(SEMI) AUTOMATIC WRAPPER GENERATION FOR PRODUCTION SYSTEMS BY KNOWLEDGE INFERENCE

By

ARCHANA RAGHAVENDRA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001

To my Family, which has been a constant source of support and encouragement in all my endeavors

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my chair, Dr. Joachim Hammer, for his advice, support and encouragement throughout the course of this work. The past one year, working under Dr. Hammer, has been a great learning experience, not only in terms of gaining technical know how but also in terms of attempting to think and see beyond the apparent. I would specially like to thank him for his patience and consideration in accommodating the thesis defense amidst his busy schedule.

I thank Dr. Douglas Dankel, for his invaluable help and suggestions provided during the course of this work. I thank Dr. Joseph Wilson, for being on my thesis committee. I also thank Dr. Michael Frank for his willingness to substitute for Dr. Dankel at the thesis defense.

I thank my family whose love and support helped me in accomplishing this task. I specially thank my friends Pooja and Lalitha for their constant encouragement throughout this work.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

(SEMI) AUTOMATIC WRAPPER GENERATION FOR PRODUCTION SYSTEMS BY KNOWLEDGE INFERENCE

By

Archana Raghavendra

December 2001

Chairman: Dr. Joachim Hammer
Major Department: Computer and Information Science and Engineering

A production system encapsulates both data and knowledge about a domain. This makes it ideal for storing the operational logic of an enterprise. However, its integration with existing legacy applications and data sources in a collaborative business environment, e.g., a supply chain, poses major challenges due to the dynamic fact base and complicated knowledge structure of production systems. The goal of this thesis is to develop methodologies to enable efficient integration of a production system into an existing sharing infrastructure. Our approach is to encapsulate the production system with a wrapper tool, which enables retrieval of data and is tailored to the needs of the enterprise.

To query an information system, the schema of the data and the query capabilities must be known in advance. Our wrapper (semi) automatically extracts the fact schema, uncovers the rule structures and displays the supported queries in the form

of an XML schema. Queries against the exported XML schema are converted by the wrapper into commands understood by the production system; for the same token, the wrapper also converts the native results into an XML document conforming to the external schema.

This thesis provides an approach to (semi) automatically inferring the semantics of the rules in a production system which significantly reduces the manual input that is needed to integrate a production system into a collaborative business environment. We have developed a fully functional wrapper prototype system for accessing facts and knowledge stored in Jess, the Java Expert System Shell. Our initial tests, which are being conducted in the Database Research and Development Center, have successfully demonstrated the inferencing as well as the query and data translation capabilities of the wrapper.

CHAPTER 1
INTRODUCTION

Where is the knowledge we have lost in information?

T. S. Eliot (1888 - 1965)

The twentieth century has witnessed the rapid emergence and widespread proliferation of digital technology. Advances in information and communication technologies have led to a dramatic increase in the volume of data and knowledge that is generated and stored by individuals and organizations. We are truly amidst what can be termed as an *Information Explosion* [Kor97] driven by developments in communication, data processing, and data storage technology. Today, there exists a broad spectrum of information systems for storing and processing information - enterprise resource planning systems, database systems, knowledge based systems, data stores for business intelligence, legacy systems, custom applications, etc, each with their own way of storing, representing, and manipulating data.

As it is truly said, *Disparity often fuels the need for integrity.* The increasing breadth of information sources has made it difficult to access and exploit fully the advantage of the existing systems. The need to link together disparate information systems e.g., ERP systems, production systems, legacy database systems to enable free flow of data and information is a common solution to many collaborative infrastructures like supply chain [Coo93] and business to business integration scenarios [Ols00]. Several distributed environments, consisting of multiple, heterogeneous data sources are often confronted with the problem of obtaining a centralized, integrated view of data and

knowledge. Consolidation and standardization of information, scattered across multiple domains, to provide a path for the flow of critical data and information is urgently needed.

Bridging the associated syntactic and semantic gaps between heterogeneous information systems requires a mixture of transformation capabilities and neutral information representations. Most information systems are designed in such a way that they are not easily inter-operable. The autonomous nature of such systems makes it difficult to extract the data and knowledge that resides in them.

The focus of this research is to enable the integration of a class of information systems called production systems in an existing sharing infrastructure. Production systems are a class of knowledge based systems that encapsulate both data and knowledge and have the capability to infer new information about the domain from the existing facts and knowledge. Before delving deeper into the thesis objectives and approach, we give a brief description about production systems.

1.1 Production Systems

As mentioned above, production systems[1] or rule based systems as they are popularly known, are a class of knowledge based systems that encapsulate both data and knowledge. At this point, it would be of relevance to clarify the distinction among “data,” “information” and “knowledge”.

[1] The terms production system, rule system, rule based system, production rule system are used interchangeably throughout this document.

Data are basically facts. Facts are numbers or terms that exist either apart or as an attribute label. Information is data associated with contextual relevance; it is interpreted data. Knowledge, on the other hand, is the interrelation and association of information, longitudinally over time. For example, $250.25 would classify as data. The statement *I have $250.25 in my checking account and a bill of $100.00 is due tomorrow* would be information about the data. *If my balance drops below $200.00, I will be fined* would constitute knowledge about the context.

Knowledge based systems work with both data and knowledge. Production systems are among the most widely used knowledge based systems. They use rule inferencing mechanisms to process the data they store.

A production system consists of three main components [Gon93]

- Knowledge base - a set of if-then rules
- Working memory - a database of facts
- Inference engine - the reasoning logic to create rules and data.

Figure 1.1 shows the conceptual architecture of a rule based system.

The knowledge base is composed of sentences. Each sentence is a representation of a fact or facts the rule system will use to make determinations. The sentences are represented in a language known as the knowledge representation language. These sentences are usually in the form of rules. Rules are made of antecedent clauses (if), conjunctions (and) and consequent clauses (then). A rule in which all antecedent clauses are true is ready to be fired or triggered. Production systems have three major properties: soundness-confidence that a conclusion is true, completeness-the system has the knowledge to be able to reach a conclusion and tractability–a surety that a conclusion can be reached. The power of these systems evolves from the clear separation of the

knowledge from its use. This separation allows us to develop different applications by having to create only a new knowledge base for each application.

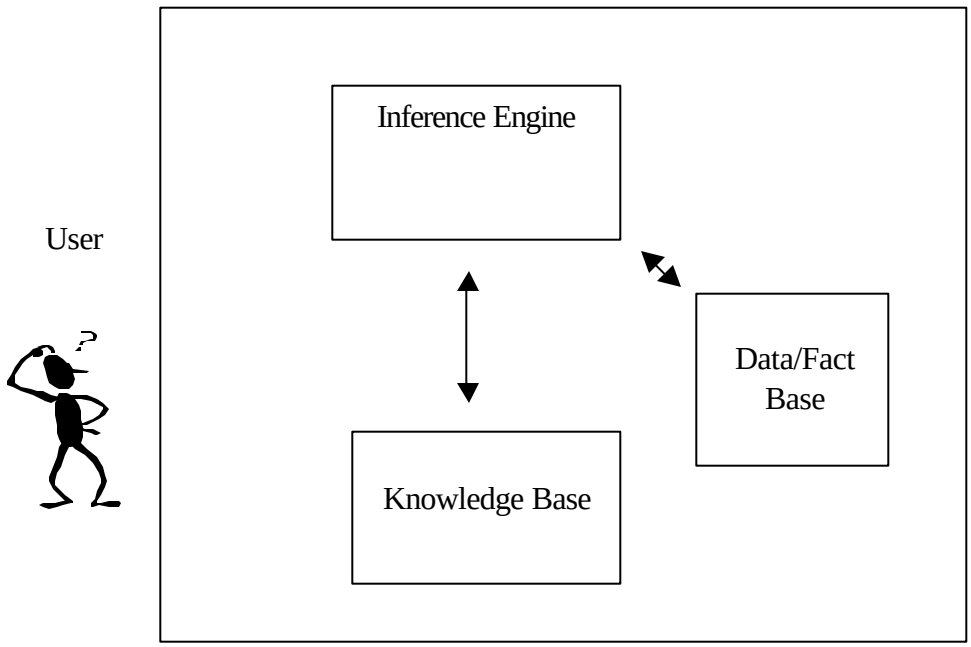

Fig 1.1 Architecture of a rule based system

Traditionally, production systems have been used mainly for inferencing in the areas of medical diagnosis [Buc84, Aik83], speech synthesis [Red76], etc. Another area, which has seen the widespread proliferation and acceptance of production systems is the enterprise sector. Production systems provide an ideal medium to store the operational logic of an enterprise. They are best suited when the business knowledge to be represented naturally occurs in the form of rules, where the relationships between rules are extremely complex, and where frequent changes in the business knowledge are anticipated. The key to a production system's success in modeling and processing business knowledge is in their ability to represent knowledge explicitly in chunks i.e., rules, which guide the operations of the business.

1.2 Motivational Example and Thesis Objective

Consider an enterprise participating in a collaborative business framework, e.g., a business to business environment. The enterprise uses a production system to store its operational logic and data. The environment requires a controlled flow of data and information from the production system to the other collaborative business units. This thesis is an attempt to enable the seamless integration of the production system in such an environment by enabling data transfer and exchange between the production system and the environment. We propose to develop a software component called *wrapper*, which enables data retrieval from the production systems by querying them. A wrapper is a software module, which encapsulates a data source and provides means to represent retrieved information in a shared data structure. The data retrieval from the source is performed by translating queries in the external model to the source specific query. A wrapper, thus essentially translates queries from a *foreign language* to the *native source specific language.* As depicted in Fig 1.2, the wrapper takes the external queries and translates them to the source specific query. The source, a production system in our case, responds to the translated queries and generates results. The results are gathered by the wrapper and converted back to a schema conforming to the external model.

1.3 Research Challenges

Production systems are a repository of data and knowledge. The knowledge encoded within them in the form of rules infers new facts and information from the existing facts. Querying them proves to be a challenging task due to the following reasons

- Unlike database systems, production systems store data as facts, which are unstructured and have no regular schema. Thus, it becomes difficult to probe such systems to expose the data that resides within them.

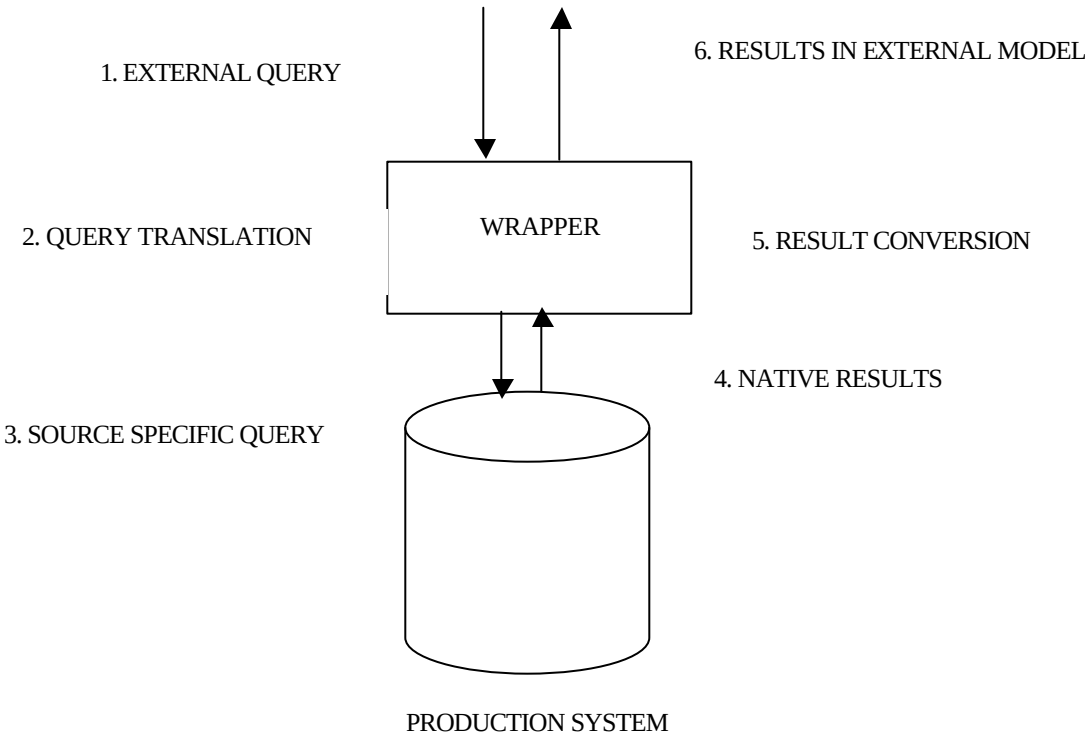

Fig 1.2 Wrapper Functionality

- In order to query an information system, the queriable information of the system should be known in advance. In other words, one should know what to query the system for. In a production system, the logic embedded in the rules determines the data and information that can be inferred or derived by the system. Thus, to extract this information, prior to the production system being active, the rules need to be analyzed and inferred. Often, these rules are complex and highly intertwined. Deciphering the semantics of such rules to predict all the possible information that can be churned out from the system is a challenging issue.

