4.1 Introduction

Knowledge representation deals with the structures used to represent the knowledge provided by the domain expert or experts [28]. Efficient knowledge representation is the key to overall success of expert systems. Although there is no single structure to represent knowledge in the most effective manner, a number of different approaches are suitable based on the nature of the problem. Some problems may require the use of more than one knowledge representation structure. The commonly used knowledge representation structures are

- production rules
- frame based schemes
- logic schemes.

Production rules are the most commonly used representation for encoding domain knowledge. However, they are inadequate for defining domain objects and static relationships among these objects [21]. Object oriented representations or frame systems can represent such domain knowledge effectively. The class taxonomies in object oriented representations provide the knowledge engineer with a powerful tool for structuring the domain descriptions and integrating the different object types into a unified coherent system model. Thus object oriented representation handles effectively those areas of knowledge representation where production rules are inadequate. The integration of object oriented representations with the production rules to form an hybrid representation can successfully meet the knowledge representation requirements in many domains.
Time plays a crucial role in real time applications. For example, the monitoring system of an aircraft jet engine must keep changes in parameters like thrust, air flow, air temperature to keep track of the engine performance. These changing quantities must be recorded to answer queries about system health, performance degradation and make projections about factors like reliability. Hence it is necessary to preserve data over a period of time to answer such historical queries. Further, as physical systems are slow changing entities, the historical data can be used to produce early warnings based on historical trends. Recently, there is a growing interest in incorporating the time dimension into the data models. Most of this work is based on the relational model. Time is added as a special attribute of the relation. Some models assume non first normal form relations and use tuples to record attribute history. However, it is widely recognised that the relational model is inadequate to capture the semantics of complex entities that occur in real time domains [95]. Object oriented data models with their properties data encapsulation, inheritance and complex attributes can capture the semantics of complex entities. However, these models are static and offer only a snapshot view of the world. Suitable attributes and facets have to be defined to incorporate the time dimension in object oriented data models.

The next section discusses an object oriented data model which incorporates the time dimension. In Section 4.3, an object oriented augmented rule structure and taxonomy has been defined. Section 4.4 presents the criteria for verification of the rule bases. The schemes for rule base verification using the Extended Petri Net model have also been presented in Section 4.5.

4.2 Object Structure and Management

The factors to be considered in choosing an appropriate time representation are [47]

- Primitive time entity: This can be either an instance representation (time point) or an interval representation.
- Time ordering: Linear ordering or a branching time or circular time
• Specification of time structure: Time can be mapped to fundamental types like integers, real numbers or derived types like calendar day

• Time boundedness: In point representation, this refers to the time span of the problem and in interval representation it refers to open or closed intervals

• Time metric: The metric for measuring the distance between two time points or intervals

Continuous streams of input data imply that the data values are instantaneous values observed in the domain. The point representation of time leads to a correct representation of data values. Hence the point representation is chosen. The continuous arrival of input data also presents an implicit ordering of all data values available to the system. So, a linear ordering is chosen. Since the rate of arrival of data depends on the combination of response times of the sensor and the monitoring systems, it is difficult to fix points in time at which data will arrive into the system. So, the time points are mapped to the set of real numbers. This enables to record data arriving at arbitrary time points. Further, it is not possible to put a time limit after which data will cease to arrive. Hence, there is no bound on the time points. Every time point will have a predecessor and successor. The time is in seconds, which is the most common metric used in most real time systems.

This ontology is quite suitable for the class of applications targeted at. But it could be unsuitable for a different class of applications where the interval representation of time is appropriate. Object oriented models are extendable. Intervals and associated semantics can be easily defined as specialisation of the basic time point model.

Data obtained in real time systems possesses two distinctive features. They are time at which the data are obtained and the life span of the data i.e. timeout. Every data value in the environment has a birth and a life span. For example in a dynamically changing system the fact that a certain pressure is 300 Ksc has little or no meaning. But the fact that the pressure at time 500 sec is 300 Ksc is meaningful and has immense value to the agent. For example if the timeout (i.e. life span) of the pressure is given as 10 sec, it means that the pressure value 300 Ksc has
no meaning after 510 sec unless it is updated by the sensor. So it is essential to represent the time of obtaining a data value in addition to the value. Thus data in a real time system is a triple \( <v, t, to> \), where \( v \) is the data value and \( t \) is the time at which the value is obtained and \( to \) is the timeout of the data item. Object oriented models do not provide for explicit representation of these properties of data. So an additional facet called \textit{timeout} is defined for attributes in the classes.

