

# **CA Aion<sup>®</sup> Business Rules Expert**

## **Rules Guide**

**r11**



This documentation and any related computer software help programs (hereinafter referred to as the "Documentation") are for your informational purposes only and are subject to change or withdrawal by CA at any time.

This Documentation may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA. This Documentation is confidential and proprietary information of CA and may not be used or disclosed by you except as may be permitted in a separate confidentiality agreement between you and CA.

Notwithstanding the foregoing, if you are a licensed user of the software product(s) addressed in the Documentation, you may print a reasonable number of copies of the Documentation for internal use by you and your employees in connection with that software, provided that all CA copyright notices and legends are affixed to each reproduced copy.

The right to print copies of the Documentation is limited to the period during which the applicable license for such software remains in full force and effect. Should the license terminate for any reason, it is your responsibility to certify in writing to CA that all copies and partial copies of the Documentation have been returned to CA or destroyed.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO THE END USER OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The use of any software product referenced in the Documentation is governed by the applicable license agreement and is not modified in any way by the terms of this notice.

The manufacturer of this Documentation is CA.

Provided with "Restricted Rights." Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in FAR Sections 12.212, 52.227-14, and 52.227-19(c)(1) - (2) and DFARS Section 252.227-7014(b)(3), as applicable, or their successors.

Copyright © 2009 CA. All rights reserved. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

## CA Product References

This document references the following CA products:

- CA Aion® Business Rules Expert (CA Aion BRE)

## Contact CA

### Contact Technical Support

For your convenience, CA provides one site where you can access the information you need for your Home Office, Small Business, and Enterprise CA products. At <http://ca.com/support>, you can access the following:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

### Provide Feedback

If you have comments or questions about CA product documentation, you can send a message to [techpubs@ca.com](mailto:techpubs@ca.com).

If you would like to provide feedback about CA product documentation, complete our short [customer survey](#), which is also available on the CA Support website, found at <http://ca.com/docs>.



# Contents

---

<b>Chapter 1: Introducing Rules</b>	<b>11</b>
Audience .....	11
Rule Basics .....	12
Rules .....	12
Rules Can Represent Knowledge .....	13
Knowledge Base .....	14
Inference Engine .....	14
Procedural Code or Rules? .....	14
Anatomy of a Knowledge-Based Application .....	15
Rules and Chaining .....	16
Location of Rule Methods .....	16
Structure of a Simple Rule .....	17
Rule Editor .....	17
Rule Analyzer .....	18
 <b>Chapter 2: Chaining</b>	 <b>19</b>
Forward Chaining .....	19
Data-Driven Inferencing .....	19
Reasoning with Forward Chaining .....	21
New Data Supply for Forward Chaining .....	21
Rule Firing .....	22
Forward Chaining Input and Output .....	22
Why Use Forward Chaining Rules? .....	23
Typical Forward Chaining Applications .....	24
Backward Chaining .....	24
Goal-Directed Inferencing .....	24
Actions of Backward Chaining .....	25
Reasoning with Backward Chaining .....	25
Backward Chaining Input and Output .....	27
Why Use Backward Chaining? .....	28
Typical Backward Chaining Applications .....	28
Conversational Systems Built with Backward Chaining .....	29
Generate the Interface .....	29
How to Link the Interface into Client/Server Environments .....	32

---

<b>Chapter 3: The Inference Block</b>	<b>37</b>
Anatomy of an Inference Block .....	37
Sample Inference Block .....	38
InferBegin and InferEnd .....	39
Posting and Scope .....	39
Direct and Indirect Posting .....	40
A Revised Example .....	40
A More Complex Example .....	41
Posting Order versus Priority Order .....	41
Object-Oriented Knowledge .....	43
 <b>Chapter 4: Runtime Issues</b>	 <b>45</b>
Posting Time or Chaining Time? .....	45
Local Attributes .....	45
Local Variables .....	45
Other Language Limitations in Rules .....	46
Dynamic Posting .....	46
Chaining Considerations .....	47
Inference Blocks and Scope .....	47
End of the Chaining Process .....	48
Multiple Execution .....	51
UNKNOWN Attributes .....	52
How to Call Methods from a Rule or Inference Block .....	55
Limitations on Chaining over Complex Data Types .....	55
Demons .....	56
Firing of Demons .....	56
Scope of Demons .....	56
Dynamic Rule Runtime Considerations .....	57
How to Load and Post Dynamic Rules .....	57
Instance Binding .....	62
Error Checking .....	66
 <b>Chapter 5: Pattern Matching</b>	 <b>69</b>
Pattern Matching Rules .....	69
Pattern Matching over One Class .....	70
Pattern Matching over Two Classes .....	70
Pattern Matching over a Class with Two Binding Variables .....	71
Pattern Matching over Interfaces .....	72
Advanced Pattern Matching .....	73
Flights.app Sample .....	73
Bindings .....	75

---

Orderby Clause .....	78
Where Clause .....	79
SingleFire Rules .....	81
Inference Engine and Multiple ifmatch Rules .....	82

## **Chapter 6: Decision Tables** **83**

Benefits of Using Decision Tables .....	84
How to Create and Open Decision Tables .....	84
Create Decision Table .....	84
Open an Existing Decision Table .....	85
How to View and Modify Decision Table Properties .....	85
Add and Modify Conditions .....	86
Add and Modify Actions .....	89
Delete a Rule .....	92
Ordering Conditions and Actions .....	93
Cutting and Pasting Conditions and Actions .....	94
Displaying a Decision Table .....	94
Customizing the Decision Table Editor .....	95
Compressing a Decision Table .....	96
Manually Collapsing Subtables .....	98
Achieving Optimal Condition Order for Compressibility .....	99
Runtime Execution .....	100
Rule Posting .....	100
Condition Evaluation .....	101
Action Execution .....	101
Chaining .....	101
Rephrasing IFRules as Decision Tables .....	102
One-to-One Mappings .....	102
ANDed Premise Expressions .....	102
Consolidating IFRules into a Single Decision Table .....	103
Dynamic Decision Tables .....	103
Dynamic Decision Table Runtime Considerations .....	104

## **Chapter 7: Truth Maintenance** **107**

Truth Maintenance Operations and Terminology .....	107
Operational Context .....	108
Tickling Demon and Pattern-Matching Rules .....	109
Available Runtime Information .....	109
TM-Assignment .....	109
TM-Retracton .....	111
TM-Confirmation .....	113

---

## **Chapter 8: Maintaining the Dynamic Rulebase** **115**

Dynamic Rulebase Administrator .....	116
The Default and Empty Rulebases .....	117
Importing a Domain Interface .....	119
Dynamic Rulebase Scenarios .....	124
Dynamic Rule Manager .....	126
Selecting a Rulebase and Opening a Domain .....	127
Creating and Maintaining Dynamic Rules .....	129
Dynamic Decision Table Editor .....	129
Creating a New Dynamic Decision Table .....	130
Opening an Existing Dynamic Decision Table .....	130
Viewing and Modifying Dynamic Decision Table Properties .....	130
Adding and Modifying Conditions .....	131
Adding and Modifying Actions .....	133
Decision Table Editor Functions .....	136
Displaying a Dynamic Decision Table .....	137
Customizing the Decision Table Editor .....	138
Compressing a Decision Table .....	139
Accessing the Dynamic Rule Repository .....	139

## **Chapter 9: Constructing Non-Persistent Dynamic Rules** **141**

Facilities for Constructing Non-Persistent Dynamic Rules .....	141
Non-Persistent Dynamic Decision Tables .....	142
Non-Persistent Rule Definer Class .....	144
Constructing Decision Table Conditions .....	144
Constructing Decision Table Actions .....	147
Posting a Non-Persistent Decision Table .....	151

## **Appendix A: Summary of Inferencing Constructs** **153**

Inference Block .....	153
Chaining Statements .....	154
Rule Types .....	154
Production Rule .....	155
Decision Table Rule .....	155
Pattern Matching Rule .....	155
Production Demons .....	156
Pattern Matching Demons .....	157
Knowability Expressions .....	157
Truth Maintenance Operations .....	157



---

<b>Appendix B: How the Inference Engine Works</b>	<b>159</b>
Forward Chaining .....	159
Forward Chaining Example .....	160
Backward Chaining .....	161
Backward Chaining Example .....	162
 <b>Appendix C: Rulebase Structure</b>	 <b>165</b>
Table Domain .....	166
Table DIMember .....	166
Table DecTable .....	168
Table Condition .....	168
Table Action .....	169
Table RPValue .....	169
Table SubAction .....	170
Table HighID .....	170
Table Users .....	171
Table CheckOut .....	171
 <b>Index</b>	 <b>173</b>



# Chapter 1: Introducing Rules

---

In CA Aion Business Rules Expert (BRE), rules represent knowledge. Rules are a natural means for capturing and automating business policies and procedures because they use simple if...then logic in a declarative form. Rules encourage business user communication and maintenance. In some cases, rules allow complex systems—systems that would be difficult to build any other way—to be modeled.

Rules test and make assignments to attributes of the business objects that are the focus of your application. Rules allow non-programmer “experts” in the business area to see and maintain application rules to validate business rule logic and quickly implement changes in rules to react to changes in the business.

This CA Aion BRE Rules Guide describes rules and how to write them.

**Note:** Unless otherwise indicated, the term Windows refers to any Microsoft Windows operating system supported by CA Aion BRE, including Windows XP, Windows 2003, Windows 2008, and Windows Vista. See the product Readme file for details about operating system support.

This guide assumes that the appropriate components of CA Aion BRE have been installed at your site. Instructions for installing the product can be found in the CA Aion Getting Started.

This section contains the following topics:

[Audience](#) (see page 11)

[Rule Basics](#) (see page 12)

[Rules](#) (see page 12)

[Rules and Chaining](#) (see page 16)

[Location of Rule Methods](#) (see page 16)

[Structure of a Simple Rule](#) (see page 17)

[Rule Editor](#) (see page 17)

[Rule Analyzer](#) (see page 18)

## Audience

This guide is intended for developers at all levels and for business rules experts.

## Rule Basics

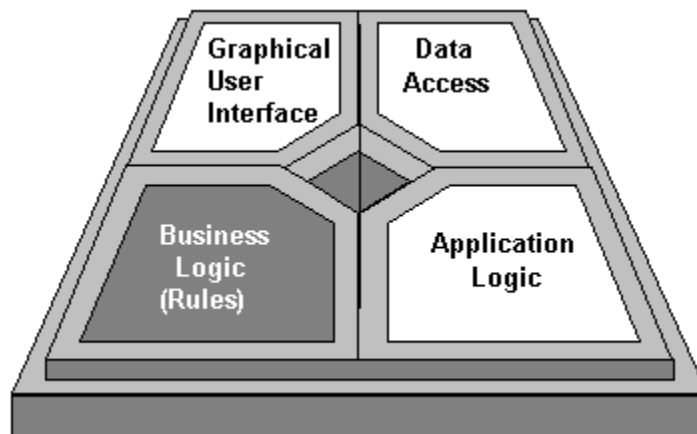
Rules are the business core of your application. They express the business logic of an application in a manner that a non-programmer can readily understand. With rules, business logic can be created and maintained by a domain specialist or a programmer; either person can write rules for your application.

## Rules

An application performs several essential tasks that are auxiliary to the analysis of business data. Written procedurally, these may include:

- A Graphical User Interface (GUI) that creates windows and dialogs for requesting input from users and for displaying output
- Database access code for retrieving raw data from a SQL or desktop database
- These are the actual values the application analyzes to reach a business decision. For example, a loan-processing application needs personal and financial information about a loan applicant. This information may come from a database of applicants.
- Application logic that determines how an application starts and ends, calls the main window, and indicates overall flow of control within the program
- Business logic where raw data is analyzed and logical conclusions reached

The user of an application can consult the business expertise modeled in rules. For instance, a loan officer could run an application that indicates whether a loan request should be approved, and for what amount.



## Rules Can Represent Knowledge

To represent knowledge in a business application, rules make use of business classes that have names such as Invoice, Customer, Salesperson, and Product. These business classes are usually more complex and business-oriented than the raw data (for example, records retrieved directly from a database) and are generally created by a programmer for use by the person who writes rules.

Attributes of business classes represent business facts. Relationships between these facts take the form of if-then statements, or rules. In this way, rules model domain-specific knowledge about business practices, policies, and regulations.

For example, in a loan-processing application, the salary of an applicant is a fact that you can represent as:

```
pApplicant.Salary=45000
```

The individual applying for the loan is represented by pApplicant and the salary is \$45,000. In programming terms, pApplicant is a pointer to an instance of the Applicant class, and Salary is an attribute of the instance. The assignment sets the value of the attribute to \$45,000. You can use a TRUE/FALSE (Boolean) attribute called Approve to represent approval of a loan request as follows:

```
pApplicant.Approve=TRUE
```

Current bank policy may be to accept only those applicants earning over \$30,000. You could represent this knowledge in the following way:

```
if pApplicant.Salary < 30000  
then pApplicant.Approve=FALSE
```

To cover all applicants, you could write rules for each possible combination of facts (salary, credit history, requested amount, and so on). Generally speaking, you do not need to worry about the order in which you write rules.

## Knowledge Base

The collection of rules in an application constitutes a knowledge base. The programs you create with rules are often referred to as knowledge-based applications. Because knowledge bases use rules inferencing, they can also be called rule bases.

The knowledge base portion of an application is so crucial that a developer may choose to isolate rules in their own application component and then make it available to other applications, using such interfaces as COM, or when using C++. With this type of isolation of rules, a domain-specialist could design a knowledge base component that programmers could integrate with different GUIs and database access components as needed. What makes this division of labor possible is the clear distinction that CA Aion BRE draws between knowledge representation (contained in rules) and inference logic (performed by the inference engine)

## Inference Engine

The inference engine is a set of algorithms that process the rules you have written. When you invoke the inference engine, it processes your rules efficiently, drawing inferences from known facts and controlling the order in which rules come into play.

The inference engine liberates software development resources to concentrate on domain-specific issues rather than the intricacies of computer programming. Yet the engine is flexible enough to adapt to various rules-inferencing situations. Special keywords in the language allow you to specify how the engine should process your rules.

Experts in the field of knowledge systems sometimes speak of “running a knowledge base through an inference engine.” In concrete terms, this means that you invoke the inference engine to process a set of rules.

## Procedural Code or Rules?

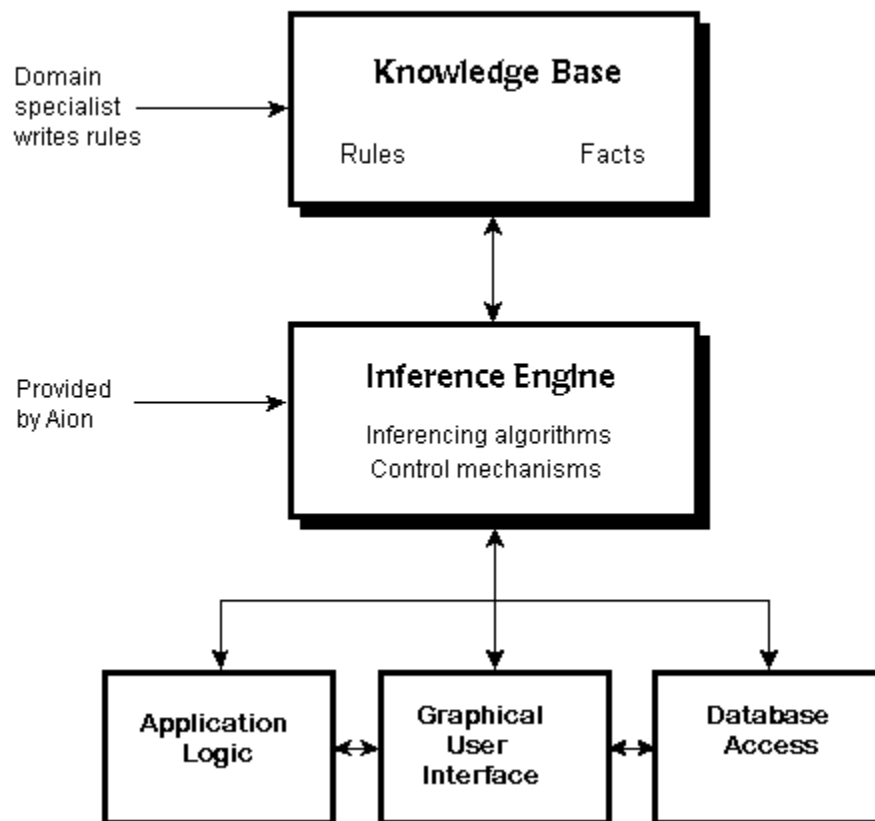
Most software applications in use today are written with procedural code, which specifies exactly how control should flow from statement to statement. The programmer must use exhaustive step-by-step instructions—complex looping constructs, conditional statements, and algorithms—for arriving at results. A very flexible tool, procedural code is the province of the software programmer.

Aion BRE rules encode business logic using if-then rules that a non-programmer can quickly master. The writer of rules does not worry about algorithms, control flow, or even rule order, but concentrates instead on translating domain-specific knowledge into a set of rules. When the application is executed, the inference engine processes the rules, drawing logical conclusions. Since rules do not specify a set of procedures but simply state business facts and relationships, they are referred to as declarative code.

## Anatomy of a Knowledge-Based Application

A CA Aion BRE knowledge-based application uses rule-based knowledge-domain-specific data interpreted by the algorithms of an inference engine-to solve business problems. A knowledge base is one part of an application that may also contain a GUI and a database access component.

### A Knowledge-Based Application



## Rules and Chaining

The inference engine examines rules one at a time. When the rule premise is true, the engine executes-or fires-the rule. When the rule premise is false, the rule is discarded for that invocation of the engine, and is referred to as having failed. When a rule has incomplete information in its premise, the engine pends the rule-putting it aside until the missing values can be supplied by querying a database, asking the user, calculating from known values, or firing other rules.

The inference engine processes rules by inferring new facts from facts that are already known. These newly established facts may enable the firing of rules that could not previously be determined. Once fired, these rules may in turn establish new facts that permit other rules to be fired. Together, they form a chain from the first rule to the last. A knowledge base consists of several such chains of rules.

As rules are fired, the inference engine invokes the actions of those rules. The action of the final rule in the chain usually determines an attribute value or executes a method of particular interest to your business process.

Because of the linked nature of knowledge base rules, rule processing is referred to as chaining. When you want the inference engine to process rules, you use one of the two chaining keywords in the language: `forwardchain` or `backwardchain`.

## Location of Rule Methods

You can locate rule methods in almost any class, although typically you place them in data classes or in business classes.

If rules access attributes in one class only, you generally place the rules in that class. In such cases, it is convenient to have the rules in the same class as the raw data they operate on. Validation of the data in query classes is often done this way.

When knowledge applies to the core purpose of an application-for example, processing a loan request-rules often reside in their own business classes. These classes often correspond to tasks in the problem model and represent higher-level business-solving processes. The rules usually access data from more than one class.



## Structure of a Simple Rule

A simple rule has the following structure:

```
rule RuleName
  ifrule premise
  then action
end
```

- The keyword **rule** begins a rule and must be followed by the rule name, which is a standard identifier or a quoted string.

**Note:** A standard identifier must start with a letter (A-Z). Remaining characters can be a letter, number, or the underscore. The identifier cannot be a reserved word. Identifiers are not case-sensitive.

- The ifrule clause, or premise, is used to examine attributes of a class or instance and specify one or more conditions that they should meet.
- The then clause, or action, invokes methods or otherwise causes data in the instance to change (for example, by assigning values to attributes).

An action is executed only when the premise conditions are satisfied.

- The keyword end marks the end of a rule.

Here is the loan processing example, using the proper syntax:

```
rule RejectUnder30K
  ifrule pApplicant.salary<30000
  then pApplicant.approve=FALSE
end
```

You can use quotation marks to include spaces in a rule name:

```
rule "Reject if salary > $30,000"
  ifrule pApplicant.salary<30000
  then pApplicant.approve=FALSE
end
```

## Rule Editor

The Rule Editor provides a structured environment for writing and editing rules. Information about the Rule Editor is available from its context-sensitive help (F1) and in the CA Aion BRE Product Guide.

## Rule Analyzer

The Rule Analyzer displays the rules that can change the value of a given attribute. From the Rule Analyzer, you can view the organization and relationship between rules and the attributes they use. You can also view these relationships by toggling the Forward Mode option in the right-click menu. When the Forward Mode option is not checked, the Analyzer is in Backward Mode.

- Use Forward Mode when you want to know which rules fire when you assign a value to an attribute, and which attributes are assigned values as a result.
- Use Backward Mode when you want to see which rules can assign values to an attribute, and to determine which attributes are needed to get a rule to fire.

Information about the Rule Analyzer is available from its context-sensitive help (F1) and in the *Product Guide*.

# Chapter 2: Chaining

---

Chaining is the process by which rules are evaluated. There are two types of chaining, forward chaining and backward chaining. Each of these types of chaining is best suited to particular kinds of tasks. This chapter explains what chains are and how they work in CA Aion BRE.

This section contains the following topics:

[Forward Chaining](#) (see page 19)

[Backward Chaining](#) (see page 24)

[Conversational Systems Built with Backward Chaining](#) (see page 29)

## Forward Chaining

Sometimes you want your application to accept all input data and then start firing applicable chains of rules. The process can continue until all the consequences of the input data have rippled through the knowledge base.

The result of executing your application is a set of attribute values on which business decisions can be based. The actions of some rules may also execute methods to perform such business actions as modifying a database, generating notification letters, or contacting a credit bureau. This type of rule processing is called forward chaining and has the following characteristics:

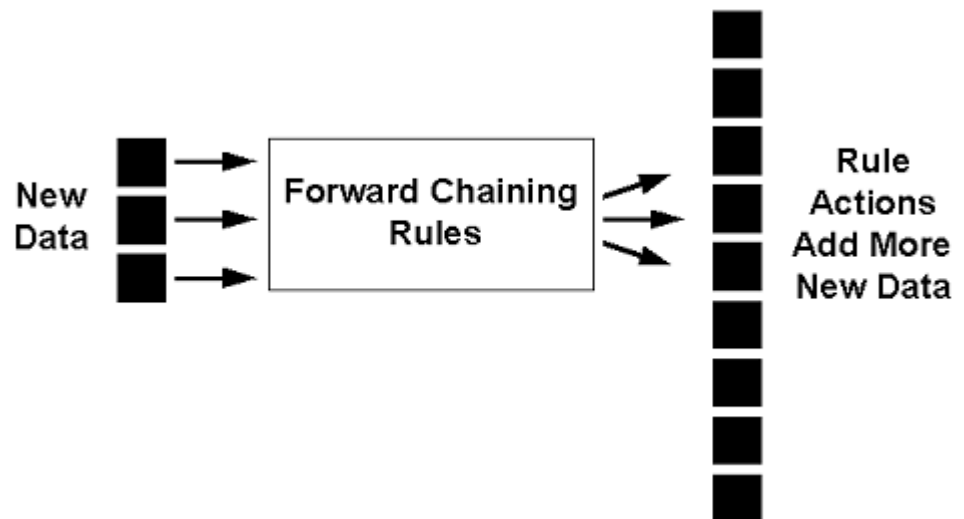
- It is data-driven reasoning
- The inference engine generally considers rules in the order posted
- The inference engine can determine the full ripple effect of input data  
(You can have the inference engine exhaust all rules before it terminates.)

## Data-Driven Inferencing

The forward chaining process starts with the attribute values provided by the user or retrieved from a database. Forward chaining continues when new data become known (that is, as new attribute values are assigned by firing rules). Forward chaining is, thus, data-driven.

The inference engine fires rules as the data in their premise becomes true. Rule firing produces new data, which in turn may cause other rules to be fired. The forward chaining process usually continues until all the knowledge has been exhausted and all possible conclusions have been reached. If you prefer, you can specify a goal attribute. If forward chaining determines a value for the attribute, then the inference engine stops.

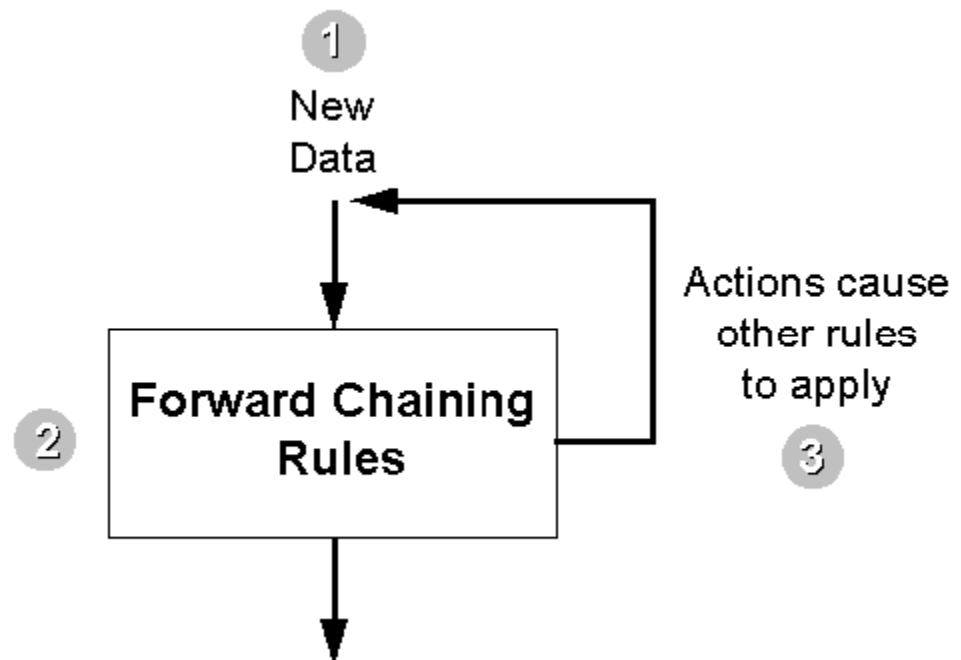
With data-driven inferencing, a small amount of input data can result in a large amount of inferred information.



## Reasoning with Forward Chaining

Forward chaining execution progresses as follows:

1. Enter new data.
2. Forward chaining rules fire when their premises are TRUE.
3. Rule actions set new data values.
4. Inference engine repeats Steps 2 and 3 until it reaches a termination condition or exhausts all rules.



## New Data Supply for Forward Chaining

New data for forward chaining may come from interactions with the application's end-user, or from other sources, such as an external database. With forward chaining it is common to ask for all needed data before starting the inference engine with the `forwardchain` keyword. You have the flexibility, however, to invoke the inference engine several times, contacting the user or a database to supply attribute values in between invocations.

## Rule Firing

A rule fires only when all the facts referred to in its premise are available to the inference engine. The inference engine does not automatically try to supply unknown values in a rule's premise. Instead, it pends such rules and revisits them when the missing values become known. For example, assume that a rule has the following premise:

```
rule "Good rating: approve any loan under 10K"  
ifrule pApplicant.CreditRating = "GOOD" and Loan<10000  
then  
    pApplicant.Approve =TRUE  
end
```

Consider what happens when the second part of the premise is true (Loan is "9000"), but the first part is not known (CreditRating is unknown). In that case, the inference engine pends the rule and waits until the value is supplied by the firing of other rules.

If you want the end-user (for example, the loan officer) to provide values during execution of the application, you can write procedural code that requests such data, perhaps using a dialog. If you want to create a question-and-answer style session, however, backward chaining may be the more efficient choice.

## Forward Chaining Input and Output

Data-driven inferencing has a characteristic pattern of soliciting data from and presenting results to the application user.

For simplicity's sake, we said that a forward chaining application generally obtains new data up front. In practice, it is often more efficient to process information in several chunks. In an application with user interaction, information is requested form by form (or dialog by dialog in an application with a graphical user interface). The first form's worth of data is processed by a forwardchain command. As a result, new attribute values may be assigned. Then your application displays a second form to solicit new data, and executes a second forwardchain command. This time, the inference engine processes rules that have been enabled by the newly solicited and inferred data. Contrast this with a question-by-question application, in which the inference engine is invoked after each new piece of data is obtained.

## Display of Forward Chaining Results

The results of forward chaining are often presented immediately to the user as an on-screen display or printed report; they may also be stored outside the application in a database. Of course, it is possible the results will serve as input for another invocation of the inference engine.

Typical forward chaining output formats include:

- Displaying a message informing the user of the rules' results as they fire
- Creating output to be written to an external database

**Note:** When the current session of the knowledge-based application is over, all data entered by the user or inferred by the inference engine is cleared. Therefore, any information that you want to store permanently should be written to a database or other file before the end of the session.

## Why Use Forward Chaining Rules?

Forward chaining answers the question: "Given some facts, what are the consequences of those facts?"

There are a number of situations that adapt themselves well to forward chaining rules, as shown in the following table.

Situation	Examples
You want to know everything that can possibly be concluded about a set of data.	Monitoring for mechanical problems on a production line Scanning a new loan application for problem areas
Many conclusions are possible from a single data item.	Filtering sensor data
You have several elements that can be configured in multiple ways. There is no one right answer; you want to know the various possibilities.	Configuring computer systems from hardware components Scheduling employees and equipment for production purposes

## Typical Forward Chaining Applications

Forward chaining is especially well suited to applications that track incoming information and need to respond to variable amounts of new data. It is excellent for synthesis-especially situations such as scheduling and configuration, which can yield many outcomes from a few inputs. Typically, forward chaining applications either collect input data in forms or infer new data as the rules are processed.

## Backward Chaining

Sometimes you are not interested in the complete ripple effect of input data. You may want to focus on a given outcome and be satisfied as soon as a particular attribute is assigned a value. In that case, you should use the `backwardchain` command to invoke the inference engine.

For example, our loan processing application can be written using either forward chaining or backward chaining. If you want to find all the “weak spots” in the loan requester's profile, you should use forward chaining. A mortgage broker would be interested in this information, so he/she could detail to his/her client all the factors that led to a loan rejection. The client could then determine whether certain factors (such as other outstanding debt) could be modified.

On the other hand, the bank's loan officer might simply want to know whether the current loan profile qualifies or not. A single disqualifying factor would be enough to cause the loan request to be rejected. Thus, a forward chaining command that exhausted all the rules would be inefficient.

The application should use the language keyword `backwardchain`, with a goal attribute of `Approve`. The inference engine examines rules to find one that assigns a value to `Approve`. If the rule cannot be fired because of an incomplete premise, the inference engine tries to form a chain of rules that leads to this rule. As soon as a value for `Approve` is determined, the inference engine stops.

Inferencing that employs backward chaining has the following characteristics:

- Reasoning is goal-directed.
- The inference engine considers rules according to their actions.
- Subgoals (explained below) are automatically set and, if possible, resolved.

## Goal-Directed Inferencing

Backward chaining is goal-directed-it uses rules to infer knowledge to accomplish goals. You can picture a chain of rules stretching backwards from a goal to its subgoals to the data that can resolve the goal.



## Actions of Backward Chaining

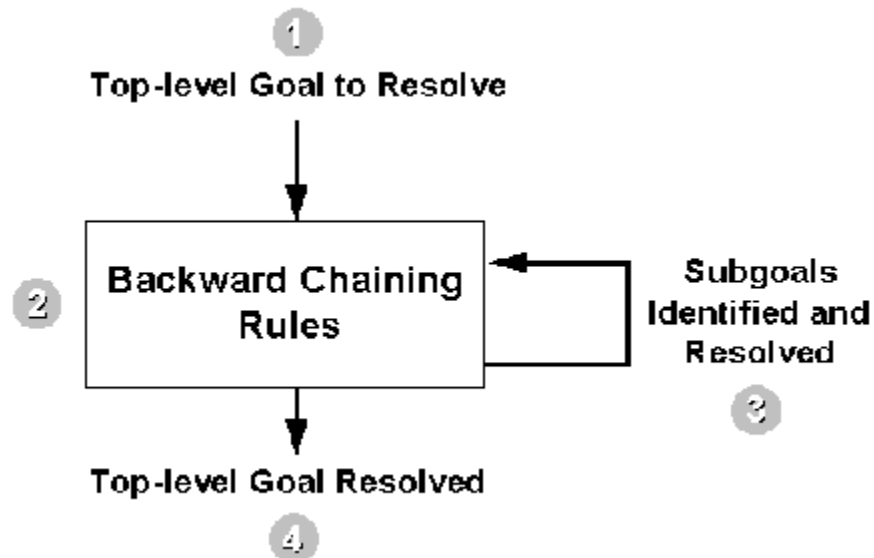
The following actions give backward chaining its name:

- The inference engine starts at the desired goal and works backwards to resolve that goal.
- The inference engine “backs into” rules by looking at the then clause first.