- The dynamic nature of production systems makes their analysis very difficult. New facts are continually generated from the existing facts. The non-deterministic nature of these systems makes it difficult to understand them and make deterministic predictions of the queriable information.

1.4 Summary of Thesis Goals

The objective of this thesis is to develop a wrapper tool, which facilitates integration of a production system with other information systems in a collaborative, sharing infrastructure. We propose a (semi) automatic wrapper, which provides data

translation capabilities by inferring the knowledge inside the production system. We use Jess, the Java Expert System Shell as a backend to develop a production system that is tailored to meet the demands of an enterprise. The characteristic feature of this wrapper is that it infers the knowledge inside the production system to determine the data and information that can be generated by the system. It thus provides query support capability for data and information, which may not exist prior to the query being issued. The main tasks involved are:

- Extraction of the fact schema and rules from the production system.
- Analysis and inference of rules to determine the query supporting capability of the system.
- Translating the external query to the production system specific query, gathering the results and displaying them.

The final deliverable, the wrapper, is an ensemble of all the components, which implement the above mentioned tasks. The wrapper encapsulates a production system, extracting the rules and the fact schema. The rules are inferred to determine the query capability of the system. This knowledge about the query support capability of the production system is represented as an eXtensible Markup Language, XML [Ext98] schema. Queries against the exported XML schema are converted into commands understood by the production system. The native results are converted into an XML document conforming to the external schema. The wrapper we propose is generic in nature and can be used to wrap any production system domain, which is built using a Jess like engine. We choose Jess as a sample production system for this research because it is well documented and is Java compatible. Jess provides a facility for easy rule and schema extraction from the Jess engine using Java APIs.

1.5 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 provides a description of knowledge based systems. Production systems, which are a class of knowledge based systems, are studied in detail. We discuss the existing methodologies, tools and techniques in the field of information systems integration and wrapper generation. We give a description of Jess, the expert system shell used as a sample production system in this research. A brief discussion on XML, its advantages as a format of data representation concludes this chapter. Chapter 3 analyzes the semantics and syntax of the knowledge structures of the production systems The architecture of the wrapper and the rule analysis procedure are described in detail. Chapter 4 gives the implementation details of the thesis. Chapter 5 summarizes the results and makes a performance evaluation of the wrapper prototype. Chapter 6 summarizes the major contributions as well as the limitations of this research and suggests the scope of future work.

CHAPTER 2
RELATED RESEARCH

In the previous chapter, we introduced production systems, their significance in business environments and outlined our approach of enabling their integration in a collaborative/shared environment with other legacy systems. This chapter provides an extended overview of the state of the art tools and technologies in the areas pertaining to all the above mentioned issues.

We begin with a study of production systems, which are a class of knowledge based systems. The architectural patterns and inferencing mechanisms of production systems are described in detail. Section 2.2 focuses on the current methodologies adopted in the integration of disparate information systems, to enable data and information flow in collaborative business environments. We then proceed to describe the research undertaken in the field of wrapper development and compare and contrast some of the approaches with our work. The final section is on XML and its related technologies. In particular, we discuss the advantages of XML as a lingua franca for neutral information models and data representation.

2.1 Knowledge Based Systems

Knowledge based systems can be described as computerized systems capable of using knowledge about some domain to arrive at a solution to a related problem If this were to be followed strictly, one could categorize many conventional software systems as knowledge based systems. Gonzalez et.al. [Gon93] describes some fundamental

concepts of knowledge based systems which distinguish them from conventional algorithmic and general search based programs. These are

- Marked separation of the knowledge from how it is used
- Use of highly specific domain knowledge
- Heuristic nature of the knowledge employed.

The MYCIN system at Stanford University [Buc84] developed these concepts to create the field of knowledge based systems. Following MYCIN, a number of other knowledge based systems have been developed such as PUFF[Aik83], HEARSAY[Red76], PROSPECTOR[Bar83] demonstrating the wide applicability of knowledge based systems in diverse domains.

As mentioned in the previous chapter, knowledge based systems are characterized by the marked separation of data and knowledge. This sets them apart from several conventional information systems e.g., database systems, which are just repositories of data. The inferencing mechanism in these systems uses the knowledge and the existing data to infer new knowledge about the domain. From a knowledge engineer's perspective, a knowledge based system is composed of a development shell and an intelligent program [Gon93]. The knowledge base, fact base and the inference engine constitute the intelligent program. Figure 2.1 depicts the conceptual architecture of the knowledge based system from a developer's perspective.

Associative (semantic) networks, frames, objects, logic and rules [Gon93] are the commonly used schemes to represent knowledge in knowledge based systems. We briefly describe associative networks, frames objects and logic based representations before describing rule based systems in detail.

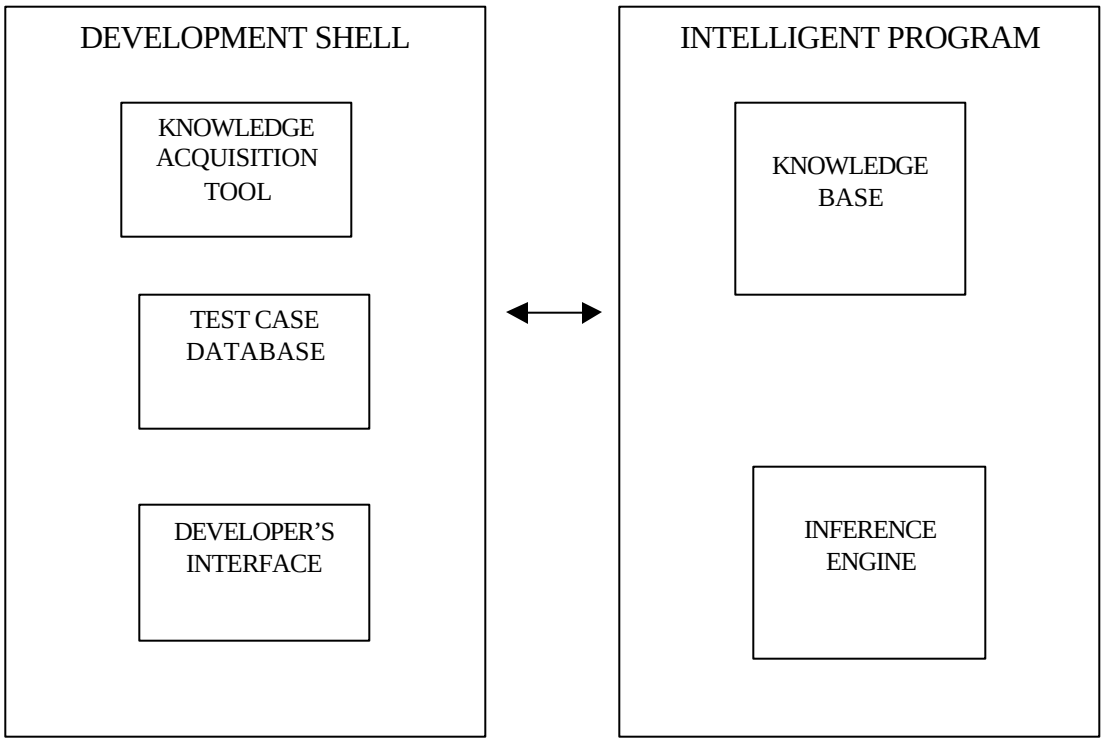

Fig 2.1 Knowledge Based System from a System Designer's Perspective

2.1.1 Associative Networks

An associative (semantic) network [Qui68] is a labeled, directed graph. The nodes in the network are used to represent various concepts or objects, and the arcs or links connecting the nodes represent the various relationships or associations that hold between the concepts. These networks were originally developed to represent the knowledge within English sentences. The main advantages of associative networks are their succinctness and explicitness [Qui68, Gon93].

2.1.2 Frames

Minsky [Min75] developed the representation scheme called frames, which provide the ability to group facts in associated clusters. Using frames, relevant procedural knowledge can be associated with some fact or group of facts. A frame provides a

structure to represent knowledge acquired through previous experience. This knowledge representation scheme helps in working with new objects or actions by using existing knowledge of previous events, concepts and situations.

2.1.3 Objects

Unlike frames, the object-oriented approach [Bob77, Kra83] creates a tight bond between the code and data, rather than separating them into two complex, separate structures. An object is defined to be an encapsulation of data, a real world entity and a description of how these data can be manipulated. The principles of abstraction, encapsulation, inheritance and polymorphism [Mey88] make the object oriented approach well suited for representing knowledge.

2.1.4 Logic

Logic based knowledge systems use formal logical systems to represent a knowledge base. Knowledge expressed in logic is then applied, using inference rules and proof procedures, to a problem-solving process. Propositional calculus [Koz98] and predicate logic [Dij90] are the common methods of representing logic. Propositional calculus deals with appropriate manipulations of logical variables that represent propositions. First order predicate calculus is used to analyze more general cases. Its expressiveness is enriched by the universal and the existential quantifiers and by the possibility of defining a predicate that expresses the true relationships among objects. The PROLOG [Col87] language is an implementation of a close version of predicate logic. Despite its rich expressiveness, first order predicate logic has many limitations [Gon93] some of them being uncertainty management, monotonic reasoning, and declarative nature.

2.1.5 Rule Based Systems

Rule based systems or production systems are the most widely used among all knowledge based systems. Classical systems such as DENDRAL [Buc84], MYCIN [Buc78], R1/Xcon [Gon93], GenAid [Gon93] are all rule based systems. Although the foundation of rule based inferencing is based on logic, these systems diverge from pure logic systems in the following ways: they are non-monotonic and accept uncertainty in the deductive process. Knowledge is represented in the form of rules using the familiar IF-THEN format, which can be used to express a wide range of associations. The IF portion of a rule is a condition, also called a premise or an antecedent, which tests the truth value of a set of facts. If they are found to be true, the THEN portion of a rule, also called the action, conclusion or the consequent is inferred as a new set of facts. Rule based systems are built around two major architectural patterns - inference networks and pattern matching.

Inference networks can be represented as a graph in which nodes represent parameters that are facts obtained as raw data or derived from other data. Interconnections between various nodes are represented by the rules within the system. This knowledge is used by the inference process to propagate results throughout the network. All these interconnections are known prior to the execution of the system. This minimizes the search time during the execution and simplifies the implementation of the inference engine and the handling of explanations. MYCIN [Buc78], PROSPECTOR [Bar83], GenAid [Gon93] are examples of large knowledge based systems that are based on the inference network architecture.

Pattern matching systems use extensive searches to match and execute rules, deriving new facts. They typically use sophisticated implementations of pattern matching techniques to bind variables, constrain the permissible values to be matched to a premise, and determine which rules to execute. Relationships between rules and facts are formed at run time based on the patterns that match the facts. PROLOG [Col87] and CLIPS [Ril98] are examples of rule shells that use pattern matching for reasoning. Pattern matching systems are very flexible and powerful. They are suited for domains where the possible solutions are unbound or large in numbers.

Rule based system use the rules of modes ponens to manipulate rules. The reasoning process of a rule based system is a progression from a set of data towards a solution. There are two means of progressing towards conclusions- forward reasoning and backward reasoning.

Forward reasoning or data driven reasoning is the process of working from a set of data towards the set of conclusions that can be drawn from these data. This strategy is especially appropriate in situations where data are expensive to collect, but few in quantity. Traditionally, pattern matching systems have been associated with forward reasoning. In many cases, the forward reasoning system proves to be inefficient due to the number of rule fact comparisons made in each match cycle. Forgy [For82] proposed an algorithm, which overcomes this inefficiency. This algorithm, known as the Rete algorithm, works on the basis of maintaining a list of satisfied rules and determining how this list changes due to addition and deletion of new facts in the system. Jess [Fri97], CLIPS [Ril98] are rule based shells which incorporate the Rete algorithm in their inference mechanism.

Backward Reasoning or goal driven reasoning works from the set of goals to the set of solutions. It is ideal for applications where there is more input than possible conclusions. Backward chaining systems are typically, not solely implemented using inference networks.

In addition to the above two methods, another method, opportunistic reasoning [Ham88] has increasingly been recognized as an alternative for certain classes of problems. In opportunistic solving, knowledge is applied at a most opportune time and in the most auspicious fashion. This form of reasoning is highly suited for applications where problem solving knowledge can be partitioned into independent modules that then cooperate in solving a problem. They have been implemented in solving several blackboard architecture systems [Jag89].

2.2 Introduction to Jess

Jess is a rule engine and scripting environment written in Java [Fri97]. It was developed by Ernest Friedman-Hill at the Sandia National Laboratories. Jess was originally inspired by the CLIPS [Ril98] expert system shell, but has a distinct Java influenced environment of its own. Jess enables us to develop applets and applications that have the capacity to reason using knowledge supplied in the form of declarative rules. Jess supports the development of rule-based expert systems which can be tightly coupled to code written in the powerful, portable Java language. The core Jess language is still compatible with CLIPS, in that many Jess scripts are valid CLIPS scripts and vice-versa. Like CLIPS, Jess uses the Rete algorithm to process rules, a very efficient mechanism for solving the difficult many-to-many matching problem [For82]. Jess adds many features to CLIPS, including backward chaining and the ability to manipulate and

directly reason about Java objects. Jess is also a powerful Java scripting environment, from which enables us to create Java objects and call Java methods without compiling any Java code.