In REX, a base class \textit{OBJECT} has been defined. All object classes are derived from the base class \textit{OBJECT}. It provides the clock service which is so essential to the real time system. In addition to clock service, \textit{OBJECT} provides generic function implementation for access to members of the classes. The structure of \textit{OBJECT} is

\begin{verbatim}
class OBJECT {
  time;
}
\end{verbatim}

A class named \textbf{Pressure} with attributes \textit{line-pressure} and \textit{accumulator-pressure} whose timeouts are 2 sec and 4 sec respectively the class definition would look like

\begin{verbatim}
class Pressure {
  time;
  line_pressure;
  time@line_pressure;
  accumulator_pressure;
  time@accumulator_pressure;
}
\end{verbatim}

Two new parameters are created when the class \textbf{Pressure} is defined viz. \textit{time@line_pressure} and \textit{time@accumulator_pressure}. The parameter \textit{time@line_pressure} indicates the valid life time of the attribute \textit{line_pressure}. It is the time after which the value of \textit{line_pressure} is no longer valid. Another policy adopted by REX, and similar to many OODB implementations is that any update to the attributes results in the creation of a new instance. All references to objects of a given class refer to the latest instance created unless explicitly specified otherwise. This is because in a real
time reasoning system a new data value from the sensor invalidates the old data value. However for calculating trends, earlier values are also available.

The update behaviour of objects is brought out in the following example. Let there be an object of class Pressure with object-id 1 at time 0. If the attribute line-pressure is updated time 1 to value 450 a new object with a different object identifier 2 is created. The two objects would be as shown below.

```
object-id :1 {
    time : 0
    line_pressure : 200
    time@line_pressure : 2
    accumulator_pressure : 5000
    time@accumulator_pressure : 4
}
object-id :2 {
    time : 1
    line_pressure : 450
    time@line_pressure : 3
    accumulator_pressure : 5000
    time@accumulator_pressure : 4
}
```

It is observed that the parameter time@accumulator_pressure remains unchanged since the value of accumulator_pressure is obtained at time 0. Any references to the object of class Pressure are directed to the object with object-id 2. Whenever the value for accumulator_pressure is requested the value 5000 is returned only if

\[ \text{currenttime} \leq \text{Pressure.time@accumulator_pressure}. \]

In this way retrieval is allowed only for attributes which still have a valid life time.

A limited number of instances of all classes are maintained in the object Instances space for access by rules. If the number of instances created crosses this limit the Object Manager shifts old instances to secondary store. Old object instances are
temporarily brought to *Object Instances space*, if required for trend calculations. Lockheed Expert system shell (LES) [64] provides for storage of temporally varying attribute values within its frame structure. However LES provides for storing only a fixed number of data values.

### 4.3 Augmented Rule Structure and Taxonomy

Real time knowledge based process monitoring system tracks process variables intelligently so as to detect faults and perform malfunction diagnosis. A majority of the process applications are characterised by large data sets and equally large malfunction hypotheses space.

Usually operators are provided with manuals which define alarms and procedures to isolate and repair faults. These manuals are organised on the basis of symptom-cause relationships and represent the compiled knowledge about the system behaviour. Real time knowledge based systems should make use of this compiled knowledge. Structured representation of knowledge becomes sine qua non for efficient search in a large domain knowledge space. Some of the well known structured techniques are frames, rules, causal models and case based indexing systems. Rules offer an easy and efficient way of representing symptom-cause relationships and associated compiled knowledge. Hence production rules are chosen as the knowledge representation structure.

The domain knowledge in temporal systems has features not present in knowledge of traditional application domains. They are

- knowledge about temporal relationships of events and their consequences
- knowledge, based on trends established in the system dynamics.

