Neuron Data Elements Environment
Intelligent Rules Element

Version 4.0

Language Programmer’s Guide

Part Number: IR-4000-0796-00

Contents

. Preface

Purpose of this Manual v
Audience vi
How to Use this Manual. vi
Organization vi
Related Manuals vii
Conventions viii

1. Representation

Introduction 1-1
Data Structures 1-2
Meta-Slot Attributes. 1-11
- Initial Value 1-11
- Private or Public Slot 1-11
- Prompt Line 1-12
- Data Validation 1-12
- Inference Priorities 1-13
- Inheritance Priorities 1-14
- Inheritability Setting 1-15
- Inheritance Strategy 1-17

Methods. 1-18
- Structure 1-18
- Order Of Sources Method 1-23
- If Change Method 1-26

Rules 1-28
- Conditions 1-29
- Hypothesis. 1-31
- Right-Hand Side Actions 1-32

Inheritance. 1-34
- Property Inheritance 1-37
- Value Inheritance 1-45
- Method Inheritance 1-50
- Conflict Resolution 1-53

Dynamic Structures. 1-58
- Dynamic Objects 1-58
- Dynamic Links 1-59
- Inheritance 1-61

Interpretations 1-66

Interpreting Slots 1-66
Interpreting Strings 1-68
Pattern Matching 1-71
Lists Created By Pattern Matchings 1-74
Multiple Pattern Matchings In One Rule 1-76
Interpretations And Pattern Matching 1-78
Pattern Matching With Data Validation 1-79
Pattern Matching With Methods & External Routines 1-80
Knowledge Islands 1-81
Knowledge Bases 1-82
Summary 1-83

2. Inference Engine Processing

Introduction 2-1
Evaluation Concepts 2-3
Rule Evaluation 2-3
Multiple Rules Evaluation 2-5
Exhaustive Evaluation 2-8
Conflict Resolution 2-9
Inference Search Disabling 2-10
Method Evaluation 2-10
Inferencing Mechanisms 2-14
Backward 2-14
Suggest 2-16
Hypothesis Forward 2-17
Gates 2-19
Forward Action-Effects 2-21
Volunteer 2-23
Contexts 2-25
Interpretations 2-28
Pattern Matching 2-32
Conflict Resolution 2-42
Summary 2-52
Controlling Inference Strategies 2-54
Rules 2-54
Methods 2-58
Application Programming Interface 2-65
Suggest 2-65
Volunteer 2-66
Restart Session 2-67
Strategy 2-67
Non-Monotonicity 2-69

Revisions 2-69
Reset 2-70
Interpretations 2-72
Pattern Matching 2-73
Conflict Resolution 2-74
Control 2-74

Multiple Knowledge Bases 2-75
Untitled.kb 2-75
Temporary.kb 2-75
Undefined.kb 2-76
Current Knowledge Base 2-76
Modular Knowledge Base Architecture 2-77
Inferencing With Multiple Knowledge Bases 2-79
Merging Multiple Knowledge Bases 2-79
Effects When Loading 2-80
Effects When Unloading 2-82

Summary 2-83

3. Primer

Introduction 3-1
Knowledge Processing - Session 1 3-3
Knowledge Processing - Session 2 3-9
Knowledge Processing - Session 3 3-16

A. Primer Decomposition

About this Appendix A-1
Rules as Building Blocks A-1
Inferencing with Multiple Rules A-4
Inferencing Flow Control A-7
Storing Data in Objects A-10
Reinitiating Inferencing A-12

B. Primer KB Text Format

About this Appendix B-1
Data Type Listing B-1
Rule Listing B-2
Class and Object Listing B-4
Meta-Slot Listing B-7
Method Listing B-9
Strategy Listing B-12

C. Primer.dat Scripts

Application Script. C-2
Start.Win C-4
Start.Win2 . . . . C-6
Start.Help C-7
FormInput.Win. C-8
End.Win. C-9