2.2.1 Jess Syntax Specifications

The atom or symbol, the basic unit of syntax, is a core concept of the Jess language. Atoms are very much like identifiers in other languages. A Jess atom can contain letters, numbers, and a set of punctuation marks. Another fundamental unit of syntax in Jess is the *list*. A *list* always consists of an enclosing set of parentheses and zero or more atoms, numbers, strings, or other lists. The first element of a list, the *car* of the list in LISP parlance is often called the list's head in Jess. Numbers and Strings are the basic data types supported by Jess.

As in LISP, all code in Jess-control structures, assignments, procedure calls-take the form of a function call. Function calls in Jess are simply lists. Function calls use a prefix notation; a list whose head is an atom that is the name of an existing function can be a function call. Function calls can be nested, the outer function being responsible for evaluating the inner function calls. Jess comes with a large number of built-in functions that do everything from math, program control and string manipulations, to giving access to Java APIs. Jess also supports newer functions to be defined using the *deffunction* construct.

Programming variables in Jess are atoms that begin with the question mark (?) character. The question mark is part of the variable's name. A normal variable can refer to a single atom, number, or string. A variable whose first character is instead a “$” is a multivariable, which can refer to a special kind of list called a multifield. Multifields are generally created using special inbuilt multifield functions and can then be bound to

multivariable: Variables in Jess cannot be declared before their first use. Jess also provides for global variables to be created which are persistent variables.

2.2.2 Data and Knowledge Structure in Jess

Rule based system maintains data in the form of nuggets called facts. This collection is called the fact base. It is somewhat akin to a relational database, especially in that the facts must have a specific structure. In Jess, there are two basic kinds of facts: ordered facts, unordered facts.

Ordered facts

Ordered facts are simply lists, where the first field (the head of the list) acts as a sort of category for the fact. Here are some examples of ordered facts:

- (shopping-list eggs milk bread)
- (person "Bob Smith" Male 35)
- (father-of danielle ejfried)

Ordered facts can be added to the fact base using the *assert* function. Each fact is assigned an integer index, the fact-id when it is asserted. Individual facts can be removed from the fact base using the *retract* function.

Unordered facts

Ordered facts are useful, but they are unstructured. Most of the time, we need a bit more organization. In object-oriented languages, objects have named fields in which data appears. Unordered facts offer this capability, the fields are traditionally called slots.

- (person (name "Bob Smith") (age 34) (gender Male))
- (automobile (make Ford) (model Explorer) (year 1999))

The structure of unordered facts has to be defined using the *deftemplate* construct before they are created. The *deftemplate* construct has the following structure

(deftemplate <deftemplate-name> [extends <classname>] [<doc-comment>]

[(slot <slot-name> [(default | default-dynamic <value>)] [(type <typespec>))]*)

The <deftemplate-name> is the head of the facts that will be created using this deftemplate. 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 '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. This thesis works with unordered facts as it is easier to capture their schema as compared to ordered facts.

Rule base

Rules in Jess are like if... *then* statements in procedural language but are not used in a procedural way. While *if... then* statements are executed at a specific time and in a specific order, according to how the programmer writes them, Jess rules are executed whenever their *if* parts, their left-hand-sides or LHSs are satisfied, given only that the rule engine is running. This makes Jess rules less deterministic than a typical procedural program. Rules are defined in Jess using the *defrule* construct. The *defrule* construct has the following structure:

(defrule <rule -name> [<doc-comment>] [<if-clause>]+ => [<then-clause>])

For example,

(defrule test (found-true) => (printout t “found in the fact base”))

This rule has two parts, separated by the "=>" symbol, which is read as "then". The first part consists of the LHS pattern (found-true). The second part consists of the RHS action (“found in the fact base”). Although it's hard to tell due to the LISP-like

syntax, the LHS of a rule consists of patterns, which are used to match facts in the knowledge base, while the RHS contains function calls.

Rules are uniquely identified by their name. If a rule named my-rule exists, and you define another rule named my-rule, the first version is deleted and will not fire again, even if it was activated at the time the new version was defined.

Rule based systems are ideal for storing the business rules and operational logic of an enterprise, which naturally occur in rule form. As mentioned previously, the focus of this research is the seamless integration of such systems in an information sharing/ collaborative environment. This section is a study on the currently deployed methodologies in integrating disparate information systems. These techniques are mostly adopted in the context of integrating collaborative business units.

2.3 Information Systems Integration

Businesses today use a wide variety of information systems to store their data and knowledge. Collaborative frameworks, e.g., business to business commerce, supply chain flows, etc often necessitate the automated flow of data and information between the business entities. The most common models for information systems integration encompass four primary techniques namely, Direct Application Integration, Closed Process Integration, Open Process Integration and Data Exchange [Yee99].

2.3.1 Direct Application Integration

The use of Direct Application Integration pattern necessitates a business application to interact directly with application APIs of different systems, translate native application data and support complex transformations. Integration brokers with built-in

support for adapters, transformations and asynchronous-content-based routing are particularly suited to address this pattern. As shown in Figure 2.2, the pattern assumes that some component of the integration broker is running on each end of the flow. In practical terms, this means that the participating entities need to run the same integration broker. Ideally, to transact data integration between independent corporate entities, the architecture of the integration broker should be secure and should allow for federated control. Security is vital because these inter enterprise transactions will increasingly occur over a public network. This necessitates security services, such as secure transport, component authentication and user authorizations, to be defined and implemented. Since each corporation is an independent entity with its own data and security models, support for federated control is also important.

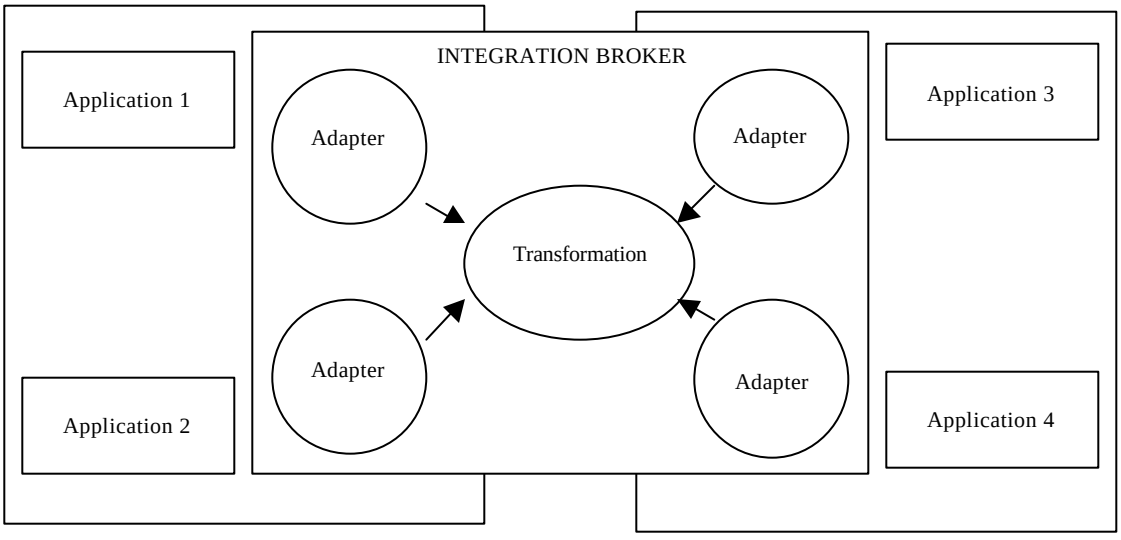

Fig 2.2 Direct Application Integration

2.3.2 Closed Process Integration

The closed process integration pattern identifies a principal participant responsible for managing processes to be bound. In this operating model, shown in Figure 2.3, the other participants are secondary. They do not have visibility into the entire process, nor do they actively manage the process. Instead, they participate in response to the process managed by the principal. Hence, the process is regarded as closed with respect to other secondary participants. Managing these interactions requires the introduction of business process integration (BPI) services. BPI services integrate logical business process elements generally expressed as activities rather than data. BPI manages these long-running transactional contracts by driving defined activities to secondary participants, such as the transmission of a business document or the acknowledgment of document receipt.

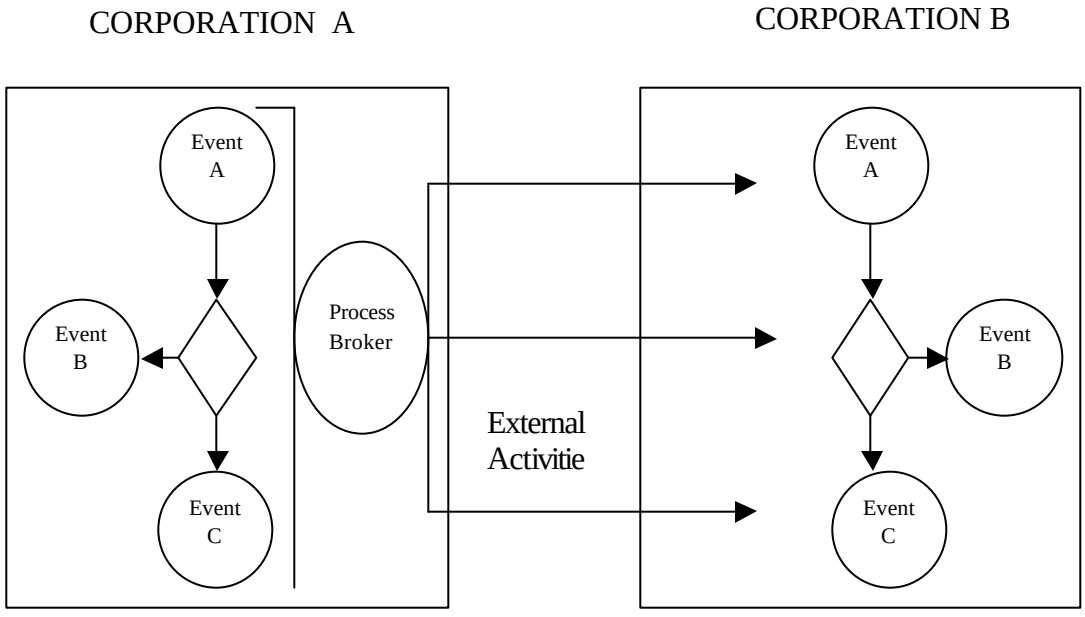

Fig 2.3 Closed Process Integration

2.3.3 Open Process Integration

Unlike the closed process integration pattern, which operates from a centralized master process manager model, the open process integration pattern introduces the notion of shared processes. The inter-system processes are managed at a peer level, with each participant actively managing processes within its domain. As shown in Figure 2.4, each participant can choose to externalize elements of its managed process domain to be shared while limiting visibility into corporate internal processes.

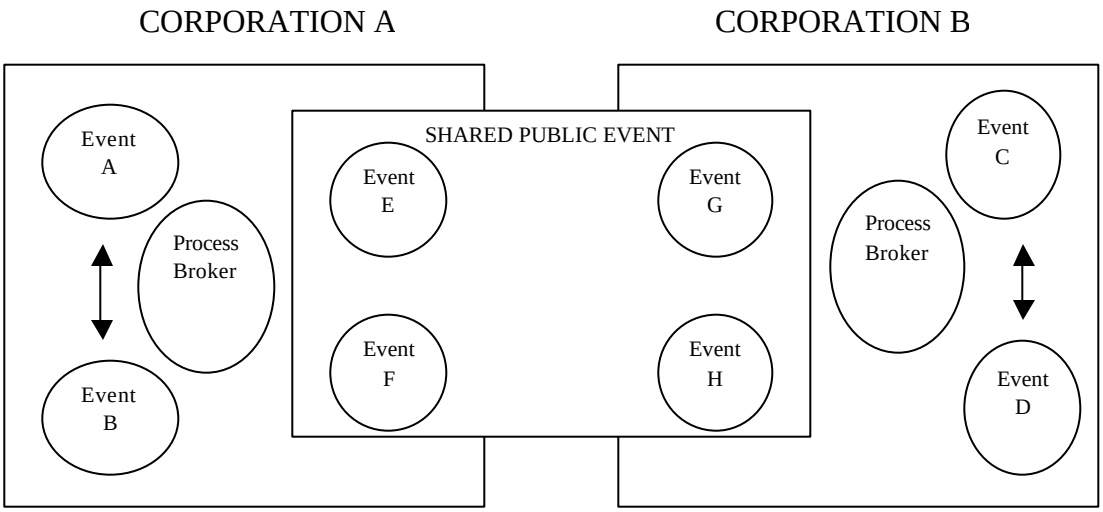

Fig 2.4 Open Process Integration

2.3.4 Data Exchange

Integrating different information systems by means of data exchange is the most prevalent pattern in deployment. It has been in place since the early days of Electronic Data Interchange, EDI [Dat87] More recently, it has been applied by the first generation of Net marketplaces and trading exchanges, using XML as the base format.