The traditional knowledge representation formalisms like rules, semantic nets and frames support representation of knowledge about static snapshots of the domain. They do not provide for representing knowledge that takes into account the dynamic nature of the domain (like temporal spacing of events and event sequences).
A knowledge representation scheme is a surrogate for the real world entities it pur-
ports to represent [11]. Further it provides a framework for modelling the problem
domain. Inadequate representations result in loss of fidelity of the model. The sig-
nificant consequence being the errors in conclusions arrived by the reasoning process.
This imperfection must be reduced to the extent possible by suitably augmenting
the representation scheme. The current rule systems lack facilities to represent the
semantics of the application and temporal information. This is also true for other
KR schemes like frames and semantics nets. In REX, the rule structure is augmented
to represent knowledge about temporal properties and behaviour. Further in con-
sonance with the aim of building an object oriented knowledge based system and
treat all entities as objects, rules are treated as objects derived from the base class
OBJECT with a specialisation hierarchy. The rule taxonomy is shown in Figure
4.1 according to the notation given by G. Booch [4]. There are three types of rules.
They are Autonomous rules, Clock synchronised rules and Spanning rules. The clock
synchronised rules and spanning rules are specialisations of the autonomous rules.
Spanning rules are further categorised as Event spanning rules and Time spanning
rules.

4.3.1 Autonomous Rules

Autonomous rules are generic rules from which all other rule types are derived. Fir-
ing of autonomous rules is not attached to any trigger events or enabling conditions,
hence the name. If the work area contents match the premises of autonomous rule,
the rule is scheduled for firing. The structure of an Autonomous rule is

Priority
Premises
Actions
Hold

The priority of the rule indicates the importance of the rule in the total rule set.
This information is used for scheduling rule firings. The premises are a conjunction
Figure 4.1: Rule Taxonomy in REX
of a set of conditions. The satisfaction of the premises makes the rule eligible for firing. The actions are the consequents of the rule. Actions assert new beliefs about the system state and these assertions are entered in the work area. An example rule is,

Rule A1:

Premise: Pressure.accumulator_pressure > 1000 Ksc
Action: Hydraulic_system.status = OK.

Real time monitoring systems track the process variables to ascertain the normal behaviour of the system under investigation and carry out fault detection by differentiating between normal and abnormal conditions. The Hold slot in the rule structure asserts a fault state of the system under investigation. Hold is an action which asserts a system malfunction and terminate a reasoning thread. Rules with non empty Hold slots have empty Action slots and vice-versa. Usually, a rule with a non empty Hold slot terminates a reasoning thread by establishing a fault state of the system. An example of a rule with a non empty Hold slot is

Rule A2:

Premise: Pressure.line_pressure < 100 Ksc and Yaw_fb > 15°
Hold: [message "System_controllability is negative."]

4.3.2 Clock Synchronised Rules

Physical systems do not react instantaneously to inputs but do so with a certain time lag. For example in an aerospace vehicle the reaction of the fins to the auto pilot control is not immediate but occurs after a definite time lag. The auto pilot of the vehicle, will expect the fins to follow the commands with this time lag. If the expected action of the fins fails to take place, a warning is signalled to the ground control immediately for remedial action. Systems monitoring such physical processes should conduct their investigation of the process by anticipating these time delays. Autonomous rules clearly do not cater to these requirements. We propose a specialisation of autonomous rules called clock synchronised rules to model such
physical phenomena. The structure of the *clock synchronised rules* is

- **Event**
- **Time limit**
- **Time operator**
- **Priority**
- **Premises**
- **Actions**
- **Hold**

*Clock synchronised rules* have three additional attributes They are Event, Time limit and Time operator. The Event is a trigger condition which enables the match and firing of the rule based on the value of Time operator. The Time operator can take as values either **AFTER** or **BEFORE**.

If the operator is **AFTER**, the match of the premises is initiated after an amount of time equal to Time limit has elapsed from the occurrence of the Event. The semantics of the **AFTER** operator are summarized in the Table 4.1.

<table>
<thead>
<tr>
<th>Delay over</th>
<th>Match Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Met</td>
<td>Not met</td>
</tr>
<tr>
<td>Fire rule</td>
<td>Ignore</td>
</tr>
<tr>
<td>Don't care</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Semantics of **AFTER** operator in Clock Synchronised Rules

The type of knowledge that can be represented with a clock synchronised rule using the **AFTER** operator is given below.

