggers, and only specific rules you define can be used in backward chaining. In truth, Jess’s reasoning engine is strictly a forward-chaining engine, and so backward chaining is effectively simulated in terms of forward-chaining rules. Still, the simulation is quite effective, and Jess’s backward-chaining mechanism has many useful applications. Backward chaining is often used as a way to pull required data into Jess’s working memory from a database on demand.

A Defquery is a way of requesting specific information in working memory from procedural code. These are usually constructed by the parser when it sees a "defquery" construct. It can be convenient to use queries as triggers for backward chaining. Syntax of defquery is as follows:

<table>
<thead>
<tr>
<th>Table 3.5.2.3: Syntax of Dequery</th>
</tr>
</thead>
<tbody>
<tr>
<td>(defquery query-name</td>
</tr>
<tr>
<td>[&quot;Documentation comment&quot;]</td>
</tr>
<tr>
<td>[(declare (variables variable+)</td>
</tr>
<tr>
<td>(node-index-hash value)</td>
</tr>
<tr>
<td>(max-background-rules value))]</td>
</tr>
<tr>
<td>(conditional element)* )</td>
</tr>
</tbody>
</table>

(defrule rule-name
 ["Documentation comment"]
 [(declare (salience value)
 (node-index-hash value)
 (auto-focus TRUE | FALSE)
 (no-loop TRUE | FALSE))]
 (conditional element)* =) (function call)* )
The defquery construct lets you create a special kind of rule with no right-hand-side. While rules act spontaneously, queries are used to search the knowledge base under direct program control. A rule is activated once for each matching set of facts, while a query gives you a java.util.Iterator of all the matches.

While normal rules act spontaneously, queries are used to search the working memory under direct program control. Whereas a rule is activated once for each matching set of facts, a query gives you a jess.QueryResult object which gives you access to a list of all the matches.

Using a defquery involves three steps:

i. Writing the query
   A query looks a lot like the left-hand side of a rule. We write a pattern which matches the facts that we're interested in.

ii. Invoking the query
   After we define a query, we can call it using the method in java like jess.Rete.runQueryStar(java.lang.String,jess.ValueVector) or the 'run-query*' function in Jess. In both cases, we need to pass the the query parameter we're interested in.

iii. Using the results
   Now that we've created a jess.QueryResult, it's time to iterate over all the matches and process them however we'd like.

It can be convenient to use queries as triggers for backward chaining. For this to be useful, jess.Rete.run() must be called while the query is being evaluated, to allow the backward chaining to occur. Facts generated by rules fired during this run may appear as part of the query results. To obtain just the number of matches for a query, you can use the count-query-results function. This function accepts the same arguments as run-query*, but just returns an integer, the number of matches.

By default, no rules will fire while a query is being executed. If you want to allow backward chaining to occur in response to a query,

Sometimes you may find that a particular rule should be treated as a special case by the conflict-resolution strategy. A rule that reports a security breach might need to fire immediately, regardless of what else is on the agenda. On the other hand, a rule that cleans up unused facts might only need to run during the idle time when no other
rules are activated. You can tell the conflict resolver to treat these rules specially using \textit{rule salience}. Each rule has a property called saliency that acts as a priority setting for that rule. Activated rules of the highest saliency always fire first, followed by rules of lower saliency (Hill, 2003) and (Strauss, 2007).

The rules of jess allow one to build systems. However these facts and rules cannot capture any uncertainty or ambiguity which is present in the domain. But extension of Jess that allows some form of uncertainty to be captured and represented using fuzzy sets and fuzzy reasoning. The NRC FuzzyJ Toolkit can be used to create Java programs that encode fuzzy operations and fuzzy reasoning.

The FuzzyJ Toolkit provides a capability for modelling fuzzy concepts and reasoning in a Java setting. Much of the work is based on earlier experience with the FuzzyCLIPS (Orchard 1998) extension to the CLIPS Expert System Shell (Riley 2001).

From the Jess language perspective there is very little added. In fact there is an implementation of the Jess Userpackage interface, the FuzzyFunctions class that adds just three fuzzy functions to Jess. These functions are \textit{fuzzy-match}, \textit{fuzzy-rule-similarity} and \textit{fuzzy-rule-match-score}. The latter two are beyond our scope. The fuzzymatch function takes two arguments: either both FuzzyValue objects or a FuzzyValue object and a string that represents a valid fuzzy expression. If one of the arguments is a string then it will be converted to a FuzzyValue using the FuzzyVariable associated with the other FuzzyValue argument. FuzzyVariable objects must both be associated with the same FuzzyVariable so that they can be compared. If there is some degree of match between the two, then the fuzzy-match function returns true, otherwise it returns false. FuzzyJess does provide a great deal more flexibility in the fuzzy patterns and does not require internal changes to any Jess parsing routines. Also, when fuzzy facts are asserted in the rules, FuzzyJess automatically takes care of the \textit{global contribution} issue. As identical fuzzy facts are asserted from different rules the contribution from each rule is accumulated.

There are a couple of other things that a user must know to use the FuzzyJess extension. You need to have access to the FuzzyJ Toolkit and FuzzyJess packages (nrc.fuzzy and nrc.fuzzy.jess). Normally these will be in a Java jar file for easy inclusion in the \textit{classpath} variable. The only other thing that is required is that instead of using a \textbf{Rete} object in programs, you must use a \textbf{FuzzyRete} object. For convenience the classes nrc.fuzzy.jess.FuzzyConsole and nrc.fuzzy.jess.FuzzyMain
have been provided and they can simply replace any use of jess.Console or jess.Main.