Data exchange integration pattern enables business transactions via a common data exchange format. This reliance on a common data exchange format rather than a

common infrastructure makes this pattern easier in some situations to implement and extend. This pattern requires that data native to an application be translated to the document format and transmitted through a gateway. As illustrated in Fig 2.5, all exchanges in and out of the corporate entities occur through a managed data exchange gateway.

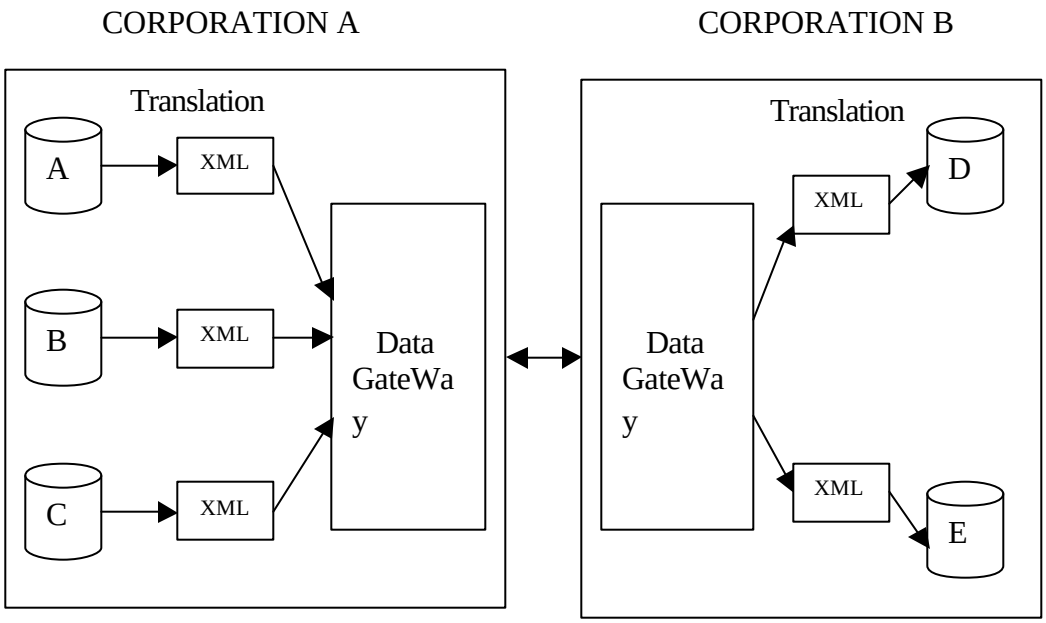

Fig 2.5 Data Exchange

This thesis is an attempt in enabling integration of production systems by means of data transfer and exchange. Our wrapper tool facilitates this process, by enabling data retrieval from production systems by querying them. It does so by translating the queries in the neutral data exchange model to the production system specific queries.

Wrapper generation has been an active area of research and interest in database communities all over the world. We now take a look in the research that has been undertaken in the area of wrapper generation.

2.4 Wrapper Technology

A *wrapper* or *translator* [Car94] is a software module which encapsulates a data source, and provides means to represent retrieved information in a shared data structure. The data retrieval from the source is performed by translating queries in the external model to the source specific query. A wrapper thus translates queries from a *foreign language* to the *native source specific language*. Wrappers are used to provide access to heterogeneous data sources, as illustrated in Fig 2.6 [Pap95]. Wrapper is an integral component of several mediator based e.g., HERMES [Sub95], TSIMMIS [Cha94] and warehouse based WHIPS [Lab97], SQUIRREL [Zho96] data integration architectures.

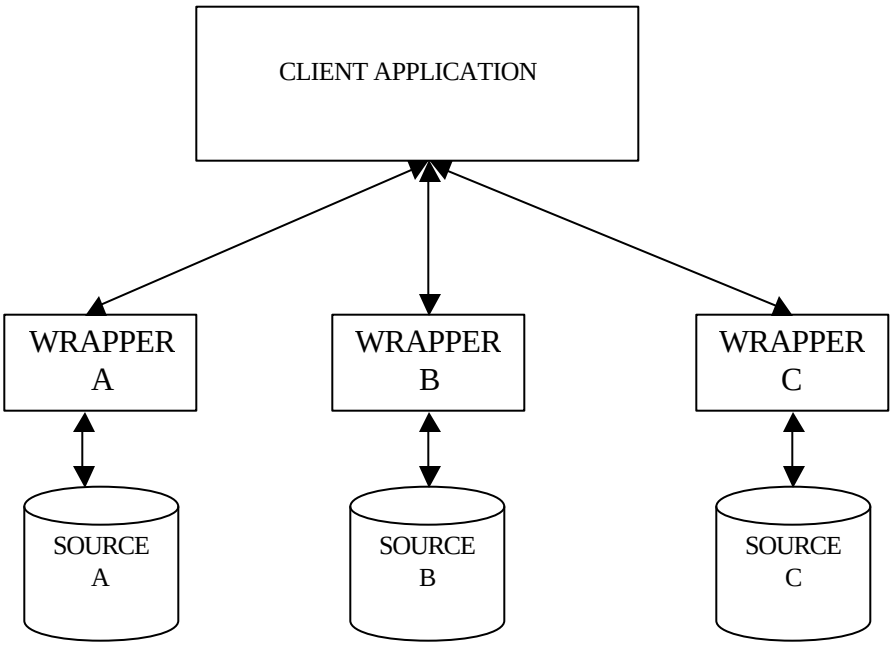

Fig 2.6 Accessing information through wrappers

Wrappers differ in the languages and models they support e.g., relational. objects, knowledge models and in the sophistication of the functionality they provide. Wrappers can be classified into the following categories based on their functionality [Wel96].

2.4.1 Generic Functionality

This refers to operations that take place in the wrapper because of the type of the data source rather than its contents. For example, conversions of SQL3 queries to SQL queries against an Oracle DBMS will be done the same regardless of the database schema. It is possible to further differentiate between conversions necessary for a family of data sources e.g., RDBs or file systems and those necessary for a particular instantiation of a data manager in the family e.g., Oracle or Sybase RDBMSs, or Unix or DOS file systems.

2.4.2 Source-Specific Functionality

This refers to the operations that take place because of the contents of the data source, i.e., its schema. Assuming that the schema remained unchanged, these conversions would take place regardless of which engine is actually managing the data e.g., unless the schema changes, the source-specific conversions would be the same whether the data was stored in Oracle or Sybase.

2.4.3 Error Reporting

Wrappers can report errors back to the caller. These may be error messages generated by the data source, perhaps converted to a desired format, or they can be the results of an error recovery operation within the wrapper.

2.4.4 Caller Model

Most wrappers assume that they are wrapping the source for use by a particular kind of caller. However, the TSIMMIS project [Pap95] recognizes that by supporting a variety of conversions from the caller before the wrapper converts to the data source, much of the wrapper's implementation can be reused.

2.4.5 Calls to External Services

Wrappers could conceivably call out to services to perform more complex services than the wrapper itself can perform. An example would be the use of an external ontology engine to semantically transform a query before submission to the wrapped data source.

Wrapper designers either construct the wrappers manually or use some tools facilitate the wrapper code development. The approaches normally adopted are manual, semi automatic or automatic generation.

2.4.6 Manual

Manual or hard coded wrappers are often tedious to create and may be impractical for several sources. For example, in case of web sources, the number of sources can be very big, new sources are added frequently, and the structure and content keeps changing [Ash97]. These factors lead to high maintenance cost of manually generated wrappers.

2.4.7 Semi-Automatic

Several semi-automatic approaches to wrapper generation [Cha94, Ash97, Liu98] have aimed at partially automating the process of wrapper generation to save the human effort involved. It was noted [Cha94] that the part of the wrapper code that deals with the details specific to a particular source is often very small. The other part is either the same among wrappers or can be generated semi-automatically. Their approach is applicable to several types of data sources like relational databases, legacy systems, web sources, etc. Their wrapper implementation toolkit is based on the idea of template based translation. The designer of the wrapper uses a declarative language to define the rules (templates) which specify the types of queries handled by a particular source. Each rule also comes with an action to be taken in case the query sent by the mediator of the integration system

matches the rule. This action causes a native query – a query in the format of the underlying source to be executed. A rule based language MSL [Pap96] is used by the authors for query formulation.

The work done by [Ash97] in the Semi Automatic Wrapper generation for web sources aimed at generating wrappers for both multiple instance sources and single instance web sources. A multiple instance source contains information on several pages that have the same format. A single instance source contains a single page with semi structured information. The documents are first analyzed to separate HTML tags that separate objects of interest and then objects are separated into different regions. Their approach relies primarily on syntactic knowledge to identify object boundaries. XWRAP [Liu98], an interactive system for semi automatic wrapper generation for web sources propounds a two phase code generation framework. The first phase utilizes an interactive facility that communicates with the wrapper developer and generates information extraction rules. The second phase utilizes the deliverables from the first phase to construct an executable wrapper program for the web source. Gruser et al. [Gru98] propose a wrapper protocol to publish the capability of the source. Their wrapper toolkit provides a graphical interface and specification languages to specify the capability of the source and functionality of the wrapper.

2.4.8 Automatic

Automatic wrapper generation calls for little or no human involvement. Such tools are usually incorporate concepts of artificial intelligence, and are based on supervised learning algorithms. Automatic techniques for wrapper generation are sometimes called wrapper induction techniques and are based on inductive and machine learning techniques. According to [Eik99] inductive learning is the task of computing a

generalization form a set of examples to some unknown concept. [Eik99] points out the classification of the inductive learning techniques used for wrapper induction:

Zero order learning

They are also called decision tree learners as their solutions are represented as decision trees. The drawback of these methods comes from the fact that they are based on propositional logic which has a number of limitations.

First order learning

Methods of this type can deal with first order logical predicates. Inductive logic programming is a method of this class, widely used due to the ability to deal with complex structures such as recursion. Two approaches- bottom up and top down are often used as a part of the first order learning. The bottom up approach fist suggests generalizations based on a few examples. This generalized model is then corrected based on the other examples. The top down approach starts with a very general hypothesis and then distills it learning from negative examples.

Some known systems of inductive learning are STALKER [Mus98] and the system described by Khusmerick, et al. [Kus97] An overview of some other systems for information extraction by inductive learning is given in [Eik99].

The wrapper we propose is a (semi) automatic wrapper having a generic functionality. It can wrap any domain built using an engine similar to Jess, regardless the fact structure and the rule base. This wrapper also has an error reporting feature which will be explained in the later sections. A characteristic feature which sets apart this thesis from the rest of the work done in the area of wrapper generation is that we attempt to query a system for data which is capable of being generated in future. Apart from data, the wrapper also supports queries for information regarding the domain. The query

models developed so far prove to be inadequate to capture the information that is capable of being generated by the system.

2.5 XML

The eXtensible Markup Language, XML [Ext98] has emerged as a new standard for data representation and exchange on the Internet. XML is a meta markup language that provides a format for describing structured data. A subset of SGML, XML provides a structural representation of data that can be implemented broadly and is easy to deploy.

XML is a powerful meta language. XML provides an easily used mechanism by which other markup languages can be developed for specialized needs or business domains. This is evidenced by the proliferation of markup languages such as Chemical Markup Language (CML), VoxML (Voice Markup Language) and VML (Vector Markup Language).

The main advantages of XML which have made it a popular are:

- XML is self-describing, i.e., its tags are meaningful and self descriptive.
- It provides a clear separation between the structure, semantic content and presentation of a document. In HTML, the tags tell the browser to display the data in a particular format; with XML tags simply describe data. Style sheets such as XSL (Extensible Style Language) or CSS (Cascading Style Sheets). This separation of data from presentation enables the seamless integration of data from many sources.
- XML is extensible in many ways: extensible tag library, extensible document structure, and extensible document elements.
- XML is non-proprietary.
- It is user-friendly because it is human-readable. XML is easy to understand because of its text-based approach. XML is web-friendly because its components can be drawn for various sources in the web.
- XML is modular in nature and allows flexible combination. For example, when a same basic XML data document is extended with different stylesheets, different views are presented to the users.

Since XML provides a standard syntax for representing data, it is perceived to be a key enabling technology for exchange of information on the WWW and within

corporations. As a consequence, integration of XML data from multiple external sources is becoming a critical task. However, XML without agreed upon Data Type Documents, DTDs does nothing to support integration at the semantic level. The names and meanings of the tags used in XML documents are arbitrary. As a result, the emergence of XML is fueling activity in various communities to agree on DTDs.

2.5.1 DTD

DTD provides a list of the elements, attributes, notations and entities contained in a document, as well as their relationships to one another. The main reason to explicitly define the language is so that documents can be checked to confirm to it. DTDs can be included in the file that contains the document they describe or they can be linked from an external URL. When included in the file, the XML declaration and the Document Type Declaration together are called the prolog of the document. DTDs enable us to see the structure of the document without going over the actual content. Also, they can be made external so that they can be shared by different documents and websites thus enforcing adherence to common standards.