## Reasoning with Backward Chaining

In backward chaining, the inference engine's execution cycle is:

1. Identify the top-level goal from the backwardchain keyword.
2. Look for a rule that resolves the goal in its then clauses.
3. If the rule is not dependent on undefined premise attributes, fire the rule. Otherwise, pend the rule and add its undefined premise attributes as subgoals for backward chaining.
4. Repeat Steps 2 and 3, but augment the search to include subgoals. Of course, you may want to write procedural code that supplies a value for a subgoal.
5. Continue chaining backward until either the top-level goal is reached, the engine exhausts all rules, or the engine detects a STOPCHAIN statement.



## Initiation of Backward Chaining

Backward chaining is initiated to satisfy a goal. This goal is to find a value for an attribute that has no value, that is, it is unknown. The inference engine treats this as the initiating or top-level goal for backward chaining.

### Indetification of Pertinent Rules

The inference engine looks for rules that have an action (then clause) addressing the unresolved top-level goal. For instance, in our example, pertinent rules are those whose action assigns a value to Approve.

### Indetification of Subgoals

As the inference engine tries to apply rules to resolve the top-level goal, it develops a list of subgoals-that is, other values that must be found before it can reach the final resolution. These subgoals derive from the premises (ifrule clauses) of rules whose actions match the top-level goal. Each unknown portion of the premise becomes a subgoal for the inference engine to resolve. A subgoal may have subgoals of its own, which may in turn have subgoals, stretching back to some known piece of data.

### Resolving Goals

Each goal that the inference engine has to resolve, whether the top-level goal or a subgoal identified during backward chaining, can be resolved in any of the following ways:

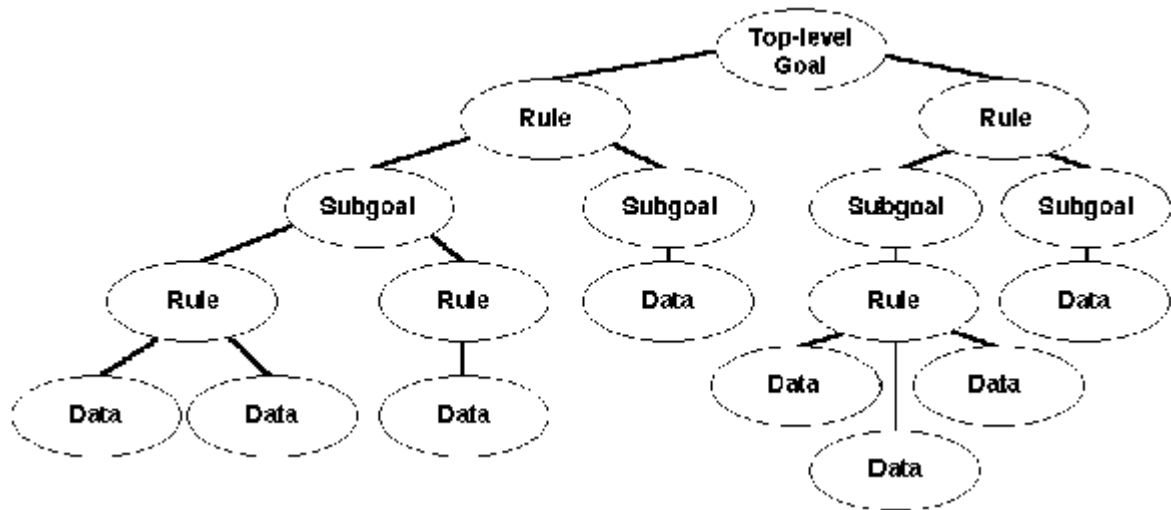
- Backward chaining with additional rules
- Asking the user for information
- Accessing values in an external database
- Executing external calculations and other external programs

The inference engine automatically searches additional rules to resolve goals and subgoals. If the other options are desired, the programmer writes procedural code to implement them.

Since backward chaining rules search for supporting data on an as-needed basis (you do not need all the data up front), the inference engine only considers those rules that lead to resolving a particular goal. Backward chaining ends when the top-level goal is successfully resolved or when all means for resolving it have been tried and have failed. You can also use a STOPCHAIN statement.

## Simple Example

The following illustration shows the fundamentals of backward chaining. Many variations are possible. For example, you can prioritize rules so that the inference engine processes them in a certain order.



An experienced programmer can assign priorities to expedite processing and micro-manage use of computer resources. A beginner, however, can rely on the inference engine to determine the ordering of rule processing.

## Backward Chaining Input and Output

Goal-directed reasoning also has a characteristic pattern of soliciting data from and presenting results to the application user. Backward chaining works best when there are relatively few possible outputs (for example, advice messages) but many possible inputs (for example, pieces of information about a problem situation), most of which are not relevant to every situation.

### Input to Rules

The main categories of input to backward chaining include:

- Pre-existing values in the knowledge base
- Questions asked of the user
- Values accessed in an external database
- Execution of internal or external calculations

**Note:** For information about constructing backward chaining systems in which the inference engine asks the user for information, see [Conversational Systems Using Backward Chaining](#) (see page 29).

### Output from Rules

The output from backward chaining usually answers the questions: "What is the value?" or "From these facts, can I conclude a particular result?" Often, a simple message dialog box will suffice. In other cases, backward chaining rules resolve multiple goals during a single execution of the inference engine. As with forward chaining, you can display results in an on-screen or printed report and save results to an external database.

### Why Use Backward Chaining?

There are a number of situations in which you should consider using backward chaining, as shown in the following table.

Situation	Examples
There is a clear set of statements-hopefully small-that needs to be confirmed or denied.	Which machine is causing this quality control problem?
A large number of questions could be asked of the user, but typically only a few of them must be asked to resolve a particular situation.	When processing an auto insurance claim for vandalism damage, it is not necessary to ask about personal injuries.
Data collection is costly or tedious. It is advantageous to collect only information that is actually needed as it is needed.	Real-time observations by the user. Results from computation or CPU-intensive models (perhaps external to the knowledge base).

### Typical Backward Chaining Applications

Backward chaining is especially well suited to diagnostic applications where the inference engine recommends a course of action based on data input. When designing a backward chaining application, you should be able to enumerate all possible conclusions that the inference engine could reach. In backward chaining, inputs tend to be open-ended and numerous, and outcomes limited to a fixed set. In general, any analysis task, such as diagnosis and classification, is susceptible to solution through backward chaining.

## Conversational Systems Built with Backward Chaining

The classic ideal of an expert system is to be capable of carrying on an intelligent conversation with the user of the system, asking only those questions that are relevant to the particular context of solving a present problem. For example, if the subject of a health diagnosis is male, an expert diagnostic system should not ask questions relevant to a woman's health.

Backward chaining is the generally accepted inferencing strategy used to construct such conversational systems. The reason for this is that backward chaining pursues subgoals only within the specific context of the main goal that the inferencing engine is pursuing. When a subgoal cannot be satisfied through firing other rules, it is typical for the inference engine to raise a question so that the system user can provide the value of the subgoal.

CA Aion BRE supports constructing conversational systems using backward chaining; however, the inference engine does not automatically generate questions to the user. The programmer must write code that tells the interfacing layer how to raise the questions. Generating the Interface describes various strategies and architectures for generating conversational systems with CA Aion BRE in different environments.

### Generate the Interface

CA Aion BRE provides the following ways to generate a dialog with the user:

- IFRULEs
- WhenSourced( ) event method

You can use the CA Aion BRE native WinLib facilities to generate a GUI. The simplest means of presenting a question to the user is to use WinLib to generate a dialog box containing the question. To do this, you must include WinLib in the application, and write the appropriate code for initiating and processing the dialog box (retrieving the answer).

**Note:** GUIV7Lib provides the ask( ) method which makes generating simple dialog boxes easier. For information about non-native ways to initiate a dialog, see [How to Link the Interface into Client/Server Environments](#) (see page 32).

## Use Rules to Initiate the Dialog

You can write rules that fire to raise questions. These rules have the following format:

```
IFRULE IsUnknown(->Attrib)
THEN
// Code that raises a question about Attrib and
// sets the value of Attrib based upon the response.
END
```

Each attribute for which a question may need to be raised should have a corresponding rule of this form. The priority of this rule should be set below that of the other rules for the attribute because you probably want this rule to be fired only after the inference engine has exhausted all other inferencing possibilities.

**Note:** It is desirable to confine these rules to a separate rule method that is always posted with the ruleset, or rulesets, containing those attributes that these rules ask about. These rules should be excluded from any verification process using VALENS.

## Use the WhenSourced Event Method to Initiate the Dialog

Another approach to raise questions uses the WhenSourced( ) event method. The WhenSourced( ) method is provided by the \_Object class of SysLib. CA Aion BRE automatically invokes the WhenSourced( ) method whenever the inference engine seeks a value for an engine-linked attribute during chaining. It is helpful in supporting backward chaining systems.

There are several steps required to have CA Aion BRE invoke the WhenSourced( ) method:

1. In the inference block, link attributes for which you want to generate questions to the current inference engine:

```
var eng is integer
eng = EngineGetCurrent( )
EngineLinkGoal(eng, ->Attrib)
```

- If your class has many attributes that participate in the inferencing process, you probably do not want to list each attribute.
- Through the CA Aion BRE metaprogramming facilities, it is possible to loop through your class to get a pointer to each attribute and link each to the current inference engine.
- When looping through a class to get a pointer to each attribute, use the GetAttribute( ) method of AttributePointer.

2. Write the code for the WhenSourced( ) method.

The WhenSourced() method is defined in \_Object. You should specialize WhenSourced() in the class where the attribute(s) to be linked are defined (or in a parent class). The signature of the WhenSourced( ) method is:

WhenSourced(pat is attributePointer, lastchance is Boolean)

where:

**pat**

Provides a pointer to the attribute that is being sourced.

**lastchance**

Is TRUE when the engine cannot resolve the attribute and chaining is about to terminate.

The simplest way is to specialize the WhenSourced( ) method to test the attributePointer parameter and execute the appropriate code for raising the question to the user.

For example:

```
If pat = ->Attrib then
    // code to raise the question
else
    // other attribute tests
end
```

A more sophisticated approach is to code a series of additional methods for each question to be generated. Following the CA Aion BRE style of naming such methods, these names would have the form:

WhenattribNameSourced

The WhenSourced( ) method would then contain just general code for invoking the appropriate method:

```
var methodname is String
methodname = "When" & GetAttributeName(pat) & "Sourced"
invokeMethod(current, methodname, 0, 0, 0)
```

If you wish to generate the question only after the inference engine has exhausted all rules for the primary goal, then you should execute the preceding code under the condition that lastchance = TRUE.

**Note:** Last chance sourcing may not exactly mimic the order of questions as using IFRULEs that fire strictly under the regime of backward chaining. For more information about the WhenSourced( ) method, see the online help.

## How to Link the Interface into Client/Server Environments

When CA Aion BRE is used in a client/server environment, constructing a conversational interface typically involves building an Aion component as a server that performs inferencing, with a client interface built in Visual Basic, C/C++, Java, or C#. For example, the Aion server may be a server running in the .NET environment (under the Common Language Runtime, or CLR) that communicates with a client component built using C# or other .NET compatible language. It is also possible to use commercially available interface building tools, for example, a user interface built with PowerBuilder may use the CA Aion BRE C interface layer.

Similar to supporting a native Aion interface, the Aion server will most likely use IFRULEs and WhenSourced( ) event methods to support client interface mechanisms within a client/server environment. However, because the Aion server does not have direct access to the client's interface mechanisms, programming to support these interface mechanisms differs from supporting a native Aion interface.

**Note:** For more information about using IFRULEs or WhenSourced ( ) to maintain a conversation interface, see [Use Rules to Initiate the Dialog](#) (see page 30) and [Use the WhenSourced Event Method to Initiate the Dialog](#) (see page 30).

To construct a conversational interface within a client/server environment, it is necessary to establish the appropriate interface links between the Aion server and the client interface. That is, the Aion server must export methods that are capable of supporting the client in initiating inferencing, obtaining the text for the question to be asked, and returning the response to Aion. Depending upon whether your client/server environment is capable of supporting stateful conversations, these methods must be programmed differently.

Client/server environments that offer only stateless servers, such as COM, require callbacks to achieve a conversational interface. To see an example of using a callback between a COM client and a COM server, see the CA Aion BRE example in \COM\animals.



Stateful conversations, in which the Aion server maintains the internal state of the inference engine over different transactions, require additional programming considerations. When returning the response to a question that the server has requested the client to ask its user, the client must be assured of getting back to the same server instance that it had originally invoked to initiate inferencing. Doing this requires thread management. To achieve the best thread management in a multi-user environment, it is recommended to use CA Aion BRE servers with a J2EE environment (with the Java interface layer) or the .NET environment (with managed C++ interface layer).

**Note:** CA Aion BRE provides examples of supporting a conversational interface. The examples are DynaInfer and the BackwardChain in the /Java example set. First, study the DynaInfer example. This example is primarily meant to illustrate the InferBegin( ) and InferEnd( ) methods (explained below). The approach explained in this chapter follows more closely the BackwardChain example. For programming considerations related to maintaining state throughout a conversational session, see [Access a Stateful Server from the Client](#) (see page 33) and [Program the Server Side](#) (see page 34) in this chapter.

### Access a Stateful Server from the Client

After the client has called a method on the Aion server to invoke inferencing, the server must return the result of the inferencing, not information about how to ask a question. To address the problem of providing information about the question to ask, the Aion server must pass back to the client an indicator that inferencing is incomplete and that a question needs to be asked. The client must then retrieve information about how to ask the question from the Aion server.

Client pseudo code for carrying on a conversational interface may look like the following:

```
AionServer.initiateInferencingBlock( )
Loop until result is found
    statusResult = AionServer.invokeInferencing()
    // This statement reinitiates inferencing on
    // subsequent iterations of the loop.
    if statusResult = needToAskQuestion then
        questionInfo = AionServer.obtainQuestionInfo()
        questionResult = askUserQuestion(questionInfo)
        AionServer.setUserAnswer(questionResult)
    elseif statusResult = inferencingComplete then
        result = AionServer.obtainInferencingResult()
    else // Server is unable to reach a result
        result = NULL
        exit Loop
    end
End
```

Other client code is possible. For instance, the CA Aion BRE BackwardChain example (in the \AionBRE\examples\Java folder) uses exception throwing in place of inferencing the statusResult code. The following section will discuss how the server side is programmed to support this client interaction.

## Program the Server Side

The client code in [Access a Stateful Server from the Client](#) (see page 33), in this chapter, demonstrates the mechanisms the Aion server must provide to the client. The Aion server must provide:

- An interface method for the client to initiate inferencing; that is, to post rules.
- A means to designate incomplete inferencing, that is, to designate the need to raise a question.

To support the preceding client code, the Aion server will need to introduce its own inferencing status code for this purpose and return it through the inference engine. This may be accomplished by invoking the stoprun( ) method using the special status code whenever a question needs to be raised on the client side. The CA Aion BRE BackwardChain example (in the \AionBRE\examples\Java folder) employs exception throwing to alert the client when the principal goal remains unknown and a question needs to be raised for a subgoal.

**Note:** Exception throwing may not be available in all environments.

- An interface method for the client to obtain information about what question to ask.
- An interface method for the client to return the response. This method will set the attribute (subgoal) requiring the response.
- A means to invoke and reinitialize inferencing.

The Aion server can use other approaches to support a conversational interface that directly use the engine history. For an example of other approaches, see the CA Aion BRE example in \AionBRE\examples\DynaInfer.

**Note:** In this section, it is assumed that either IFRULEs or WhenSourced ( ) event methods are used to set up the information in the Aion server that is required by the client to ask the intended question.

Instead of executing code to raise the question directly to the user, the IFRULEs and WhenSourced( ) event-methods must set up sufficient information in the Aion server for the client to ask the appropriate question. That is, the Aion server must save information about the attribute that the inference engine is currently pursuing. To preserve this information, the Aion server usually saves the attributePointer to attribute.

The Aion server must also make this information about the attribute available to the client in a usable form, for example, as a string representing the name of the attribute. This information is made available to the client through an appropriate interface method.

**Note:** CA Aion BRE constraints provide a convenient mechanism to specify valid responses to questions that the user is asked during backward chaining. Constraints can be made available to the client through the `getAttributeConstraint( )` method of `attributePointer`. For more information about constraints, see Constraints in the chapter “Aion Objects Overview” in the *User Guide*.

The `attributePointer` is reused to set the value of the attribute when the client returns the response.

**Note:** For more information, see the `SetAttributeValue( )` method in the `AttributePointer` class of `SysLib` in the online Reference. For information about invoking and reinitializing inferencing, see [Invoke and Reinitialize Inferencing within a Stateful Conversation](#) (see page 35).

### Invoke and Reinitialize Inferencing Within a Stateful Conversation

Because the CA Aion BRE application must stop inferencing to return the proper inferencing status to the client, the principal issue in linking a conversational interface in a client/server environment is how the inference engine will maintain state during the conversation and resume inferencing in an efficient way. If the standard inference block (Infer/End) is used in this context, then the state of the inference engine is lost when CA Aion BRE exits the inference block. Upon the return to inferencing, the CA Aion BRE application would have to rerun the complete chain with the new information until it reaches another unknown attribute, at which point the process, including executing the complete chain again, is recycled.

Fortunately, CA Aion BRE provides an alternative way of creating an inference block, called dynamically bound inference blocks, using the `InferBegin( )` and `InferEnd( )` methods. These methods allow the inference engine to maintain state across different invocations of inferencing from the client.

**Note:** For more information about dynamically bound inference blocks, see [InferBegin and InferEnd](#) (see page 39). For an example, see the `DynaInfer` example.

When using `InferBegin( )` and `InferEnd( )`, it is necessary to return to the same inference engine when using dynamically bound inference blocks. To accomplish this:

- The CA Aion BRE server must provide separate interface methods for initiating an inference block (which should also post rules) and for invoking inferencing.

For example, the `InferBegin( )` invocation cannot be in the same method that invokes backward chaining; otherwise, a new inference engine will be created with each iteration.

- The initiating interface method is called only once per consultation session by the client, whereas the inference invoking method is always called in a loop, as shown in [Access a Stateful Server from the Client](#) (see page 33) in this chapter.

**Note:** For more information about developing a conversational interface using Java, see Supporting Backward Chaining in the chapter “Generating and Using the Java Interface Layer” in the *User Guide*. The information contained in that section is also relevant for developing a conversational interface using a C# client.

# Chapter 3: The Inference Block

---

The previous chapters discuss simple rules and the operation of the `backwardchain` and `forwardchain` keywords. To write and process rules, you need one more construct—the inference block.

An inference block is written in a method and contains all code dealing with inferencing. It begins with the keyword `INFER` and finishes with the keyword `END`. Two things always occur in an inference block:

- Rules are posted
- A chaining command is issued (`backwardchain` or `forwardchain`)
- An inference block may contain more than one chaining command.

An inference block may contain other language statements, including method calls and regular if-then statements (not to be confused with the `ifrule` keyword used in rules).

A class may contain several inference blocks, which may be dynamically nested.

This section contains the following topics:

[Anatomy of an Inference Block](#) (see page 37)

[Posting and Scope](#) (see page 39)

[Object-Oriented Knowledge](#) (see page 43)

## Anatomy of an Inference Block

In an inference block, two main events happen. First, the rules are posted, then the inferencing engine processes the rules.

## Sample Inference Block

As an example, consider the following, in which ->Approve is a pointer to the attribute Approve. Indentation, line spacing, and comments are for convenience only:

```
INFER
// Loan Rule 1
rule "Good Rating: approve any loan under 10K"
ifrule CreditRating = "GOOD" AND Loan<10000
then
    Approve=TRUE
End
// Loan Rule 2
rule "High Income: approve any loan under 10K"
ifrule Income > 50000 AND Loan<10000
then
    Approve=TRUE
End
// Loan Rule 3
rule "High Income/Good Rating: approve any amount"
ifrule CreditRating = "GOOD" AND Income > 50000
then
    Approve=TRUE
End
// Loan Rule 4
rule "If no reason to approve, then reject"
ifrule ISUNKNOWN(->Approve)
then
    Approve=FALSE
End
backwardchain(->Approve)
TellUser( )
end
```

This example demonstrates that a domain expert, such as a loan officer, could write rules without knowing anything about if-then-else or procedural constructs. When the knowledge base includes 400 rules, instead of just four, this simplicity translates into real savings in development time.

The use of backwardchain means that as soon as the inference engine arrives at a value for Approve, it stops. Our loan officer does not care about all the ways a loan could qualify; he just wants a Yes or a No.

The example could be modified to meet the needs of a mortgage broker if we use forwardchain instead. In the case of a rejection, she could discover all the rules that assigned Approve a value of FALSE.

The `TellUser( )` statement demonstrates that you can call methods from within an inference block. In this case, it might display a dialog box that states whether the loan request was approved or rejected. If we used `forwardchain` instead, `TellUser( )` might display fired rules that assigned a value to `Approve`.

## InferBegin and InferEnd

Inference blocks can be made even more flexible by using the `InferBegin` and `InferEnd` methods. Unlike inference blocks that are statically bounded via `INFER/END` statements, `InferBegin` and `InferEnd` can create dynamically bounded inference blocks that can span multiple methods. In other words, starting a block, posting its rules, chaining the rules, and terminating the block may all be performed by different methods. For example:

```
VAR hInfer IS INTEGER
// Begins inference block
hInfer = InferBegin("ResolveGoal1", INFER_HISTORY)
VAR bSuccess IS BOOLEAN
// Ends inference block
bSuccess = InferEnd(hInfer)
```

This increased flexibility lends itself to scenarios whereby a client drives a server and the server's inferencing block needs to remain open across client-server interaction. The client and server may reside in the same or different components.

`InferBegin` and `InferEnd` are semantically equivalent to `INFER` and `END` respectively, and can be substituted for them if desired. However, they cannot be mixed in the same inference block. Attempting to terminate an `InferBegin( )` block with an `END` statement results in a syntax error (because of an extraneous `END` statement). Attempting to terminate an `INFER` block with `InferEnd( )` results in a syntax error (because of a missing `END` statement).

## Posting and Scope

When control enters the inference block, all rules are posted according to their order in the block. Posting a rule means it is available to the inference engine to be fired, failed, or pended during forward or backward chaining. Although all posted rules are available, the inference engine may reach a goal before all posted rules have been examined.

Rules are available only to the inference block in which they are posted; therefore, each inference block creates its own scope for the processing of rules. The same rules can be processed by different invocations of the inference engine; however, they must be posted separately in each inference block.

## Direct and Indirect Posting

It is not always practical to post rules directly, that is, by writing them explicitly in the inference block as shown in the example. You may want to reuse rules in several inference blocks. You may want to streamline the inference block, which can become too full as the number of rules grows.

There is a convenient and modular alternative-indirect posting of rules. You write the rules in their own method, and then call the method from the inference block. Because they contain nothing but rules, these methods are known as rule methods, or sometimes, rule packets. With large knowledge bases, the indirect style of posting rules is a necessity. We can rewrite our example using rule methods.

## A Revised Example

In this example, revised from Sample Inference Block, we create a rule method called `LoanRules( )` to contain the first three rules-the ones that test income level and credit status. We chose to leave the last rule out of the `LoanRules( )` method. For an explanation, see the next section.

```
INFER
LoanRules( )    // Post loan rules 1, 2, & 3
rule "If no reason to approve, then reject"
ifrule ISUNKOWN (-> Approve)
then
  Approve = FALSE
end
backwardchain (->Approve)
TellUser( )
end
```



## A More Complex Example

With your rules tucked away in the `LoanRules( )` method, it becomes easy to modify the inference block to carry out more sophisticated tasks:

```
INFER
LoanRules( )      /*Post loan rules*/
AskUserRules( )   /*Post rules to ask user for values*/
CreditCheckRules( ) /*Post rules to check credit rating*/
rule "If no reason to approve, then reject"
ifrule ISUNKNOWN(->Approve)
then
    Approve=FALSE
end
backwardchain (->Approve)
TellUser( )
end
```

We created a method called `AskUserRules( )` that has rules for soliciting such information as income and requested loan amount. One of the rules calls a dialog box to ask the user for this information, if it is missing.

We also added a method called `CreditCheckRules( )` that has a rule to check if the `CreditRating` data exists. If not, it automatically connects to the Credit Bureau and runs a credit check.

The `LoanRules( )` method does not include the rule called "If no reason to approve, then reject." This last-resort rule should be fired only when the engine has pursued dependencies (that is, unresolved attributes) in any of the other three rules. If all four rules were placed in the `LoanRules( )` method, you might encounter the following problem: The first three rules might pend due to incomplete information, and then the last-resort rule would fire before the user had been asked or the Credit Bureau contacted.

## Posting Order versus Priority Order

The preceding example demonstrates how you can improve your application by arranging the inference order in which rules are posted. The credit-checking rules (for `CreditRating`) are posted after the ask-user rules (for `Income` and `Loan`) because connecting to the Credit Bureau costs time and resources. As such, we want to avoid firing the credit-checking rules unless we absolutely need to do so. And, if we can approve the loan based on `Income` and `Loan` without examining `CreditRating`, we should do so. By positioning the credit-checking rules after the ask-user rules, we accomplish this efficiently.

In many situations, you may want more control over the rule order. You may find it inconvenient to move large numbers of rules around after they are written, or you may wish to post a last-resort rule ahead of other rules.

The posting order is the order in which the inference engine encounters rule definitions during execution of the inference block. Aion posts rules consecutively from the top of the block. Rules may reside directly in the inference block or in rule methods that you call from it. The posting order for a block includes both kinds of rules.

Priority order governs the order in which the inference engine examines rules during forward or backward chaining. Priority indicates which is the next available rule. By default, the posting order and the priority order are the same. You can specify a different priority order by assigning priority numbers to rules.

Priority numbers work in the following way:

- The greater the value, the higher the priority.
- Priority numbers must be integers and may be positive, negative, or zero.
- If several rules have the same priority number, the inference engine decides order among them by using the posting order.

**Note:** If you do not specify a priority, the priority number for a rule is zero by default.

The following table of seven hypothetical rules illustrates these points. Assume that all rules are in the same inference block:

Order by Posting		Order by Priority	
Rule #	Priority #	Rule #	Priority #
1	29	1	29
2	-15	5	4
3	0	4	3
4	3	6	3
5	4	3	0
6	3	7	0 (default)
7	none	2	-15

You use the keyword `priority` to assign priority numbers. Consider the example in the previous section. From a packaging standpoint, the last resort rule could be grouped with the `LoanRules( )` method to improve the overall organization of the code.

We avoided putting it there in the first place, you may remember, because we do not want the last-resort rule to fire if other sources of information—Credit Bureau or loan officer—could alter the outcome. Priority numbers can solve this problem, letting us place all four rules together in the `LoanRules( )` method.

For instance, you might assign a negative integer to the last-resort rule. You could also use priority numbers, if you did not want to depend on posting order. In that case, the other three rules in `LoanRules( )` might have a priority of 5, and the rules in `CreditCheckrules( )` and `AskUserRules( )` a priority of 3.

The `LoanRules( )` method would look like this:

```
rule "Good Rating: approve any loan under 10K"
priority 5
ifrule CreditRating = "GOOD" AND Loan<10000
then
    Approve=TRUE
end
rule "High Income/Bad Rating: approve any loan under 10K"
priority 5
ifrule Income > 50000 AND Loan<10000
then
    Approve=TRUE
end
rule "High Income/Good Rating: approve any amount"
priority 5
ifrule CreditRating = "GOOD" AND Income > 50000
then
    Approve=TRUE
end
rule "If no reason to approve, then reject"
priority -2
if rule ISUNKNOWN(->Approve)
then
    Approve=FALSE
end
```

The inference block would be streamlined, easy to understand and maintain:

```
INFER
    LoanRules( )
    AskUserRules( )
    CreditCheckRules( )
    backwardchain (->Approve)
    TellUser( )
end
```

## Object-Oriented Knowledge

An Aion application is completely object-oriented. The class hierarchy encompasses all functionality-including rules and inferencing.

Rules and inference blocks participate in object orientation because they are an integral part of the method in which you write them. The characteristics of a method affect the rules and inference blocks in it. For instance:

- If a method has access rights to particular attributes and methods, so do its rules and inference blocks.
- If a method has a private or protected access type, then access to its rules and inference blocks is limited, too.
- If you want rules and inference blocks to be globally accessible in your application, give the method an access type of public.
- If the attributes and methods belong to a different class, you specify class or instance name when accessing them in a rule or inference block.
- If rules and inference blocks reside in an instance method, they use the same instance with which you called the method.
- Unqualified names in rules refer to attributes and methods of the current instance and class. (Unqualified names are those that do not specify a class or instance name.)
- Rules and inference blocks participate in inheritance.
- The current instance may belong to the class in which the rule is defined or to a descendant class. If you make an unqualified method call in a rule, you do not know ahead of time to which class the instance belongs. Hence, polymorphism applies to rules and inference blocks, too.
- Rules and inference blocks can be specialized by specializing the method in which they reside.

# Chapter 4: Runtime Issues

---

This chapter discusses posting, chaining, demons, and runtime considerations for dynamic rules. Use the information in this chapter to assist in designing and coding your Aion applications.

This section contains the following topics:

[Posting Time or Chaining Time?](#) (see page 45)

[Chaining Considerations](#) (see page 47)

[Demons](#) (see page 56)

[Dynamic Rule Runtime Considerations](#) (see page 57)

## Posting Time or Chaining Time?

Consider what happens to a rule method at runtime. As a method is executed, rules are posted in preparation for processing by the inference engine. The posted rules are not executed until a `backwardchain` or `forwardchain` keyword is reached. This temporal separation in the execution of rule methods and their rules has several important implications for programming in Aion.

Keep in mind these two distinct “times” to understand how rules behave. Posting time is when the method is executed and the rules posted. Chaining time is when the inference engine executes the rules.

## Local Attributes

Local attributes are attributes of the class to which a rule belongs. Local attributes accessed in a rule refer to the same instance and class as the rule's method. Aion establishes the current instance or current class for a rule at posting time. Attributes are correctly resolved when rule premises are evaluated or actions executed. Within an inference block, different rules may be associated with different current instances and classes.

## Local Variables

Local variables, which you declare in a method, behave differently from local attributes, which belong to the class. To understand local variables, you should know that each rule premise or action is treated as if it were an individual function. By the time a rule is executed, its method has already passed out of scope, along with the method's local variables.

Local variables are re-initialized every time a rule action is executed. Values do not persist between examinations or executions of a rule.

If you want to use a method's local variable in a rule action, declare the variable at the beginning of the action. You should do this in the interest of comprehensible code. If you do not, however, Aion declares it implicitly for you. You can use the local variable within the action only.

## Other Language Limitations in Rules

When you use the Aion language in a rule premise or action, these restrictions apply:

- Return-statements cannot be used in a rule.

**Note:** The method you would intend to make return would already have passed out of scope.

- Arguments that you pass to the rule's method cannot be referenced in a rule.

**Note:** The arguments have the same lifetime as the method.

## Dynamic Posting

A complex application can contain hundreds, even thousands, of rules. Some uses of your application may require only a subset of these rules to arrive at a solution. To enhance performance of the inference engine, limit the number of rules posted to only those necessary for each case or problem.

You can dynamically control the posting of rules. When you test one or more conditions before posting a group of rules, we speak of dynamic posting-which is accomplished by rules or by procedural logic.