*Knowledge*: If the pressure valve is opened and even after 5 seconds the pressure is less than 40 Ksc then it can be concluded that the fluid level in the tank is low and pump motor should be switched on.
Chapter 4. DATA AND KNOWLEDGE REPRESENTATION SCHEME

Table 4.2: Semantics of the BEFORE operator in Clock Synchronised Rules

<table>
<thead>
<tr>
<th>Time</th>
<th>Match Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Met</td>
<td>Not met</td>
</tr>
<tr>
<td>Delay over</td>
<td>Don't care</td>
</tr>
<tr>
<td>Delay not over</td>
<td>Fire rule</td>
</tr>
<tr>
<td></td>
<td>Try again</td>
</tr>
</tbody>
</table>

Rule:

- Event: Hydraulic_system.Pressure_valve = OPEN
- Time limit: 5 seconds
- Time operator: "AFTER"
- Premise: Hydraulic_system.Pressure < 40 Ksc
- Hold: [message "fluid level low - switch pump motor"]

In physical systems, some events are expected to happen at a given time and at a given pace. For example if an auto pilot issues a yaw command, the vehicle is expected to achieve the desired turn at a desired rate and by the end of a desired period. If the yaw rate is achieved too quickly it may induce undesirable body accelerations. These situations are modelled using the BEFORE operator. The situation is depicted in the Table 4.2.

The following is the knowledge about an aerospace vehicle's responses to its auto pilot's commands. If the auto pilot has issued a Yaw command, the Yaw feedback is to become equal to the command. In a hypothetical vehicle, let us assume that the Yaw feedback has to become equal to Yaw command 2 seconds after the command is issued but before 6 seconds. This can be represented with two clock synchronised rules, one with the BEFORE operator and another with the AFTER operator. The rules are

**Cl:**

- Event: Yaw.and == +10°
- Time limit: 2 seconds
Time operator; "BEFORE"
Premise: Yaw.fb == -10°
Action: Yaw.Controlstatus = Fast

C2:
Event: Yaw.and == +10°
Time limit: 6 seconds
Time operator: "AFTER"
Premise: Yaw.fb ≠ -10°
Hold; Yav.Controlstatus = Lag

4.3.3 Spanning Rules

Early warnings based on historical trends are a necessary feature in reactive systems as preventive action can be contemplated before the alarm actually occurs. This requires the study of historical data before issuing warnings. Knowledge for this kind of reasoning is captured using Spanning rules. Spanning rules are derived from Autonomous rules and their structure is shown below.

From time
Spanning premises
Priority
Premises
Actions
Hold

The additional attributes of Spanning rules are From time and Spanning premises. Spanning premises are conditions ranging over data history to calculate trends like increase, decrease and rate of change. The From time parameter indicates the time over which parameter history is to be obtained to calculate the trends.
Spanning rules can be either triggered by events occurring in the system under investigation or periodically. Two specialisations of spanning rules are derived viz. event spanning rules and time spanning rules.

Event Spanning Rules

Event spanning rules are triggered by an event. The structure of this rule is

- From time
- Event
- Spanning premises
- Priority
- Premises
- Actions
- Hold

The knowledge to be modelled using an event spanning rule looks like: if hydraulic oil level is below 3000 litres and rate of decrease of pressure is more than 10% in the last 100 seconds then a leak in the pipe system is suspected. This is modelled as an event spanning rule as shown below.

\[ SI: \]

- Event: \texttt{Hydraulic\_system.OilLevel < 5000 l}
- From Time: \texttt{100 seconds}
- Spanning Premise: \texttt{Decrease(Hydraulic\_system.Pressure) > 10%}
- Hold: [message "leak in pipe system suspected, check up"]
Time Spanning Rules

Some critical parameters of physical systems are monitored periodically to observe trends and make conclusions about the status of the physical system. Such situations are modelled by a *Time spanning rule*. The *Time spanning rule* is derived from the *Spanning rule* class. The structure of these rules is

- **From time**
  - Cycle time

- **Spanning premises**
  - Priority
  - Premises

- **Actions**
  - Hold

The attribute Cycle time specifies the time period in which the rule should be fired once. This rule is used for periodic monitoring as in the example below.