2.5.2 XML Schema

XML schema provide a means for defining the structure, content and semantics of XML documents. XML Schema has recently been approved as a W3C recommendation. It is a new approach to providing a reference to a document. It is written in XML. A schema gives all the information that a DTD does, and, since it is an XML document, it is more readable than a DTD. Additionally, a schema allows to validate a content is appropriate for a given tag. Schema classifies the XML elements as simple types or complex types. Simple types are those elements, which do not have nested elements or attributes. A complex type includes all elements in the XML

document, which have nested elements and attributes describing the element. Using XML schema, we can specify that a tag must enclose numeric data within a certain range, or belong to one of a specific list of values

2.5.3 XSLT

XSLT is a language for transforming XML documents into other XML documents. XSLT is designed for use as part of extensible Stylesheet Language, XSL which is a stylesheet language for XML. A transformation expressed in XSLT describes rules for transforming an XML document parsed into a source tree into a new resulting tree which represents the new document. The transformation is achieved by associating patterns with templates. A pattern is matched against elements in the source tree. A template is instantiated to create part of the result tree. The resulting tree is separate from the source tree. A transformation expressed in XSLT is called a stylesheet. This is because, in the case when XSLT is transforming into the XSL formatting vocabulary, the transformation functions as a stylesheet. However, XSLT is not intended as a completely general-purpose XML transformation language. Rather it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL.

2.5.4 DOM

The document object model (DOM) is an API for HTML and XML documents. It defines the logical structure of the documents and the way a document is accessed and manipulated. In the DOM, documents have a logical structure having to a tree like representation. This is often called the structure model. DOM structure models exhibit structural isomorphism; i.e., if any two DOM implementations are used to create a representation of the same document, they will create the same structure model, with precisely the same objects and relationships.

XML, with all its related technologies, represents the most versatile and robust format for exchanging business information since the development of open Electronic Data Interchange (EDI) standards. Many corporate entities and vertical groups have embarked on a development effort to establish XML as their preferred format for Enterprise Application Integration (EAI) and business to business (B2B) exchanges.

.

CHAPTER 3
ANALYSIS AND DESIGN

The first step towards building a wrapper for any information system begins with an analysis of the system being wrapped. This chapter begins with a study of the production system's fact base. We explain the extraction and representation of the fact schema. The discussion proceeds to analyze the knowledge base. Rules, which form the production system knowledge, are classified and their structures, understood. We carve out a subset of rules, which form the focus of our analysis. The problem of determining the queriable information from the system is modeled and a solution is presented. We then proceed to describe the architecture of the wrapper and present the steps involved in the generation of the wrapper.

3.1 Extracting Fact Schema

In a production system, data are stored as facts. To enable the external user to achieve a better understanding of the fact base, we extract the schema of the data stored within the production system. It also helps an external user in issuing queries on the production system. In the previous chapter, we gave an overview of the fact structures supported in Jess. We focus our attention on unordered facts, which have a definite structure or template. Unordered facts in Jess have a schema defined by their template. The template is somewhat similar to the schema of a relational table in a relational database. For example,

(employee (slot name) (slot id) (slot addr) (slot salary))

is a template for an employee fact. The slots name, id, salary and addr are the attributes of the employee template. All the employees in the fact base would have the above structure.

Apart from these template defined attributes, there could be some other entities which are generated by the rules in the production system. For example, there could be a rule which generates hourly-rate for an employee using a mathematical calculation based on his salary. All such data entities, which do not exist in the fact base but are capable of being generated when the inference engine is active, need to be captured and presented to the user. Jess provides variables to store all such attributes. We use Java APIs to extract the templates and the variables associated with each of them. They are converted into an XML representation and presented to the user. Consider an employee template, which has the template structure as shown above and has the attributes hourly-rate and discount associated with it. The XML model representing this would be

```
<schema>
     <relation>
          <name>employee</name>
          <attr type =”int”>id</attr>
          <attr type= “string”>name</attr>
          <attr typr = “string”>addr</attr>
          <attr type = “real”>sal</attr>
          <var>
               <vname type =”real”>hr-
rate</vname>
               <vname type
=”real”>pension</vname>
          </var>
     </relation>
 </schema>
```

The DTD for the above schema would be:

```
<!ELEMENT  schema (relation+)>
<!ELEMENT  relation (name,attr+,var)>
<!ELEMENT  name   (#PCDATA)>
<!ELEMENT   attr     (#PCDATA)>
     <!ATTLIST attr type CDATA
#IMPLIED>
<!ELEMENT   var    (varname+)>
<!ELEMENT  varname (#PCDATA)>
     <!ATTLIST varname type CDATA
#IMPLIED>
```

3.2 A Semantic Classification of Production System Rules

Knowledge in a production system is encoded in the form of rules. These rules determine the data and information that can be generated by the inference engine. It is important to note that we are attempting to query the system for not only the data and knowledge that resides within, but also for the data and information that can be inferred by the production system. Our aim is to enable an external entity have an understanding of the production systems' queriable information. In order to accomplish this, we need to systematically analyze the rules.

Rules in a production system can be classified into three broad categories according to their semantic functionality - Structural Assertions, Action assertions and Derivations.

3.2.1 Structural Assertions

This category comprises all those rules, which convey some structural information about the existing facts in the production system. In reference to our motivational example introduced in Chapter 1, two sample rules in this category could be:

- If an employee fact with id = 'id' is found, display the name, address and salary associated with the employee

- If the salary of the employee > 'amount' display the employee attributes

These rules provide a support for the extraction of data and knowledge that reside within the system. They are capable of supporting queries similar to relational database queries. Examples of such queries could be:

- Display the name, address and salary of all employees in the fact base with id = 'id' (selection)
- Display the name and id of all employees with salary > 'amount' (selection)

3.2.2 Derivations

A base fact is a fact that is a given in the domain of consideration and is stored in the production system. A derived fact is created by inference or a mathematical calculation from the existing set of facts and rules. A derivation is a kind of rule, which may either be a Mathematical Calculation or a Logical Inference. A mathematical calculation produces a derived fact according to a specified mathematical algorithm. An Inference produces a derived fact using logical induction from particulars or deduction from general principles. Examples include

- The hourly rate of an employee is **calculated** from his salary divided by factor
- The date of delivery of an order is **calculated** from the manufacturing time added to the date of order
- Manufacturing time of an order is **calculated** from the order qty multiplied with the manufacturing cost of one unit
- The discount on an order is **inferred** from the discount applicability of the customer who issued the order

Rules of this nature support queries on data that does not exist in the system but is capable of being generated. Examples of such queries are:

- Given the employee id, display the hourly rate
- Given the order-number, display the manufacturing costs associated.

In the above examples, hourly rate and manufacturing costs are not stored in the system but are capable of being generated by the production system rules.

3.2.3 Action Assertions

Rules in this category comprise all those statements, which limit or control the actions of the domain (business enterprise, in our case). These rule statements are associated with the dynamic aspect of the application domain. They basically specify constraints on the results that actions can produce. Examples include:

- If employee salary exceeds limit, then display the bonus for the last two years
- If stock qty is less than the threshold, then issue a reorder
- If customer credit is higher than average, then issue a discount.

Action assertion rules are capable of supplying "information" rather than raw data. For example, these rules can support a query like

- Tell me if the given stock item needs to be reordered (response – yes/no)
- Tell me if I am eligible for a discount given my credit rating (response-yes/no)
- Given a transaction activity history, tell if the cost can be reduced (response-yes/no/maybe?)

Providing a query support for information retrieval is a challenging issue. Commonly used query models provide a support for representing data. They are inadequate for capturing information.

3.3 Syntax Analysis of Rules

Rules in a production system have the basic structure

If (clause) + -> (action) +

A rule is fired when all the clauses on the left-hand side of the rule are satisfied i.e., there exist facts in the fact base which match the "if" clauses. The above structure can be broken down into the following structures

3.3.1 Type 1

On(event)+ -> (action)+

- On employee salary as input, calculate the hourly rate

(event) (action)

3.3.2 Type 2

On (event)+ if (condition)+ -> (action) +

- On (employee id as input,) if (a match for employee record found) then (display the other associated employee attributes)

3.3.3 Type 3

On (condition) -> (action)+

- Whenever the number of employees exceeds 'limit', issue an order
- Whenever stock quantity dips below the 'level', issue a reorder request

The *event* clause in rules of Type 1 and 2 is a trigger. The trigger can be external or can come internally from the action clause of other rules.

Action clauses in the rules can be categorized as terminating or non-terminating, observable or non-observable. Terminating actions terminate a rule, they do not lead to further triggering of rules. Non-terminating outcomes can trigger other rules. They provide the internal stimulus to other rules. Observable actions are all those actions, which can be seen externally e.g., display of data. Non-observable actions cannot be seen, but change the state of the fact base by inserting/deleting new facts. These actions can be composed of both terminating and non-terminating clauses.

Rules of Type 3 are triggered automatically whenever the conditions that match their clause in the left hand side are satisfied. Considering the above example, a reorder issue (action) occurs whenever the stock quantity dips below the desired level. Now, we can never say at what instance of time would this happen. With such rules, we can never

deterministically predict when the outcomes, issuing of order and issuing a reorder request (from the above examples) would be observed. The outcome of such "condition action" rules is dependent on the state of the fact base and it is very difficult to predict whether a query for the outcomes of these rules will be answered. Therefore, we narrow down our problem to a subset of rules, of Type 1 and 2.

Our aim is to predict all possible outcomes the production system can produce when the inference engine is active. This would enable an external entity to perceive what data or information can be obtained from the system. This calls for a complete "break down" of all the rule structures to ascertain the observable actions. Interdependencies among rules make this a difficult task. As mentioned previously, rules can be highly intertwined. A rule might be capable of triggering several other rules. All such transitivities need to be deciphered and inferred to make a deterministic evaluation of the system outcomes. For example,

R1: On employee id as input, if employee record found then output employee salary, assert trigger for r2, assert trigger for r3.
R2: On employee salary as input, calculate employee hr-rate
R3: On employee credibility factor as input, calculate discount applicable

Thus in the above scenario, the action of R1 is to trigger both R2 and R3. Thus, on employee id as input to the system, assuming the employee record is found, the outcomes are employee salary, hr-rate and discount allowance. Likewise, there can be many more scenarios of rule transitivities, much more complicated and intertwined. These dependencies need to be deciphered fully to come up with a deterministic set of outcomes capable of being generated by the system.

We propose a solution to the above problem by means of a dependency graph [Gon93]. A dependency graph captures all the rule transitivities present in the rule

structures. An analysis of this graph would give us the set of all the possible outcomes deducible from the system.

3.4 Resolving Rule Transitivities

We model the problem of predicting the system outcomes as follows:

Let $r_1,r_2,\ldots,r_n$ be the rules of type 1 and 2 in consideration.

$t_1,t_2,\ldots,t_n$ be the set of all the triggers

$f_1,f_2,\ldots,f_n$ be the set of all the "if" clauses in the left hand side of rules of type 1 and 2

$c_1,c_2,\ldots,c_n$ set of all the possible observable outcomes .

The problem of resolving rule transitivity now can be stated as,

Given

r_1 : on t_1 -> c_1,t_3

r_2 : on t_2 if (f_1) -> c_2

r_3 : on t_3 -> c_3,c_4,t_4,t_5

r_4 : on t_4 -> c_5,c_6

r_5 : on t_5 -> c_5,c_7

Deduce the following:

c_1, c_3, c_4, c_5, c_6,c_7 can be obtained on t_1

c_2 can be obtained on t_2 (assuming f_1 to be present)*

c_3, c_4, c_5, c_6,c_7 can be obtained on t_3

c_5,c_6 can be obtained on t_4

c_5, c_7 can be obtained on t_5

* There might be situations when rules of Type 2 fail to deliver any results when triggered. This is because the if clauses in such rules may not be satisfied. The wrapper is

required to handle such situations by issuing error statements explaining the insufficiency of data in fact base.

We propose a solution to obtain the above deductions with the aid of a dependency graph. A dependency graph is a directed graph where each vertex in the graph represents a rule and a directed edge between any two vertices indicates the presence of dependence between the two rules. As depicted in Fig 3.1, the directed edge from vertex r1 to r3 signifies the trigger t3, which present in the action clause of r1, can trigger rule r3. This essentially captures the concept that the observable outcomes of r3 are capable of being generated by r1.

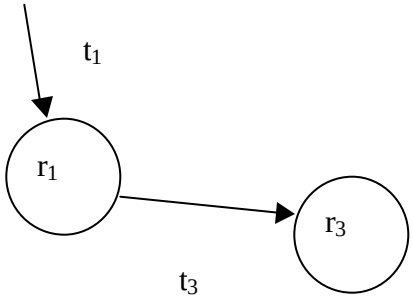

Fig 3.1 Dependence between the rules

Figure 3.2 represents the dependency graph for the scenario presented above. In some sense, this is not a true graph in a mathematical sense as there could be incoming edges to a vertex, which do not have a starting vertex. As in Fig 3.2, the vertex r_1 has an incoming edge t_1, which has no originating vertex. t_1 is an external stimulus that can trigger rule r_1.

We now proceed to examine all the vertices of the dependency graph, one by one to gather the all the observable outcomes capable of being generated from these rules.