## Meta-Rules

Some rules exist to post other rules. When a rule is fired, its action may post one or more other rules. Rules that augment knowledge during the inference process are referred as meta-rules. For example, the following rule adds the knowledge required to diagnose skin disease only after a generic set of symptoms has been identified:

```
RULE "Skin-Disease"  
IFRULE SkinDiseaseSymptoms  
THEN  
    SkinDiseaseKnowledge.diagnose      // post rules  
END
```

## Conditional Posting

You can also employ regular if-then statements to control which rule methods are called, as in the following:

```
INFER
  /* Conditional Knowledge Posting */
  BusinessObject.MyRules1           // post common rules
  IF Individual.NotPolicyHolder
    THEN BusinessObject.MyRules2    // post specific rules
  END
  /* Inference Logic */
  FORWARDCHAIN
END
```

## Comparison of Meta-Rules and Conditional Posting

In general, you use conditional posting when posting due to non-inferencing criteria. You post using meta-rules when posting due to an inferencing result.

# Chaining Considerations

When you are coding the section of the inference block that contains the chaining command, it may help you to know a little more about the following:

- Inference block scoping
- Stopping the chaining process
- Multiple execution of premises, actions, and the chaining command
- Unknown attributes
- Calling methods from rules
- Limitations regarding chaining over complex data types

## Inference Blocks and Scope

An inference block provides the scope for the inferencing operations. All rules posted in the same inference block are in the same inference scope. When pursuing a goal or exhaustively firing rules, the inference engine examines all rules currently in scope.

## Nested Inference Blocks

There may be times when you want to interrupt the current scope, create a limited scope for resolving a sub-problem, and then resume the interrupted scope. Aion accommodates this need with nested inference blocks. A nested inference block processes locally posted rules only; it does not examine or fire rules from inference blocks of higher or lower scope.

When combined with conditional statements, nested inference blocks provide for truly dynamic inferencing, in which a line of reasoning is pursued only under selected circumstances.

**Note:** Nested inference blocks are allowed, but not nested chaining. You cannot issue one chaining command when another is still executing. Do not invoke chaining from within a rule action.

## Rule Readying

During inferencing in a nested inference block, readying of rules in higher scopes does occur. The engine notes actions (such as assignments to attributes or creation of an instance) that might affect a rule in a higher scope. The engine then changes the rule state from Pended to Ready. Later, when the higher scope is reentered, the newly readied rules will be considered by the engine.

## End of the Chaining Process

Once you have started the forward or backward chaining process, three conditions can bring it to a halt:

- The goal is achieved (value determined for the goal attribute).
- Rules are exhausted.
- A STOPCHAIN is executed.

## Normal Return Codes

The chaining command returns an integer value that tells you what made the inferencing process terminate. Return values are defined as constant class attributes in the `_ENGINE` class. You can access these return codes in your application logic. Under certain circumstances, you may want to determine new input and execute the chaining command again.

There are four normal return codes for `forwardchain` and `backwardchain`:

Return code	Rule processing terminated because...
CHAIN_COMPLETED	The goal attribute was assigned a value.



Return code	Rule processing terminated because...
CHAIN_FAILED	A chaining error was detected. For example, nested chaining was attempted.
CHAIN_OUTOFRULES	The rules were exhausted and the goal attribute was not assigned a value.
CHAIN_STOPNV	A stopchain statement did not define any other return code.

## Backward Chaining

The backward chaining syntax is as follows:

```
<return code> = backwardchain (<GoalPointer>)  
backwardchain (GoalPointer)
```

where <GoalPointer> defines an attributepointer to the goal.

You must specify a goal attribute for backward chaining. Depending on how you designed your rules, the CHAIN\_OUTOFRULES return code may signal that inferencing did not reach a definite conclusion.

## Forward Chaining

Forward chaining often continues until all the consequences of input data have rippled through the rules. Usually, it is not pursuing a goal attribute. The CHAIN\_OUTOFRULES return code would signal that your inferencing has been a success.

You can also set a goal attribute for forward chaining. Unlike backward chaining, the goal attribute is optional. In this situation, a CHAIN\_OUTOFRULES return code might be viewed as a failure, and CHAIN\_COMPLETED as success.

The syntax can conform to any one of the following lines:

```
<return code> = forwardchain ( <GoalPointer> )  
forwardchain ( )  
forwardchain
```

where <GoalPointer> defines an attributepointer to an optional goal.

## Stopchain Statement

You can stop chaining under other circumstances than the defaults. Issue the stopchain statement anywhere in a rule's action to terminate chaining. In general, you test for some condition before using stopchain, as in the following example:

```
RULE "No charge if senior citizen"
IFRULE SeniorCitizen
THEN
    charge = 0
    STOPCHAIN
END
```

## Return Codes

The stopchain statement supports an optional return code, which returns to the chaining command. The stopchain return code becomes the return code for the whole forwardchain or backwardchain statement.

```
stopchain // no return-code
stopchain <RetCode> // with return-code
```

The integer expression <RetCode> defines the command return code.

For your convenience, the following standard return codes have been defined as constant class attributes in the class `_Engine`. A standard meaning is noted for each, but they can mean whatever you decide.

Return code	Handy when...
STOP_DONE	...chaining completes normally.
STOP_ERROR	...you use stopchain because an error has been detected.
STOP_EXIT	...you use stopchain to force the termination of chaining under special circumstances.

However, <RetCode> can evaluate to any integer value.

You can employ the preceding standard return codes, or you can create your own return codes. Restrict these return codes to non-negative values, because the inference engine employs negative values for the standard return codes.

Examples of usage:

```
stopchain
  stopchain STOP_DONE
  stopchain(123)
  stopchain 3 * Method1("Duck")
```

**Note:** Aion does not check non-negative, developer defined return codes nor does it range check those codes. The restriction is simply a convention for avoiding conflict with current and future Engine-defined return codes.

### StopChain Does Not Terminate Rule Action

The stopchain statement terminates chaining only-it does not terminate the current rule's action. When stopchain is encountered, the engine records the stopchain return code, if any, and then execution of the rule's action continues. The engine terminates chaining at completion of the rule's action.

### Stopchain Priority

If, during processing of a given rule, multiple stopchain statements are executed, the return code is determined by the last-executed stopchain statement.

Sometimes during processing of a rule, a stopchain statement is executed and the goal attribute is resolved. The return code reflects the stopchain statement-not the goal completion.

### Multiple Execution

Parts of rules and inference blocks can be executed several times during the processing of rules. For the sake of performance, you may wish to consider the following:

## Multiple Execution of Chaining

Sometimes it is necessary to execute the chaining statement multiple times within the inference block. A classic example is the case of a backward chaining engine that may not resolve its main goal the first time.

The code can be programmed to detect such a condition and resolve some of the remaining sub-goals by either querying the user or applying default values. You then reexecute the inference command, and the inference engine uses the new values to process the rules. Consider the following example:

```
INFER
  BusinessObject.MyRules1          /* Rule Posting */
  LOOP
    BREAKIF BACKWARDCHAIN(Goal) = CHAIN_COMPLETED
    hnd = _Engine.GetCurrentEngine( ) /* Retrieve engine's */
    goals = _Engine.EngineGetGoals(hnd) /* sub-goal list */
    AskUser(goals) /* have user resolve one/more sub-goals */
  END
END
```

## UNKNOWN Attributes

The discovery and resolution of UNKNOWN attributes is one of the fundamental tasks of chaining. Rule attributes that have not been assigned a value are said to be UNKNOWN. Once an UNKNOWN attribute is assigned a value (even NULL), it becomes KNOWN. Attributes that contain an initial value are KNOWN attributes.

**Note:** NULL is a value. An attribute with a value of NULL is not UNKNOWN.

## Test for UNKNOWN

Before starting or restarting an inferencing process, you may need to know if a goal attribute is UNKNOWN. If you find it is KNOWN, you can call the engine method `_Engine.GoalMakeUnknown( )` to clear the value before using the chaining command again. For example `_Engine.GoalMakeUnknown(->x)`.

Two operators can be used to test the whether a variable or attribute is KNOWN or UNKNOWN. The operators are `isknown` and `isunknown`. For example:

```
RULE DefaultX
IFRULE ISUNKNOWN(->x)
  x = DEFAULT
END
RULE StopOnX
IFRULE ISKNOWN(->x)
STOPCHAIN
END
```

## UNKNOWN in Called Method Causes Rule to Pend

For example, consider what happens when you call a method from within the premise of a rule. If the engine encounters an UNKNOWN attribute, the rule is pended. During backward chaining, the attribute is added to the list of subgoals.

**Note:** This is not true for methods not written in the Aion language.

## When UNKNOWN Does Not Cause Rule to Pend

In general, when the engine executes a rule, an UNKNOWN attribute causes the rule to pend. There are three exceptions:

Exception	Reason
the isunknown operator	You can use it without causing the rule to pend because isunknown returns a Boolean value. However, the isknown operator can cause a rule to pend.
ifmatch rules when demons whenmatch demons	These rules and demons never pend.
AND/OR Node	These nodes are short-circuited in a special way.

## AND/OR Nodes and UNKNOWN

The AND and the OR operators in rules behave in a way that adapts the standard C-language short-circuiting to the world of rules inferencing. Let's consider the AND operator first:

A AND B

When the Aion inference engine encounters that construct in a rule, it first tests whether A is UNKNOWN. If it is, the engine does not pend the rule immediately. Instead, it goes ahead and tests B. If B is false, then there is no point in pursuing A at this time. Hence, the rule is not pended; it is failed. Consider the OR operator next:

A OR B

What happens if A is UNKNOWN? In this case, too, the engine does not pend the rule at once. It checks B next. If B is TRUE, then it fires the rule without going through the trouble of pending it first and resolving A.

## Partial Execution of Premise and Action

During the inferencing process, a rule premise can be partially executed several times until it evaluates to either TRUE or FALSE. A rule action may also be partially executed many times until all attributes referenced in the action have been assigned values.

**Note:** To improve inferencing performance, avoid time-consuming method calls and any side-effects during the execution of a premise or action.

The following examples illustrate how to remove time-consuming conditions from a premise:

Rules	Examples
Original rule	<pre>RULE InefficientRule IFRULE ExpensiveCondition( ) AND CheapCondition( ) THEN Attribute1 = 123 END</pre>
More efficient rules	<pre>RULE MoreEfficientRule IFRULE CheapCondition( ) AND ExpensiveCondition( ) THEN Attribute1 = 123 END</pre>

The original rule is inefficient because it always evaluates `ExpensiveCondition`, even when `CheapCondition` is FALSE. The second rule is more efficient than the original rule because it avoids evaluation of `ExpensiveCondition` if `CheapCondition` is FALSE. However, it will still evaluate `ExpensiveCondition` when `CheapCondition` is unknown. An even more efficient rule is:

```
RULE MostEfficientRule
IFRULE CheapCondition( )
THEN
IF ExpensiveCondition( ) THEN
Attribute1 = 123
END
END
```

This form of the rule is the most efficient one because it evaluates `ExpensiveCondition` only if `CheapCondition` is known to be TRUE.

## How to Call Methods from a Rule or Inference Block

When calling methods from a rule or inference block, two things you should take into consideration are:

- Non-inferencing statements
- Automatic in-lining of methods

### Non-Inferencing Statements

All language statements can be used in the inference block, not just those statements with inferencing keywords. We have seen how the regular if-then statement can be used to dynamically manage rule posting. You can also call methods as you would outside the inference block. For example, you might call a method to initialize data. Then you could issue the chaining command, followed by a method call to process the resulting data.

```
INFER
    KnowledgeObject1.MyRules1
    Individual.LoadData
FORWARDCHAIN
    Individual.ProcessData
END
```

### Automatic In-Lining of Methods

When you call a method from within a rule, the method is expanded in-line automatically. Keep this in mind if you are calling lengthy methods from within rules. You may want to assign the return value from such a method to an attribute and then access only the attribute from your rule.

Also keep in mind that rules can pend if the engine encounters an UNKNOWN attribute referenced by the in-lined method.

#### **More information:**

[UNKNOWN in Called Method Causes Rule to Pend](#) (see page 53)

## Limitations on Chaining over Complex Data Types

Beyond the simple data types such as integer, real, string, and boolean, Aion also offers complex data types such as List and Array. Chaining, however, is limited to the simple data types. That is, complex data types such as lists and arrays cannot serve as goals of chaining. Although syntactically permitted at the present time, such statements as `backwardchain(->MyList)`, where `MyList` is a list of simple data type, will not invoke any consideration of the rules. This limitation applies to both backward and forward chaining.

It should also be noted that Truth Maintenance operations do not apply to complex data types. For example, values assigned to a list during truth maintenance will not be retracted if the reasoning is retracted. For more information, see [Truth Maintenance](#) (see page 107).

Moreover, inferencing behavior becomes unpredictable when lists are used in rule premises, for example, IFRULE/WHEN AttribList includes "specificValue". The use of complex data types in rule and demon premises is discouraged.

## Demons

Demons are very special rules that monitor events and fire when the events occur. For example, a demon might monitor a gauge value. When the gauge reaches a critical level, the demon's action fires automatically.

Demons become active immediately upon posting; they do not require a chaining statement to activate them. When the demon is posted, the engine evaluates the premise and, if TRUE, executes the demon's action. Thereafter, the engine monitors the status of demon premise attributes to detect modification, and when modified, the demon is refired. Demons have two forms, depending on how many instances you are monitoring:

- **when**-intended for monitoring a single instance at a time
- **whenmatch**-used to monitor several instances via pattern matching

## Firing of Demons

Any assignment to a monitored attribute of a demon causes the premise of the demon to be re-evaluated. Note the following characteristics:

- If the premise evaluates to TRUE, the demon is fired immediately-that is, demons are fired "inline" with the actions of other rules.
- A demon may be fired multiple times, unlike an ifrule, which is discarded after it fires.

## Scope of Demons

Demons remain active until the inference block in which they were defined is exited. Normal inference block scoping does not apply to demons. Actions in a nested inference block can fire a demon posted by a higher-level inference block.

**Note:** Posting too many Demon rules can slow down the performance of your application.



## Dynamic Rule Runtime Considerations

Dynamic rules are a powerful feature of Aion for defining rules to the knowledge bases at execution time. For more information about dynamic rules, see the “System Considerations for Supporting Business Rules” chapter in the User Guide. Dynamic rules differ from static rules, which are defined at edit-time and are compiled into the knowledge base executable. Dynamic rules are typically achieved by storing rules in an external medium called a rulebase. Rulebases are most commonly relational databases, such as an MS Access, a Sybase SQL Server or an Oracle database. Dynamic rules that are stored in an external medium are called persistent dynamic rules. Aion also provides support for dynamic rules that are created by logic in the knowledge base itself at execution time, see [Constructing Non-Persistent Dynamic Rules](#) (see page 141).

**Note:** Aion provides a default rulebase in MS Access format, Rulebase.mdb, that contains no rules. This rulebase can be found in the \Rulebase subdirectory of the Aion BRE install directory. For more information about this default rulebase, see [The Default and Empty Rulebases](#) (see page 117).

This section covers the runtime considerations pertaining to persistent dynamic rules, which is the most common form of dynamic rules. The principal runtime concerns regarding persistent dynamic rules center on how to load them from an external medium and post them to the inference engine. Aion also requires special consideration be given to instance binding, which, for dynamic rules, involves issues that are not found with static rules. These issues must be addressed at rule posting time. The section concludes by considering how errors that occur during dynamic rule processing (loading and posting) can be handled.

### How to Load and Post Dynamic Rules

Once the domain expert defines decision rules in the rulebase, the Aion application developer must load and post these rules in the Aion knowledge base.

**Important!** The Aion application must include the DynRDLib library from the \BRE\init directory.

For more information about DynRDLib, see the “DynRDLib” chapter in the online Reference.

**Note:** The Aion BRE applications in the BRE\examples\DynaRule directory provide illustrations of loading and posting dynamic rules.

### Connect to the Rulebase

A rulebase may be any relational database with which Aion can connect, either natively or through ODBC.

DynRDLib provides a default connection that points to an ODBC Source named "Aion Rulebase". This ODBC Source, which is created during installation if you have chosen to install the ODBC drivers, points to the default rulebase, Rulebase.mdb, in the \BRE\Rulebase subdirectory.

DynRDLib also provides a service to change the settings of the default connection, RuleBaseServices.SetConnectionProperties( ). This service allows you to establish or change the properties of your connection to a rulebase. For example, to stipulate your own data source, use the following code before calling any other services of DynRDLib:

```
RuleBaseServices.SetConnectionProperties(DRV_ODBC, "your_source_name")
```

SetConnectionProperties( ) also allows you to invoke any of the direct Aion database connections. To connect to an Oracle 8 rulebase on a server called "ora\_prod8" under the userid/password "scott/tiger", invoke SetConnectionProperties( ) with the following arguments:

```
RuleBaseServices.SetConnectionProperties(DRV_ORACLE8, "", "ora_prod8", "scott",  
"tiger")
```

When a direct connection is used to connect to a rulebase, the second argument of SetConnectionProperties( ) is the name of the database rather than the name of the data source. (Oracle does not require a database name; therefore it is nulled-out by passing an empty string.)

When SetConnectionProperties( ) is required, it must be called before attempting to use any database service to load a domain or any dynamic rules from that domain. For more information about SetConnectionProperties( ), see the "DynRDLib" chapter in the *online Reference*.

## Load a Domain

External dynamic rules must first be loaded into the knowledge base. This may be done at any time prior to attempting to post the dynamic rules to the inference engine. Loading is a normal database operation. Thus, loading does not have to be performed within an INFER block.

**Important!** The first step is to load the domain that contains the target rules. Dynamic rules can be loaded only through the domain to which they belong.

The following code loads a domain from the rulebase

```
var DomainName is string = "name"
    // The name of the Domain that you wish to load.
var pDom is &Domain
    // Domain is a class provided by DynRDLib
pDom = LoadDomainByName(DomainName)
    // Perform error processing if pDom is NULL
```

As an alternative to loading a specific domain by name, the Domain class also provides a method to load all domains from the current rulebase: LoadAllDomains( ).

## Load Dynamic Rules

Loading dynamic rules requires a Runtime Facility provided by a DynRDLib class specifically designed to handle the type of rule that is to be loaded. For dynamic decision tables, this facility is the class DecTableRuntime. The runtime facility provides the LoadRuleByName( ) method by which dynamic rules are loaded.

Dynamic rules must be loaded from a domain. In the following example, we are loading a dynamic decision table from the domain we loaded previously by specifying the pointer to the domain as an input argument to DecTableRuntime.LoadRulebyName( ) method.

```
var RuleName is string = "name"
var pDT is &DecTableRuntime
pDT = DecTableRuntime.LoadRulebyName(RuleName, pDom)
    // Perform error processing if pDT is NULL.
```

The loaded dynamic rule is, in effect, an instance of the runtime facility. As we shall see in the following section, the runtime facility also provides a method for posting instances to the inference engine.

The preceding example shows that rules are loaded by name. What does the Aion application programmer do if the names of rules are unknown? There are several alternatives:

- The application programmer can use the `Domain.GetRuleNames( )` method once the Domain is loaded to retrieve a list of all rule names in a domain. The list may be restricted to just those rules that apply to a specific domain interface source (Aion application library) or list of sources by using the `Domain.SetActiveSources( )` method before calling the `GetRuleNames( )` method.
- The `RuleLookup` class methods can be used to do more robust rule name lookups

Once a list of rule names is obtained, a loop can easily be constructed to process this list and to load each rule using the methods of the runtime facility.

For more information about loading rules, see [Loading Invalid and Inactive Rules](#) (see page 61).

## Post and Inference Over Dynamic Rules

As is the case with static rules in Aion, dynamic rules must be posted to the inference engine within the context of an INFER block. The runtime facility used to load the dynamic rule also provides an instance method for posting dynamic rules to the inference engine. In the case of dynamic decision tables, the posting service is offered by `DecTableRuntime.PostDTbl( )`. The following example posts the dynamic decision table that was loaded in the preceding example.

```
INFER
  var hRule is integer
  hRule = pDT.PostDTbl( )
  // pDT is &DecTableRunTime, see preceding code
  // Perform error processing if hRule is NULL
  // Invoke forward or backward chaining
END
```

For more information about posting dynamic decision tables, see [Dynamic Decision Table Runtime Considerations](#) (see page 104).

Once the dynamic rule is posted, it is treated by the inference engine as if it were a statically defined rule. Rules can be reposted at any time without being reloaded from the database as long as they have not been cleared. For more information about clearing rules, see [Housekeeping Considerations](#) (see page 61).

**Note:** Error processing is critical for the posting process because the posting facility must perform extensive validation of the dynamic rule before it can be posted to the inference engine. For more information about the error processing facilities, see [Error Checking](#) (see page 66).

## Housekeeping Considerations

Once dynamic rules are posted to the inference engine, it is no longer necessary for them to exist as objects in the knowledge base (unless, of course, they must be reused in another cycle of the knowledge-base process). It is possible to clear (delete) all resources for the domain, including any loaded rules belonging to that domain, by invoking the `ClearDomain( )` method provided by the runtime facility, for example,

```
pDom.ClearDomain( )  
    // pDom is &Domain initialized when the domain was loaded.
```

If not invoked after posting, `ClearDomain( )` should always be invoked before exiting the method that loaded the Domain. For more information about the runtime facility for dynamic rules, see [Loading Dynamic Rules](#) (see page 59).

**Note:** The runtime facility also makes available a `ClearRule( )` method to delete individual rules. This method is iteratively called within `ClearDomain( )`.

## Invalid and Inactive Rules Loading

At any one time, a rulebase is likely to consist of valid rules, invalid rules, and rules that are inactive. Invalid rules might be rules that are only partially completed. Inactive rules may be former versions of current rules that have been kept in the rulebase for historical purposes or rules that are to be activated at some future time. For more information about inactivating rules, see [Dynamic Rule Manager](#) (see page 126). Invalid rules are inactive by default.

Aion accommodates loading both invalid and inactive rules. The loading facility does not care if the rule being loaded is valid, invalid, or inactive! Therefore, it is important to learn what facilities Aion provides for controlling loading and posting rules.

Rules are loaded by name or (internal) ID. It is the responsibility of the Aion application developer to insure that the rule is of the appropriate type for the application. The methods `Domain.GetRuleNames( )` (as well as `Domain.GetRuleIDs( )`) provides boolean parameters to specify whether names of inactive / invalid rules are to be returned. To load only valid rules, the application developer must set these parameters to `FALSE` when retrieving the names of the rules of a domain. `LoadRuleByName( )` may then be safely called in a loop with the returned list.

**Note:** The `include_inactive` and `include_invalid` parameters of `Domain.GetRuleNames( )` and `Domain.GetRuleIDs( )` are `TRUE` by default.

If inactive and invalid rules are loaded, the application developer can determine the status of a rule by calling the `IsEditStyle( )` method on the loaded rule instance. The method takes a constant as an input parameter and returns a boolean indicating whether the rule instance is of the edit status indicated by the input parameter. Relevant edit styles are:

- `DTBL_EF_INACTIVE`: Is the current rule instance inactive?
- `DTBL_EF_INVALID`: Is the current rule instance invalid?

Aion allows posting inactive rules. The posting facility, for example, `PostDTbl( )`, provides a parameter to indicate whether inactive rules should be posted. This parameter, `post_inactive`, is set to `FALSE` by default.

Naturally, invalid rules cannot be posted. However, it should be noted that a rule that is valid from the rulebase perspective may still be determined to be invalid within the knowledge base that posts it. For example, a domain interface member referenced by the rule may have been deleted from the Aion application but not from the rulebase.

**Note:** For more information about keeping the Aion application and rulebase synchronized, see [Dynamic Rulebase Administrator](#) (see page 116). For more information about error checking during rule posting, see [Special Error Codes for Invalid and Inactive Rules](#) (see page 67).

## Instance Binding

Instance binding implicitly occurs in the context of static rules. IFRULEs, including static decision tables, and WHEN demons are bound to the instance that posts them. Attributes and methods used in these rules are bound to the attributes (values) and methods of the posting instance. If rules reference other instances, the posting instance must contain a pointer attributes to the other instances that its rules use. However, the domain interface exposes methods outside of their ordinary OO context. Instance domain interface members are not explicitly associated with specific instances. Therefore, there are important considerations related to instance binding that the Aion application developer must keep in mind when it comes to dynamic rules that reference instance domain interface members.

**Note:** Binding is not an issue for the domain interface members that are class methods, because all public class methods are globally available.

The lack of OO context for instance domain interface members has important consequences. A dynamic rule, unlike IFRULEs and static decision tables, can be associated with multiple OO contexts at the same time. In other words, a dynamic rule can be associated with multiple Aion instances, where each instance belongs to a different class. When dynamic rules are used there does not need to be a special instance (the posting instance) that contains pointer attributes to other instances needed by a rule.

Dynamic rules do not observe the same binding conventions as static rules. Aion must figure out, for each posted dynamic rule, what Aion instance will serve as the binding for each reference to an instance domain interface member in the rule. If there is only one instance of the class to which a domain interface member belongs, Aion will automatically use that instance as the binding for the reference of that domain interface member. In this case, binding is unambiguous. But in other situations Aion needs some help from the Aion application developer to bind to the appropriate instance.

When a domain interface member belongs to a class that has more than one instance at rule posting time, including even the class of the posting instance, the situation changes. In this case, it is the application developer's responsibility to inform Aion which instance or instances should serve as bindings for references to domain interface members in the dynamic rule. One way to fulfill this responsibility is to invoke the posting facility with a list of pointers to the intended instances as its input argument (for more information about posting dynamic rules, see *Posting and Inferencing over Dynamic Rules*). For example, the relevant signature of the `PostDTbl( )` method is:

```
PostDTbl(BindList is list of &_Object = NULL) : integer
```

The list of pointers to binding instances) does not have to be in any particular order; Aion will figure out which pointers provide bindings for which domain interface member references.

**Note:** For more information about posting methods, see the Source Information column referenced for each type of dynamic rule in *Posting and Inferencing Over Dynamic Rules*.

## Binding Conventions for Dynamic Rules

To summarize Aion's binding conventions for dynamic rules:

- No binding is required for references to domain interface members that are class methods.
- No binding is required for references to instance domain interface members when there is only one instance at posting time of the class owning the domain interface member.

- A binding must be provided for any references to instance domain interface members when there are two or more instances at posting time of the class owning the domain interface member. A pointer to the appropriate instance can be passed in the input argument of the posting method, for example, PostDTbl( ).

**Note:** The pointer current may be passed in the argument if there are two or more instances of the class one of whose instance posts the dynamic rule.

- A binding must be provided for any references to domain interface members that represent an \_Interface method when there are two or more instances whose class implements the \_Interface either directly or indirectly. In other words, if class A, which implements \_Interface X-able has one instance, and class B, which also implements X-able, has one instance, a binding must be provided for the reference to the X-able domain interface member.

These binding conventions are compatible with inheritance: the binding instance's class may be a descendent of the class in which the domain interface member is defined, and the method invoked on the binding instance may be a specialization of the method represented by the domain interface member. In each situation, the invoked method of the binding instance or the invoked class method must be enabled.

### Alternative Approach to Instance Binding

Aion also provides a “manual”, or “pinpoint”, means of instance binding that allows the Aion application developer to bind references for domain interface members on a Condition-by-Condition or Action-by-Action basis. A combination of two methods is used. The first step involves obtaining a pointer to the Condition or Action for which one wants to bind the reference by invoking one of the runtime component methods listed in the following table. These methods are provided by dynamic rule's runtime facility (for information about a dynamic rule's runtime facility, see “Loading Dynamic Rules”). The second step is to perform the binding by invoking one of the SetBindingByXXX( ) methods that belongs to the DIMember class in DynRDLib on instance pointed to by the handle.

The following table summarizes the runtime component methods for the dynamic decision tables. These methods return a pointer to particular component in a dynamic decision table and other information about that component.

DecTableRuntime Component Methods	Returned Handle
GetDTblCondition( )	Obtains a pointer to a specific Condition
GetDTblCheckmarkAction( )	Obtains a pointer to a specific Checkmark Action



<b>DecTableRuntime Component Methods</b>	<b>Returned Handle</b>
GetDTblValueAction( )	Obtains a pointer to a specific Value Action
GetDTblActionAction( )	Obtains a list of pointers to the actions comprising a multiple domain action (MDA).

For more information about using these methods, see “Dynamic Decision Table Runtime Considerations.”

Having obtained the domain interface member whose reference must be bound, one can then bind the reference either by using a pointer to the intended instance, `SetBindingByPointer( )`, or by the name of the intended instance, `SetBindingByName( )` as follows:

```
pDI.SetBindingByName(instance_name) // OR
pDI.SetBindingByInstance(pInst)
```

where 'instance\_name' is a String and 'pInst' is a pointer to an `_Object`. The `SetBinding` methods return a boolean to indicate success or failure of the binding operation.

Rules pertaining to manual, or pinpoint, binding are:

- Manual binding must be performed prior to invoking the posting method, for example, `PostDTbl( )`, which now may be invoked without input arguments if all required references are established by this approach.
- A combination of binding strategies may be used to bind instances to a dynamic rule (see the `DPDRules` example).

Manual binding must be used when a dynamic rule references two instances of the same class. Because the input argument to posting method of a dynamic rule is not sorted, two pointers to the same class cannot be passed. Each instance must be manually bound to the reference of the specific domain interface member in the dynamic rule.

## Error Checking

The DynRDLib contains a class, RuleBaseServices, that provides class methods for obtaining information about the current state of the dynamic rules in the knowledge base. One of these services provides information on the latest recorded error. It is, therefore, a recommended practice to test for any error when making non-querying operations to the dynamic rule structures specifically supported by DynRDLib. The operations in DynRDLib generally return a value indicating whether they were successfully. This return value may be a boolean, FALSE indicating failure, an integer, zero indicating failure, or NULL indicating failure. The Aion application developer is responsible for finding out the type of return value for an invoked operation and setting up the error testing appropriately.

RuleBaseServices provides two methods for obtaining information on the last error encountered:

```
GetErrorInfo(errorMessage:String) : errorCode:Integer  
GetErrorValues(...) // long list of arguments
```

GetErrorInfo( ) returns a formatted error message, which is compiled from error details; whereas GetErrorValues( ) returns the detailed error information.

A typical use of the GetErrorInfo( ) is provided in the DPDRule example:

```
var errCode is integer  
var errMsg is string  
var returnValue is Boolean  
// DynRDLib operation returning returnValue  
If returnValue = FALSE then  
    errCode = RuleBaseServices.GetErrorInfo(errMsg)  
    MessageBox(errMsg, "Error")  
    // Cleanup operations  
End
```

**Note:** In the previous section, we introduced the SetBindingBy methods of the DIMember class. The DIMember class resides in DynRDLib, and therefore cannot avail itself of the RuleBaseServices to track error conditions occurring in its methods. DIMember provides its own error information method: GetDIErrorInfo( ), for example,

```
errCode = pDI.GetDIErrorInfo(errMsg)
```

This method must be invoked for error checking after a call to a DIMember operation, for example, SetBindingByName( ). The DIMember class also provides a GetDIErrorValues( ) method for detailed information on an error.

**Note:** For more information about the methods involved in error processing in DynRDLib and DynRDLib, see “DynRDLib” and “DynRDLib” chapters in the *Online Reference*.

### Special Error Codes for Invalid and Inactive Rules

Aion accommodates loading invalid and inactive rules, see Loading Invalid and Inactive Rules. Rule posting is sensitive to these possibilities:

- The dynamic rule posting facility allows inactive rules to be posted. However, an inactive rule will be rejected with a `DYNRERROR_Attempt_to_post_inactive_rule` error code if the `post_inactive` parameter of the posting facility is `FALSE`.
- A known invalid rule cannot be posted and will be rejected by the posting facility with a `DYNRERROR_Attempt_to_post_invalid_rule` error code. Different processing may be justified in this situation in contrast to valid rulebase rules that are rejected by the posting application, for example, if a rulebase domain interface member has been deleted from the application.



# Chapter 5: Pattern Matching

---

The rules we have seen so far deal with only one instance of a business class. This is acceptable for an application that a loan officer or mortgage broker consults on an ad hoc basis. But you might need an application to process large numbers of loan requests from several branch banks. Pattern Matching rules help you do this quickly and efficiently.

This section contains the following topics:

[Pattern Matching Rules](#) (see page 69)

[Advanced Pattern Matching](#) (see page 73)

## Pattern Matching Rules

In procedural programming, you use some kind of looping construct and then iterate over all the instances of a business class. However, coding this process can be too time-consuming and complicated for a non-programmer.