Knowledge: *If the bearing temperature of the reaction wheels has increased by 10% in the last 5 seconds then it can be concluded that the bearing lubrication is low.*

S2:

- Cycle time: 5 seconds
- From Time: 10 seconds
- Spanning Premise: Increase(Bearing. temperature) > 10%
- Hold: [message "bearing lubrication low, check up"]

The four different rule types defined above enable modelling of knowledge and events peculiar to real time monitoring applications. Rules in the rule base are instances of one of the rule classes. This conceptual view of the rule taxonomy is carried to the storage level by storing rules in an object base similar to object store using the same access methods provided by the object manager. Rules are thus
treated as first class objects with all properties of independent objects.

4.3.4 Condition Elements

The premises, spanning premises, events and actions of rules in REX are formed by conjunction of Condition elements. The condition elements are further categorised into regular conditions and spanning conditions.

Regular Conditions

Objects of the Regular conditions class form the premises, events and actions. Regular condition class is a derived class of base class OBJECT. The structure is shown below.

```
LHS
rel-op
RHS
method name
```

The LHS and RHS are expressions on the classes and attributes in the object taxonomy of REX. The relational-operator can be ==, !=, <, >, <=, >=. The condition expressed using a regular condition is translated in a C++ method and attached to the object. The evaluation of this method returns true or false

Spanning Conditions

Objects of the class Spanning conditions form the Spanning premises in Spanning rules. Again like regular conditions the spanning condition class is derived from the base class OBJECT. The structure is shown below.
Function
Class
Attribute
rel-op
gg
value

The attribute function specifies the trend. It could be Decrease, increase or rate of change. The class name, attribute name attributes identify the parameter whose trend is to be calculated. The three functions decrease, increase and rate of change are generic functions implemented for the class Spanning conditions.

4.3.5 Hold

Hold asserts a fault state of the physical system under investigation. This is a message to the operator console. The Hold action terminates a reasoning thread. Rule with a Hold assertion will have empty Action slots.

4.3.6 Event

The Event in a Clock synchronised or a Spanning rule specifies the trigger for matching or firing the rule. Events are instances of the class Regular condition and are usually formed by a conjunction of more than one condition element. The evaluation of events is triggered by updates to the Attribute Table.

4.4 Knowledge Base Verification

Hayes-Roth points out that rule based systems lack a suitable verification methodology or a technique for testing the consistency and completeness of a rule set [77]. The verification of rule bases is classified into two categories. They are intra rule and inter rule verifications. In intra rule verification, a given rule is verified for completeness and freedom from contradictions. A rule in the knowledge base is defined to be
Chapter 4. DATA AND KNOWLEDGE REPRESENTATION SCHEME

Complete if all slots in the rule objects are filled with legal entries. An autonomous rule should possess legal non empty premise slot and either a non empty action or hold slots. Autonomous rule which possesses non empty action and hold slots is declared invalid. A spanning rule should have non empty spanning premise and from time slots. The premises and actions of any rule should be free from contradictions. The contradictions fall into two categories: value contradicting and operator contradicting. Value contradicting premises have identical attributes and relation operators and dissimilar values. Operator contradicting premises possess identical attribute value pairs but have complementary operators. The complementary operator table is given below

<table>
<thead>
<tr>
<th>=</th>
<th>!=</th>
<th>_</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>&lt;=</td>
<td></td>
</tr>
</tbody>
</table>

Two spanning premises are function contradicting if they have the same object attribute value pairs but possess contradicting functions. The decrease function contradicts the increase function and vice versa. Two actions are said to be contradicting if they assign different values to the same attribute. All the above verifications can be incorporated in the syntax analysis of the rule base.

Inter rule verification pertains to the complete rule base. Rules identified to be of the following categories are invalid rules [25, 45, 73, 85] - contradicting, dead end rules, and nonstarter rules.

Two autonomous rules are defined to be contradictory if they either

- possess identical premises, but contradictory action/hold slots or
- possess operator contradicting premises, but identical action/hold slots
- or possess identical premises, actions/hold slots, but different priorities.

An example of contradicting Autonomous rules is given below

Rule A1:
Premise:

**Pressure** < 100 Ksc and **Yaw** > 15°

**Hold:**

[message **System.controllability** is negative.]

Rule A2:

Premise:

**Pressure** > 100 Ksc and **Yaw** > 15°

**Hold:**

[message **System.controllability** is negative.]

The above two rules possess *operator contradicting premises* and hence are invalid.

Two *Clock synchronised rules* are defined to be contradicting if they

- satisfy the conditions imposed for contradicting autonomous rules

- or possess identical premises, actions, but have operator contradicting events.