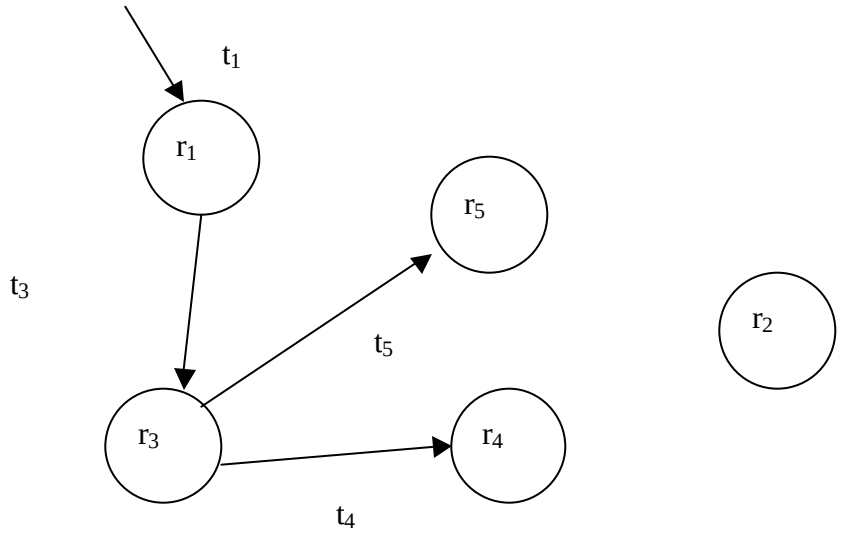

Fig 3.2 Dependency Graph: capturing rule transitivities

We now proceed to examine all the vertices of the dependency graph, one by one to gather the all the observable outcomes capable of being generated from these rules. It is to be noted that any incoming edge on a vertex represents the trigger, which can activate the corresponding rule, and all outgoing edges to other vertices signify the other rules that can be activated by the rule in consideration. Thus, we start at any vertex and record the trigger (incoming edge), the observable outcomes at that vertex, and follow all the outgoing edges to reach the other vertices. This process is repeated for each vertex, till there are no outgoing edges from any of the subsequent vertices. We illustrate this with an example.

Let us start our analysis with vertex r_1 and generate the outcomes that can be produced by this rule. The set of outcomes is developed as we follow the edges coming out of r_1. The stages are depicted in Fig 3.3

State A : c_1 on t_1

State B : c_1, c_3, c_4 on t_1

State C : c_1, c_3,c_4,c_5, c_7 on t_1

State D : c_1,c_2,c_3,c_4,c_5,c_6,c_7 on t_1

This completes the analysis of rule r_1. Likewise, we do this for all the rules to get the final set of queriable information capable of being generated when the inference engine is active.

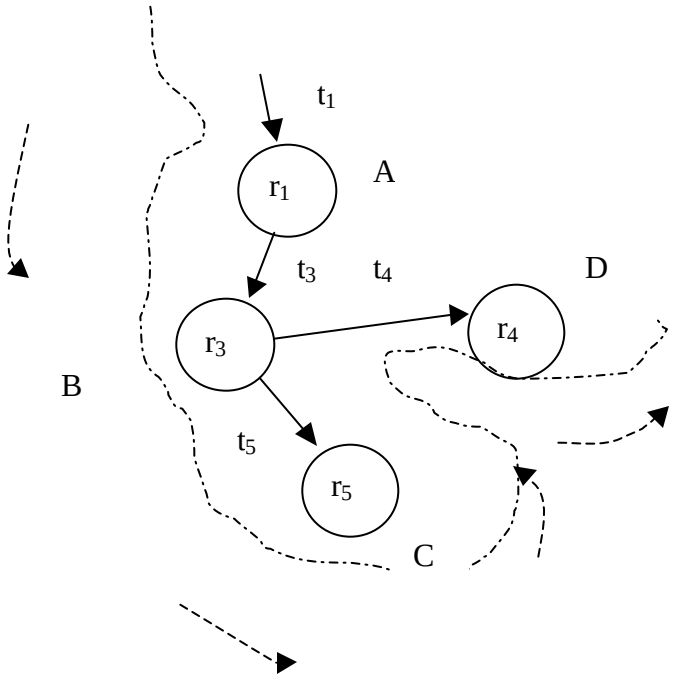

Fig 3.3 Dependency Graph Traversal

We represent this queriable information obtained from the rule analysis in the form of an XML document. As an example,

```
<queriable information>
     <query>
          <results>
               <op>employee.sal</op>
                <op>employee.hr_rate</op>
                 <op>employee.pf</op>
          </results>
          <on>
               <ip>employee.id</ip>
          </on>
     </query>
</queriable information>
```

The DTD for the above XML representation would be

```
<!ELEMENT   queriable
information (query+)>
<!ELEMENT   query (results,
on)>
<!ELEMENT   results (op+)>
<!ELEMENT   op  (#PCDATA)>
<!ELEMENT   on (ip+)>
<!ELEMENT   ip (#PCDATA)>
```

3.5 External Query Model

In this thesis, we have adopted a very basic, tag based structure to model the external queries. Each query in this model has the following format:

<fact name <(input criterion)> <desired output(s)>>

The element fact name is the name of the template class to which the fact being queried belongs. It is followed by a set of input conditions and the outputs desired.

An example would make things clear. We refer to our employee fact base which has the template shown below:

(employee (slot name) (slot id) (slot sal) (slot addr))

Let hr-rate and pension be two derivations and discount applicability (yes/no) be the action assertion associated with this fact base. We explain below the representations of queries searching for data generated by Structural Assertions, Derivations and Action Assertion rules.

Consider a query which searches for the salary of an employee with id =10. This, in a relational model would be:

Select sal from employee where id =10

This, in our model, is represented as

<employee <id=10> <sal>>

A derivation query like

Obtain the hr-rate of an employee with sal=2000

becomes

<employee <sal=2000> <hr-rate>>

A query seeking information about the discount applicability of an employee is modeled as

<employee <id=10> <?discount>>

This model serves well to represent simple selections and projections. It is not very sophisticated and does not have the capability to represent complex, nested queries and joins. Also, queries which seek "information" i.e., action assertions are not well represented. It was stated earlier that very little emphasis has been laid on the development of this model. It is very rudimentary; its development remains one of the future challenges.

This section concludes with a brief discussion on the model used to represent the native results obtained by querying the production system. We propose a simple XML based representation. As an example, consider the results shown below. These were obtained by querying an employee fact base for all the attributes of all the records.

```
<results name =”employee”>
     <op> name =bill,addr = gnv,sal=2000,id =20 </op>
     <op> name =joe addr = iowa,sal=2500,id =22 </op>
     <op> name =mary,addr = ohio,sal=3000,id =23 </op>
</results>
```

Results obtained from structural assertions and derivations follow the above structure. Action Assertions, which give information, rather than data are captured using this model as shown below:

```
<results name=”employee”>
     <op> Discount Applicability –
Yes </op>
</results>
```

As with the query model, this representation suffices to capture the results from basic structural operation queries like selects and projections and derivative results. This model needs to be developed further to enable the representation of results from complex nested queries and joins.

3.6 Wrapper Architecture

Figure 3.4 shows the conceptual design of the wrapper. The functionality of each of its components is described below:

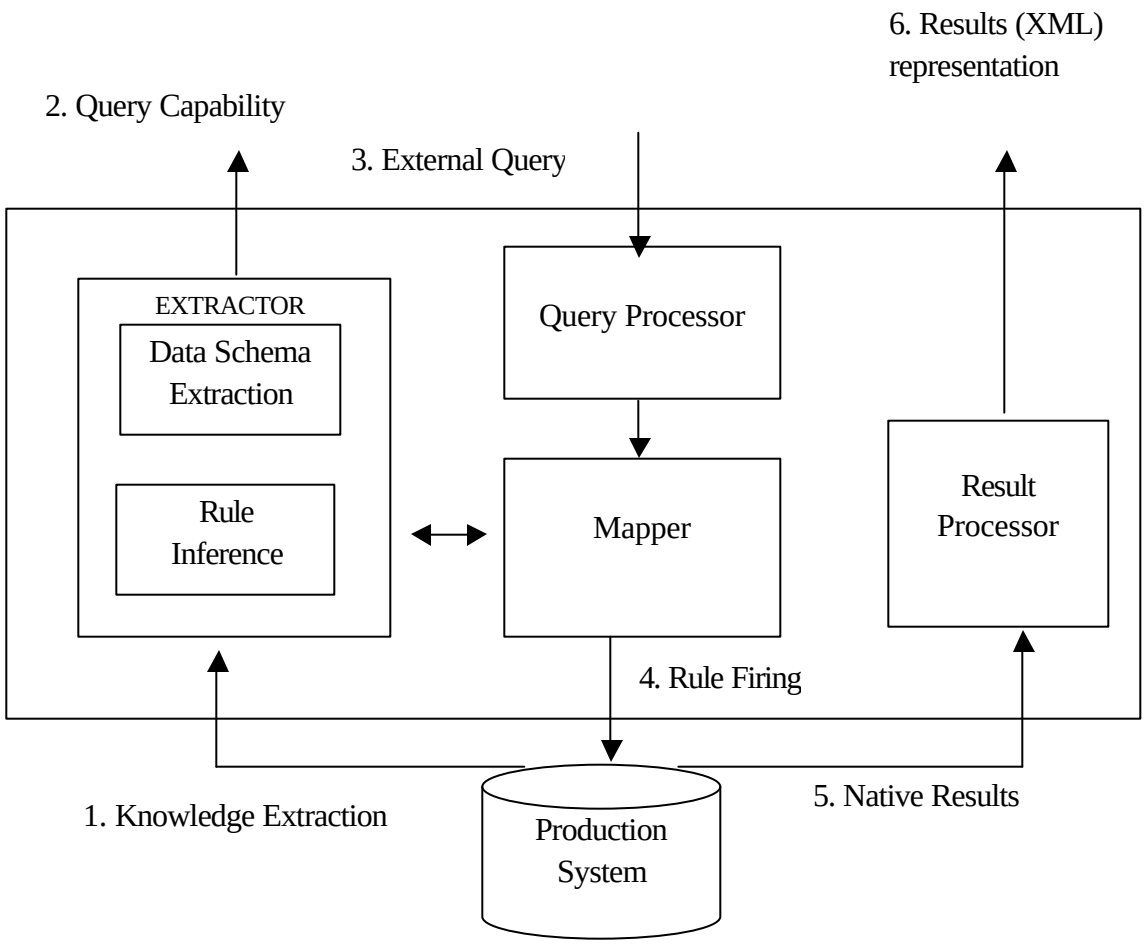

Fig 3.4 Architecture of the Wrapper

As illustrated in the figure, the main components of the wrapper are – Extractor, Query Processor, Mapper and Result Processor.

3.6.1 Extractor

The extractor forms the heart of our wrapper. It is composed of two sub components, the fact schema extractor and the rule extractor. The fact schema extractor extracts the schema of the facts from the production system. It converts the schema in the Jess specification to the external XML specification conforming to the DTD described in section 3.1 The rule extractor extracts all the production system rules. Using the

techniques of rule analysis described in the previous section, it generates the set of all the possible outcomes that can be obtained from the system under specific inputs. This information is given out as an XML document to the external entity thereby enabling him to issue relevant and meaningful queries on the system.

3.6.2 Query Processor

This component handles the incoming queries. All the incoming queries are in the format described in section 3.3. The incoming query has three main elements— the fact template on which the query is issued, the inputs (if any) and the desired outputs. The query is parsed and these elements are extracted form the query and encapsulated in an internal data structure. This internal object is then passed on to another component, Mapper that maps this query against the corresponding production system specific trigger.

3.6.3 Mapper

This component is responsible for mapping the external query against the production system specific trigger. This module collaborates with the rule extractor component, which has already analyzed and inferred the rules to generate a mapping table. This table is a list of all the possible system outputs, which can be generated by the system given the desired inputs. It is an internal, main memory representation of the external XML representation of the queriable information of the system generated by the extractor. This table maintains information about triggers, which lead to the desired outcomes under the given set of input conditions. The mapper does a sequential search on this table, comparing the inputs and outputs of each table entry against the input and outputs of the query object formulated by the query processor. If a match is found, the corresponding trigger along with the query inputs (if any) is inserted in the knowledge

base. The relevant rules get activated to produce the desired results. In case a match is not found, the mapper issues an error statement, asking the external entity to reformulate the query.

3.6.4 Result Processor

The result processor gathers the results generated by the production system and converts them to the external XML representation. There might be cases(section 3.3) when the mapper finds a match but no results are generated. In such cases, the result processor issues relevant statements to the user explaining the absence of the pre-existing facts in the fact base.

CHAPTER 4
IMPLEMENTATION DETAILS

In the previous chapter, we outlined the design issues and the architecture of the wrapper. In this chapter, we focus on the implementation details. We explain the program modules, their functionalities, and the data structures used in the program.

4.1 Implementation Details

The wrapper prototype has been developed in Java using JDK 1.8. As mentioned previously, the wrapper functionality is divided into two main phases. The first phase extracts the queriable information from the production system. The second phase involves query translation and result processing. We now examine the step-by-step implementation of these phases. The program modules and data structures used to implementing these phases are explained in detail.