Aion has a special kind of rule for this situation that is powerful yet easier to learn-pattern matching rules. A pattern matching rule can iterate over many instances of a class or over the instances of several classes. Pattern matching also iterates through a class to include child classes. For example, pattern matching on a class student also includes instances of classes freshman, sophomore, junior, and senior. When an instance meets certain conditions, the action of the rule sets a value or executes a method. Because this process is akin to holding up each instance to a model to see whether a given set of attributes matches, it is called pattern matching.

The format of a pattern matching rule is as follows:

```
rule Rulename
ifmatch Bound variable(s)
where pattern (attributes to be matched)
then action
END
```

Prior to the rule definition, you must bind one or more variables to your class(es). After the rule, you would write the forwardchain command, since pattern matching is available only with forward chaining.

## Pattern Matching over One Class

We can rewrite one of our previous rules to cover a whole class of individual applicants:

```
INFER
  bind App to Applicant
  rule "Approve under $10,000"
  ifmatch App
  where App.loan < 10000 AND App.CreditRating = "GOOD"
then
  App.Approve=TRUE
end
forwardchain
end
```

In the example, Applicant is a class, and the bind statement binds the variable App to the class. The rule iterates over all instances of the Applicant class. The other loan application rules could be re-written in similar fashion.

## Pattern Matching over Two Classes

If your data is at all complex, you probably have more than one business class. In Aion, you can write pattern matching rules for the instances of two or more classes. In the interest of simplicity and performance efficiency, we will consider pattern matching over two classes. We will abandon our loan-request application example in favor of a weekly scheduling application that uses pattern matching and forward chaining.

The scheduling application has two classes, a Worker class and a Task class. Each worker has certain shifts that he is available for work during the week. He is also certified at certain skills, such as forklift operator, quality-assurance inspector, assembly line supervisor, etc. Each task is for a specific work shift during the week, and requires a worker with particular skills.

Following is a table of attributes that the two classes might have:

Attributes of Worker Class	Attributes of Task Class
ID	ID
List of WorkshiftsUnassigned	Workshift
List of SkillsMastered	SkillNeeded
List of TasksAssigned	Assigned

Since a worker has multiple workshifts, skills, and tasks, we use the Aion keyword list that works like automatic arrays. For information about the Aion language, consult the online Reference. The Workshift attribute of Task Class is one of the 21 possible time periods in a week, assuming three shifts per day.

The attribute Assigned holds Boolean values-a TRUE or FALSE indicating whether a Task instance has been assigned to a Worker instance yet. One possible pattern matching rule is the following:

```
INFER
bind InstWorker to Worker
bind InstTask to Task
rule "Assign outstanding tasks to available workers"
ifmatch InstWorker, InstTask
where  InstWorker.WorkshiftsUnassigned includes InstTask.Workshift
and    InstWorker.SkillsMastered includes InstTask.SkillNeeded
then
    Add(InstWorker.TasksAssigned, InstTask.ID)
    Remove(InstWorker.WorkshiftsUnassigned, Task.Workshift)
    InstTask.Assigned = TRUE
end
forwardchain
end
```

Therefore, InstWorker.SkillsMastered includes InstTask.SkillNeeded. This returns TRUE if an instance of Worker has mastered the skill required by a task.

## Pattern Matching over a Class with Two Binding Variables

There are times in data analysis when you do not want to merely find instances that have a particular value for an attribute. Instead, you want groups of instances that share the same value for the attribute, regardless of what it is. To understand the distinction, think of the difference between finding all employees who make a given salary, say \$50,000, and finding groups of employees who make the same salary-which may be \$35,500, \$45,600, \$51,378, etc.

Aion gives you the capability to find such groupings by using pattern matching. In essence, you have one class whose instances fall into two groups. You can create a pattern matching rule that treats the class as if it were two. To do so, you create two binding variables, one for each subgroup.

Consider a university in which graduate students serve as instructors of introductory-level classes. The student population consists of two subgroups-those who are just students and those who are also instructors. If you have one class called Student, pattern matching allows you to declare two variables for the class-one for pupils and one for instructors. Then you write ifmatch rules just as you would for two classes.

## Pattern Matching over Interfaces

During data analysis, you may want to find instances for one or more classes which implement a particular interface (behavior). For example, you may want to pattern match over instances of all classes that implement an interface named Flyable. You can perform such matching by associating bind variables to interfaces (as well as to classes) - for example,

```
bind bFlyable to Flyable    // interface
bind bCargo to Cargo       // class
rule "Send cargo by any flyable means"
ifmatch bFlyable, bCargo
end
```

The inference engine examines instances in the same fashion as it would if the bind variables were Class-associated. It examines the instances:

- When the Pattern Matching rule/demon is posted
- As the instances are created
- As relevant instance attributes change value
- As instances are deleted

Pattern matching over interfaces is very useful when considering that, by binding interfaces to class instances and then pattern matching using those bind variables, one is able to find patterns in otherwise unrelated classes, for example,

```
//assume HasFeathers is an interface
bind bFeathered to HasFeathers //interface
rule "Defeather all feathered objects"
ifmatch bFeathered
where
bFeathered <> NULL
then
Defeather(bFeathered)
end
```

The engine pattern matches over all instances of all Classes that implement the interface (directly or indirectly) and will Defeather those instances.

**Note:** Interface associated bind variables can only reference methods associated with interfaces.



## Advanced Pattern Matching

Pattern matching rules allow you to reason conveniently and efficiently about instances of one or more classes. If you are new to pattern matching in Aion, read the Pattern Matching section in this chapter.

### Flights.app Sample

You can gain an understanding of pattern matching's more advanced aspects by examining the sample Aion application called Flights. To access it, select File, Open and double-click Flights.app in the examples directory.

The Flights application is based on a well-established example from the field of artificial intelligence. It demonstrates the concept of exploding knowledge: Every time a pattern-matching rule fires, new instances are created, which may cause the rule to fire again and create more instances. These expanding cycles continue until a goal is reached.

The goal of Flights is to get from a starting city to a destination city by the shortest route. The user chooses from a list of seven United States cities. The catch is that direct flights are few and short. To make it across the United States, you must take several flights—for example, Los Angeles to Denver, Denver to Chicago, Chicago to New York.

The sample may be clearer if you conceive of Flights as Rails. Think of taking the train between any two of the seven cities. Railroad tracks are fixed in number and only go “direct” to the closest cities. All other journeys consist of several legs.

Following is a list of the 20 flights you can use to link together a route:

Start City	Stop City	Start City	Stop City	Mileage
ATL	CHI	CHI	ATL	708
ATL	NOL	NOL	ATL	480
CHI	NOL	NOL	CHI	919
CHI	SFO	SFO	CHI	2173
CHI	PHX	PHX	CHI	1742
MIA	NOL	NOL	MIA	860
NOL	PHX	PHX	NOL	1496
NYC	CHI	CHI	NYC	809
PHX	SFO	SFO	PHX	762

Start City	Stop City	Start City	Stop City	Mileage
NYC	MIA	MIA	NYC	1334

## The Rules

Flights consists of just two rules, both of them pattern-matching:

### FlightRules( ) Method

```
bind pnode to Node
bind pflight to Flight
var temp is pointer to Node
// Expand
RULE MatchFlight
ifmatch pnode, pflight
    where pnode.CurrentCity = pflight.StartCity
    orderby pnode.TotalMiles + pflight.Miles
then
    if (not Node.Exists(pflight.StopCity)) then
        temp = Node.Create( )
        temp.CurrentCity = pflight.StopCity
        temp.TotalMiles = pnode.TotalMiles + pflight.Miles
        temp.VisitedCities = pnode.VisitedCities
        add(temp.VisitedCities,pflight.StopCity)
    end
end
```

### StopRule( ) Method

```
bind pnode to Node
// Stop condition
RULE MatchingFlight
ifmatch pnode
    where pnode.CurrentCity = TargetCity
then
    stopchain
end
```

### Exists( ) Method

```
// Takes string argument StopCity
var temp is pointer to Node
var nodes is list of pointer to Node
```

```
nodes = list(Node)

for nodes, temp
  if temp.CurrentCity = StopCity then
    //   DebugNode = temp
    return(TRUE)
  end
end
return(FALSE)
```

### What Are Flight Class and Node Class?

The instances of Flight class are the 20 direct flights listed above. Each Flight instance has these attributes: StartCity, StopCity, and Miles. The last attribute is the distance between the cities in miles.

The Node class represents routes. A route consists of one or more connecting flights. A Node instance has these attributes: CurrentCity, VisitedCities, and TotalMiles.

- The CurrentCity attribute tells you how far you have progressed.
- The VisitedCities come from the StartCity and StopCity attributes of the flights you have taken so far.
- The TotalMiles calculates the combined distance of those flights.

**Note:** The first instance of Node is a single city-the starting city of your route. The VisitedCities is NULL, and TotalMiles is equal to zero.

## Bindings

There are two distinct uses of the word binding in pattern-matching rules:

- Binding a pointer to an instance
- Rule-instance binding

It is important to understand the difference.

## Binding a Pointer to an Instance

Consider a simpler example for a moment—one with a single class:

```
bind pCust to Customer
rule  SpecialCustomerCreditLimit
ifmatch pCust
where  pCust.Kind = "special"
then   pCust.SetCreditLimit(10000)
END
```

The bind declaration must be made before posting the corresponding rule. This is one kind of binding—the binding between an instance of Customer class and the pointer variable pCust.

## Rule-instance Binding

The ifmatch rule considers the instances one at a time. The where clause tests to see if there is a match—that is, if the Kind attribute is special. If there is a match, the inference engine binds the Customer instance to the rule so that the action clause can be fired.

This is the second kind of binding—one that impacts pattern-matching efficiency. Since these bindings are relatively costly in computing resources, the engine refrains from making them until it has found instances that meet the where clause conditions. When we discuss ramifications of pattern matching bindings, we are referring to the rule-instance binding.

## Bindings in Flights

Let's return to the Flights example. The first line of FlightRules( ) binds a variable called pnode to the class Node, and variable pflight to the class Flight. In each case, the variable points successively to an instance of the class.

Each combination of a Flight instance with a Node instance is compared. When a combination meets the conditions in the where clause, a binding is created and the rule fired for that binding. The result may be a new Node instance based on the data of the two instances.

Say that you are starting at Atlanta (ATL) and want to go to San Francisco (SFO). The first Node instance has these attributes:

CurrentCity	VisitedCities	TotalMiles
"ATL"	NULL	0

The where clause in FlightRules( ) compares CurrentCity of the Node instance with StartCity of each Flight instance:

```
where pnode.CurrentCity = pflight.StartCity
```

Two Flight instances meet the criteria: ATL-CHI and ATL-NOL. Each combination of instances is bound with the MatchFlight rule:

Node Instance	Flight Instance
ATL	ATL-NOL
ATL	ATL-CHI

The rule action fires once for each rule-instances binding:

```
then
  if (not Node.Exists(pflight.StopCity)) then
    temp = Node.Create( )
    temp.CurrentCity = pflight.StopCity
    temp.TotalMiles = pnode.TotalMiles + pflight.Miles
    temp.VisitedCities = pnode.VisitedCities
    add(temp.VisitedCities,pflight.StopCity)
  end
```

Since nodes do not exist for either of the flight termination cities, the result is that two new instances of Node are created, bringing the total to three:

CurrentCity	VisitedCities	TotalMiles
"ATL"	NULL	0
"NOL"	"ATL"	480
"CHI"	"ATL"	708

## Explosion of Knowledge

Whenever instances are created by firing an ifmatch rule, the inference engine evaluates the rule premise with these new instances, and fires the action if appropriate. The number of available instances can thus "explode" during inferencing. When evaluating the premise, the engine must consider a number of instance-combinations equal to the cross product of the current instances of Node and Flights. For example, if there are 3 nodes and 20 flights, the engine must consider 60 bindings ( $3 \times 20 = 60$ ).

## Achieving the Goal of Flights

Any Node instance that had "SFO" as the CurrentCity would be considered as a possible solution for Flights. The simplest way to achieve the goal would be to continue until all possible routes to SFO were created, and then select the shortest one. For purposes of efficiency, however, Flights.app was written differently using the orderby clause.

## Orderby Clause

Rule-instance bindings are not fired immediately after they are created. First the inference engine creates all bindings that satisfy the where clause. Then it proceeds to fire them in an order that you can specify, using the orderby clause of an ifmatch rule.

The orderby clause can contain one of the following:

- **MOSTRECENT** keyword-The rule-instance bindings are fired, starting with the last one created (LIFO). This keyword can also be used in non-orderby code.
- **LEASTRECENT** keyword-The rule-instance bindings are fired, starting with the first one created (FIFO). This keyword can also be used in non-orderby code.
- One or more integer expressions that allow you to rank the bindings-The inference engine starts with the binding that has the lowest ranking.

Controlling this order often increases the efficiency of your pattern matching. If a solution can be reached before firing all bindings, you can optimize inferencing performance.

If an orderby clause is not specified, the engine fires bindings starting with the first binding created (LEASTRECENT).

If a rule on a parent class has a LEASTRECENT or MOSTRECENT clause, and instances matching the criteria exist in various subclasses, the bindings are ordered correctly only within each class and not across all instances of all subclasses. For example:

```
inst_a = PARCL.Create( )  
inst_d = SUBCL.Create( )  
inst_b = PARCL.Create( )  
inst_d = SUBCL.Create( )  
inst_c = PARCL.Create( )
```

The binding order is `inst_a`, `inst_b`, `inst_c`, `inst_d`, `inst_e` (rather than `inst_a`, `inst_d`, `inst_b`, then `inst_e`). To order bindings by recency across all subclasses, create an integer attribute to hold the order in which the instances were created, then use `ORDERBY` on that attribute.

### New Instances Join the Firing Queue

Whenever a rule is fired that creates new instances, the engine considers the instances for generating additional bindings. The new bindings (if any) immediately join the old bindings in the firing queue. When the engine decides which binding to fire next, it checks the value in the `orderby` clause.

### Orderby in Flights

The inferencing in `Flights.app` is highly efficient because of the following: of the existing bindings, the engine always fires the one that would create the `Node` instance with the smallest `TotalMiles`. In this way, you can always be sure that each new `Node` instance is the shortest route to the `CurrentCity`. By the time you get to `CurrentCity = "SFO,"` you automatically have the shortest route.

The `orderby` clause in `Flights.app` looks like this:

```
orderby pnode.TotalMiles + pflight.Miles
```

### Where Clause

The `where` clause in `Flights` compares instances from the two classes to see whether they meet certain conditions. This process is similar to doing a `table JOIN` in SQL. The clause looks like this:

```
where pnode.CurrentCity = pflight.StartCity
```

If the `CurrentCity` of a `Node` instance equals the `StartCity` of a `Flight` instance, the two are bound to the rule and enter the firing queue.

In other situations, you put conditions in the `where` clause to “filter” instances of a single class before combining the instances of multiple classes. This is similar to doing a SQL `SELECT` on records of a single table prior to doing a `JOIN`.

For Flights, imagine that the original list of flights included some in countries outside the United States. To figure the shortest route from Atlanta to San Francisco most efficiently, you would want to consider only flights within the United States. Assuming that the Flight class has a new attribute called Country, you could modify the where clause of the MatchFlight rule in the following way:

```
RULE MatchFlight
ifmatch pnode, pflight
    where pnode.CurrentCity = pflight.StartCity
        and pflight.Country = "USA"
    orderby pnode.TotalMiles + pflight.Miles
then
    if (not Node.Exists(pflight.StopCity)) then
        temp = Node.Create( )
        temp.CurrentCity = pflight.StopCity
        temp.TotalMiles = pnode.TotalMiles + pflight.Miles
        temp.VisitedCities = pnode.VisitedCities
        add(temp.VisitedCities,pflight.StopCity)
end
```

### Order Unimportant in Where Clause

For maximum efficiency, the engine must filter out non-U.S. flights before matching Flight and Node instances. In Aion, however, you need not worry about the order in which these statements appear in the where clause. The inference engine analyzes your statements and applies them in the most efficient order.

### Where TRUE clause

Aion does not support the where TRUE clause. The where clause must reference all bind variables defined in an ifmatch rule. If it does not, a syntax error occurs.



## SingleFire Rules

Rule authors can classify a pattern-matching rule as a SingleFire rule. The `singlefire` keyword is inserted into the rule when you check the Rule Editor's SingleFire checkbox. After firing a SingleFire rule on the first appropriate binding, the inference engine changes the rule's state from READY to FIRED. After that, the engine ignores events associated with that rule's bind variables. For example:

```
bind bP to Person
bind bD to Duck
rule "Associate Persons and Ducks"
ifmatch bP, bD
where
    bP.Age > 10
    and bP.Age = bD.Age
singlefire
then
    // Some actions
end
```

This option applies only to IFMATCH rules; it does not apply to WHENMATCH demons.

**Note:** The SingleFire option is mutually exclusive with the demon option; that is, when the demon option is checked, the SingleFire option is disabled.

An application can restore a SingleFire rule back to a READY state using the existing SysLib method:

```
_Engine.EngineRuleReset(...)
```

Once reset, the rule again reacts to events associated with its bind variables, but only to subsequent events and not to events that occur while it is in a FIRED state.

When calculating bindings for a SingleFire rule, the inference engine attempts to minimize the number of bindings. If the rule uses LEASTRECENT (default) ordering and does not specify ORDERBY clauses, the engine immediately fires the rule when it finds the first binding. Otherwise, it calculates all of the bindings before firing the rule.

## Inference Engine and Multiple ifmatch Rules

So far we have considered only the one ifmatch rule in FlightRules( ). Flights has another rule, however-the one in StopRules( ):

```
bind pnode to Node
// Stop condition
RULE MatchingFlight
  ifmatch pnode
  where pnode.CurrentCity = TargetCity
  then
    stopchain
  end
```

This rule tests to see if the CurrentCity of any existing Node instance equals the user's final destination, or TargetCity. To see how it is used, examine the inference block in InvokeFlightRules( ):

```
INFER
  StopRule( )
  FlightRules( )
  forwardchain
end
```

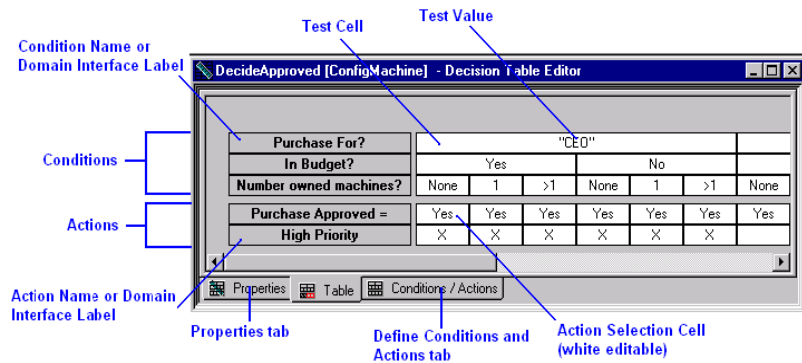
Because there are no priority clauses, priority order is the same as posting order. Thus, StopRule( ) has a higher priority than FlightRules( ).

With respect to priority order, the inference engine treats ifmatch rules similarly to the way it treats IFRULEs. Whenever an UNKNOWN attribute is resolved or a new instance is created, the engine goes back to the top of the priority order and visits the stop rule.

Therefore, every time a new instance of Node is created, the engine visits the stop rule to see if pnode.CurrentCity = TargetCity. As soon as a match happens, inferencing ends. Since the orderby clause guarantees that each node is truly the shortest route from Atlanta, the first node that ends in San Francisco achieves the goal.

# Chapter 6: Decision Tables

A decision table is a tabular rule that evaluates a set of conditions and executes a set of actions based on that evaluation. The Decision Table Editor uses a graphical display to show the order and flow of the logic and the actions that can result.



A decision table is similar to a collection of IFRULEs with related premises and actions (see the section Rephrasing IFRULEs as Decision Tables in this chapter). Like IFRULEs, decision tables can use attributes and methods, and can be used in forward or backward chaining. However, decision tables do not match over instances of one or more classes like an IFMATCH rule.

Through the use of domain interfaces, decision tables have the capability of being dynamic. (For more information about domain interfaces and dynamic rules, see the chapter "System Considerations for Supporting Business Rules" in the *User Guide*.) In particular, this means that decision tables can be stored outside the Aion IDE and maintained by domain experts. For more information about rule management by domain experts, see [Maintaining the Dynamic Rulebase](#) (see page 115). The section Dynamic Decision Tables discusses handling dynamic decision tables in the Aion knowledge base.

This section contains the following topics:

[Benefits of Using Decision Tables](#) (see page 84)

[How to Create and Open Decision Tables](#) (see page 84)

[How to View and Modify Decision Table Properties](#) (see page 85)

[Customizing the Decision Table Editor](#) (see page 95)

[Compressing a Decision Table](#) (see page 96)

[Runtime Execution](#) (see page 100)

[Chaining](#) (see page 101)

[Rephrasing IFRules as Decision Tables](#) (see page 102)

[Dynamic Decision Tables](#) (see page 103)

## Benefits of Using Decision Tables

Using a decision table instead of a set of production rules can offer the following benefits:

- It is easier to ensure completeness and consistency, and to avoid redundancy in specifying a discreet set of conditions and test values, and relating those conditions to a set of actions.
- Aion decision tables include a runtime algorithm to optimize the execution of the decision table rule by ignoring conditions that do not contribute to determining a specific action.

Decision tables are generally favored over rules by domain experts, because decision tables provide a simple view, or “full picture,” that may not be easily gleaned from a list of discreet production rules.

## How to Create and Open Decision Tables

Creating a Decision Table entails opening the Decision Table editor, adding new Conditions and Actions, and generating the table. After you have created a Decision Table, you can modify it using the Decision Table editor.

### Create Decision Table

#### **To create a decision table**

1. From the Logic menu, choose New, Decision Table to display the New Decision Table dialog.
2. Type a name for the new decision table in the Name field.
3. Choose an Owning Class from the drop-down list.
4. Click OK. A new method object is created in the application, and the Decision Table Editor opens.

When first created, the Decision Table Editor defaults to the empty Table page. Click the Properties tab to choose properties for the method. Click the Conditions/Actions tab to specify conditions and actions.

To create a new Condition, see [Add and Modify Conditions](#) (see page 86). To create a new Action, see [Add and Modify Actions](#) (see page 89).

After you have created conditions or actions, the decision table is created (or updated) when you choose Save on the toolbar. The decision table displays when you choose the Table tab. If Auto Refresh is turned on, the table will be automatically generated when you click on the Table tab.

## Open an Existing Decision Table

You can open an existing decision table method in the Decision Table Editor or in the Method Editor. However, it is advisable always to use the Decision Table Editor to view or edit a decision table method.



**Note:** In the CA Aion BRE IDE, decision table methods are identified by the icon shown above.

- To open a decision table in the Decision Table Editor, from within the Project Workspace or Aion Explorer, highlight the Decision Table object and do one of the following:
  - Double-click
  - Right-click, then choose Open from the pop-up menu
- To open a Decision Table in the Rule Analyzer, right-click the Decision Table object, then choose Analyze from the pop-up menu.
- To open a Decision Table in the Method Editor, do one of the following:
  - Select the object, and from the Aion Logic menu choose Properties
  - Right-click the object and choose Properties from the pop-up menu.

**Note:** Opening a Decision Table method in the Method Editor is not recommended.

The code for the decision table is displayed on the Implementation tab of the Method Editor.

**Important!** Do not use the Method Editor to modify a decision table. Make changes to the table using the Decision Table Editor.

## How to View and Modify Decision Table Properties

To view or modify decision table properties, click the Properties tab in the Decision Table Editor.

- The Method Name field displays the name of the method used to post the rule.
- The Rule Name field is the name of the rule (within the method). This name is used by the Debugger and trace to identify the rule.
- Priority sets the priority used by the engine to order this rule with other posted rules. This value can be any integer or integer constant, or it can be left blank.

- The Class Method checkbox is checked if the method is a class method (instead of an instance method).
- This box is dimmed for inherited methods or methods not under the \_Object class.
- The Layout buttons permit you to choose whether to display the table in Landscape (horizontal) or Portrait (vertical) view.

**Note:** You can also change views by right-clicking in the Decision Table Editor Table page and selecting Toggle Layout from the pop-up menu.

## Add and Modify Conditions

Conditions define the premises of a Decision Table rule. A Condition is composed of a Condition Name and a number of test cells. Test cells display ranges or values specified in the Condition row. Conditions can be defined either by specifying (the implementation of) a condition or by specifying a Domain Condition.

### To add a condition

1. Do one of the following:
  - Right-click in the Condition area of the Decision Table Editor, then select New from the right-click pop-up menu.
  - From the Logic menu, choose New, Condition.
  - Click the Add Condition button on the Decision Table toolbar.
2. Select Specify Condition or Domain Condition from the pop-up menu.

The condition properties pane appears to the right of the Conditions/Actions page.

- a. For Specify Condition: enter a name of your own choosing to describe the condition in the Name field.  
If no name is entered, the default condition name, newConditionRow, will be used.
  - b. For Domain Condition: select a condition domain interface member from the Name field drop-down list.
3. In the Test Values field, enter test values for comparing with values obtained from the knowledge base by the condition's implementation.

If no operator is specified, equal (=) is implied.

- Values must be literal expressions or constants; they cannot be non-constant attributes or methods.
- String values must be enclosed in quotes.

- Valid operators include: <, >, <=, >=.
  - To specify a numeric range, use the range operator .. between the extreme values of the range. For example, >18..<=65, which would be read "Greater than 18 and less than or equal to 65."
  - Boolean constants are TRUE, FALSE; Yes, No.
4. Choose Test unknown value if you wish to include Unknown as one of the values the inference engine explicitly tests for.
  5. Choose Test for Other Values if the rule should check for condition values other than those listed in the Test Values.

If this checkbox is turned on, an ELSE cell is displayed in the table to indicate the decision path to take when values other than those explicitly listed are encountered.

6. For Specify Condition only: enter the source from which Aion will obtain the value to be tested by this condition in the Implementation field.

Values can be any of the following:

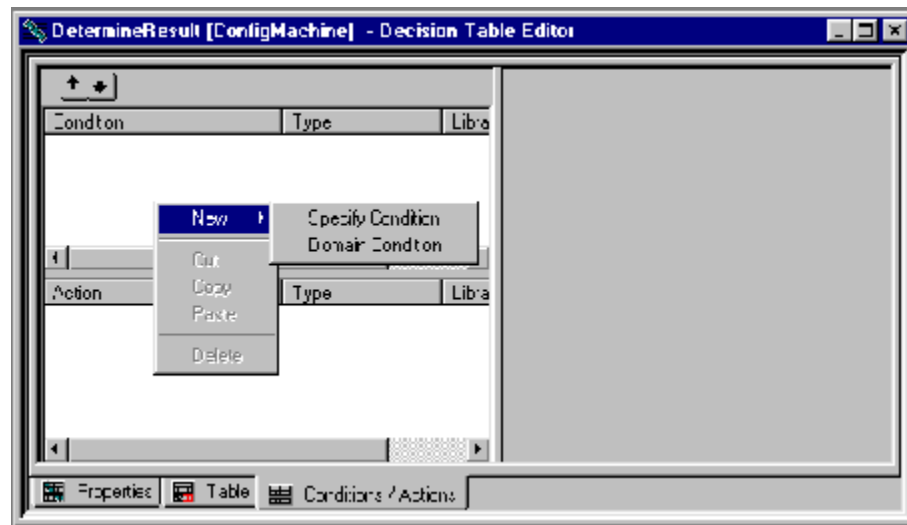
- The name of an Aion class, for example, Department.
- A boolean expression; for example, Age > 50.
- An Aion method that returns a value; for example, GetDepartment( ).

#### **To modify Condition properties**

1. In the Decision Table Editor Table page, right-click a Condition Name cell.
2. Select the Edit <name> pop-up to display condition properties in the Conditions/Actions page.

To view the effect of changes, click the Table tab. Changes are automatically applied if Auto Refresh is checked in the Table page pop-up menu. Or you can select Refresh from the Table page pop-up menu to apply changes.

**Note:** You can tell Aion to use Auto Refresh to refresh the table each time a condition or action is modified or added by checking Automatically Refresh in the Decision Table Options dialog (select Tools, Options, Decision Table).



What do these choices represent?

By using Domain Interface Members to support dynamic rules, Aion has introduced a new way to construct decision tables: with Domain Interface members. The Specify Condition style represents the 8.1/8.1.1 style of defining conditions and constructing decision tables, while Domain Condition permits static decision tables to be defined with Domain Interface members. The use of Domain Condition is encouraged.

- Specify Condition: you must create a name for the Condition and specify its implementation in terms of an existing Attribute or Method.
- Domain Condition: you may use a previously defined Domain Interface Condition member as the Condition by selecting its label from the Name drop-down list. In this case the Implementation is automatically provided: it is the method designated by the selected Domain Interface member label.

**Note:** A decision table may consist of both specified implementations and Domain Interface members.



Rules for entering test value specifications are:

- Use of ',' (the comma) is not allowed.
- In the Decision Table Editor in the Aion IDE, string test values must be enclosed in quotes.
- Real test values have to be specified in USA settings without the use of ','  
**Note:** Real and integer values are displayed according to the country format Setting.
- When specifying a range, place < or > before equals (=). Start range is > value or >= value. Optionally use '..' with an end range < value, <= value.

## Add and Modify Actions

Actions define the activities to be taken by the decision table rule if the conditions are met. Actions are composed of an action and a number of selector cells. Actions can be defined in three ways: Specify Action, Domain Action, and Multiple Domain Action.

### To add a action

1. Right-click in the Action pane and select New from the right-click pop-up menu.

Or

From the Logic menu, choose New, Action

2. Select Specify Action, Domain Action, or Multiple Domain Action from the pop-up menu.

The action properties pane appears to the right of the Conditions/Actions page.

- a. For Specify Action: enter a name to describe the action in the Name field. If no name is entered, the name defaults to newActionRow.
- b. For Domain Action: select the label of an Action domain interface member from the Name field drop-down list. Notice that, if present, the type of input argument to the domain interface member is indicated; otherwise (none) is indicated. The description of the selected domain interface member is provided.
- c. For Multiple Domain Actions: enter a name to describe the action in the Name field. If no name is entered, the name defaults to newActionRow. A list of labels of Action domain interface members that are possible values for Multiple Domain Action is provided

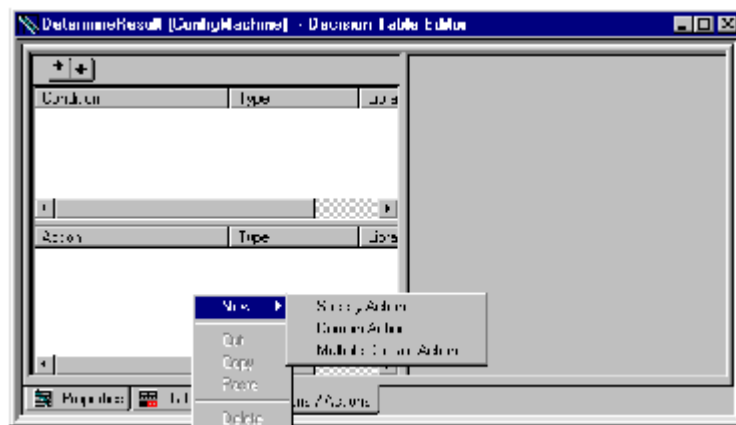
3. For Specify Action only: enter the statement(s) that are to be executed when the action is to be performed in the Implementation field. This can be any Aion executable statement or block of code, including
  - An assignment statement; for example, Department = "CS"
  - A fully specified Aion method; for example, SetDepartment("CS").Actions should be limited to a few lines of code.

### To modify action properties

1. In the Decision Table Editor's Table page, right-click an Action Name cell.
2. Select the Edit <name> pop-up to display action properties in the Conditions / Actions page.

To view the effect of changes, click the Table tab. Changes are automatically applied if Auto Refresh is checked in the Table page pop-up menu. Or you can select Refresh from the Table page pop-up menu to apply changes.