However, a rule based expert system shell (Jess) provides a convenient and suitable way to encode many types of applications. Fuzzy logic programs fit nicely into the rule based paradigm. An integration of the FuzzyJ Toolkit and Jess is FuzzyJess. FuzzyJess provides a great deal more flexibility in the fuzzy patterns and does not require internal changes to any Jess parsing technique (Orchard, 2001).

For simple systems and for performance purposes creating fuzzy systems using only the FuzzyJ Toolkit may be appropriate. But as systems grow larger and the number and type (fuzzy, crisp, fuzzy-crisp) of rules make the system more complex, FuzzyJess may have an advantage since it could be more robust and likely will be easier to maintain. Certainly hybrid approaches of Java and FuzzyJess will also be used in many applications.

### 3.5.3 Rete Algorithm

The performance of the simple but inefficient pattern-matching algorithm can be improved by thinking about the source of its inefficiency. The typical rule based system has a more or less fixed set of rules, whereas the working memory changes continuously. However, it is an empirical fact that in most rule based systems, much of the working memory is also fairly fixed over time. Although new facts arrive and old ones are removed as the system runs, the percentage of facts that change per unit time is generally fairly small.

The rules finding facts algorithm is therefore unnecessarily inefficient, because most of the tests made on each cycle will have the same results as on the previous iteration.

An algorithm that could somehow remember previous pattern-matching results between cycles, only updating matches for facts that actually changed, could do far less work and get the same results.

Jess uses a very efficient version of this idea, known as the **Rete algorithm**. Rete is Latin for net (it’s pronounced “ree-tee”). The Rete algorithm is implemented by building a network of interconnected nodes.
Charles Forgy's (Forgy, 1982) classic paper describing the Rete algorithm became the basis for several generations of fast rule-based system shells: OPS5, its descendant ART, CLIPS, Jess, and others. Each system has enhanced and refined the algorithm to improve performance or flexibility. In this thesis, authors use the algorithm as implemented in Jess.

Briefly, the Rete algorithm eliminates the inefficiency in the simple pattern matcher by remembering past test results across iterations of the rule loop. Only new or deleted working memory elements are tested against the rules at each step. Furthermore, Rete organizes the pattern matcher so that these few facts are only tested against the subset of the rules that may actually match.

Every node represents one or more tests found on the LHS of a rule. Each node has one or two inputs and any number of outputs. Facts that are being added to or removed from the working memory are processed by this network of nodes. The input nodes are at the top of the network, and the output nodes are at the bottom. Together, these nodes form the Rete network, and this network is how Jess's working memory is implemented.

At the top of the network, the input nodes separate the facts into categories according to their head—for example, books go through one path, and borrowers go through another. Inside the network, finer discriminations and associations between facts are made, until the facts get to the bottom. At the bottom of the network are nodes representing individual rules.

When a set of facts filters all the way down to the bottom of the network, it has passed all the tests on the LHS of a particular rule; this set, together with the rule itself, becomes either a new activation record or a command to cancel a previously existing activation record (recall that an activation record is an association of a list of facts with a rule that they activate).

Between the inputs and the outputs, the network is composed of two broad categories of nodes: one-input nodes and two-input nodes. One-input nodes perform tests on individual facts, and two-input nodes perform tests across multiple facts.
3.5.4 Integrating Java and Jess

Java is a general-purpose, concurrent, class-based, object-oriented computer programming language. There are two main ways in which Java code can be used with Jess: Java can be used to extend Jess, and the Jess library can be used from Java. In general, all extracted code would need to appear inside a “try” block, inside a Java method, inside a Java class, to compile; and all Java source files are expected to include the "import jess.*;" declaration.

The jess.Rete class is the rule engine itself. Each jess.Rete object has its own knowledge base, agenda, rules, etc. To embed Jess in a Java application, you'll simply need to create one or more jess.Rete objects and manipulate them appropriately. To use Jess as a library from Java programs, the file jess.jar (in the lib directory) must either be on your class path, be installed as a standard extension, or your development tools must be set up to recognize it. The details of doing these tasks are system and environment dependent, but setting the class path usually involves modifying an environment variable, and installing a standard extension simply means copying jess.jar into $JAVA_HOME/jre/lib/ext directory.

You can create and manipulate Java objects directly from Jess. Using them, you can do virtually anything you can do from Java code, except for defining new classes. Jess converts freely between Java and Jess types when it can. Java objects that can't be represented as a Jess type are called external address values. Jess can also access member variables of Java objects using the set-member and get-member functions.

3.6 Steps in Implementation of Model

The detail procedural analysis was carried out. After going through the analysis, the procedure which was adopted to develop a model is mentioned below.

- Detail study of biometric system, Knowledge based (rule based) system
- Study of JESS, Java concepts and Integration
- Analysis of possible attack points
- Design multiple agents and required logs
- The framing of rules using the Intrusion knowledge and incorporation into JESS
- Develop user interface and agents using JESS and Java
• Testing of agents
CONCLUDING REMARKS

In spite of the various advantages of biometric process, it is vulnerable to attacks which can compromise on its security intentions. Intruder can attack on different points of biometric system. For example biometric template database, network channel, biometric device, feature extraction module, matcher module etc. The research design which authors proposed contains a knowledge based system to detect source of intrusion and suggests best possible prevention technique and suitable preventive and corrective actions for different intrusions. This model also uses security audit as well as alarming and reporting mechanisms. The malicious activity database is stored for future intrusion detection. To detect the source by tracking, backward chaining approach is used. The rules are defined and are stored in the Rule engine of the system.

The intelligent model uses AI and expert system as backbone of this system. In this research design, authors have suggested multi-agent system. Those three intelligent agents are implemented at biometric template database, feature and matcher modules and biometric device.

To develop these modules the authors have used integration of java, Jess, FuzzyJess.