The main program is a Java class which instantiates the Jess engine and creates a wrapper instance.

```
class  wrapperClass {

Rete Jess = new Rete();    // invoke a
jess engine

Wrapper jessWrap = new Wrapper();   //
invoke the wrapper

}
```

The facts and the rules (in the Jess specification) are loaded in the Jess engine. The wrapper invokes the following modules to implement the first phase ie, fact schema extraction, rule extraction and inference.

```
class Wrapper {

extractSchema();                    //
extract the fact schema

extractRules();                        //
extract the rules

resolvDependence();               //
analyze and infer the rules

finalOutcomes();                    //
generate the query capability
```

The method extractSchema extracts the schema of the unordered facts from the production system fact base. It uses the Java APIs – listDeftemplates(),getNSlots(), getSlotName(), etc to accomplish this. It also extracts the variables related to each template. Variables (section 2.3) represent storage for data and information that can be generated by inference. The fact templates and associated variables are encapsulated in the storage class factObj. factObj has the following structure:

```
class factObj {
   String factName;
        int    nSlots;
   String[] slotName;
   String[] slotType;
   int slotCount;
   static int   factCount;
   String[] var;
   int    varCount;
}
```

The attribute factName stores the name of the template. slotName and slotType capture the template slots and their types.The string array var is used to store the variables associated with the template.

The method extractRules extracts the rules from the Jess engine, using Java APIs listDefrules(),getName(), etc. extractRules calls a method parse(), which parses the rules, extracts the relevant pieces of information i.e., the rule name, rule trigger, the if clauses and the action clause of the rule body and encapsulates it in the ruleObj class. The ruleObj class has the structure shown below:

```
class ruleObj {
   String ruleName;
   int    ruleNum;
   String docString;
   Trigger trigger;
   Fact[] fact;
   Conc[] conc;
   static int ruleCount = 0;
   int factCount = 0;
       int concCount;

}
```

The members are: ruleName - stores the name of the rule, trigger -stores the trigger i.e., the event clause of the rule, fact- stores the entire “if” clauses in the left hand side of the rule, conc- stores all the actions associated with the rule. The objects trigger, fact and conc are objects of class Trigger, Fact and Conc respectively. The class Trigger stores all the triggers in the production system. As mentioned in Chapter 3, triggers represent the event clauses in the body of of rules of type 1 and 2.

```
class Trigger {
   String tname;
   String[] input_var;
   int    varCount;
}
```

The attribute tname stores the name of the trigger. The attribute input_var is a provision of any inputs accompanying the trigger.

The class Fact is used to store all the "*if*" clauses (Chapter3) in the left hand side of a rule.

```
class Fact {
   String name
   String[] attr
   String[] var
   int varCt
}
```

The attribute name stores the name of the unordered fact, the members attr and var store the attributes and variables of the clause that is seen for a match in the fact base.

The class Conc captures all the action statements in the "then" part of a rule

```
class Conc {
  String opr;
  String oprstr;
  boolean flag;
  Trigger  trigger;
}
```

As mentioned in Chapter 3, the actions clauses can comprise triggering and non triggering statements. The boolean attribute flag indicates whether the action is terminating or non terminating in nature. Accordingly, the content is stored as either a trigger object or as a terminating string.

Having extracted the fact schema and the rules, we proceed to analyze the rules to determine the queriable information. As mentioned in Chapter 3, rule analysis is performed by generating a dependency graph and analyzing each vertex in the graph. This is implemented by the modules resolvDependence() and finalOutcomes(). The method resolvDependence generates the dependency graph, depicting the dependencies

between the rules. The method finalOutcomes() analyzes each node of this graph and generates the set of outcomes that can be generated by the production system under the specific input conditions.

The method resolvDependence() generates the dependency graph which is represented internally as a transitivity list. The data structure Graph is used to represent the dependency graph in the form of a transitivity list of nodes. Each node is an object of type dependence.

```
class dependence {
   int    ruleNum;
   String trigger;
   Fact[] fact;
   Conc[] conc;
       int links;

}
```

We could have used the class ruleObj to model the nodes of the dependency graph. However, ruleObj has a lot of other attributes which are irrelevant for processing at this stage. The adoption of a separate class here makes programming more clear and convenient. All the members of this class are used for the same purpose as the members of ruleObj. The member links is present in the head nodes of each chain of nodes in the transitivity list. It keeps a count of the nodes that are in that chain. The method finalOutcomes() analyzes each node in this list to generate a final set of input – output combinations. It uses an array, outcomeList to capture these combinations. The data structure outcomeList is an array of objects of class capability. The class capability has the structure shown below:

```
class capability{
   String trigger;
   int ruleNum;
   String[] input;
   String[] output;
   int[] cruleNum;
   int ccount
       static int fcount;
   int rcount;
   int ip;
   int op;
}
```

The member input stores all the inputs (if needed) to generate the outputs. Output stores all the outcomes that can be obtained under the given input conditions. The member trigger captures the triggering string which when asserted in the Jess fact base would lead to the outcomes. The methods showSchema() and showOutcomes() of the class wrapper convert the internal representations of the fact templates and the possible outcomes to the specific XML representations (section 3.1,3.4). This concludes the first phase of the wrapper generation.

The second phase of the wrapper, which is query translation and result extraction is performed by the following modules.

```
class Wrapper {
     …….
     ……..
     …….
            getInput();
     findTrigger();
            fireTrigger();
}
```

The module getInput() implements the component query translator, as explained in Chapter 3. It takes the incoming queries (section 3.4) and converts them into an internal

main memory representation. The external query is composed of three main elements – the fact name (all queriable data and information are associated with a fact template), the inputs (if any) and the desired outputs. These elements are extracted from the input query and encapsulated in an internal query representation object. The object is an instance of class query having the following structure:

```
class query {
     String   name;
     String[] input;
     String[] output;
     String[] var;
     int ip;
     int op;
}
```

The member, input stores the input variable in the input query. The outputs in the external query are stored in the array output. The array var stores all the input values. For example, consider the external query:

<employee <id=10> <sal>>

It is stored as a query object q where:

```
q.name = employee
q.input[0] = id
q.var[0] = 10
q.output[0] = sal
```

The method findTrigger () implements the component mapper, which maps this query against the corresponding trigger. findTrigger () searches for a matches for the internal query object in the array of outcomeList . As explained previously, the outcomeList is a sequence of outcomes capable of being generated under the specific inputs. The trigger, which would generate each of such outcomes, is also stored in the

list. This module searches for a match between the inputs and outputs of the internal query object and each element of the outcomeList. If a match is found, it is asserted in the knowledge base. This is handled by the method fireTrigger. In case no match is found, the module generates an error statement asking the external entity to reformulate the query. The module showResults() gathers the results that are generated by the Jess engine, converts them to the XML representation.

CHAPTER 5
PERFORMANCE EVALUATION

In this chapter, we provide a qualitative analysis of the wrapper prototype. We first demonstrate the correctness of the results generated by the wrapper by testing and verification methods. We proceed to justify the use of the wrapper against the other possible ways of extracting the data from the production system.

All the experiments described below were performed the DataBase Research and Development Center. We ran the test cases on Lima workstation (Pentium II 400 MHz, 130,472 KB RAM). The production system was set up using Jess v5.0.

5.1 Testing

A black box testing approach was adopted to check the wrapper functionality for the validity and accuracy of results. We developed query test cases to search for the data and information generated by rules having the semantic functionality of structural assertions, action assertions and derivations. The syntactic structure of rules in each of the above category confirmed to the type 1 and type 2 classification, explained in Chapter 3. The rules, conforming to the above specifications were supplied to the Jess engine in the Jess language. The rules spanned a broad range in terms of the degree of inter dependencies among them. The rule set ranged from simple rules with a single terminating action clause to highly complex rules, each rule having the capability to trigger four to five other rules.

For each external query, we developed a Jess program which enables the same data, as sought by the query to be extracted from the Jess engine. The queries were fed to the wrapper and simultaneously, the corresponding Jess programs were executed. The results generated by the wrapper were compared and tallied against those generated by the individual Jess programs .In all of our test cases, the results from both the sources were found to be consistent with each other.

We present the test case scenarios. To start, the Jess knowledge and fact base were fed with facts and rules. A snapshot of the fact and knowledge base is shown below:

```
Defining the Templates for unordered facts

(deftemplate employee (slot id) (slot name) (slot
addr) (slot sal))
(deftemplate order (slot order-num) (slot qty) (slot
cost) (slot date-of-order))
(deftemplate stock (slot stk-num) (slot stk-qty))
(deftemplate items (slot item-id) (slot item-name)
(slot unit-price))
(deftemplate cust_order (slot order-num) (slot item-
id) (slot qty) (slot date-of-order))
(deftemplate customer (slot cust-id) (slot cust-
name) (slot order-num))

Defining the variables associated with each fact
template

(defglobal ?*employee.hr-rate* )
(defglobal ?*employee.pension* )
(defglobal ?*cust_order.discount* )
(defglobal ?*cust_order.manufacturing_cost* )
```

Facts entered in the facts base:

```
(employee (id 1) (name joe) (addr newyork) (sal
2345))
(employee (id 2) (name mary) (addr gainesville)
(sal 2338))
(employee (id 3) (name bill) (addr iowa) (sal
1200))

 (order (order-num 12) (qty 20) (cost 12000)
(date-of-order 01/21/01))
 (order (order-num 13) (qty 2) (cost 120) (date-
of-order 01/23/99))
 (order (order-num 14) (qty 56) (cost 20000)
(date-of-order 05/06/01))

 (stock (stk-num 23) (stk-qty 24))
 (stock (stk-num 2) (stk-qty 0))
 (stock (stk-num 12) (stk-qty 0))

 (items (item-id 12) (item-name rivets) (unit-
price 250))
 (items (item-id 2) (item-name bolts) (unit-price
25))
 (items (item-id 23) (item-name jacksaw) (unit-
price 200))

(cust_order (order-num 10) (item-id 2) (qty 25)
```

The following rules were built in the rule base:

```
Defining the rules
;1. Structural Assertion

; a. Enter the employee id to get the
salary (selection+ projection)
(defrule get-salary-for-employee " Enter
employee id to get the salary " ?cmd <-
(compute-salary ?id )(employee (id ?id)
(sal ?sal)) => (printout res " employee.sal
" ?sal crlf) (retract ?cmd))

;b. Display all the attributes of all
employees (projection)
(defrule select-all-employee "Select all
```

.. cntd

```
; c. Get the order numbers placed by the customer ,
input- customer id.
(defrule get-all-orders "get all the order nums
placed by the customer" (getallorder ?cust-id)
(customer (cust-id ?cust-id) (order-num ?order-num))
(cust_order (order-num ?order-num) (item-id ?item-
id)) => (printout res " items.item-id " ?item-id
crlf))


; d. Enter the order number to get the cost
associated with the order (selection+ projection)
(defrule get-cost-of-order "Enter the order num to
get the cost " ?cmd <- (get-cost ?order-num ) (order
(order-num ?order-num) (cost ?cost)) => (printout res
" order.cost " ?cost crlf) (retract ?cmd))


;2 Derivations

; a. Calculate employee hourly rate given employee id
(defrule get-rate "enter id to get rate " ?cmd <-
(get-rate ?id)(employee (id ?id) (sal ?sal)) =>
(printout res " employee.hr-rate " (* ?sal 0.65)
crlf) (retract ?cmd))


; b. Calculate the manufacturing cost associated with
the cust_order, input- cust_order num
(defrule get-manufacturing-cost "enter order-num to
get the manufacturing cost associated"?cmd <- (get-
mancost ?order-num)(cust_order (order-num ?order-num)
(qty ?qty)) =>(printout res
"cust_order.manufacturing_cost " (* ?qty 200) crlf)
(retract ?cmd))

(defrule get-sal-rate-pension " Enter the id to get
salary, hr-rate and pension " ?cmd <- (get-sal-rate-
pension ?id)(employee (id ?id) (sal ?sal)) =>
(printout res " employee.sal " ?sal crlf) (assert
(get-rate ?id )) (assert (get-pension ?sal)) (retract
?cmd))
```

```
… cntd

;3 Action Assertions

; a. Tell if a stock item needs to be reordered
(yes/no) input- stock-num
(defrule reorder-info "enter stk-num to find if it
needs to be reordered" ?cmd <- (find-reorder-info
?stk-num ) (stock (stk-num ?stk-num ) (stk-qty
?stk-qty &:(eq ?stk-qty 0))) => (printout res "
Reorder info yes/no " "Yes" crlf) (retract ?cmd))

; b. Tell if discount can be obtained for a
cust_order. input, cust-order-num
(defrule find-discount "enter order-num to find if
discount is applicable " ?cmd <- (find-discount
?order-num) (cust_order (order-num ?order-num
)(qty ?qty &: (> qty 20))) => (printout res "
```

5.1.1 Queries exploiting Structural Assertions

1. Select sal from employee where id=1

<employee <id=1> <sal>>

<results name=”employee”>
<op> sal = 2345 </op>
</results>

2.Select attributes of all the employees

<employee <> <name,id,addr,sal>>

<results name = “employee”>
<op> id =1, name = joe, addr = newyork, sal 2345 </op>
<op> id= 2, name = mary, addr= gainesville, sal 2338 </op>
<op> id =3 ,name = bill, addr= iowa,sal =1200 </op>

</results>

3.Select cost from order where order-num= 13

<order <order-num=23> <cost>>

<results name = “order”>
<op> cost =120 </op>
</results>

4. Select stk-qty from stock where stk-num=6

<stock <stk-num =6> <stk-qty>>

Sorry, insufficient facts in fact base to answer the query.(stock item with stk-num

does not exist in the fact base)

5.1.2 Queries Exploiting Derivations

1. Calculate hourly rate of employee given employee id

<employee <id=2> <hr-rate>>

<results name =”employee”>
<op> hr-rate= 151.9 </op>
<results>

2. Calculate the manufacturing cost associated with the cust_order, given the

order number

<cust_order <order-num=12> <manufacturing-cost>>

<results name =”cust_order”>
<op> manufacturing-cost =4000 </op>
</results>

3. Calculate the employee pension, hr-rate and salary given id

<employee <id=2> <pension,hr-rate,sal>>

<results name =”employee”>
<op> pension = 25000, hr-rate = 151.9, sal = 2345 </op>
</results>

5.1.3 Queries Exploiting Action Assertions

1. Given a stock item number, tell if it needs to be reordered (yes/no)

<stock <stk-num=2> <?reorder>>

<results name="stock">
<op> Reorder info – Yes </op>
</results>

2. Given cust-order number, tell if discount can be obtained for the order

<cust-order <order-num=10> <?discount>>

<results name="cust_order">
<op> Discount Applicability – Yes </op>
</results>

5.2 Comparison with Other Approaches

In justifying the use of this wrapper tool, we first look at the other possible alternatives that could be used to integrate the production system in a collaborative environment.