**Note:** You can tell Aion to use Auto Refresh to refresh the table each time a condition or action is modified or added by checking Automatically Refresh in the Decision Table Options dialog (select Tools, Options, Decision Table).



- **Specify Action:** Like Specify Condition, Specify Action follows the 8.1 style of defining Actions.

You must create a name for the Action and specify the implementation of that action by means of an explicit method call or attribute-value assignment.

Where the implementation is a method call without arguments, this style is satisfactory. However, when the implementation requires different arguments to be passed to a method, or different values to be assigned to an attribute, this style tends to produce large decision tables, because each different argument or value has to be explicitly defined in a separate Action. Domain Interfaces provide a means to address this problem. Both Domain Action and Multiple Domain Action make use of Domain Interface Members. Using Domain Interface members provides additional capabilities not provided by specifying an Action. For this reason, the use of Domain Action and Multiple Domain Action is encouraged over Specify Action.

- **Domain Action:** Like Domain Condition, Domain Action allows you to choose a previously defined Domain Interface Action as the Action by selecting its label from the Name drop-down list, and, again, the implementation is automatically provided by the Domain Interface Member. However, in the case of Domain Actions, there are two possibilities:
  - The Domain Interface Member does not take an input argument. In this case, a specified action and the Domain Action look much the same in the decision table itself: invocation of both is indicated by an "X" in the appropriate Action Selector cell of the decision table.
  - The Domain Interface Member takes an input argument. In this case, you must specify the appropriate value to be passed as that argument in the Action Selector cell of the decision table where the Action is to be invoked. This technique reduces the table bloat that was produced when actions involved different arguments or attribute-value assignments.
- **Multiple Domain Action:** This option allows you to specify Domain Interface Actions in the Action Selector cells of a decision table.

One row can be used to invoke different actions as long as these actions are mutually exclusive. In this case, you must create a name of the Action, as is done in the Specify Action style. The difference is that during decision table construction, an Action Selector cell of a Multiple Domain Action automatically transforms into a drop-down list when it becomes the focus. The list consists of Domain Interface Member labels that may be selected for that cell. Valid Domain Interface Members are those that (1) are defined to be of type Action, and (2) do not require an input argument.

## Summary: Specifying Selector Cells

If no selector cells for Actions are specified for a given combination of condition cells, the corresponding action cells are highlighted in the Decision Table Editor. (By default, highlighting is red.) This highlighting is a warning that there is a potential "hole" (that is, a possible outcome for which no action is specified) in the table logic. This could be valid if the combination of values is impossible, that is:

Has Driver License = TRUE  
and Age < 16

A blank selector cell indicates that the decision table will invoke no activity for an Action for the combination of Conditions. To invoke a result for the decision table, the selector cell must be specified. Selector cells are specified in three ways depending upon the nature of the action.

- Action specified by name and domain interface member with no input argument:

Place an "X" in the selector cell by left clicking in the cell. The "X" may be toggled off by left clicking in the cell a second time.

- Domain interface member with an input argument:

Type the value of the input argument in the selector cell. Rules for action selector values are:

- Use of ',' (the comma) is not allowed.
- In the Decision Table Editor in the Aion IDE, string selector values must be enclosed in quotes.
- Real test values have to be specified in USA settings without the use of ','

**Note:** Real and integer values are displayed according to the country format Setting.

- Boolean values will be shown in True/False format.

- For multiple domain actions:

Select the appropriate action domain interface member from the drop-down list. To cancel the action for that combination of conditions, select the first element in the drop-down list, which is always the NULL value.

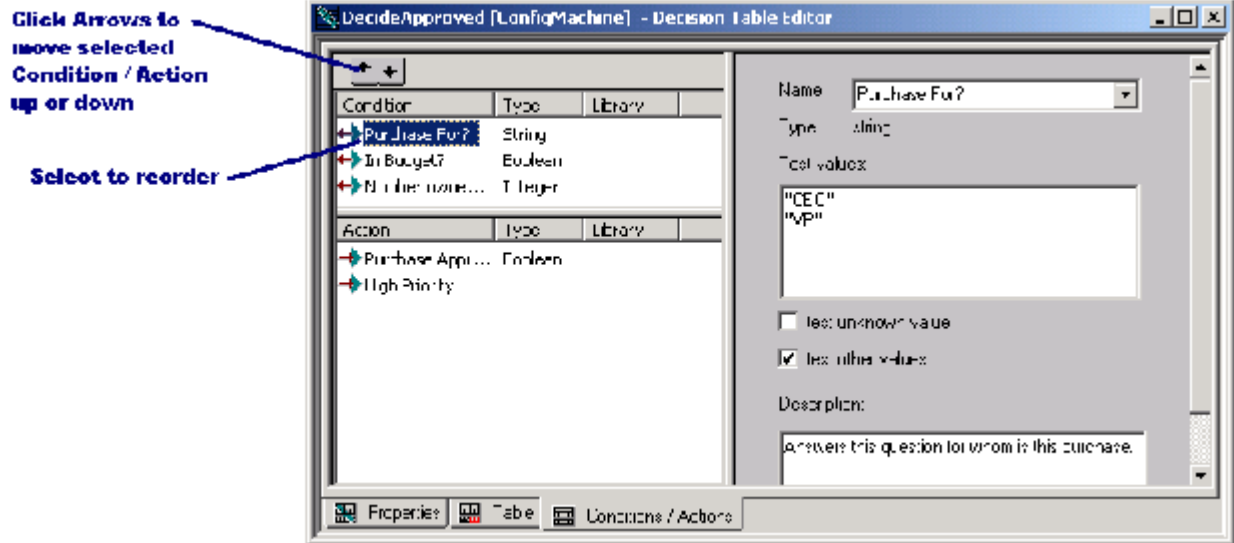
## Delete a Rule

### To delete a rule from the rulebase:

1. Select the rule in the Domain Workspace of the Decision Table Editor.
2. Choose Edit, Delete, or right-click to display a pop-up menu and choose Delete.

## Ordering Conditions and Actions

When the Conditions/Actions tab is displayed, you can use the Move Up and Move Down arrows to change the order of conditions or actions in the generated decision table.



**In the Conditions/Actions page, to change the order of the conditions or actions of the decision table:**

1. Select a condition or action in its respective pane.
2. Click the arrow buttons on the Decision Table Editor toolbar to move the selected condition or action up or down.

**Note:** Aion also provides an automatic means to order conditions to achieve the minimal number of combinations of conditions, see [Achieving Optimal Condition Order for Compressibility](#) (see page 99).

## Cutting and Pasting Conditions and Actions

### To cut and paste conditions and actions:

1. Select a condition or action in its respective pane.
  - Select Cut or Copy from the right-click menu
  - OR
  - From the Edit menu, select Cut or Copy.
2. Select Paste from the right-click menu or the Edit menu. The Condition(s) or Action(s) are pasted after the selected item.

**Note:** To rearrange Conditions or Actions, see Ordering Conditions and Actions in this chapter.

### To cut, copy, or paste subtable contents:

Right-click in a Decision Table column heading, and select Cut Values, Copy Values, or Paste Values from the Collapse pop-up menu.

## Displaying a Decision Table

You may control the appearance and/or behavior of a decision table on the Table page by right clicking in a non-table area of the page. A pop-up menu appears with the following options:

Menu Option	Explanation	Related Topic
Show compressed	See Compressing a Decision Table in this chapter	Selecting the compression algorithm can be performed on the Decision Table Options dialog (Tools, Options, Decision Table). For the procedure, see <a href="#">Customizing the Decision Table Editor</a> (see page 95).
Optimum condition order	See <a href="#">Achieving Optimal Condition Order for Compressibility</a> (see page 99)	
Toggle layout	Changes display between landscape and portrait.	Layout can be set on the Properties page, see <a href="#">How to View and Modify Decision Table Properties</a> (see page 85).

Menu Option	Explanation	Related Topic
Refresh	Redraws the decision table according to the latest changes to the definitions of the conditions and/or actions.	
Auto refresh	Toggles automatically refreshing of the decision table. Automatic refreshing cause the table to redrawn when it is redisplayed following any changes to the definitions of the conditions and/or actions.	Auto refresh can be set for all decision tables in the Decision Table Options dialog (Tools, Options, Decision Table). For the procedure, see <a href="#">Customizing the Decision Table Editor</a> (see page 95).

## Customizing the Decision Table Editor

You can use the Decision Table Options dialog (Tools, Options, Decision Table) to set options that control how the editor displays Decision Tables. Use the fields and options in the Decision Table Options dialog as follows:

- Compressed View-Specifies what algorithms should be used to display Decision Tables in compressed view
  - Hide All Combinations with no Actions-Automatically hides any condition test values of the Decision Table that do not result in an action
  - Hide Groups with no Actions-Automatically hides condition test values of the Decision Table if an entire group of test values do not result in an action
  - Combine Values with same Actions-Combines condition test values that lead to the same action. For example, if A and B had the same result, they would be combined into a column labeled "A or B."

- Table Display-Contains options related to the display of the table
  - Automatically Refresh-Refreshes the Decision Table display immediately after changes are made
  - Minimum Action Cell Width-Sets the minimum number of characters that a cell will display.
  - Maximum Characters in Cell-Sets the maximum number of characters (width) that a cell will display of its contents.
  - No Action Color-Lets you choose the color used to indicate an action cell with no action specified

Domain Interface Filter-Indicates whether domain interface members defined in the application's included libraries are included in the domain interface members to which the application has access for constructing conditions and actions.

You can also select the font in which the Decision Table displays information. See the Fonts section of the "Creating and Editing Applications" chapter of the User Guide.

## Compressing a Decision Table

Compression is an edit-time feature concerned with making optimal use of screen real estate when displaying a decision table. Values are suppressed when those values are not relevant for distinguishing between different actions. Compression makes the Decision Table easier to understand by hiding irrelevant parts of the Decision Table and combining redundant parts.

### **To compress the decision table:**

1. Right-click in the Decision Table Editor's Table page.
2. In the right-click pop-up menu, check Show Compressed.

Or:

From the View menu, select Show Compressed.

A rectangular marker appears in the upper left corner of the Table page to show that you are in compressed view.



There are three kinds of compression used by the Decision Table Editor. These can be accessed from Decision Table Options (see Customizing the Decision Table Editor).

- **Hide All Combinations with no Actions**-If any condition combination (from the topmost condition test value ) does not result in an action, the condition test value and its subtable are not displayed. By default this compression algorithm is turned off.
- **Hide Groups with no Actions**-If an entire group of condition combinations does not result in any actions, that group of condition test values are not displayed in the Decision Table Editor.
- **Combine Values with same Actions** - If two or more condition test values result in the same actions, that is, they have the same subtables, these test values are combined into a single subtable and the values are OR'd together.

The following examples demonstrate each kind of compression using a Condition row with three possible values, "CEO", "VP", and "ELSE", and an Action row with two possible outcomes, Approved and Rejected.

In the first example, the uncompressed Decision Table shows no actions are executed for the "VP" column:

Purchase For?	"CEO"		"VP"		"ELSE"	
In Budget?	Yes	No	Yes	No	Yes	No
Purchase Approved =	Yes	Yes			Yes	
High Priority	X	X				

When you compress the Decision Table, the "VP" column is removed from the display:

Purchase For?	"CEO"		"ELSE"	
In Budget?			Yes	No
Purchase Approved =	Yes	Yes		
High Priority	X	X		

In the second example, values "CEO" and "VP" always result in the same action.

Purchase For?	"CEO"		"VP"		"ELSE"	
In Budget?	Yes	No	Yes	No	Yes	No
Purchase Approved =	Yes	Yes	Yes	Yes	Yes	No
High Priority	X	X	X	X		

A compressed Decision Table combines those cells into a cell that reads "CEO [OR] VP" as shown in the following table:

Purchase For?	"CEO","VP"	ELSE	
In Budget?	-	Yes	No
Purchase Approved	Yes	Yes	No
High Priority	X		

If a cell has the same action as an "ELSE" cell, the cell value is conceptually OR'd with the ELSE cell but the value is not shown with the "ELSE" cell. For example, if the Condition cells with a common action were "VP" and "ELSE", the Condition row would have two cells that read "CEO" and "ELSE."

**Note:** Actions of compressed subtables are not editable.

## Manually Collapsing Subtables

Aion allows you to manually specify particular parts of the decision table that you may wish to collapse temporarily so that you can focus on a particular area of a table.

### To collapse or expand the decision table in either normal or compressed view

1. Right-click in the test cell at the top of the condition you wish to collapse, and select Collapse (Collapse, Collapse Down, Collapse All, Expand, Expand Down, Expand All) from the right-click pop-up menu.
2. Left click on the test cell at the top of the condition you wish to collapse.

Left clicking on a collapsed Condition Test cell or right clicking on the cell and selecting "expand down" restores the subtable.

**Note:** Actions under manually collapsed subtables are not editable.

## Achieving Optimal Condition Order for Compressibility

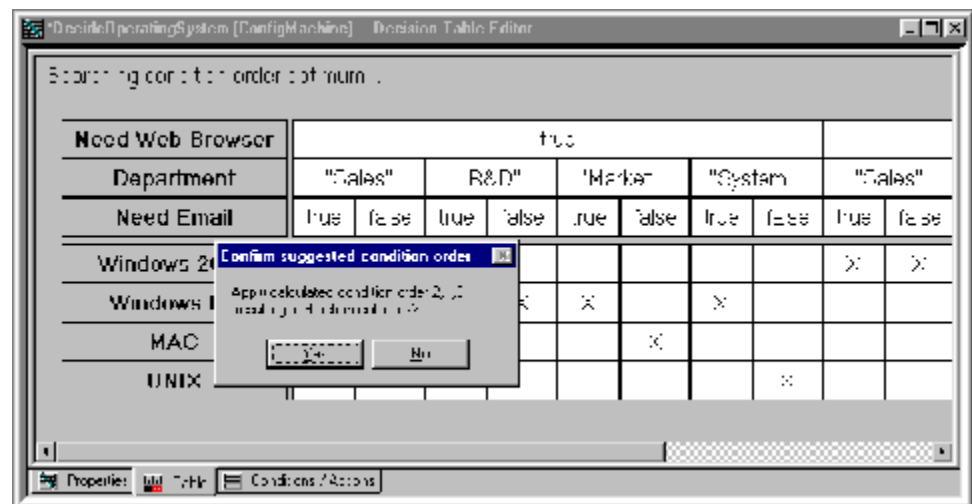
Aion also provides a sophisticated algorithm for determining the optimum order of the Conditions in your decision table to achieve optimum compressibility.

**To determine whether the order of the conditions of the decision table produces the fewest number of results when the table is compressed**

1. Right-click in the Decision Table Editor's Table page.
2. In the right-click pop-up menu, check Optimum Condition Order.

If a more optimum condition order is possible, you will be asked whether you would like the table to be reorganized in that order.

If the algorithm can find an optimum order, you will be asked if you wish to have the table reorganized in that order. In the following example, Aion has determined that greater compressibility (fewer action columns) could be achieved if the first and second Conditions (Need Web Browser and Department) were reversed, that is, the second Condition should be first. Answering "Yes" automatically reorders your Conditions and displays a compressed view of your decision table.



If the algorithm cannot determine a condition ordering that is smaller than the current order of the Conditions, you will receive the message "Current condition order is a minimum."

**Note:** Using optimal compressibility may also result in improved execution of the decision table.

## Runtime Execution

The following Decision Table, which will be called `ClassifyPerson`, is used as an example throughout this section:

Gender	Male				Female			
Age	<21		>=21		<21		>=21	
Hair Color	Red	Brown	Red	Brown	Red	Brown	Red	Brown
Group1	X	X	X		X		X	X
Group2		X				X		

`ClassifyPerson` groups people according to selected physical characteristics. Depending on a person's characteristics, a person may belong to zero, one or two groups. For example, a younger man with brown hair belongs to both groups whereas an older man with brown hair belongs to neither group. This Decision Table consists of three Conditions (Gender, Age, Hair Color) and two Actions (Group1, Group2).

## Rule Posting

An Aion application posts a Decision Table by invoking the Decision Table's method. As with an IFRULE, a Decision Table may be posted with an inferencing priority.

The engine posts Decision Tables in a READY state. The engine resets the rule to a FIRED state when the engine evaluates the conditions and finds one or more actions associated with the evaluation outcome. In the above example, the engine fires the rule for:

- All young men of any hair color
- Older men with red hair
- All women of any age and hair color

The engine resets the rule to a FAILED state when, upon evaluating the rule's Conditions, the engine cannot find any Actions associated with the evaluation outcome. In the above example, the engine fails the rule for older men with brown hair.

The engine resets the rule to a PENDED state when the engine detects an UNKNOWN attribute

## Condition Evaluation

The engine evaluates Conditions in the order specified within the Decision Table. So, in `ClassifyPersons`, the engine always evaluates Gender before Age, and Age before HairColor. You can change the Condition evaluation order by rearranging Conditions; see the section [Ordering Conditions and Actions](#) (see page 93).

The engine employs test values in the order specified within their Condition row. In `ClassifyPersons`, the engine always tests Gender against "Male" before it tests Gender against "Female."

If test values overlap - for example, Age: <21, >18..<40, >30 the engine stops testing upon finding the first applicable test value. For example, if Age evaluates to 19, the engine stops with the first test value. If Age evaluates to 35, the engine stops with the second test value. The engine employs the third test value only for Ages >=40.

If desired, an ELSE cell can be used to test for all other values that are not defined in the condition, except UNKNOWN. If a condition is UNKNOWN, it does not yet have a value, so the inference engine does not take the ELSE path.

The engine avoids pending Decision Table-rules if, despite UNKNOWN attribute(s), it can safely determine the Actions to be performed. For example, the engine can safely fire the rule for a red-haired person-even though the person's gender and age are unknown because all red-haired persons belong to Group1 but not Group2.

## Action Execution

The engine may perform multiple Actions as an outcome of Condition evaluation. When multiple Actions apply, the engine performs them in the order specified within the Decision Table-rule. In `ClassifyPersons`, for a brown-haired younger man, the engine performs Group1 actions before Group2. If the engine detects UNKNOWN attributes while performing Actions, it pends the rule-just as it would for an IFRULE's Action.

## Chaining

Decision Table-rules fully participate in chaining along with rules of any other type. The engine can use a Decision Table-rule for both forward chaining and backward chaining. Once the engine has fired or failed a Decision Table-rule, the rule no longer participates in chaining, unless reinstated due to Truth Maintenance side-effects.

When backward chaining, the engine selects a Decision Table-rule if any of its Actions can resolve the primary goal or any current subgoals. When firing a Decision Table-rule, the engine evaluates Conditions and performs Actions as it does for forward chaining, that is, the engine does NOT somehow focus on particular Conditions or Actions which are known to contribute to goal resolution. As a result, the rule may fire without contributing to goal resolution. And, as noted above, once fired or failed, the rule no longer participates in chaining.

## Rephrasing IFRules as Decision Tables

Decision Table rules are generalizations of IFRules, which means that IFRules can be rephrased in terms of Decision Tables.

### One-to-One Mappings

The following IFRule could be rephrased as shown in the decision table, where that Get Person's Age and Set Eligibility are condition names or domain interface labels for the corresponding application methods.

```
IFRULE pPerson.GetAge() < 21 THEN
    pPerson.SetEligibility(FALSE)
END
```

Get Person's Age	<21
Set Eligibility	FALSE

### ANDed Premise Expressions

If an IFRule premise is an ANDed expression, the rule could be rephrased as a decision table with multiple conditions. For example, the following IFRule can be rephrased as the decision table shown immediately following:

```
IFRULE pPerson.GetAge() >= 21
    AND pPerson.GetHeight() > 60
THEN
    pPerson.SetEligibility(TRUE)
END
```

Get Person's Age	>=21
Get Person's Height	>60
Set Eligibility	TRUE

## Consolidating IFRules into a Single Decision Table

If multiple IFRules can be logically grouped within the same decision table, runtime efficiencies can result. For example, you could rephrase the following IFRules as the decision table immediately following:

```
IFRULE pPerson.GetAge() >= 21
    AND pPerson.GetHeight() > 60
THEN
    pPerson.SetEligibility(TRUE)
END
IFRULE pPerson.GetAge() >= 21
    AND pPerson.GetHeight() < 60
THEN
    pPerson.SetEligibility(FALSE)
END
IFRULE pPerson.GetAge() < 21
THEN
    pPerson.SetEligibility(FALSE)
END
```

Get Person's Age	>=21		ELSE	
Get Person's Height	>60	ELSE	>60	ELSE
Set Eligibility	TRUE	FALSE	FALSE	FALSE

## Dynamic Decision Tables

Aion provides the ability to make decision tables dynamic. For more information about dynamic rules, see the chapter “System Considerations for Supporting Business Rules” in the User Guide. Making decision tables dynamic means that decision tables can be saved in a persistent medium and loaded at runtime or created within the Aion knowledge base at runtime. For more information about creating decision tables at runtime, see the [Constructing Non-Persistent Dynamic Rules](#) (see page 141).

Aion provides a default decision table editor for editing persistent dynamic rules outside the IDE (see [Dynamic Decision Table Editor](#) (see page 129)). The dynamic decision table editor is not significantly different from the static decision table editor in the Aion IDE. The dynamic decision table editor can use only domain interface members; thus it cannot specify an implementation of a Condition or Action. For more information about the difference between specifying an implementation for a condition or action of a decision table at edit time and using a domain interface member, see the sections [Adding and Modifying Conditions](#) (see page 131) and [Adding and Modifying Actions](#) (see page 133).

## Dynamic Decision Table Runtime Considerations

For general information pertaining to the runtime considerations pertinent to any dynamic rule, see Dynamic Rule Runtime Considerations.

The name of the runtime facility provided by DynRDLib to support dynamic decision tables is DecTableRuntime. The posting method provided by DecTableRuntime is PostDTbl( ). The following example posts a dynamic decision table (an instance of DecTableRuntime):

```
INFER
    var hRule is integer
    hRule = pDT.PostDTbl( )
        // pDT is &DecTableRunTime
        // Perform error processing if hRule is NULL
    // Invoke forward or backwardchaining
END
```

The relevant signature of the PostDTbl( ) method is:

```
PostDTbl(BindList is list of &_Object = NULL, ...) : integer
```

where BindList is a list of pointers to binding instances. For more information about binding instances, see [Instance Binding](#) (see page 62). See the DPDRules example for an example of binding instances at posting time. Note particularly the way in which a binding is provided for references to domain interface members implemented in the Decision class:

```
hRule = pDT.PostDTbl(list(pDecision))
```

**Note:** BindList does not have to be in any particular order; Aion will figure out which pointers provide bindings for which domain interface member references in the dynamic decision table.

An alternative to using BindList involving manual binding of instances is discussed in Dynamic Rule Runtime Considerations. Manual binding requires invoking the appropriate GetDTblXXX( ) method to obtain a pointer to the domain interface member of a dynamic rule that requires a binding instance. In the following, “pos” is an input argument indicating the position of a domain interface member within the Condition or Action of a dynamic rule and “pDI” is a pointer to that domain interface member.

```
GetDTblCondition(in pos is Integer, out pDI is &DIMember, out TestValues is String)
```

```
GetDTblCheckmarkAction(in pos is Integer, out pDI is &DIMember, out Selector is list of Boolean)
```

```
GetDTblValueAction(in pos is Integer, out pDI is &DIMember, out Selectors is String)
```

```
GetDTblActionAction(in pos is Integer, out Description is String, out Selectors is list of &DIMember).
```



Output arguments other than pDI may be ignored for instance binding except in the case of GetDTblActionAction( ). The second output argument provides the list of pointers to the domain interface members that make-up the possible actions invoked by this multiple action Action. In the case of multiple actions, the references to each instance domain interface member in the action list must have a proper binding.

Manual instance binding for dynamic decision tables is illustrated in the DPDRule example. In the following, "pDT" is a pointer to a dynamic decision table instance, "2" identifies the second Condition in that dynamic decision table, "pDI" is a pointer to a DIMember instance, and "Testvalues" is a string. We obtain pDI with the statement:

```
pDT.GetDTblCondition(2, pDI, TestValues)
```

Having obtained the pointer to a domain interface member, one can then bind the reference either by using a pointer to the intended instance, SetBindingByPointer( ), or by the name of the intended instance, SetBindingByName( ). Here we bind a particular instance of the class Duck (the "Pet Duck" instance), by means of the SetBindingByName( ) method:

```
if pDI.SetBindingByName("Pet Duck") = FALSE then  
    // Error processing.
```

The MDDRules example illustrates an approach that avoids having to know ahead of time the exact position of the domain interface member for whose reference in the rule you wish to provide a binding. By using the meta-programming capabilities of the dynamic rule libraries, it is possible to identify the position of a particular domain interface member in a dynamic decision table at run time based upon the name of the domain interface member.

**Note:** For more information about the methods discussed in this section, see the "DynRLib" chapter of the *online Reference*.



# Chapter 7: Truth Maintenance

---

This chapter describes the Aion truth maintenance inferencing feature and how to use it to do non-monotonic (or what-if) reasoning. For the sake of brevity, this chapter often refers to truth maintenance by the acronym TM.

The kind of inferencing we have discussed so far is called monotonic reasoning, which assumes that attribute values change and rules fire or fail at most once during the rules inferencing process. Monotonic reasoning is great for applications that seek to arrive at a particular answer or establish the consequences of certain known data.

There are occasions, however, where you would like to see how conditionally changing a particular piece of data could alter the result. Consider a loan processing application, in which modifying just one input, such as the amount of outstanding debt, could alter the bank's decision to approve or reject the loan.

Such inferencing, in which attribute values can change value and rules can refire multiple times during the course of current rules processing, is called non-monotonic, or sometimes what-if, reasoning. Aion provides non-monotonic reasoning through the use of the truth maintenance feature.

With truth maintenance, you make a conditional assignment to an attribute. If you subsequently confirm or retract that assignment, the inference engine confirms or retracts the consequences of that assignment.

This section contains the following topics:

[Truth Maintenance Operations and Terminology](#) (see page 107)

## Truth Maintenance Operations and Terminology

There are three truth maintenance operations in Aion:

Operation	Syntax
TM-Assignment	<attrib> ?= <value>
TM-Retract	retract ( <attribpointer> )
TM-Confirmation	confirm ( <attribpointer> )

These operations affect the retractability of the specified attribute. When an attribute is non-retractable, its assigned value (if any) may be changed-but not retracted or undone. All attributes are initially non-retractable. When an attribute is retractable, its assigned value is tentative-that is, the value may be discarded in favor of the attribute's previous non-retractable value.

Over time, an attribute may alternate between a retractable and non-retractable state.

- TM-assignment assigns a tentative value to an attribute and sets the attribute to the retractable state.
- TM-retraction discards the tentative value, restores the non-retractable value, and sets the attribute back to the non-retractable state.
- TM-confirmation makes the tentative value permanent, and then sets the attribute back to the non-retractable state.

## Operational Context

Truth maintenance operations can execute anywhere within an INFER block except as part of a rule's premise evaluation. The INFER block must specify the history option or inherit it from an ascendant INFER block.

The Aion interpreter enforces these restrictions via runtime error messages. When outside of this context, generated code ignores TM-retraction and TM-confirmation and treats TM-assignment as a non-truth-maintenance assignment.

Truth maintenance operations may be usefully employed for both forward and backward chaining. TM-assignments within one INFER block may be TM-retracted and/or TM-confirmed within nested INFER blocks of a lower scope. Keep in mind the following restrictions:

- Truth maintenance operations must be applied to either class attributes or instance attributes-but not to local variables of a method.
- Truth maintenance operations may not be applied to list, array, or binary attributes.
- Truth maintenance operations may not be specified within a pattern-matching (IfMatch) rule or in a WhenMatch demon.

## Tickling Demon and Pattern-Matching Rules

The following table summarizes how truth maintenance affects pattern-matching rules and demons. If the truth maintenance operation can tickle the rule-type in question, Yes appears in the corresponding cell. For demons, “tickling” means firing. For ifmatch rules, it means readying.

	<b>ifmatch Rules</b>	<b>when Demons</b>	<b>whenmatch Demons</b>
TM-Assignment	No	Yes	No
TM-Confirmation	Yes	Yes	Yes
TM-Retractation	No	No	No

## Available Runtime Information

The `_Engine` class defines numerous class methods, such as `EngineGetHistory()`, for accessing runtime information concerning inferencing progress and results. Although available for non-truth-maintenance scenarios, this information is particularly useful in truth maintenance scenarios for explaining or justifying inferencing results. The `_Engine` class methods are described in the “SysLib” chapter of the *online Reference*.

## TM-Assignment

TM-assignment assigns a tentative value to an attribute and sets the attribute to the retractable state. For example:

Make a regular assignment in a rule.	<code>Attrib1 = 5</code>
In a subsequent rule, make a TM-assignment.	<code>Attrib1 ?= 32</code>

In the above table, `Attrib1` becomes a retractable attribute. The attribute's value may later be TM-retracted back to 5 or TM-confirmed as 32. Prior to a TM-assignment, an attribute may be non-retractable or it may already be retractable.

## Implicit TM-Assignments

If an ifrule fires while it is dependent on a retractable attribute, all non-TM-assignments executed as part of that rule's action are implicit TM-assignments-even though they are coded as non-TM-assignments. For example, consider what happens if bAttrib1 is currently a retractable boolean attribute and the following rule fires:

```
rule "Illustration of Implicit TMAssignment"  
ifrule bAttrib1  
then  
    AttribA = 23 // implicit TMAssignment  
    AttribB = 99 // implicit TMAssignment  
end
```

As a result of the rule's action, AttribA and AttribB are now retractable attributes. When an attribute receives an implicit TM-assignment, it is subsequently TM-retractable or TM-confirmable-just as though it had received an explicit TM-assignment.

## Ignored TM-Assignments

The inference engine ignores a TM-Assignment that would end up not changing the current value of the attribute, as in the following circumstances.

### Successive TM-Assignments with Same Value

The engine ignores a TM-assignment to a retractable attribute if the assignment would end up not changing the attribute value. For example, if Attrib1 is currently TM-assigned a value of 5, and, during subsequent rule processing, it is again TM-assigned a value of 5 (explicitly or implicitly), the engine ignores the second TM-assignment.

In the case of an explicit TM-assignment followed by an implicit TM-assignment of the same value, the engine ignores the second TM-assignment but the engine also "upgrades" the earlier TM-assignment to an implicit one. The rationale here is that implicit TM-assignments carry with them a somewhat higher confidence level than do explicit TM-assignments. Therefore, the engine should therefore upgrade the earlier explicit TM-assignment to reflect this higher confidence level.

The engine does not ignore successive TM-assignments of different values to an attribute. For more information about TM-retractions, see TM-Retraction.

### Non-TM-Assignment Followed by TM-Assignment with Same Value

The engine also ignores a TM-assignment to a non-retractable attribute if the assignment would not change the attribute value. For example, if `Attrib1` is currently non-TM-assigned a value of six, and, during subsequent rule processing, it is TM-assigned a value of six (explicitly or implicitly), the engine ignores the second assignment.

### TM-Retraction

TM-retraction discards a retractable attribute's tentative value and restores the attribute to its previous non-retractable state and value. Consider the following, which restores `Attrib1` to the value 5 and makes the attribute non-retractable. If, prior to TM-assignment, the attribute's value was undefined, it is restored as an undefined value:

Make a regular assignment in a rule.	<code>Attrib1 = 5</code>
In a subsequent rule, make a TM-assignment.	<code>Attrib1 ?= 32</code>
In a third rule, you can retract the TM-assignment. The value returns to 5.	<code>retract (-&gt;Attrib1)</code>

### Ignored TM-Retractions

The engine ignores attempted TM-retractions of a non-retractable attribute.

### Implicit TM-Retractions

The engine makes implicit TM-retractions in the following circumstances.