Two *Time spanning rules* are defined to be contradicting if they

- satisfy the conditions imposed for contradicting autonomous rules

- or they have function contradicting spanning premises but identical actions/hold slots.

Two *Event spanning rules* are defined to be contradicting if they

- satisfy the conditions imposed for contradicting autonomous rules

- or possess identical premises, actions, but operator contradicting events

A rule is a *dead end rule* if the actions of the rule do not participate in the premises of at least one other rule. This is because, in REX, it has been assumed
that a reasoning chain terminates in an Hold. If the attributes updated by a rule's action do not participate in the premises of another rule, it implies that a reasoning chain has terminated in an Action. So, REX cannot have dead end rules though it is a forward chaining system. A rule is a non starter rule if the premises of the rule can never be satisfied. This happens if data that matches the premises is not generated either by the external world or by the actions of another rule.

4.5 Rule Base Verification Using Extended Petri Nets

The criteria for verification of rule bases can be implemented in the Extended Petri Net model in a simple way. The algorithms to carry out inter rule verification using Extended Petri Net representation are given below. [71].

Let us define four functions

- `get_next_rule()`, which gets the next rule (transition), by suitable table look-up procedure.
- `get_next_transition()` which gets the next transition
- `get_prev_transition()` which gets the previous transition and
- `Val(r_{t_i})` gives the place, value pairs $\forall p_i \in O(r_{t_i})$

The procedure for detecting contradictions between two rules $R_i$ and $R_j$ (the corresponding rule transitions $r_{t_i}$ and $r_{t_j}$) is $\text{contradict}(r_{t_i}, r_{t_j})$.

```c
contradict(r_{t_i}, r_{t_j}){
contradict = false;
if (l(r_{t_i}) = l(r_{t_j})) {
    if( Val(r_{t_i}) \neq Val(r_{t_j}) ) {
        contradict = true;
        return(contradict);
    }
}
```
else if(I(r_i) ≠ I(r_j) and Val(r_i) = Val(r_j)) {
    p_{t_i} = get_prev_transition(r_i)
    p_{t_j} = get_prev_transition(r_j)
    p_{t_i} in contradictlist(p_{t_j})
    contradict = true;
    return(contradict);
}
else if(I(r_i) = I(r_j) and Val(r_i) = Val(r_j)) {
    if( priority(r_i) ≠ priority(r_j))
        contradict = true;
        return(contradict); }
else if(V( \mu_k \in I(r_i)) if( ∼p. \notin I(r_j))
        contradict = true;
        return(contradict);
}

The following procedure given \( R_i \) detects if \( R_i \) is a non starter rule.

\texttt{nonstarter}(r_t) { 
  if \( \forall(r_t_j \in r_t \land r_t \neq r_t_j) \)
  if (I(r_i) \notin O(r_t_j) \lor I(r_t_j) \neq \text{external input})
    return(true);
    return(false); }

The following procedure given \( R_i \) detects if \( R_i \) is a deadend rule.

\texttt{deadend}(r_t) { 
  if (O(r_t).type ≠ HP and \n      \forall(r_t_j \in r_t \land r_t_i \neq r_t_j) \)
    if O(r_t) \notin I(r_t_j)
      return(true);
      return(false); }

4.6 Summary

In this chapter, a unified object oriented data and knowledge representation is proposed. The scheme is designed taking into consideration the temporal nature of the real time domains. The data representation supports the concept of data validity through the timeout facet of attributes. The knowledge representation scheme is an object oriented rule base. Rules are treated as objects in an uniform taxonomy with data objects. Various rule types are defined in the taxonomy to represent different situations. Clock synchronised rules represent event based knowledge. Spanning rules represent knowledge about trends in the system. Treating rules as first class objects is a special feature of this representation. This allows uniform storage and retrieval of both data and rules. Further, the extensibility provided by the object oriented representation, helps in extending the rule model to meet emerging needs of the domain. Other shells using an object oriented representation of data, treat rules as separate entities operating on the data objects [9]. Another shell treats rules as methods of an object [43], thus restricting the scope of the rules. Finally the verification criteria for the knowledge base are presented. In the next chapter, the match algorithm used in the REX Reasoning Subsystem is presented.