One approach to querying a Jess based production system would be to write small snippets of Jess code for each query. This approach apart from being labor intensive and time consuming can only be done by the knowledge engineer, i.e., the person who designed the system. For a naïve user, who does not have an understanding of the knowledge and the capability of the system, this approach is not helpful.

Many expert system shells provide for interactive capabilities by providing a user interface. This interface is usually built by the knowledge engineer while the system is being constructed. A user interface however, is hard coded and customized to a specific application domain. Even for such interfaces, the user is expected to have an idea of the internal inferencing capability of the system. Also, if the knowledge within the domain changes, the interface needs to be built right from scratch. The advantage of this wrapper

is that it adapts itself to any domain, which needs to be queried. Also, it enables a naïve user to perceive the data and information generating capability of the system.

CHAPTER 6
CONTRIBUTIONS AND FUTURE WORK

6.1 Summary

This thesis is an attempt to enable an efficient, seamless integration of production systems in an existing data/information sharing infrastructure. Production systems are characterized by their capability to infer new data and knowledge from the existing set of data. They are best suited when knowledge about a domain naturally occurs in rule form, e.g., business rules of an enterprise. Our wrapper tool facilitates data exchange and transfer between a production system and other information units operating in a collaborative environment. Most data exchange and transfer environments embark upon a neutral information model to support data exchange. The wrapper enables data retrieval from the production system by translating the queries in the neutral model to the production system specific queries.

Our wrapper component works in two phases. In the first phase, it encapsulates the production system, extracts the schema of the data and rules. Rules are inferred and analyzed to determine the data and information that can be generated by the system in future. This knowledge about the query support capability of the production system is given out to the external user as an XML schema. This schema enables an external user to formulate queries. In the second phase, the wrapper takes the queries and translates them to the production system specific triggers. This translation is done automatically inside the wrapper. The trigger is asserted in the production system fact base and fires the

rule engine. The native results generated, are gathered by the wrapper, converted to an XML format and displayed out to the user.

6.2 Contributions

The most important contribution of this work is the inferencing of knowledge embedded inside a production system. The wrapper facilitates the production system to be queried for not only the data that resides inside the system but also for all the data and information that can be generated by the system. Analyzing and inferencing the knowledge structures to make a deterministic evaluation of the production system's query capabilities forms the core of this work.

The final deliverable of this work, the wrapper, is generic in nature. Although our experimental domain was tailored to meet the needs of an enterprise, this tool can wrap any application domain.

Many production systems are accompanied by a user interface which facilitates data and information retrieval. This interface is usually built by the knowledge engineer while the system is being constructed. A user interface however, is hard coded and customized to a specific application domain. Also, even for such interfaces, the user is expected to have an idea of the internal inferencing capability of the system. If the knowledge within the domain changes, the interface needs to be built right from scratch. The advantage of this wrapper is that it can mould itself to any domain, "learn" the domain capability and facilitate data translation.

This work is an important contribution in the field of reverse data engineering. In attempting to wrap a production system, we have made an attempt to analyze the knowledge within the system domain. This reengineering work is often needed when the

operational knowledge of a domain (business units in this context) is lost among huge lines of code or embedded deeply within convoluted knowledge structures. This thesis lays the foundation for systematically analyzing the semantics and structures of these rules, to extract the business knowledge stored within them.

6.3 Future Work

Rule analysis, which forms the core of this work, has been limited to a subset of rules with a definite structure. Extension of the wrapper to exploit the data and information generation capabilities of rules of Type 3 remains an issue.

In this thesis work, we did not adequately focus on the external query model. A query model that can support queries for both data and information needs to be developed and standardized.

This thesis provides groundwork for analyzing the knowledge structure and semantics of production systems. This can be extended further to capture the essence of the semantics. For example, consider a rule which calculates hourly rate of a person based on a mathematical logic say, *hr-rate = salary* factor*. Currently, we are focussing on the fact that this rule is capable of generating the information, hr-rate about the person. This can be extended to capture the logic behind this computation, which is *salary*factor*. This logic, which drives the operations of several enterprise systems, is often lost and buried among huge lines of code. Extracting this operational logic from existing legacy systems remains a major challenge in the field of reverse engineering.

REFERENCES

[Aik83] Aikins, J. S., J. C. Kunz, E. H. Shortliffe and R. J. Fallat. PUFF: An Expert System for Interpretation of Pulmonary Function Data. Computers and Biomedical Research, 16:199-208,1983.

[Ash97] Ashish, N. and C. Knoblock. Wrapper Generation for Semi Structured Internet Sources. In Proceedings of the Workshop on Management of Semistructured Data, In conjunction with PODS/SIGMOD-97, Ventana Canyon Resort, Tucson, Arizona, 1997.

[Bar83] Barr, A. and E. Feigenbaum. The Handbook of Artificial Intelligence. Addison-Wesley Publishing Company, Reading, MA, 1983.

[Bob77] Bobrow, T.G. and T. Winograd, An Overview of KRL – A Knowledge Representation Language. Artificial Intelligence, 1:3-46, 1977.

[Buc78] Buchanan, B.G. and E.A. Feigenbaum. DENDRAL and Meta DENDRAL: Their Applications Dimension. Artificial Intelligence, 11: 5-24,1978.

[Buc84] Buchanan, B. G. and E. H. Shortliffe, Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project. Addison-Wesley Publications, Reading, MA, 1983.

[Car94] Carey, M.J., L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J.H. Williams and E.L. Wimmers. Towards Multimedia Information System: The Garlic Approach. IBM Almaden Research Center, San Jose, 1994.

[Cha94] Chawathe, S., H. G. Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of Heterogeneous Information Sources. In Proceedings of IPSJ Conference, 7-19, Tokyo, Japan,1994.

[Col87] Colmerauer, A. Opening the Prolog III Universe. Byte, 12:177-182, August 1987.

[Coo93] Cooper, M. C. and L. M. Ellram. Characteristics of Supply Chain Management and the Implications for Purchasing and Logistics Strategy. 4:13-24, 1993.

[Dat87] Data Interchange Standards Organization, http://www.disa.org/edi,1987.

[Dij90] Dijkstra, E.W. and C.S. Scholten, Predicate Calculus and Program Semantics, Springer-Verlag, New York, 1990.

[Ebus00] e-Business Integration Drives EAI. www.eaijournal.com, 1997.

[Eik99] Eikvil, L. Information Extraction from World Wide Web - A survey. http://www.nr.no/research/samba/tm_survey.ps, July1999.

[Ext98] Extensible Markup Language, http://www.w3.org/XML/,1998.

[For82] Forgy, C.L. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem Artificial Intelligence, 19:17-37,1982.

[Fri97] Friedman-Hill, E. Jess – The Java Expert System Shell http://herzberg.ca.sandia.gov/jess, 1997.

[Gon93] Gonzalez, A. J. and D. D. Dankel. The Engineering of Knowledge Based Systems. Prentice-Hall Publications, Englewood Cliffs, New Jersey, 1993.

[Gru98] Gruser J. R., L. Raschid, M. E. Vidal and L. Bright. Wrapper Generation for Web Accessible Data Sources, In Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems, New York City, New York, 14-23, August 20-22, 1998.

[Ham88] Hammond, K.J. Opportunistic memory: Storing and Recalling Suspended Goals. Proceedings of a Workshop on Case-Based Reasoning, J.L. Kolodner (ed.), 154-168, Clearwater Beach, FL, 1988.

[Jag89] Jagannathan V., R. Dodhiawala, and L. S. Baum, Blackboard Architectures and Applications, Academic Press, New York, 1989.

[Kor97] Korth, H. F. and A. S. Silberschatz, Database Research Faces the Information Explosion, CACM 40(2): 139-142, 1997.

[Koz98] Kozen, D. Results On The Propositional Calculus, Theoretical Computer Science, 27: 333-354,1983.

[Kra83] Krasner, G. SmallTalk-80: Bits of History, Words of Advice. Addison-Wesley, Reading, MA, 1983.

[Kus97] Kushmerick, N., D.S. Weld and R. Doorenbos. Wrapper Induction for Information Extraction, In Proceedings of the 15th International Joint Conference on AI (IJCAI-97), 729-735, Nagoya, Japan, 1997.

[Lab97] Labio, W. J., Y. Zhuge, J. L. Wiener, H. Gupta, H. Garcia-Molina, J. Widom. The WHIPS Prototype for Data Warehouse Creation and

Maintenance. In Proceedings of the ACM SIGMOD Conference, Tucson, Arizona, May1997.

[Liu98] Liu L., C. Pu and W. Han. XWRAP: An XML enabled Wrapper Construction System for Web Information Sources, In Proceedings of International Conference on Data Engineering (ICDE), 611--621, 2000.

[Mey88] Meyer, B. Object Oriented Software Construction. Prentice Hall, New York, 1988.

[Min75] Minsky, M. A Framework for Representing Knowledge. In the Psychology of Computer Vision, McGraw-Hill Publications, NewYork, 1975.

[Mus98] Muslea I., S. Minton and C. Knoblock. STALKER: Learning Extraction Rules for Semistructured, Web-based Information Sources. In Proceedings of AAAI'98: Workshop on AI and Information Integration, Madison, Wisconsin, July 1998.

[Ols00] Olsen,G. An Overview of B2B Integration, EAI Journal, 5: 24-32, 2000.

[Pap95] Papakonstantinou Y., A.Gupta, H.G. Molina and J. Ullman. A Query Translation scheme for Rapid Implementation of Wrappers, ACM Computing Surveys, 18:323-364,1995.

[Pap96] Papakonstantinou, Y., H.Garcia-Molina and J. Ullman. Medmaker: A Mediation System Based on Declarative Specifications, International Conference on Data Engineering, New Orleans, 1996.

[Qui68] Quillian, M.R. Semantic Memory: In Semantic Information Processing, MIT press, Cambridge, MA, 1968.

[Red76] Reddy, R., L. Erman, R. Fennell, and R. Neely. The HEARSAY Speech Understanding System: An Example of the Recognition Process. EIII Transactions on Computers, C-25: 427-431, 1976.

[Ril98] Riley, G. CLIPS, A Tool for Building Expert Systems http://www.ghg.net/clips/CLIPS.html, 1998.

[Sub95] Subrahmanian,V.S., S. Adali, A. Brink, R. Emery, J. Lu, A. Rajput, T. Rogers, R. Ross and C. Ward. HERMES: A Heterogeneous Reasoning and Mediator system. Technical report, University of Maryland, College Park, 1995.

[Yee99] Yee A. Order out of chaos: Understanding Business to Business Integration Patterns. http://b2b.ebizq.net/ebiz_integration/yee_1.html, September 1999.

[Wel96] Wells, D. A survey report on wrappers http://www.objs.com/wrappers, 1996.

[Zho96] G. Zhou, R. Hull, and R. King. Generating Data Integration Mediators that use Materializations. Journal of Intelligent Information Systems, 6: 199-221, 1996.

BIOGRAPHICAL SKETCH

Archana Raghavendra was born on November 6, 1977, in Bhilai, Madhya Pradesh, India. She received her bachelors in computer science and engineering from Regional Engineering College, Warangal, India in May 1999.

She joined the University of Florida, Department of Computer and Information Science and Engineering in August 1999 to pursue her interests in computer science. She has been working under Dr Joachim Hammer in the Database Research and Development Labs since January 2000.

Her areas of interest are operating systems, e-business and data modeling. She wants to pursue a career in software development after graduation.