### Successive TM-Assignments with Different Values

Successive TM-assignments of different values to an attribute are interspersed with implicit TM-retractions. Thus, the following sequence implicitly TM-retracts `Attrib1` back to 11 before TM-assigning it the value 13:

Make a regular assignment in a rule.	<code>Attrib1 = 11</code>
In a subsequent rule, make a TM-assignment.	<code>Attrib1 ?= 12</code>
In another rule, make a second TM-assignment.	<code>Attrib1 ?= 13</code>
The engine makes an implicit TM-retraction first.	
Make a TM-retraction. The value returns to 11, not 12.	<code>retract (-&gt;Attrib1)</code>

### TMAssignment Followed by non-TM Assignment with Different Value

A TM-assignment followed by a non-TM-assignment to the same target attribute but with a different value will be interspersed with an implicit TM-retraction. So, for example, this example implicitly TM-retracts `Attrib1` back to the value 11 before non-TM assigning the value 13 to it.

Make a regular assignment in a rule.	<code>Attrib1 = 11</code>
In a subsequent rule, make a TM-assignment.	<code>Attrib1 ?= 12</code>
In another rule, make a regular assignment.	<code>Attrib1 = 13</code>
The engine makes an implicit TM-retraction first.	

A TM-assignment followed by a non-TM-assignment to the same target attribute and with the same value does not cause an implicit TM-retraction. For more information about TM-confirmations, see TM-Confirmation.

Operations such as the `_Engine` class method `GoalMakeUnknown(->Attrib)` are not considered assignments (TM- or otherwise) and therefore do not trigger implicit truth maintenance operations.

### Side-Effects of TM-Retractions

When TM-retracting an attribute, the engine may also retract associated attributes and reset the states of associated rules.

### Algorithm

The engine first restores the attribute back to its previous state and value. The engine next identifies rules that have fired or failed subsequent to the attribute's last TM-assignment. For each such rule:

<b>Fired or failed rule</b>	<b>Fired rule only</b>
Engine resets the rule's state to <code>READY</code> , if the firing or failing was dependent on the retracted attribute.	For each attribute that was TM-assigned (explicitly or implicitly) via the rule's action, the engine recursively applies this algorithm to the attribute.

The affected rules may reside within the current `INFER` block or any `INFER` block of higher scope.



## TM-Confirmation

TM-confirmation restores a retractable attribute back to a non-retractable state; but retains the attribute's tentative value as the attribute's value for that state.

Thus, if `Attrib1` was TM-assigned the value 13, the following statement would make `Attrib1` non-retractable yet retain the value 13:

```
confirm(->Attrib1)
```

## Ignored TM-Confirmations

The engine ignores attempted TM-confirmations of a non-retractable attribute.

## Implicit TM-Confirmations

The engine makes implicit TM-confirmations in the following circumstances.

### TM-Assignment Followed by non-TM Assignment with Same Value

A TM-assignment followed by a non-TM assignment to the same target attribute but with the same value will be interspersed with an implicit TM-confirmation. Consider the following example, in which you implicitly TM-confirm `Attrib1` with the value of 11):

Make a TM-assignment in a rule.	<code>Attrib1 ?= 11</code>
In a subsequent rule, make a regular assignment. The engine implicitly TM-confirms the value 11 first	<code>Attrib1 = 11</code>

## Automatic TM-Confirmation at Inference Block Termination

When an inference block terminates execution, the engine examines all attributes TM-assigned (explicitly or implicitly) via the actions of the block's fired rules. For each such attribute, if the attribute is still retractable, the engine implicitly TM-confirms that attribute.

## Side Effects of TM-Confirmations

When TM-confirming an attribute, the engine may also confirm associated attributes and reset the states of associated rules. It does so in the following way.

## Algorithm

The engine first confirms the attribute with its current value. It then attends to rules with the retractable attribute in their premise, treating pending rules differently from fired rules.

## Pended Rules

The engine identifies all pended rules which are dependent on the confirmed attribute. For each such rule, the engine resets the rule's state to READY. The affected rules may reside within the current INFER block or any ascendant INFER block.

## Fired Rules

The engine identifies rules that have fired subsequent to the attribute's last TM-assignment and that were:

- Dependent on the confirmed attribute as a retractable attribute; but
- Not dependent on any other retractable attributes.

For each such rule, the engine does the following: for each attribute implicitly TMAssigned via the rule's action, the Engine recursively applies the algorithm to the attribute.

### **Note:**

- As a side effect, the Engine does not confirm attributes that were explicitly TM-assigned by rule actions.
- The affected rules may reside within the current INFER block or any INFER block of higher scope.

# Chapter 8: Maintaining the Dynamic Rulebase

---

Aion provides two principal facilities for maintaining persistent dynamic rules. (For more information about dynamic rules, see Dynamic Rules in the User Guide.) These facilities are:

- **Dynamic Rulebase Administrator.** The Dynamic Rulebase Administrator provides facilities for administering the rulebase. Rulebase administration includes controlling user access to rulebase management functions and keeping the rulebase in sync with the Aion applications that use dynamic rules. In particular, the Dynamic Rulebase Administrator enables a domain interface of an Aion application to be imported into a rulebase. (For a definition of domain interface, see the Domain Interfaces section in the User Guide.) The Dynamic Rulebase Administrator also provides facilities to keep the rulebase synchronized with the changes made to the domain interface in the Aion application. For more information about the Dynamic Rulebase Administrator, see the [Dynamic Rulebase Administrator](#) (see page 116).
- **Dynamic Rule Manager.** The Dynamic Rule Manager enables a domain expert to create and maintain dynamic rules outside the Aion development environment and to access the Dynamic Rule Repository. The Dynamic Rule Manager provides dynamic rule editors that are similar to the rule editors in the Aion development environment. It also allows access to the Dynamic Rule Repository for saving and viewing rule history. For more information about the Dynamic Rule Manager, see the [Dynamic Rule Manager](#) (see page 126).

The Dynamic Rule Manager provides a user interface into the Aion rulebase. However, in a robust business application, Aion users will want to augment the rulebase with customized fields or even additional tables, and therefore will require extensions to the Dynamic Rule Manager. These extensions will take the form of utilities and user-written application programs to manage the new rulebase features. Aion also provides a library (DynRELib) that includes facilities for updating the core dynamic rule structures in the Aion rulebase (if you choose to write your own rule editor). For a description of the structure of the default Aion rulebase, see the appendix “Rulebase Structure”.

**Note:** For more information about DynRELib, see the “DynRELib” chapter in the *online Reference*.

This section contains the following topics:

[Dynamic Rulebase Administrator](#) (see page 116)  
[Dynamic Rule Manager](#) (see page 126)  
[Dynamic Decision Table Editor](#) (see page 129)

## Dynamic Rulebase Administrator

The Dynamic Rulebase Administrator provides the following functionality:

- The Dynamic Rulebase Administrator allows user access permissions to be specified at the user/domain level. User access permissions define what functionality of the Dynamic Rule Manager a business user is entitled to use. For more information about specifying user access permissions, see Establishing User Access Permissions in the “Dynamic Rule Management” chapter of the User Guide.
- The Dynamic Rulebase Administrator initializes an empty rulebase with the domain interface members defined in an Aion application. See Importing a Domain Interface in this chapter.

**Note:** The rulebase for an application can be created under any relational Database Manager supported by Aion's DataLib, such as Oracle, Sybase, DB2, or any ODBC supported database. Aion provides a default empty rulebase called Rulebase.mdb in the \Rulebase subdirectory of the Aion, for Windows, install directory for Windows. For more information about the default rulebase, see [The Default and Empty Rulebases](#) (see page 117).

- The Dynamic Rulebase Administrator allows the Aion application developer to organize the domain interface members of an Aion knowledge base into domains, which may be identified during system analysis. For more information about domains and their role in system development, see the sections Domain Interfaces and especially Role of the Domain Interface in System Development in the User Guide.
- The Dynamic Rulebase Administrator provides facilities for synchronizing the domain interface members in the rulebase with the domain interface members defined in the Aion application as the Aion application itself changes. See [Dynamic Rulebase Scenarios](#) (see page 124).
- The Dynamic Rulebase Administrator allows the domain's definitions within the rulebase to be cloned when a clone operation is requested. When menu item Clone is selected, a pop up dialog box will appear and end users can enter the new domain name.
- The Dynamic Rulebase Administrator provides option to save all the decision tables in the rulebase as HTML.

- The Dynamic Rulebase Administrator provides the option to save the cross reference information within that domain when a domain is selected, otherwise the cross reference information within the entire rulebase will be saved.

**Note:** When Save Term References as HTML is selected, a popup dialog box appears and you can enter a term pattern or a term name. This option is also available for Save Rule References as HTML. The following Wild card search is supported:

\*

Represents any string of zero or more characters

**Example:** Get\* — Finds all terms started with "get"

?

Lists any single character

**Example:** G?t — Find all terms started with "g", ended with "t", and have one and only one character between g and t.

\

Lists escape character

**Example:** \*\? — Find all terms ended with a '?'

The search is case-insensitive.

**Note:** The Dynamic Rulebase Administrator is itself an Aion application developed with COBSLib.

## The Default and Empty Rulebases

During installation, Aion installs two rulebases in the \Rulebase subdirectory. Both rulebases are in MS Access format:

- Default rulebase (Rulebase.mdb)
- Empty rulebase (EmptyRulebase.mdb)

The default rulebase is Rulebase.mdb. The characteristics of the default rulebase are:

1. At startup it contains no rules.
2. The ODBC Source, "Aion Rulebase", points to this rulebase. A connection to this Source is provided by DynRDLib.
3. It allows read/write access.
4. It is required by the Dynamic Rulebase Administrator for initial startup.

You can use the default rulebase to begin your own rulebase development. If you use this rulebase for your own development project, you may want to consider whether you should move the rulebase to a development directory and create your own ODBC Source pointing to it.

**To connect to an ODBC Source for the default rulebase (rulebase.mdb):**


1. Open the Windows Control Panel and double-click the ODBC Data Sources icon to open the ODBC Data Source Administrator
2. Click the Add button. This displays the Create New Data Source dialog.
3. Scroll down the list box and highlight the desired driver; for example, Microsoft Access Driver (\*.mdb).
4. Click the Finish button. This displays the ODBC Microsoft Access Setup.
5. Enter the Data Source Name required by your application in the Data Source Name Field. To use the Aion-supplied default ODBC driver, use the data source name "Aion Rulebase." Otherwise, enter the name that you have chosen for the Data Source in your application.
6. Optionally, enter a description of the Source (the database being accessed).
7. Click the Select button under Database.
8. Navigate the directories to find the rulebase to connect. The Aion default rulebase is rulebase.mdb. The default location of this rulebase is the \Rulebase subdirectory of the Aion install directory.
9. Highlight the name of the target rulebase in the Data Name list box. Click OK.
10. Click OK on the ODBC Microsoft Access Setup
11. Click OK on the ODBC Data Source Administrator.

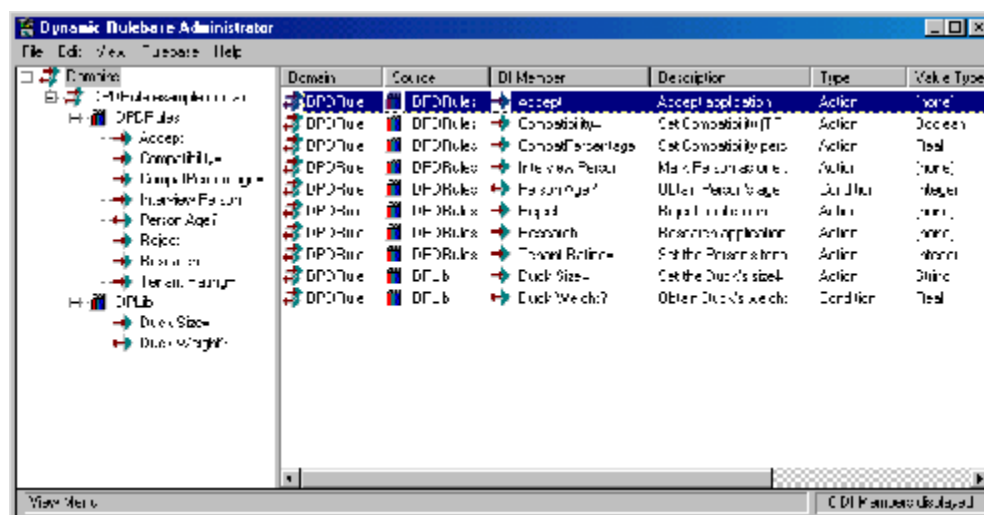
The other rulebase is the empty rulebase-EmptyRulebase.mdb. It also contains no rules. However, the empty rulebase allows read only access. It is a permanent backup of the default rulebase. To use this rulebase (that is, to restore the default rulebase), copy it to the install \rulebase directory, rename it, and change its properties to allow read/write access.

The \rulebase directory is located on the first level below the root which is created by installing Dynamic Rule Manager(DRM).

Aion also provides SQL scripts for creating rulebases, SQLScript.sql, in the \Rulebase subdirectory. This text file contains SQL for creating a rulebase under MS Access, Sybase, Oracle, and DB2.

## Importing a Domain Interface

To open the Dynamic Rulebase Administrator, click on the Dynamic Rulebase Administrator icon  that is included in the Aion program group.



The left panel is called the Rulebase Workspace. It consists of a tree list presenting (in hierarchical order):

- The rulebase (the word “Domains”; this is the common root of all trees)
- The domains within in this rulebase.
- The Aion applications, or Sources, from which the domain interface members within this domain have been imported.
- The domain interface members that have been imported from this Source.

The right panel is called the DI Member Detail View. It consists of a listing of the domain interface members for the current highlighted item in the Rulebase Workspace and shows:

- The Domain to which this member belongs and the Source (Aion application) from which it was imported.
- The domain interface member name (label) and its description.
- The domain interface member type: Condition or Action. The type is also indicated by the icon to the left of the domain interface member name, if View, Detail View Options, Display SubImages is activated.
- The domain interface member value type. If the domain interface member is a Condition, then the value type represents the value type returned by the underlying (Get) accessor method. If the domain interface member is an Action, then the value type represents the type of input argument, if one is required, to the underlying implementing method.

- The date and time that the Aion application (Source) containing this domain interface member was last saved prior to this member being synchronized with the rulebase.

**Note:** This date does not necessarily represent the date that this domain interface member was changed in the application. For example, the domain interface member may be changed on January 1 but the application might not be synchronized with the rulebase until January 8. Between January 1 and January 8 the application would have probably been saved several times with other changes unrelated to this domain interface member. The Last Update would be the date on which the application was last saved (perhaps January 7) before its synchronization on January 8.

- The most recent activity (imported or updated) and the date it was performed on this domain interface member in the rulebase.

The View menu provides controls for displaying the contents of this panel.

Importing the members of a domain interface involves the following steps:

1. Connecting to a Rulebase and Opening an Aion application.
2. Establishing a domain in which to import domain interface members from the application.
3. Synchronizing the domain interface members into the domain.

### Connecting to a Rulebase and Opening an Aion Application

Select a rulebase from the File, Settings menu option to open the Dynamic Rule Settings dialog. In the Dynamic Rule Settings dialog:

1. Enter your name in the User Name field

Specify the database access information of the rulebase with which you wish to connect:

- Select the Interface. Interface is either ODBC or a native database interface to a database manager
- Specify the Database, that is, the name of the rulebase. In case of ODBC, select the ODBC Source.
- Complete Server, User ID, and Password as appropriate for the database being accessed.



The connection can be tested before attempting to actually open a domain in the rulebase by clicking the Test button.

The other principal driver of the Dynamic Rulebase Administrator is an Aion knowledge base. An Aion knowledge base must be opened in the Administrator in order for the Administrator to access the domain interface that has been defined in that application. To open an Aion application in the Administrator:

2. Choose File, Open Source and select the desired application from the directory box.
3. Click Open.

The name of the Dynamic Rulebase Administrator window will change to reflect the name of the opened source (application).

Editing (deleting and renaming) is performed by highlighting an item in the Rulebase Workspace. You may delete or rename domains and sources, but domain interface members can only be deleted.

**Important!** Renaming sources should be performed only if the original Aion application (library) has been renamed. Mistakenly renaming a source to a non-existent application could prevent all elements of the domains using that source from being accessible by Aion.

Highlighting and right clicking on a domain interface member in the DI Member Detail View is equivalent to highlighting and right clicking the name of domain interface member in the Rulebase Workspace. You may use either the Edit menu from the main menu bar or right click on the highlighted item and select the Edit option from the pop-up menu. Options on the Edit menu may be grayed out if they are not available at the level of the highlighted item. Editing may be performed without an Aion application being opened.

Administering the rulebase (creating new domains and importing and synchronizing domain interface members) is performed by highlighting an item in the Rulebase Workspace or in the Detail View and selecting the desired option from the Rulebase menu. Alternatively, the Rulebase options may be selected from the pop-up menu by right-clicking a highlighted item.

## Establishing a Domain

Whenever the rulebase level (the root node of the tree) is the selected item in the Rulebase Workspace, the Administrator automatically selects the last domain to which domain interface members have been imported as a default target domain.

**Note:** The rulebase level is selected by default when the rulebase is opened.

To establish a domain other than the default, highlight the name of the domain or any element below the domain (a Source or domain interface member belonging to that domain) in the Rulebase Workspace. You may also select an item in the Detail View to establish a domain and source.

To establish a new domain, choose New Domain from the Rulebase menu. The New Rulebase Domain dialog will appear. Enter the name of the new domain and click OK. The name of the domain will be added to the Rulebase Workspace.

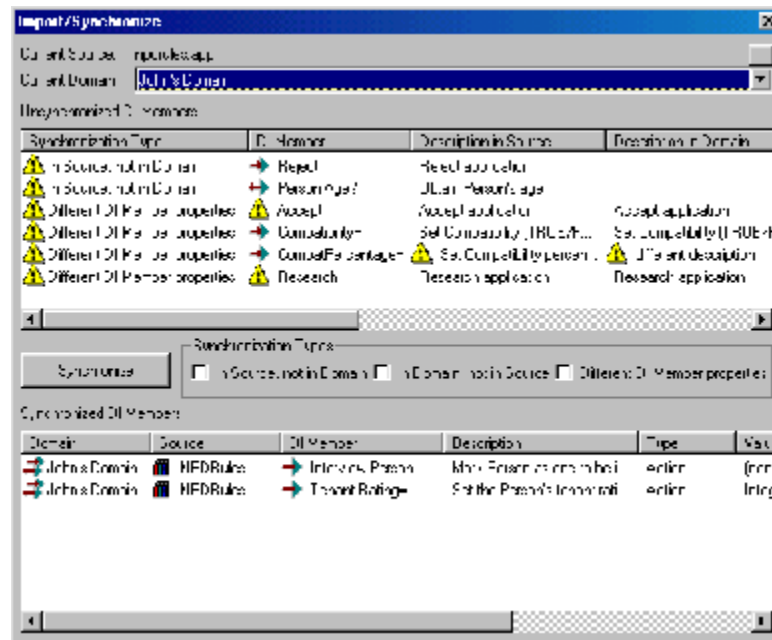
### Synchronizing Domain Interface Members

An application may export its domain interface members to several different domains. Indeed, it is possible to export the same domain interface member to multiple rulebase domains. Thus, rules in different domains can share the same domain interface members (share the same business vocabulary).

Similarly, a domain may contain domain interface members imported from different Aion applications.

Once an Aion application is opened in the Administrator and a domain established, domain interface members must be imported from the application and the domain must be synchronized with changes made to the application since the last import or synchronization.

To access the Import/Synchronize dialog, choose the Import/Synchronize option from the Rulebase menu.



The Import/Synchronize dialog provides two lists:

- Unsynchronized DI Members-A list of domain interface members that do not currently match between the application (Source) and rulebase (Domain).
- Synchronized DI Members-A list of domain interface members in the domain that currently do correspond to domain interface members in the Source.

The list of unsynchronized domain interface members specifies the reason that a match cannot be found between the Source and Domain (Synchronization Type column). To synchronize an item from the upper list, highlight the item and click the Synchronize button. Unless the item is being deleted from the domain, the item will move to the lower list. It is possible to highlight all items in a particular category by selecting one or more of the Synchronization Type checkboxes. For an explanation of these types, see [Comparing Domain Interface Members in the Application and Rulebase](#) (see page 124). Clicking the Synchronize button synchronizes all items that are highlighted.

## Importing Domain Interface Members

### **To import domain interface members:**

1. Open a rulebase and an Aion application.
2. Highlight a domain to which domain interface members will be imported in the Rulebase Workspace,

OR

### **To create a new domain**

- a. Choose Rulebase, New Domain from the Administrator menu, or right-click in the Rulebase Workspace and choose Rulebase, New Domain for the pop-up menu.
  - b. When the New Rulebase Domain dialog appears, enter the name of the Domain and click OK.
3. Choose Rulebase, Import/Synchronize from the Administrator menu,  
OR  
Right click on the highlighted domain and select Rulebase, Import/Synchronize from the pop-up menu.  
The Import/Synchronize dialog displays.
  4. Check the "In Source, not in Domain" Synchronization Style,  
OR  
Highlight the specific items in the Unsynchronized DI Member list that are In Source, not in Domain that you wish to import.
  5. Click the Synchronize button.

## Comparing Domain Interface Members in the Application and Rulebase

The checkboxes in the Synchronize Types group determine what groups of application (Source) or rulebase domain interface members are selected for synchronization in the Unsynchronized DI Members list.

- In Source, not in Domain-Shows the domain interface members in the application that have not yet been imported into the selected domain of the rulebase.
- In Domain, not in Source-Shows the domain interface members in the selected domain of the rulebase that do not have corresponding domain interface members in the application.
- Different DI member properties-Shows the domain interface members in the Source and selected domain with the same names but whose properties are different.

Properties of a domain interface member include:

- Type (whether it is a Condition or Action domain interface member)
- Input argument (whether it has an input argument and the type of that argument)
- Return type (the data type of the return value from the method implementing the domain interface member)
- Description

**Note:** A caution icon in the Import/Synchronize dialog indicate properties that do not agree for a domain interface member.

For more information about synchronizing the application domain interface and the rulebase, see [Dynamic Rulebase Scenarios](#) (see page 124).

## Dynamic Rulebase Scenarios

As an Aion application is developed, features of the domain interface will undoubtedly change. Domain interface members may be added or deleted; properties and even the names of the existing domain interface members may be altered. The Dynamic Rulebase Administrator, designed to keep the rulebase synchronized with later modifications made to Aion applications, facilitates making corresponding changes in the rulebase.

For step-by-step procedures used in these scenarios, see [Importing Domain Interface Members](#) (see page 119).

**Scenario 1: Add a new domain interface member to the domain interface**

Open the rulebase and application in the Dynamic Rulebase Administrator and, in the Rulebase Workspace, select the domain to which the new member is to be imported. If the member is to be imported to a newly constructed domain, it does not matter what domain is current. Select Rulebase, New Domain; then Rulebase, Import/Synchronize.

Highlight the name of the new domain interface member in the Unsynchronized DI Members list or select the "In Source, not in Domain" Synchronization Type to select all new domain interface members. Click the Synchronize button to import the domain interface member into the rulebase. You have a choice to add the domain interface members to just the current domain or to all domains in the rulebase by checking the Apply to all Domains checkbox.

**Scenario 2: Change the properties (other than the name) of an existing domain interface member**

Open the rulebase and application in the Dynamic Rulebase Administrator and, in the Rulebase Workspace, select the domain to which the member that has been changed belongs. Select Rulebase, Import/Synchronize.

Highlight the name of the changed domain interface member in the Unsynchronized DI Members list or select the "Different DI member properties" Synchronization Type to select all changed domain interface members. Click the Synchronize button. You have a choice to applying the changes to just domain interface members in the current domain or in all domains in the rulebase by checking the Apply to all Domains checkbox.

**Scenario 3: Change the name (label) of a domain interface member that has already been imported into the rulebase**

After opening the rulebase in the Dynamic Rulebase Administrator (you do not have to open the application), right click on the domain interface for which you want to change its name in the Rulebase Workspace or DI Member Detail View. Select Edit, Rename from the pop-up menu. Enter the new name of the domain interface member in the Rename DI Member dialog. Click Ok. You have a choice to rename just the selected domain interface member or all the domain interface members with that name in the rulebase by checking the Apply to all Domains checkbox on the Rename DI Member dialog.

**Scenario 4: Delete an imported domain interface member from the application**

After opening the rulebase in the Dynamic Rulebase Administrator (you do not have to open the application), right click on the domain interface member that you wish to delete in the Rulebase Workspace or DI Member Detail View. Select Edit, Delete from the pop-up menu. Click Ok in the Delete DI Member dialog. You have a choice to delete just the selected domain interface member or all the domain interface members with that name in the rulebase by checking the Apply to all Domains checkbox on the Delete DI Member dialog.

If the domain interface member has already been deleted in the application, you may also select a domain that contains this member and go to the Import/Synchronize dialog. Select the "In Domain, not in Source" Synchronization Type. Click the Synchronize button. The domain interface member will be deleted from the domain. Again, you have the choice to delete the domain interface member just from the current domain or from all domains in the rule by checking the Apply to All Domains checkbox.

**Note:** If the domain interface member is already used in rules, it cannot be deleted. A list of the rules using that domain interface member will be displayed. The Dynamic Rule Manager must be used either to delete the rules or to change them to use another domain interface member. The Dynamic Rulebase Administrator can then be used to delete the domain interface member from the domain.

**Important!** Attempting to inference using a dynamic rule whose domain interface members have been deleted from the application will cause a runtime error.

### Scenario 5: Move an imported domain interface member from one application to another

Open the rulebase and application in the Dynamic Rulebase Administrator. In this scenario synchronizing the rulebase with the application requires two steps. The first step is to delete the domain interface from the domain in which it resided. See [Scenario 4: Delete an imported domain interface member from the application](#) (see page 125).

The second step is to select the Source in which this member now belongs. This may be done by clicking the browse (ellipsis) button on the Import/Synchronize dialog and selecting the desired Source. Highlight the name of the domain interface member in the Unsynchronized DI Members list or select the "In Source, not in Domain" Synchronization Type to select all new domain interface members with respect to that domain. Click the Synchronize button.

## Dynamic Rule Manager

Once the Domain Interface has been imported into the rulebase, the domain expert can use the Dynamic Rule Manager to begin populating a Domain in the rulebase with dynamic rules. The Dynamic Rule Manager organizes the rules into Domains. For information about the concept of Domain, see the "System Considerations for Supporting Business Rules" chapter of the *User Guide*.

**To create dynamic rules, the domain expert**

1. Selects a domain in the Dynamic Rule Manager to which the rule will belong.
2. Selects Domain Interface members that have been previously imported into the domain for constructing the Conditions and Actions for dynamic rules. For more information about populating a domain with Domain Interface members, see the Dynamic Rulebase Administrator in this chapter. Domain Interface members are referenced through domain interface labels.
3. Specifies the Condition test values and Action values that will define the rule.
4. Constructs the dynamic rule and saves it to the rulebase.

Optionally, the domain expert may also check the new rule into the Dynamic Rule Repository. For more information about the accessing the Dynamic Rule Repository, see [Accessing the Dynamic Rule Repository](#) (see page 139).

## Selecting a Rulebase and Opening a Domain

To access the Dynamic Rule Manager, click the Dynamic Rule Manager icon provided in the Aion Program Group.

Select a rulebase by using the File, Settings menu option to open the Dynamic Rule Settings dialog.

**To open a rulebase**

1. In the Dynamic Rule Settings dialog, enter your name in the User Name field.
2. Specify the database access information of the rulebase with which you wish to connect:
  - a. Select the Interface. The Interface is either ODBC or a native database interface to a database manager
  - b. Specify the Database, that is, the name of the rulebase. In case of ODBC, select the ODBC Source.
  - c. Complete Server, User ID, and Password as appropriate for the database being accessed.

The connection can be tested before attempting to actually open a domain in the rulebase by clicking the Test pushbutton.

**To open a domain using the Dynamic Rule Manager:**

1. Choose File, Settings to verify that the Dynamic Rule Manager is pointing to the correct rulebase. To open a different rulebase, see To open a rulebase: earlier in this chapter.
2. Choose File, Open Domain. The Select Rule Domain dialog appears.  
Select the desired domain from the Domain drop-down list.

3. Optionally, check the Use as Default Domain checkbox.

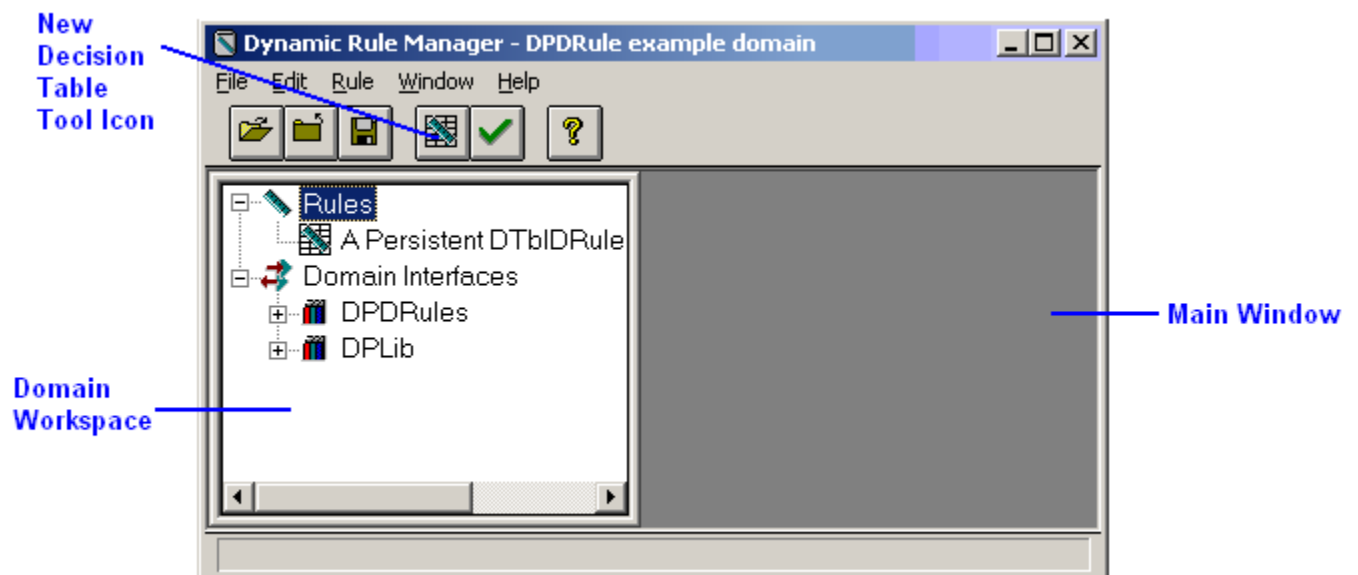
**Note:** A domain opens automatically if you previously checked Use as Default Domain. If the desired domain does not open by default, the Dynamic Rule Manager may not be pointing to the correct rulebase.

4. Click OK.

To open another domain in the same rulebase, choose File, Open Domain.

**Note:** A Domain is automatically opened if you have checked a domain as the default domain in the Select Rule Domain dialog. If the desired default domain is not opened automatically, it may be because the manager is not pointing to the desired rulebase.

The Dynamic Rule Manager appears as follows when a domain is opened:





## Creating and Maintaining Dynamic Rules

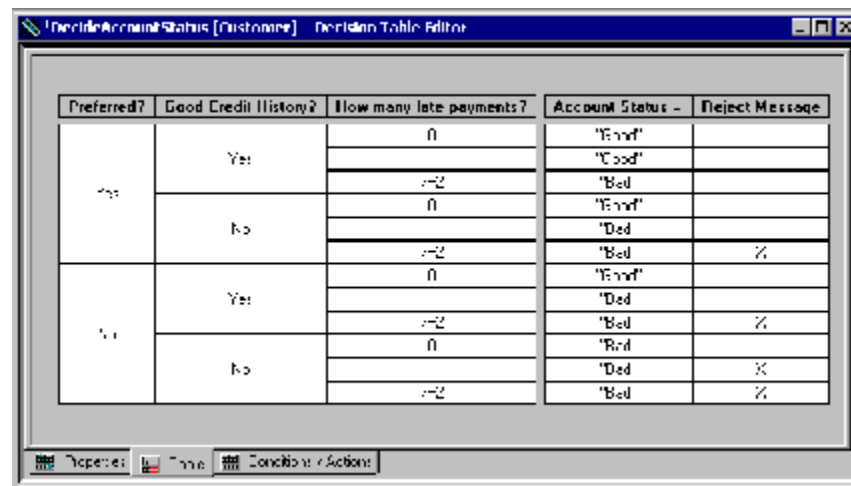
The Dynamic Rule Manager is used to create, view, or maintain dynamic rules:

- To create a new dynamic rule, select Rule, New from the Dynamic Rule Manager menu, or click the appropriate new rule icon on the Dynamic Rule Manager toolbar.
- To view or maintain an existing dynamic rule, double click the rule in Domain Workspace or highlight the rule and select Rule, Open for the Dynamic Rule Manager menu.

**Note:** Currently, Aion supports creating and maintaining only dynamic decision tables.

## Dynamic Decision Table Editor

The Dynamic Decision Table Editor provides much the same functionality as the static decision table editor in the Aion BRE development environment.



Decision tables are displayed in the Dynamic Rule Manager with the icon shown. This icon serves to identify decision tables in the Domain Workspace as well as representing the New Decision Table icon on the toolbar.

## Creating a New Dynamic Decision Table

### To create a new decision table

1. Choose Rule, New, Decision Table... to display the New Rule dialog.
2. Enter the decision table name and click OK to display the Dynamic Decision Table Editor in the Output Window.
3. The editor opens to a blank Table page. The typical procedure would be to click the Conditions/Actions tab to begin defining the conditions and actions that make up the dynamic rule.

For more information about creating decision tables, see [Create Decision Table](#) (see page 84).

## Opening an Existing Dynamic Decision Table

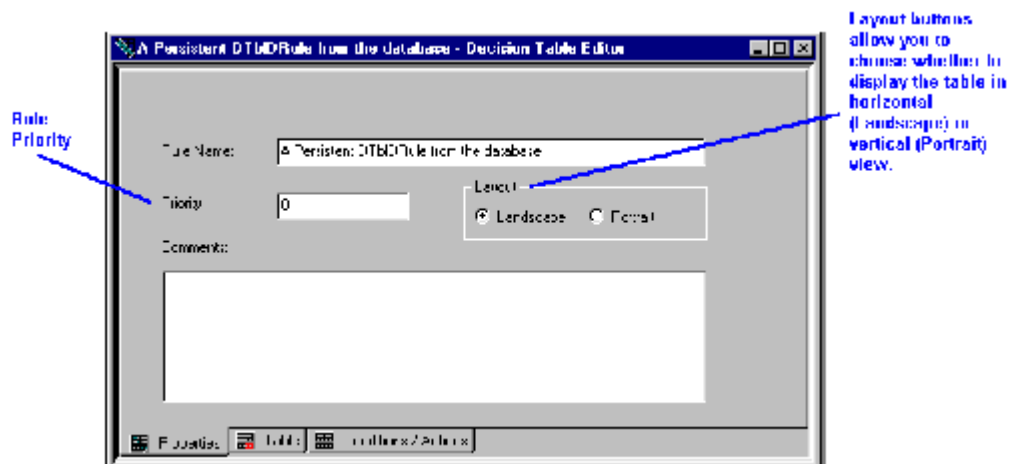
### To view or edit a dynamic rule:

1. Double-click the desired dynamic rule icon in the Rules tree in the Domain Workspace.  
  
Or, highlight the name of the rule in the Domain Workspace and choose Rule, Open from the Dynamic Rule Manager menu.
2. Click the Properties, Table, or Conditions/Actions tab as appropriate.

## Viewing and Modifying Dynamic Decision Table Properties

To view or modify properties of the decision table, click on the Properties tab of the decision table editor.

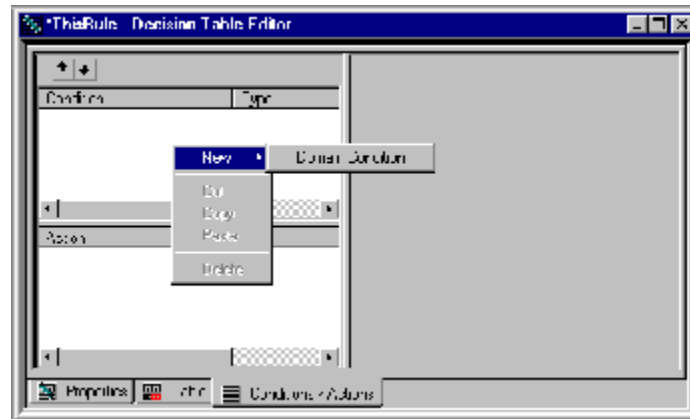
The Property page of the Dynamic Decision Table Editor appears as:



## Adding and Modifying Conditions

Conditions define the premises of a Decision Table rule. A Condition is composed of a Condition Name and a number of Test cells. Test cells display ranges or values specified in the Condition row.

In the Dynamic Decision Table Editor, one can use only domain interface members, which are referenced through their labels. To add a Condition, select Domain Condition for the Condition pop-up menu.



Domain Condition allows you select a domain interface member from the Name drop-down list.

### To define a dynamic decision table condition

1. In the Dynamic Rule Manager, choose Rule, New, Decision Table... to open the Decision Table Editor.
2. Click the Conditions/Actions tab.
3. Right-click in the Condition area to display the Condition menu, and select New, Domain Condition.
4. Select a name for the new Condition from the domain interface Members drop-down list. (This list contains all domain interface members that are Conditions.)
5. Define the values for the selected domain interface Member.
  - Use of ',' (the comma) is not allowed.
  - String test values will be automatically quoted. To test for a quoted value, specify it as a quoted string prefaced by '=', as in the test value = "my quoted string". If quotes are not specified (= my quoted string), the editor assume the test value is simply = my quoted string. ELSE, UNKNOWN and NULL values will not be quoted. When it is necessary to test for the string value "Null" instead of the NULL value, specify = "Null".

- Real test values have to be specified in USA settings without the use of ','

**Note:** Real and integer values are displayed according to the country format Setting.

- When specifying a range, place < or > before equals (=). Start range is > value or >= value. Optionally use '..' with an end range < value, <= value

6. You can arrange the order of Conditions by highlighting a Condition and clicking the up/down arrows.
7. Repeat Steps 3-5 as necessary.
8. Check Test Unknown Value/Test Other Values, as appropriate.

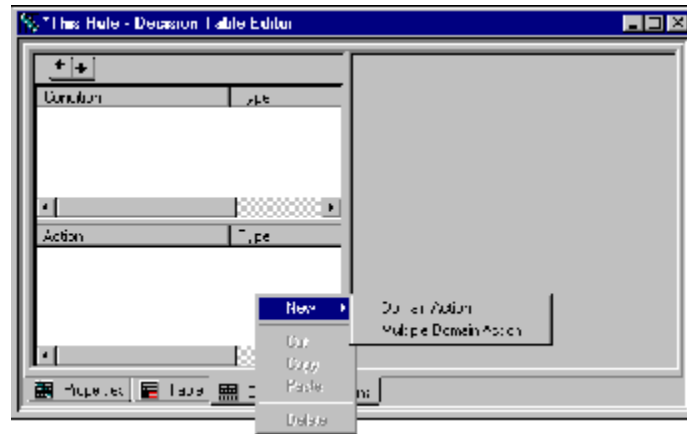
Test values for a condition are specified according to the following rules:

- Use of ',' (the comma) is not allowed.
- In the Dynamic Decision Table Editor, string test values are automatically quoted. For more details on automatic string quoting, see the preceding procedure To define a dynamic decision table condition.
- Real test values have to be specified in USA settings without the use of ','  
**Note:** Real and integer values are displayed according to the country format Setting.
- When specifying a range, place < or > before equals (=). Start range is > value or >= value. Optionally use '..' with an end range < value, <= value

## Adding and Modifying Actions

Actions define the activities to be taken by the Decision Table rule if the conditions are met. Actions are composed of an Action and a number of Selector cells.

In the Dynamic Decision Table Editor, one can use only domain interface members, which are referenced through their labels. There are two types of Actions that can be selected from the Action pop-up menu.



What do these types of Actions represent?

- Domain Action

Like Domain Condition, Domain Action allows you to choose a previously defined Domain Interface Action as the Action by selecting its label from the Name drop-down list. However, in the case of Domain Actions, there are two possibilities:

- The Domain Interface Member does not take an input argument. In this case, a specified action and the Domain Action look much the same in the decision table itself. Invocation of both is indicated by an "X" in the appropriate Action Selector cell of the decision table.
- The Domain Interface Member takes an input argument. In this case, you must specify the appropriate value to be passed as that argument in the Action Selector cell of the decision table where the Action is to be invoked. This technique reduces the table bloat that was produced when actions involved different arguments or attribute-value assignments.

- **Multiple Domain Action**

This option allows you to specify Domain Interface Actions in the Action Selector cells of a decision table. Thus, one row can be used to invoke different actions as long as these actions are mutually exclusive. In this case, you must create a name of the Action, as is done in the Specify Action style. The difference is that during decision table construction, an Action Selector cell of a Multiple Domain Action automatically transforms into a drop-down list when it becomes the focus. The list consists of Domain Interface Member labels that may be selected for that cell. Valid Domain Interface Members are those that (1) are defined to be of type Action, and (2) do not require an input argument.

**To define a dynamic decision table action**

1. In the Dynamic Rule Manager, choose Rule, New, Decision Table... to display the Decision Table Editor.
2. Click the Conditions/Actions tab.
3. Right-click in the Action area to display the Action menu.
4. Select New, Domain Action or New, Multiple Domain Action, depending on the number of methods you require.

Domain Action-May be a domain interface Member that takes at most one input argument.

Multiple Domain Action-May take a list of different domain interface Members, none of which can have an input argument.

5. Enter a name for the new Action.

For a Domain Action, use the name of the domain interface Member to be executed when that Action is indicated in the decision table. For a Multiple Domain Action, enter a name of your choice.

6. Repeat steps 3-5 as necessary.
7. Optionally, arrange the order of Actions by clicking the up/down arrows.
8. Click the Table tab. If Auto Refresh is checked, the decision table will be built. Otherwise, right-click anywhere on the table and select Refresh or Auto RefreshSpecify the appropriate actions for each combination of Condition values:

- **For Domain Actions**

If the domain interface member does not have an input argument, a checkmark appears in the Action cell when it is clicked.

If the domain interface member has an input argument, enter a value for that argument in the Action cell. This value is passed at execution time to the method that implements the domain interface member. The following conditions apply to entering values in Action cells:

- Use of ',' is not allowed.
- String action values will be quoted automatically.
- Real testvalues have to be specified in USA settings without the use of ','.

**Note:** Real and integer values are displayed according to the country format Setting.

■ For Multiple Domain Actions

Clicking the Action cell displays a drop-down list containing domain interface Members that may be specified as the value for this Action cell. The first member of the drop-down list is blank in case you need to cancel a previously selected domain interface member.

Multiple Domain Actions only support domain interface members that do not have an input argument.

9. Choose File, Save to save the decision table.

**Note:** It is possible to save incomplete and invalid rules. For example, this can occur when the:

- Table has no actions
- Table has no action values (table is empty)
- Table has an invalid value for a type, "This" is not a Boolean
- Table has invalid range specification < 23 .. > 100

This feature allows you to save your work and return to it later. When the table is invalid, the Active checkmark on the Properties page will be disabled if it is checked on.

## Summary: Specifying Selector Cells

If no selector cells for Actions are specified for a given combination of condition cells, the corresponding action cells are highlighted in the Decision Table Editor. (By default, highlighting is red.) This highlighting is a warning that there is a potential "hole" (that is, a possible outcome for which no action is specified) in the table logic. This could be valid if the combination of values is impossible, that is:

Has Driver License = TRUE  
and Age < 16

A blank selector cell indicates that the decision table will invoke no activity for an Action for the combination of Conditions. To invoke a result for the decision table, the selector cell must be specified. Selector cells are specified in three ways depending upon the nature of the action.

- Domain interface member with no input argument:

Place an "X" in the selector cell by left clicking in the cell. Left clicking in the cell a second time may toggle the "X" off.

- Domain interface member with an input argument:

Type the value of the input argument in the selector cell. Rules for action selector values are:

- Use of ',' (the comma) is not allowed.
- In the Dynamic Decision Table Editor, string selector values are automatically quoted.
- Real test values have to be specified in USA settings without the use of ',' .

**Note:** Real and integer values are displayed according to the country format Setting.

- Boolean values are shown in True/False format.

- For multiple domain actions:

Select the appropriate action domain interface member from the drop-down list. To cancel the action for that combination of conditions, select the first element in the drop-down list, which is always an empty value.

## Decision Table Editor Functions

The functions described below are the same in the static and dynamic rule editors. For more information, see [Decision Tables](#) (see page 83).

### Deleting a Rule

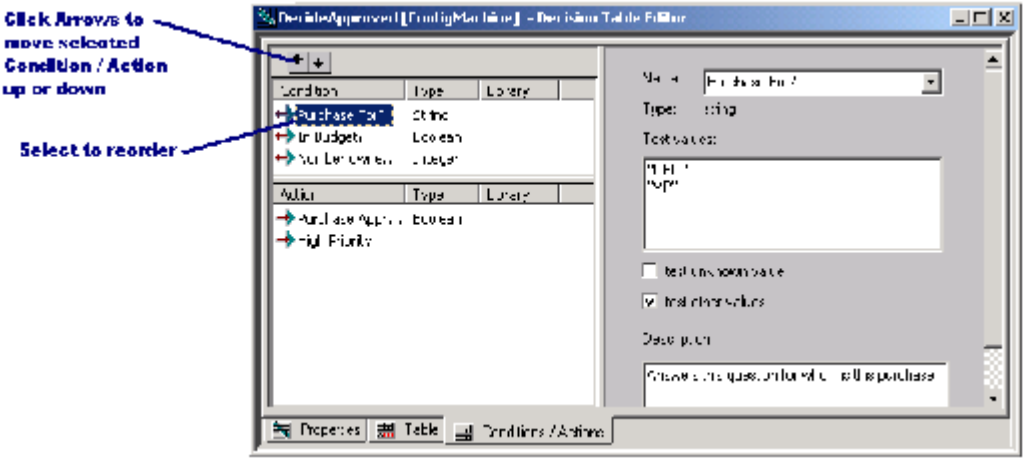
**To delete a rule from the rulebase:**

1. Select the rule in the Domain Workspace of the Decision Table Editor.
2. Choose Edit, Delete, or right-click to display a pop-up menu and choose Delete.



Ordering Conditions and Actions

When the Conditions/Actions page is displayed, you can use the Move Up and Move Down arrows to change the order of Conditions or Actions in the generated decision table.



For step-by-step procedures for changing the order of Conditions or Actions, see [Decision Tables](#) (see page 83).

**Note:** Aion also provides an automatic means to order conditions to achieve the minimal number of combinations of conditions. For more information, see [Achieving Optimal Condition Order for Compressibility](#) (see page 99).

Cutting and Pasting Conditions and Actions

For step-by-step procedures for cutting and pasting conditions and actions, and for cutting and pasting sub-table contents, see [Decision Tables](#) (see page 83).

Displaying a Dynamic Decision Table

You can control the appearance and/or behavior of a decision table on the Table page by right clicking in a non-table area of the page. A pop-up menu appears with the following options:

Menu Option	Explanation	Related Topic
Show compressed	See Compressing a Dynamic Decision Table	
Optimum condition order	See Achieving Optimal Condition Order for Compressibility	

Menu Option	Explanation	Related Topic
Toggle layout	Changes display between landscape and portrait.	Layout can be set on the Properties page, see <a href="#">Viewing and Modifying Dynamic Decision Table Properties</a> (see page 130)
Refresh	Redraws the decision table according to the latest changes to the definitions of the conditions and/or actions.	
Auto refresh	Toggles automatically refreshing of the decision table. Automatic refreshing cause the table to redrawn when it is redisplayed following any changes to the definitions of the conditions and/or actions.	

## Customizing the Decision Table Editor

You can use the Decision Table Options dialog to set options that control how the editor displays decision tables. Select the Decision Table Options command from the File menu. For more information about setting these options, see [Customizing the Decision Table Editor](#) (see page 95).

**Note:** The Domain Interface Filter group is not available in the Dynamic Rule Manager.

You can also select the font in which the Decision Table displays information. Select the Fonts command from the File menu.

**Note:** For more information about setting fonts, see the Fonts section of the “Creating and Editing Applications” chapter of the *User Guide*.

## Compressing a Decision Table

Compression is an edit-time feature concerned with making optimal use of screen real estate when displaying a decision table. Values are suppressed when those values are not relevant for distinguishing between different actions.

Compressing a decision table is done by right clicking on the background of the Table page and selecting "Show compressed" from the pop-up menu. For examples of compression, see [Compressing a Decision Table](#) (see page 96).

**Note:** Actions of compressed subtables are not editable.

## Manually Collapsing Subtables

Aion allows you to manually specify particular parts of the decision table that you may wish to collapse temporarily so that you can focus on a particular area of a table. For step-by-step procedures for compressing and collapsing the decision table, see [Compressing a Decision Table](#) (see page 96).

**Note:** Actions under manually collapsed subtables are not editable.

## Achieving Optimal Condition Order for Compressibility

Aion also provides a sophisticated algorithm for determining the optimum order of the Conditions in your decision table to achieve optimum compressibility. For step-by-step procedures for compressing and collapsing the decision table, see [Compressing a Decision Table](#) (see page 96).

For more information about determining the order of conditions to achieve greatest compressibility, see [Achieving Optimal Condition Order for Compressibility](#) (see page 99).

## Accessing the Dynamic Rule Repository

The Dynamic Rule Manager also provides access to the Dynamic Rule Repository. The Dynamic Rule Repository supports a business rule process for maintaining business rules. (For more information about business rule maintenance scenarios supported by the Dynamic Rule Repository, see "Business Rule Maintenance Scenarios in the Dynamic Rule Management" chapter of the *User Guide*). The Dynamic Rule Repository is not Aion-related software and may not be available in all environments. Dynamic rule repository functionality is provided by means of a Source Code Control (SCC) compatible source code control program (such programs include Microsoft Visual SourceSafe and CA Harvest Change Manager). Dynamic rules can be saved as source code within the database of the source code control program. The Dynamic Rule Manager uses the standard SCC application programmer's interface (API) to provide a subset of the functionality of the source code control program for dynamic rules.

Advantages of using the Dynamic Rule Repository include controlling access to rules through the check-out/check-in protections offered by a source code control program. These protections prevent two business experts from changing the same rule at the same time.

To access the Dynamic Rule Repository, right click the name of a rule in the Domain Workspace. This action invokes a pop-up menu that opens up the source code control program API functions provided within the Dynamic Rule Manager.

# Chapter 9: Constructing Non-Persistent Dynamic Rules

---

Dynamic rules are assembled for execution at runtime rather than being compiled into the knowledge base executable. One way to achieve dynamic rules is to store rules in a rulebase and load them into the knowledge base at run time. Such rules are known as persistent dynamic rules. The Aion mechanisms that support persistent dynamic rules also support their opposite, namely, non-persistent dynamic rules. Non-persistent dynamic rules are rules created at runtime within the knowledge itself. Instead of programming a rule, the Aion application developer programs logic that literally constructs the definition of a rule as the knowledge base is executing. Rules defined in this manner do not exist beyond the execution of the knowledge base.

Using non-persistent rules could be beneficial if the structure of the rules is highly dependent upon changing, external conditions. For example, non-persistent rules could be used when the conditions and actions that should be considered, and how they should be considered, depends on available data. Another potential use of non-persistent rules is to construct rules that the knowledge base reads from an XML document. In this case, the knowledge base must also have access to an XML translator.

**Note:** Non-persistent dynamic rules are intended for use by advanced Aion application developers.

This section contains the following topics:

[Facilities for Constructing Non-Persistent Dynamic Rules](#) (see page 141)  
[Non-Persistent Dynamic Decision Tables](#) (see page 142)

## Facilities for Constructing Non-Persistent Dynamic Rules

The facilities required by Aion for building non-persistent dynamic rules are:

- The library DynRLib. This library provides the basic mechanisms for defining and posting non-persistent dynamic rules. These mechanisms include:
  - The DIMember class. The instances of this class are individual domain interface member objects that are referenced in a dynamic rule.
  - The XXXRule\_Defn\_Interface interface, where XXX represents a type of dynamic rule supported by Aion. These interfaces provide the methods by which a dynamic rule can be defined to the inference engine.

- Posting facilities provided by XXXRule\_Services, where 'XXX' represents a type of dynamic rule supported by Aion. Posting facilities provide the PostRule( ) method.

**Note:** Currently Aion supports only dynamic decision tables. The only Rule Services provided by DynRLib is DecisionTableRule\_Services and the only interface is DecisionTableRule\_Defn\_Interface.

- An application class that implements the methods specified in the XXXRule\_Defn\_Interface interface for the type of non-persistent rule that will be created in the application.
  - A non-persistent rule is created as an instance of the class that implements the XXXRule\_Defn\_Interface interface.
  - The methods of XXXRule\_Defn\_Interface are used by the posting facility to parse the values for the conditions and actions (DIMembers) comprising the non-persistent rule.

- Any additional application structures that may be necessary to implement the methods of the XXXRule\_Defn\_Interface interface.

These structures provide the condition and action values that comprise the non-persistent rule and are used to associate these values with the domain interface members (instances of DIMember class) that are referenced in the rule.

- An agent (class) that has responsibility for constructing the non-persistent rule. This class, defined by the Aion application developer, typically also posts and inferences over rules.

## Non-Persistent Dynamic Decision Tables

The NPDRules example in the \Examples\DynaRule\NPDRule directory serves as the basis of following explanation of non-persistent dynamic decision tables. In NPDRules, the following classes and methods fulfill the facilities that are required to create non-persistent dynamic rules.

**Note:** For more information about these facilities, see [Facilities for Constructing Non-Persistent Dynamic Rules](#) (see page 141).

- A class that implements the DecisionTableRule\_Defn\_Interface interface.  
In NPDRules, this class is Sample\_DTBl\_Definer.

**Note:** For more information about constructing an implementation of this interface, see [Non-Persistent Rule Definer Class](#) (see page 144).

The Sample\_DTBl\_Definer class can serve as a typical model for defining such classes, although other ways of implementing the methods of DecisionTableRule\_Defn\_Interface are possible. The functionality of Sample\_Dtbl\_Definer is completely general, and it may be used in other applications just as it is implemented here.

- Mechanisms to define the condition and action values and to hold the association of these values with the domain interface members of the non-persistent rule.

In NPDRules, these mechanisms are provided by the ConditionData and ActionData classes. Interface methods implemented in Sample\_Dtbl\_Definer made reference to instances of these structures to obtain the values of conditions and actions of the dynamic rule.

Although other application strategies are possible for defining the required mechanisms, these classes are also completely general and can be reused in other applications just as they are implemented in NPDRules.

- An agent that constructs and posts the non-persistent rules.

In NPDRules, this agent is the Examples class. In its PostDTbl( ) method, the Examples class:

1. Creates the domain interface members that are referenced by the non-persistent decision table being constructed.
2. Specifies the condition and action values for each domain interface member and associates these values with the appropriate domain interface member
3. Posts the decision table.

PostDTbl( ) invokes the \_DefinedDTblConditions( ) and \_DefinedDTblActions( ) methods. These methods construct the instances of the ConditionData and ActionData classes that define the condition and action values for the methods referenced by the non-persistent decision table. These methods contain highly domain specific content.

**Note:** For more information, see [Constructing Decision Table Conditions](#) (see page 144) and [Constructing Decision Table Actions](#) (see page 147). For more information about posting the decision table that has now been constructed, see [Posting a Non-Persistent Decision Table](#) (see page 151).

## Non-Persistent Rule Definer Class

In NPDRules, the non-persistent rule Definer class, `Sample_DTbl_Definer` class must implement two principal types of methods:

- The `DecisionTableRule_Defn_Interface:GetDTblXXX( )` methods, which are general purpose methods for retrieving information about the condition test values or action selectors of a dynamic decision table. These methods are used by Aion during posting. For more information about these methods, see “DynRDLib” in the *online Reference*.
- The non-interface method `InitRuleData( )` method, which is called to define a non-persistent rule. Its input arguments are:

`argRuleName` is String  
`argRulePriority` is integer  
`argConditionArray` is array of `&ConditionData`  
`argActionArray` is array of `&ActionData`

The “trick” is to pass this method the pointers to structures that hold the condition and action values that define the desired decision table. Conditions and Actions are defined as instances of the `ConditionData` and `ActionData` classes, respectively.

## Constructing Decision Table Conditions

In NPDRules, see the `_DefineDTblConditions( )` method in the `Examples` class. Notice that the argument of this method (an array of pointers to `ConditionData`) is an input to the `InitRuleData( )` method in the `Sample__DTbl_Definer` class. In this method, instances of the `ConditionData` class are constructed, each of which represents a Condition of a decision table. Pointers to these instances are saved in an array, which serves as the method's output argument.

The first step is to dimension, or redimension, the array so that it will hold the pointers to Conditions that you want in the decision table.

```
Array.Redimension(ConditionArray, 2)    // Two Conditions
```

A Condition of a dynamic decision table must always reference a domain interface member. A domain interface member is represented by an instance of the class `DIMember` in `DynRLib`. The next step in building a definition for a non-persistent dynamic decision table is to create an instance of the `DIMember` class that defines the domain interface member you want to reference in your condition. Simply specifying a label for the domain interface member can do this. (Of course, this must be a valid domain interface member defined in the application or one of its imported libraries.)

```
var pDI is &DIMember  
pDI = DIMember.Create(NULL, "Person Age?", NULL)
```



The ConditionData instance that defines a Condition in a non-persistent decision table must:

- Point to this domain interface member
- Describe the test values that are relevant in the Condition

For example:

```
var iEle is integer = 0
iEle = iEle +1
ConditionArray(iEle) = ConditionData.Create( )
    // Create a ConditionData data instance and set an
    // array element to point to that instance.
ConditionArray(iEle).SetMember(pDI)
    // Point this ConditionData instance to the Person
    // Age? Domain interface member
var TestValues is string
TestValues = ">=21..<=150" &CHAR_TAB & "unknown"
    // Specify the test values for this Condition just as
    // they would be specified in the decision table
    // editor. Values must be separated by a CHAR_TAB
ConditionArray(iEle).SetTestValues(TestValues)
```

**Note:** For more information about constructing Condition test values, see [Condition Test-Value String Format](#) (see page 145).

**Important!** An instance binding may be required for a Condition. This binding may be set during the Condition construction phase (see the example in NPDRules) or be passed as an input argument in the posting method (see [Posting a Non-Persistent Decision Table](#) (see page 151)). For more information about instance binding, see [Dynamic Rule Runtime Considerations](#) (see page 57).

The preceding procedure is repeated for each Condition in the decision table.

### Condition Test-Value String Format

The test-value string of a Condition specifies all the test values associated with the Condition. The application specifies condition test-value strings as arguments to the ValidateConditionTestValues( ) method in the DecisionTableRule\_Services class and the GetDTblCondition( ) method in the DecisionTableRule\_Defn\_Interface class.

Within the test-value string, tab characters (CHAR\_TAB) delimit test values. Text is not case-sensitive and all values must be in quotes. At most, there can be one ELSE test value specified for a Condition and, if specified, it must be the last test value.

## Examples

The following are examples of valid test-value strings.

### To describe the following set of condition values

12	=NULL	<>13	UNKNOWN	ELSE
----	-------	------	---------	------

use:

```
"12"  
  & CHAR_TAB & "=null"  
  & CHAR_TAB & "<>13"  
  & CHAR_TAB & "unknown"  
  & CHAR_TAB & "else"
```

### To describe the following set of condition values

<100	>=100 .. <200	>=200
------	---------------	-------

use:

```
"<100"  
  & CHAR_TAB & ">=100 .. <200"  
  & CHAR_TAB & ">=200"
```

### To describe the following set of condition values

True	false
------	-------

use:

```
"true" & CHAR_TAB & "false"
```

### To describe the following set of condition values:

"Max"	"Sally"	>="Xerxes"	NULL	ELSE
-------	---------	------------	------	------

use:

```
CHAR_QUOTE & "Max" & CHAR_QUOTE  
  & CHAR_TAB & "=" & CHAR_QUOTE & "Sally" & CHAR_QUOTE  
  & CHAR_TAB & ">=" & CHAR_QUOTE & "Xerxes" & CHAR_QUOTE  
  & CHAR_TAB & "null"  
  & CHAR_TAB & "else"
```

Notice the use of CHAR\_QUOTES in the final example. CHAR\_QUOTES are necessary to enclose the string being concatenated in quotes, which is required of test value cells containing string values.

## Constructing Decision Table Actions

In NPDRules, see the `_DefinedTblActions( )` method in the Examples class. Notice that the argument of this method (an array of pointers to `ActionData`) is an input to the `InitRuleData( )` method just as in `_DefineTblConditions( )`. In this method, instances of the `ActionData` are created, each of which represents an Action of a decision table. Pointers to these instances are saved in an array, which serves as the method's output argument.

Thus, the process of defining a non-persistent decision table Action is quite similar to the process of defining a Condition. In particular, an instance of the `DIMember` class must be created and an instance of `ActionData` must point to that `DIMember` instance. The only difference between the two processes is that three types of Actions that can be specified for a decision table. These actions are:

- Checkmark actions
- Value actions
- Multiple Domain Action actions (MDAs).

Aion provides constants that specify the type of action. In NPDRules, see the `SetActionType( )` method in the `ActionData` class that is used to define an `ActionData` instance as supporting a particular type of action.

- `DYNRACTIONTYPE_CHECKMARKS` defines an Action whose domain interface member does not take an input argument.
- `DYNRACTIONTYPE_VALUES` defines an Action whose domain interface member does take an input argument.
- `DYNRACTIONTYPE_ACTIONS` defines an Action that can invoke other domain interface members.

In particular, for all Action types, the number of Action selectors must exactly equal the number of decision table outcomes. If the decision table has six possible outcomes, each Action must specify exactly six selectors. There are no "defaults" if fewer selectors are specified.

## Checkmark Actions

The selectors for a checkmark action are Booleans, where TRUE represents an invocation of the domain interface member. Create a list of Booleans that describes the set of actions making up the Action. Use the Boolean value FALSE to indicate that the domain interface member is not to be invoked for the combination of values.

**To describe an action row that looks like the following example**

<b>Action1</b>		X	X	
----------------	--	---	---	--

Use the following list of Booleans: (False, True, True, False)

## Value Actions

The selector for a value action is a string of concatenated values that represent the input arguments to the domain interface member. Create a tab-delimited string of concatenated values that describes the Action, where each value is expressed as strings. The requirements governing how this value-string is created are similar to defining test values for Conditions. To designate a selector cell with no value, concatenate a tab, immediately followed by another.

**Note:** For more information about patterns of constructing value strings for value actions, see [ValueAction Selector Format](#) (see page 149).

## Multiple Domain Action (MDA) Actions

This type of action is the most complex. First, each MDA Action selector must be created as an instance of the DIMember class. It is then necessary to create a list of pointers to these instance that describes the set of actions making up the MDA Action. To designate a selector cell with no value, include a pointer with the value of NULL in the list.

**To describe an MDA action row that looks like the following example**

<b>Action1</b>	Set Status		Sound Alarm	Sound Alarm
----------------	------------	--	-------------	-------------

Use the list (pSetStatus, NULL, pSoundAlarm, pSoundAlarm), where each pointer points to the appropriate DIMember instance.

Multiple Domain Action actions require two pieces of information:

- A description (a user chosen name for the MDA Action)
- The list of selectors (pointers to domain interface actions)

Again, should any Action require an instance binding, including those domain interface members created as selectors of an MDA Action, the binding must be established during this Action construction phrase or passed as an argument in the non-persistent decision table posting method, see [ValueAction Selector Format](#) (see page 149).

The above procedure must be repeated for each Action in the decision table.

### ValueAction Selector Format

For Actions of the type DYNRACTIONTYPE\_VALUES, the selector is a string specifying all the value selectors associated with the Action.

Within the string, tab characters (CHAR\_TAB) delimit values. Successive tab characters indicate non-selection. The total number of value selectors and non-selectors should match the number of decision table dynamic rule outcomes exactly. Otherwise, a subsequent decision table dynamic rule posting or validation service will fail.

Value selectors must be constants of a data type compatible with the data type of the first parameter of the Action's domain interface member method. Special considerations include the following:

- For parameters of any data type, NULL is always a valid selector value.
- For parameters of data type string, string constants must be in quotes.
- For parameters of data type real, value selectors can be either integer or real constants.

### Examples

The following are examples of valid value-selector strings for each data type.

#### Integer Selectors

To describe an action row with the following values:

Action1	123	-13		NULL
---------	-----	-----	--	------

Use the following example:

```
"123"  
  & CHAR_TAB & "-13"  
  & CHAR_TAB           // Non-selection of 3rd outcome  
  & CHAR_TAB & "null"
```

**Note:** The absence of a value in the third column must be indicated by an extra CHAR\_TAB to "jump over" the cell.

**Real Selectors**

To describe an action row with the following values:

<b>Action1</b>	12.356	-12	-12.356e+2	NULL
----------------	--------	-----	------------	------

Use the following example:

```
"12.356"  
  & CHAR_TAB & "-12"  
  & CHAR_TAB & "-12.356e+2"  
  & CHAR_TAB & "null"
```

**Boolean Selectors**

To describe an action row with the following values:

<b>Action1</b>	True	false	NULL
----------------	------	-------	------

Use the following example:

```
"true"  
  & CHAR_TAB & "false"  
  & CHAR_TAB & "null"
```

**String Selectors**

To describe an action row with the following values:

<b>Action1</b>	"abc"	"123 56 abc"	NULL
----------------	-------	--------------	------

Use the following example:

```
CHAR_QUOTE & "abc" & CHAR_QUOTE  
  & CHAR_TAB & CHAR_QUOTE & "123 56 abc" & CHAR_QUOTE  
  & CHAR_TAB & "null"
```

Notice the use of CHAR\_QUOTES to indicate that the string value is enclosed in quotes, as required for selector cells containing a string value.

## Posting a Non-Persistent Decision Table

When posting a persistent dynamic decision table, Aion requires the use of the `PostDTbl( )` method, which is provided by the `DecTableRuntime` services in `DynRDLib`. For posting non-persistent decision tables, there is a separate posting method, `PostRule( )`, provided by `DecisionTableRule_Services` in `DynRLib`. The former is an instance method of the persistent decision table itself that has been loaded from the (external) rulebase. The latter is a class method. The relationship between them is that the `PostDTbl( )` method calls the `PostRule( )` method. Thus, in posting a non-persistent dynamic rule, the Aion programmer is explicitly invoking the mechanisms that Aion itself uses internally to post persistent dynamic rules.

Because `PostRule( )` is a class method and not a method of the non-persistent decision table itself, when posting a non-persistent decision table it is necessary to pass the pointer to the non-persistent rule definer object, that is, to the instance of the class that implements the `DecisionTableRule_Dfn_Interface`. The signature of `DecisionTableRule_Services:PostRule( )` is:

```
PostRule(pDefiner is &DecisionTableRule_Dfn_Interface, pInstBindings is list of
&_Object = NULL) : integer
```

Where `Sample_DTbl_Definer` is the class implementing the dynamic decision table `_Interface`, you can post a non-persistent decision table with the following code:

```
var hRule is integer
var pDefiner is &Sample_DTbl_Definer
pDefiner = Sample_DTbl_Definer.Create( )
.
.
.
// Insert code to set the properties of the pDefiner
// instance as shown above. Invoke InitRuleData( ) with
// the appropriate arguments.
.
.
// Now post the decision table that has been defined:
hRule = DecisionTableRule_Services.PostRule(pDefiner)
    // Here we assume no (further) instance bindings are
    // required.
```





# Appendix A: Summary of Inferencing Constructs

---

This appendix includes a list of the inference block constructs.

This section contains the following topics:

[Inference Block](#) (see page 153)

[Chaining Statements](#) (see page 154)

[Rule Types](#) (see page 154)

[Production Demons](#) (see page 156)

[Pattern Matching Demons](#) (see page 157)

[Knowability Expressions](#) (see page 157)

[Truth Maintenance Operations](#) (see page 157)

## Inference Block

All rule definition and chaining must occur within the runtime context of an inference block:

```
INFER
    PostSomeRules( )
    FORWARDCHAIN( )
END
```

If the inference engine needs to retain historical information for Truth Maintenance operations or debugging purposes, the inference block should be specified with the HISTORY option:

```
INFER HISTORY
    PostSomeRules( )
    FORWARDCHAIN( )
END
```

## Chaining Statements

There are two inferencing strategies, forward chaining and backward chaining. Forward chaining can optionally specify a goal using an attribute pointer, and it can optionally return an integer return code:

```
FORWARDCHAIN( )  
FORWARDCHAIN(->goal)  
VAR rc IS INTEGER  
rc = FORWARDCHAIN( )
```

Backward chaining must specify a goal using an attribute pointer, and it can optionally return an integer return code:

```
BACKWARDCHAIN(->goal)  
VAR rc IS INTEGER  
rc = BACKWARDCHAIN(->goal)
```

During inferencing, the STOPCHAIN statement will terminate chaining:

```
STOPCHAIN( )
```

The STOPCHAIN statement may optionally specify an integer expression which will become the chaining statement's return code:

```
STOPCHAIN(Obj.GetErrorCode( ))
```

## Rule Types

This section discusses the following types of Rules:

- Production Rule
- Decision Table
- Pattern Matching Rule

## Production Rule

A production rule applies to single instances of classes. It can be employed for both forward chaining and backward chaining.

```
RULE StartingRule // RuleName
IFRULE Obj.GetValue( ) < 3 // RulePremise
THEN
// RuleActions
    Obj.SetStatus(TRUE)
    Person.Notify(Obj)
    STOPCHAIN(100)
END
```

The rule may optionally specify a posting PRIORITY as an integer expression:

```
RULE "Another Rule" // RuleName
PRIORITY 100
IFRULE Person.Age <12 AND Dog.Age > 1
    THEN
        Person.AssignDog(Dog)
END
```

## Decision Table Rule

A decision table rule is a generalization of a production rule. It applies to single instances of classes. It can be employed for both forward chaining and backward chaining.

Decision tables consists of multiple conditions and actions. Decision tables are defined and maintained using the Decision Table Editor, shown in the following figure: {bmc Art\\DT\_256.BMP}

## Pattern Matching Rule

A pattern matching rule applies collectively to all instances of one or more classes. It can be employed only for forward chaining.

Pattern matching rules symbolically refer to instances using bind variables. The bind variables must be declared prior to the rule's definition:

```
BIND bParent TO Parent // for Parent class
BIND bChild TO Child // for Child class
```

The rule then references the bind variables as instances of the associated classes:

```
RULE MatchParentsWithChildren
IFMATCH bParent, bChild
WHERE
    bParent.GetNumKids( ) > 0
    AND bParent.LastName = bChild.LastName
THEN
    School.SendReportCard(bChild, bParent)
END
```

The rule may optionally specify a posting PRIORITY. It can also optionally specify a list of integer expressions indicating how the resulting bindings should be ordered [ORDERBY]. It can also optionally specify whether bindings should be returned in LEASTRECENT or MOSTRECENT order [LEASTRECENT is the default].

```
BIND bChild1 TO Child
BIND bChild2 TO Child
RULE "Match children with the same first name"
PRIORITY 20
IFMATCH bChild1, bChild2
WHERE
    bChild1 <> bChild2
    AND bChild1.FirstName = bChild2.FirstName
ORDERBY AverageAge(bChild1, bChild2), AverageHeight(bChild1, bChild2)
MOSTRECENT
THEN
    Report.AddItem(bChild1, bChild2)
END
```

## Production Demons

A production demon is the demon version of a production rule. It applies to single instances of classes. It can be employed for both forward chaining and backward chaining.

```
RULE CheckForMaxTemperature
    WHEN Boiler.Temperature > MaxSafeTemperature
THEN
    SendAlert("Boiler is now too hot!")
END
```

The demon may optionally specify a posting PRIORITY as an integer expression.

## Pattern Matching Demons

A pattern matching demon is the demon version of a pattern matching rule. It applies collectively to all instances of a single class. It can be employed for both forward chaining and backward chaining.

```

BIND bCat TO Cat
RULE "Check for fat cats"
WHENMATCH bCat
WHERE
bCat.Weight > 20
THEN
bCat.StartDiet( )
END
```

The demon may optionally specify a posting PRIORITY as an integer expression.

## Knowability Expressions

These expressions determine an attribute's UNKNOWN status. Each expression accepts an attributepointer argument and returns a Boolean value:

```

// Never causes a rule to pend
ISUNKNOWN(->Person.Age)
// May cause a rule to pend
ISKNOWN(->Salary)
```

## Truth Maintenance Operations

A TMAssignment associates an attribute with a retractable value:

```
Person.Status ?= OK
```

A TMRetractation restores the attribute back to its previous nonretractable value:

```
RETRACT(->Person.Status)
```

A TMConfirmation establishes an attribute's current retractable value as its new nonretractable value:

```
CONFIRM( ->Person.Status)
```

For truth maintenance operations, the inference block must be specified with the HISTORY option:

```
INFER HISTORY  
    ...  
END
```

# Appendix B: How the Inference Engine Works

---

The Aion inference engine is sophisticated enough to process your rules very efficiently. Occasionally, however, you may feel the need to understand how the engine does its job. Various aspects of the process have been discussed throughout this and the preceding chapter. We will put the pieces together here. If the terms are unfamiliar, please go back and read the other material first.

This section contains the following topics:

[Forward Chaining](#) (see page 159)

[Backward Chaining](#) (see page 161)

## Forward Chaining

The inference engine starts by posting rules, which may exist directly in the INFER block or in rule methods called from the INFER block. Initially, all rules have a state of Ready.

The engine visits the rules in priority order. This is not necessarily the order in which they were posted, but the order specified by their priority keyword.

- If the premise can be evaluated to TRUE, the rule fires.
- If the premise can be evaluated to FALSE, the rule fails.
- Whether TRUE or FALSE, the rule is finished. Its state is changed to Fired or Failed, and the engine will not visit it anymore.
- If any attributes in the premise are UNKNOWN, the rule is pended—that is, the engine changes the rule's state from Ready to Pended. The engine adds the attribute to a list of attributes to be watched.

Then, the engine proceeds to the next Ready rule in the priority order and fires, fails, or pends it.

Whenever a rule fires and its action makes an assignment to an attribute for a pended rule, the engine takes notice. It changes the rule dependent on the attribute from Pended to Ready and, thus, readies the rule for further consideration. The engine then goes back to the top of the posted-rule list and takes another pass at the Ready rules, visiting them in priority order.

Forward chaining can end in three ways. You can:

- Allow inferencing to continue until all rules have fired, failed, or pended, and no Ready rules remain.
- Define a goal attribute, which will halt the inferencing process as soon as it has been assigned a value. Goal attributes are optional for forward chaining.
- Use a stopchain command.

### Forward Chaining Example

In the following example, we gather facts from symptoms. There are three Symptom attributes. One or more of these should be assigned values prior to inferencing. There are three Fact attributes. All of these should be UNKNOWN prior to inferencing. Some of these may be assigned values during inferencing. The inferencing code is as follows:

```
// Gather Facts from Symptoms
var rc is integer
infer
rule "GF1: Condition under which Fact1 is true"
ifrule not Fact2
then
    Fact1 = TRUE
end
rule "GF2: Condition under which Fact1 is false"
ifrule Fact2 and Fact3
then
    Fact1 = FALSE
end
rule "GF3: Condition under which Fact2 is true"
ifrule Symptom1 and Symptom2
then
    Fact2 = TRUE
end
rule "GF4: Condition under which Fact2 is false"
ifrule not Symptom1 and Symptom3
then
    Fact2 = FALSE
end
rule "GF5: Condition under which Fact3 is true"
ifrule Symptom2 and Symptom3
then
```



```
        Fact3 = TRUE
    end
    rule "GF6: Condition under which Fact3 is false"
    ifrule Symptom1 and not Symptom2
    then
        Fact3 = FALSE
    end
    rule "GF7: Make sure we have enough Symptoms"
    priority 100
    ifrule isunknown(->Symptom1)
        or isunknown(->Symptom2)
        or isunknown(->Symptom3)
    then
        stopchain
    end
    // Perform chaining
    rc = forwardchain( )
    end // of infer
    return rc
```

If there are no symptoms, rule GF7 terminates chaining using the STOPCHAIN command. The facts all remain UNKNOWN and the chaining return code is CHAIN\_STOPNV.

Note that GF7 is a high PRIORITY rule. Its priority assures that the Engine will examine it before examining any of the other rules. Some rules (GF1 and GF2) establish facts from other facts. These rules are therefore dependent on other facts and will pend until those other facts are established. Other rules establish facts directly from symptoms and will pend if symptoms are UNKNOWN. Assuming that at least one symptom has been defined, one or more of the facts are defined and the chaining return code is CHAIN\_OUTOFRULES.

## Backward Chaining

Posting is the same as in forward chaining, and all posted rules start in the Ready state. Since backward chaining is goal-driven, however, the engine processes rules a little differently—it starts by creating a goal list. Initially, the goal list contains only the main goal as specified by the chaining statement.

The engine visits the first rule in priority order and ascertains if the rule action contains an assignment to the goal attribute. If not, the engine moves on to the next rule in priority order.

If so, the engine tries to evaluate the premise:

- A TRUE premise causes the rule to fire. The rule's state changes to Fired.
- A FALSE premise causes the rule to fail. The rule's state changes to Failed.
- If the premise contains one or more UNKNOWN attributes, they are added to the goal list as subgoals. The rule's state changes to Pended. The engine then goes back to the start of the priority order and visits the first Ready rule.

The engine proceeds through the Ready rules in priority order. With each rule, it checks to see if the action contains assignments to any of the attributes in the goal list. If so, the engine tries to evaluate the rule premise. If not, the rule is passed over.

The engine starts back at the top of the priority order whenever the following happens:

- A rule is fired, and its action makes an assignment to a subgoal. All pended rules that contain the subgoal change to the Ready state.
- A rule is pended. The UNKNOWN attribute is added to the goal list.

As in forward chaining, the engine continues until the main goal is assigned, no Ready rules remain, or you issue the stopchain command. Backward chaining requires, however, that you specify a goal attribute.

## Backward Chaining Example

This example continues where the last example left off. Having established facts, we are now drawing conclusions and from those conclusions, we are drawing an overall conclusion. There are two Conclusion attributes. Both of these should be UNKNOWN prior to inferencing. Some of these may be assigned values during inferencing. There is one OverallConclusion attribute. This attribute is the goal of the backward chaining and it should be UNKNOWN prior to inferencing. The inferencing code is as follows:

```
// Draw Overall Conclusion from Facts
var rc is integer
  infer
  rule "DC1: Can conclude Outcome1"
  ifrule Conclusion1 and not Conclusion2
  then
    OverallConclusion = "Outcome1"
  end
  rule "DC2: Can conclude Outcome2"
  ifrule Conclusion1 and Conclusion2
  then
    OverallConclusion = "Outcome2"
  end
```

```
rule "DC3: Can conclude Conclusion1 as true [1 of 2]"
ifrule Fact1 and not Fact2
then
    Conclusion1 = TRUE
end
rule "DC4: Can conclude Conclusion1 as true [2 of 2]"
ifrule Fact2 and Fact3
then
    Conclusion1 = TRUE
end
rule "DC5: Can conclude Conclusion2 as true"
ifrule not Fact2 and Fact3
then
    Conclusion2 = TRUE
end
rule "DC6: Can conclude Conclusion2 as false"
ifrule Fact2 and Fact3
then
    Conclusion2 = FALSE
end
// Perform chaining
rc = backwardchain(->OverallConclusion)
end // of infer
return rc
```

Because rules DC1 and DC2 define values for the overall conclusion, the Engine attempts to fire them first. This would happen regardless of the placement or priority of these rules within the INFER block. Since these rules are dependent on other conclusions, they will pend until those other conclusions are established. The Engine attempts to fire the remaining rules because they establish values for conclusions needed by DC1 and DC2.

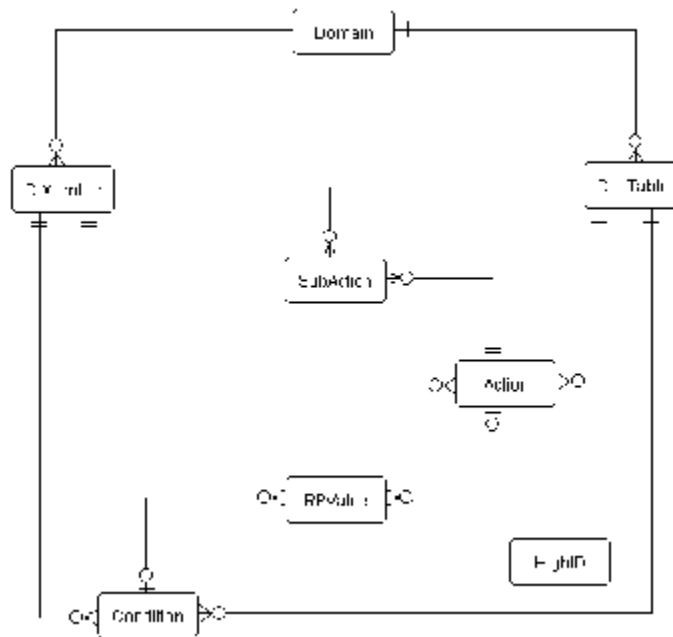
At the end of inferencing, there are two possible outcomes:

- Either DC1 or DC2 has fired (but not both). In this case, the overall conclusion is defined and the chaining return code is CHAIN\_COMPLETED.
- Neither DC1 nor DC2 has fired. In this case, the overall conclusion remains UNKNOWN and the chaining return code is CHAIN\_OUTOFRULES.



## Appendix C: Rulebase Structure

The following Entity/Relationship diagram depicts the default rulebase structure for dynamic decision tables.



**Note 1:** Conceptually, the relationship between Domain and DIMember is many-to-many: a domain may contain many domain interface members and a domain interface member can belong to many domains. However, the relationship is implemented as a one-to-many relationship: the same domain interface member has multiple occurrences in the DIMember table for each domain to which it belongs.

**Note 2:** Conceptually, the relationship between DecTable and Condition and DecTable and Action is mandatory: a (valid) decision table must have at least Condition and at least one Action. These relationships are optional to allow partially completed decision tables to be saved in the rulebase.

**Note 3:** Conceptually, the relationships between Condition and RPValue and Action and RPValue are one-to-one. However, it is necessary for the rulebase to accommodate the possibility that the length of the string describing the Condition or Action value may exceed the length of the rulebase field. In this case, Aion will automatically parse the value string and create multiple RPValue entries for a single Condition or Action.

This section contains the following topics:

[Table Domain](#) (see page 166)  
[Table DIMember](#) (see page 166)  
[Table DecTable](#) (see page 168)  
[Table Condition](#) (see page 168)  
[Table Action](#) (see page 169)  
[Table RPValue](#) (see page 169)  
[Table SubAction](#) (see page 170)  
[Table HighID](#) (see page 170)  
[Table Users](#) (see page 171)  
[Table CheckOut](#) (see page 171)

## Table Domain

Defines each domain in the current rulebase. Domains provide a means to group dynamic rules within a common area, subject matter or discipline.

Field	Type	Size	Description	Key
DomainID	Number (Long)	4	System assigned unique identifier for this domain.	Primary, ascending
Name	Text	64	Name of the domain	
EditFlags	Number (Long)	4	Bit mapped field representing a set of flags for this domain.	

## Table DIMember

Defines an occurrence of a domain interface member within a domain.

Field	Type	Size	Description	Key
DomainID	Number (Long)	4	System assigned unique identifier for the domain to which this domain interface member belongs.	Foreign, ascending

Field	Type	Size	Description	Key
DIID	Number (Long)	4	System assigned unique identifier for this domain interface member.	Primary, ascending
Label	Text	96	Name of this domain interface member.	
DIType	Number (Integer)	2	Code identifying the Type of domain interface member: 1 = Condition 2 = Action with no argument 3 = Action with one input argument	
ValType	Number (Integer)	2	Code identifying the type of return value (for Conditions) or input argument (for Actions) 0 = None 1 = String 2 = Boolean 3 = Integer 4 = Real	
Descript	Memo	-	Comments on this domain interface member.	
Source	Text	64	Aion application (library) from which this domain interface member was imported.	
LastUpdt	Text	20	Date and time that the Source containing this DIMember was last saved prior to this member being synchronized with the rulebase.	
Status	Text	64	The last activity and the date/time that this activity was performed on this member. Activity values are "imported" and "updated".	
EditFlags	Number (Long)	4	Bit mapped field representing a set of flags for this DIMember.	

## Table DecTable

Defines a decision table (property information) within a domain.

Field	Type	Size	Description	Key
DomainID	Number (Long)	4	System assigned unique identifier for the domain to which this decision table belongs.	Foreign, ascending
RuleID	Number (Long)	4	System assigned unique identifier for this decision table.	Index, ascending
RuleType	Number (Integer)	2	Code identifying the type of this dynamic rule. (1 = Decision Table).	
Name	Text	200	Name assigned to this decision table.	
Descript	Memo	-	Comments on this decision table.	
Priority	Number (Long)	4	Priority assigned to this decision table: determines order in which the decision table is evaluated by the inference engine.	
EditFlags	Number (Long)	4	Bit mapped field representing a set of flags for this domain.	

## Table Condition

Defines a Condition for a dynamic rule.

Field	Type	Size	Description	Key
PartID	Number (Long)	4	System assigned unique identifier for this Condition.	Index, ascending
PartType	Number (Integer)	2	Code identifying the type of Condition. (1 = Condition)	
RuleID	Number (Long)	4	System assigned unique identifier for the dynamic rule to which this Condition belongs	Foreign, ascending
PartSeq	Number (Integer)	2	The position of this Condition amongst the Conditions of the owning decision table.	
DIID	Number (Long)	4	System assigned unique identifier for the domain interface member from which this Condition is derived	Foreign, ascending



## Table Action

Defines an Action for a dynamic rule.

Field	Type	Size	Description	Key
PartID	Number (Long)	4	System assigned unique identifier for this Action.	Index, ascending
PartType	Number (Integer)	2	Code identifying the type of Action: 2 = Action with no argument 3 = Action with one input argument 4 = Multiple Domain Action	
RuleID	Number (Long)	4	System assigned unique identifier for the dynamic rule to which this Action belongs	Foreign, ascending
PartSeq	Number (Integer)	2	The position of this Action amongst the Actions of the owning decision table.	
DIID	Number (Long)	4	System assigned unique identifier for the domain interface member from which this Action is derived.	Foreign, ascending
GrpLabel	Text	96	Name assigned to Multiple Domain Action	

## Table RPValue

Defines the values and how these values occur in a decision table for Conditions and Actions. Each RPValue entry is owned by a Condition or Action.

Field	Type	Size	Description	Key
ValID	Number (Long)	4	System assigned unique identifier for this RPValue entry	Index, ascending
PartID	Number (Long)	4	System assigned unique identifier for the Condition or Action that this RPValue entry defines.	Foreign, ascending
ValSeq	Number (Integer)	2	The sequence of the RPValue entry amongst those owned by the Condition or Action. (Typical this will be 1, see <a href="#">Note 3</a> ) (see page 165).	

Field	Type	Size	Description	Key
ValStr	Memo	-	A string containing the test values of the owning Condition or the list of Action values. For Actions of type 2, relevant columns in which this Action is invoked; for Actions of type 3, the input arguments to be passed to the domain interface member.	

## Table SubAction

Defines the action domain interface members that are to be invoked in a Multiple Domain Action Action.

**Note:** Entries in this table are owned only by Actions of PartType = 4.

Field	Type	Size	Description	Key
ValID	Number (Long)	4	System assigned unique identifier for this SubAction entry.	Index, ascending
PartID	Number (Long)	4	System assigned unique identifier for the Action that this SubAction entry belongs.	Foreign, ascending
DIID	Number (Long)	4	System assigned unique identifier for domain interface member from which this SubAction is derived.	Foreign, ascending
ValSeq	Number (Integer)	2	The sequence of this SubAction entry amongst those owned by the Action.	
ValStr	Memo	-	A string identifying the relevant columns in which this SubAction is invoked.	

## Table HighID

Controls the assignment of IDs within the rulebase.

**Note:** This table is for system use only.

Field	Type	Size	Description	Key
MaxID	Number (Long)	4	Maximum allowable ID in this rulebase.	

Field	Type	Size	Description	Key
HIOID	Number (Long)	4	Currently assigned highest ID	

## Table Users

Defines the level of privileges that a user may exercise in a domain.

Field	Type	Size	Description	Key
Name	Text	64	System ID of users of the rulebase.	Index, ascending, dups
DomainID	Number (Long)	4	Unique identifier for the domain on which this user has privileges.	Index, ascending, NULL, dups
Permissions	Number (Long)	4	Code defining the level of privileges this user enjoys in the specified domain.	

## Table CheckOut

Retains a record of rules that have been checked-out of the rule repository.

**Note:** This table is required only if the Dynamic Rule Manager invokes a source code control program to serve as a rule repository (not shown in database illustration).

Field	Type	Size	Description	Key
USERID	Text	64	System ID of the user who has checked out the rule.	Index, ascending, dups
RuleName	VarChar	200	Name of the rule that has been checked out by the user	Index, ascending
DomainID	Number (Long)	4	Identifier for the domain to which this rule belongs.	Index, ascending, dups
OutTime	Text	32	Time stamp: when this rule was checked out of the repository by the user.	



# Index

---

## A

- action rows • 133
  - Decision Table • 133
- adding actions • 89
  - Decision Table • 89
- algorithms • 14
  - inference engine • 14, 22
- applications • 24, 28
  - backward chaining • 28
  - forward chaining • 24
- ask( ) method • 29
  - GUIv7Lib • 29
- attributes • 45, 52, 109, 111
  - local • 45
  - TMAssignment • 109
  - TMRetracton • 111
  - UNKNOWN • 52

## B

- backward chaining • 24, 25, 27, 28
  - applications • 28
  - defined • 24
  - input/output • 27
  - reasoning • 25
  - when to use • 28
- Backward Mode • 18
  - Rule Analyzer • 18
- BackwardChain • 32
  - example • 32
- bases • 14
  - knowledge • 14
- business • 37
  - logic • 37

## C

- C/C++/C# • 32
  - a client interface built in • 32
- callbacks • 32
- chaining • 16, 19, 24
  - rules • 16
- code • 14
  - procedural • 14
- COM • 32
- Common Language Runtime (CLR) • 32
- Condition Name • 131

- Decision Table • 131
- conditions • 86
  - Decision Table • 86
- conditions rows • 131
  - Decision Table • 131
- conversational interface • 33
  - pseudo code to carry on a • 33
- conversational systems • 29
- creating decision tables • 84
- customer support • iii

## D

- data • 21
  - supplying • 21
- data-driven inferencing • 19
- Decision Table • 84, 86, 89, 95, 96, 100, 101, 103, 131, 133, 138
  - action execution • 101
  - action rows • 133
  - adding actions • 89
  - chaining • 101
  - compressing • 96
  - condition evaluation • 101
  - conditions • 86
  - conditions rows • 131
  - consolidating IFRules into • 103
  - creating • 84
  - customizing • 138
  - Decision Table Editor • 95
  - modifying actions • 89
  - opening • 84
  - options • 95
  - rule posting • 100
  - runtime execution • 100
- Decision Table Editor • 95
- dialog box • 29
  - using the interface layer • 29
- direct posting • 40
  - rules • 40
- DynaInfer • 32
  - example • 32
- dynamic • 46
  - d inferencing • 46
- dynamic decision tables • 142
  - non-persistent • 142
- Dynamic Rule Manager • 126

---

- dynamic rulebase • 115, 124
  - maintaining • 115
  - scenarios • 124
- Dynamic Rulebase Administrator • 116
- dynamic rules • 57, 102, 116, 126, 141
  - managing • 126
  - non-persistent • 141
  - persistent • 141
  - rephrasing IFRules as • 102
  - rulebase administration • 116
  - runtime considerations • 57
- dynamically-bound inference block • 35, 39

## E

- editing rules • 17
  - Rule Editor • 17
- environment • 32
  - maintaining client/server • 32
  - NET • 32
- examples • 32, 40
  - BackwardChain • 32
  - conversational interface • 32
  - DynaInfer • 32
  - rule posting • 40
- expert system • 29

## F

- forward chaining • 19, 21, 22, 23, 24
  - applications • 24
  - defined • 19
  - input/output • 22
  - reasoning • 21, 25
  - supplying new data • 21
  - when to use • 23
- Forward Mode • 18
  - Rule Analyzer • 18

## G

- generate a dialog box • 29
- goal-directed • 24
  - inferencing • 24
- GUI • 29
  - presenting a question • 29
- GUIv7Lib • 29
  - ask( ) method • 29

## I

- indirect posting • 40
  - rules • 40

- InferBegin • 39
- InferBegin( ) • 35
- inference block • 38, 39, 40
  - example • 38, 40
  - posting • 39
  - scope • 39
- inference engine • 14, 22
  - algorithms • 14
  - rules premise • 22
- inferencing • 19, 24, 46
  - dynamic • 46
  - goal-directed • 24
  - inferencing data-driven • 19
- InferEnd( ) • 35
- initiating a dialog • 30
  - using rules • 30
  - using WhenSourced( ) • 30
- input • 22, 27
  - backward chaining • 27
  - forward chaining • 22
- invokeInferencing( ) • 33

## J

- J2EE environment • 32

## K

- keyword • 16, 17, 19, 24, 37, 41
  - backward chaining • 19
  - backwardchain • 16, 24
  - defined • 17, 19, 24, 37, 41
  - END • 17, 37
  - forwardchain • 16, 19
  - ifrule • 17
  - INFER • 37
  - priority • 41
  - rule • 17
- knowledge • 43
  - bases • 14
  - location • 16
  - methods • 16
  - object-oriented • 43

## L

- Last chance sourcing • 30
- local • 45
  - attributes • 45
  - variables • 45
- location • 16
  - knowledge • 16

---

- methods • 16
- logic • 37
  - business • 37

## M

- maintaining • 32, 35
  - state of server • 35
  - x client/server environment • 32
- matching • 69
  - pattern • 69
- methods • 16
  - knowledge • 16
  - location • 16
- modifying actions • 89
  - Decision Table • 89
- multi-user environment • 32

## N

- NET environment • 32
- non-persistent dynamic rules • 141
  - defined • 141
  - facilities • 141

## O

- object-oriented knowledge • 43
- opening • 84
  - decision tables • 84
- output • 22, 27
  - backward chaining • 27
  - forward chaining • 22

## P

- persistent dynamic rules • 141
  - defined • 141
- posting rules • 39
- PowerBuilder • 32
  - x a user interface built with • 32
- premises • 22
- presenting a question • 29
  - GUI • 29
- procedural code • 14
- pseudo code • 33
  - to carry on a conversational interface • 33

## R

- reasoning • 21, 25
  - backward chaining • 25
  - forward chaining • 21

- Rule Editor • 17
- rules • 14, 16, 17, 22, 23, 30, 37, 40
  - chaining • 16, 19, 24
  - examples of posting • 40
  - forward chaining • 23
  - initiating a dialog • 30
  - overview • 37
  - satisfying premise • 22
  - simple • 17
  - versus procedural code • 14
- runtime considerations • 57
  - dynamic rules • 57

## S

- satisfying rule premise • 22
- scopes • 39
- setUserAnswer( ) • 33
- simple • 17
  - rules • 17
- SingleFire rules • 81
- state of server • 32, 35
  - maintaining • 32, 35
- stateless servers • 32
- statusResult • 35
- supplying data • 21

## T

- technical support • iii
- Test cells • 131
  - Decision Table • 131
- TMAssignment attribute • 109
- TMConfirmation attribute • 113
- TMRetraction attribute • 111
- truth maintenance • 109
  - operations • 109

## U

- UNKNOWN attributes • 52

## V

- variables • 45
  - local • 45
- Visual Basic • 32
  - a client interface built in • 32

## W

- WhenSourced( ) • 30
  - x initiating a dialog • 30

---

WinLib • 29

X

x client/server environment • 32