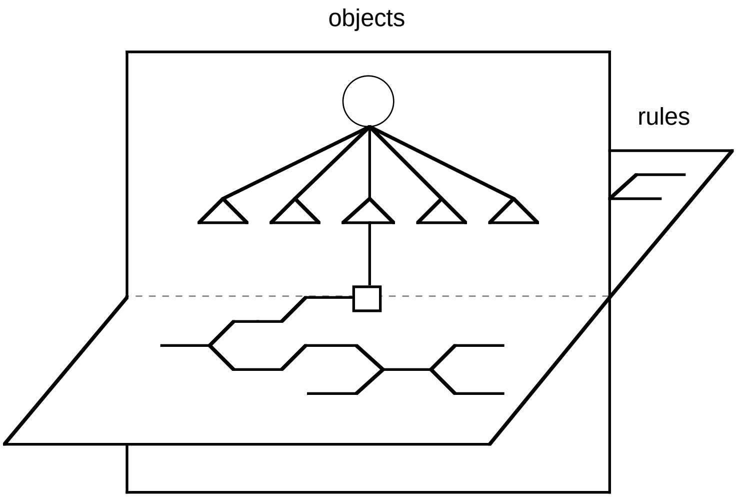
. Index

Preface

Purpose of this Manual

This manual explains the basic concepts of the Intelligent Rules Element and demonstrates the concepts through a tutorial example. The Rules Element is a general purpose knowledge-based application development tool. It provides a friendly graphical user interface to help you create a knowledge-based application, a rich set of data structures to represent the domain knowledge, and a powerful inference engine to complete tasks in the domain.

The Rules Element is also a hybrid tool, which means it integrates rules and objects as well as many additional features:



This manual does not dwell on the syntax and all of the lowest level features. Rather, it first explains the various application structures in detail, and then it explains how the Rules Element inferencing mechanisms process the application structures. This processing, which is the intersection of the rule and object planes, represented by the figure above, is the heart of the Rules Element.

Audience

This manual is for application developers who need to know how the Rules Element works. The manual contains two major parts as follows:

- Part One - provides an in-depth explanation of the various features of the Rules Element using both conceptual drawings and graphics from the development environment.
- Part Two - provides hands-on exercises that demonstrate the concepts described in Part One.

This manual does not assume any knowledge of the Rules Element, programming, or of AI terminology and techniques in general. A brief introduction to Rules Element concepts can be obtained from the Elements Environment Getting Started manual and familiarity with this material will facilitate understanding of the more detailed discussion presented this book.

How to Use this Manual

This manual can be used in one of two ways. The manual can be read from beginning to end to learn how the Rules Element works. Each section builds on the ideas from previous sections so this approach makes sense.

However, once you understand the fundamentals of how the Rules Element works, it can be used as a reference manual. Specific points are explicitly described to allow easy random look-ups at any time in the future.

The exercises in Part Two of this manual, while not exhaustive, should be carefully read and understood to ground the understand of the concepts presented in Part One. The appendices give important supplementally information that applies to both Parts One and Two of this manual.

Organization

To locate specific subjects, refer to the general table of contents, the chapter table of contents, or the index. This manual has three chapters and three appendices:

Chapter One, "Representation," describes each of the Rules Element's representation structures. It begins by explaining the Rules Element's object-oriented structures, including objects, classes, and methods. It then describes how rules are used. The chapter concludes by describing more advanced object-oriented features, such as inheritance and dynamic structures, as well as knowledge islands and knowledge bases which are macroscopic organization structures.

Chapter Two, "Inference Engine Processing," explains how the Rules Element inference engine processes the application representation structures. It describes the basic elements of the agenda, a dynamic, priority-based scheduling system. It then explains how some advanced features affect the agenda including: interactions with the external environment via the application programming interface, non-monotonic reasoning, and working with multiple knowledge bases.

Chapter Three, "Knowledge Base Processing" takes you on a tour of the Rules Element's dynamic behavior while processing a small primer application designed for this manual. The example provides step-by-step actions to load and run the primer while accessing the facilities of the Rules Element and its companion front-end development tool, Open Interface Element.

Appendix A, "Primer Decomposition" explains the concepts that underlie the knowledge structures found in a typical Rules Element application. Specific examples are drawn from the primer knowledge base used in Chapter Three.

Appendix B, "PrimerKB Text Format" gives the text format listing of the primer knowledge base used in Chapter Four. The listing includes comments that help to clarify the knowledge base.

Appendix C, "Primer.Dat Scripts" gives a listing of the primer graphical user interface scripts used in Chapter Four. The listing help clarify the role of the script language in the development of knowledge-based applications.

Related Manuals

The following manuals contain information related to the Intelligent Rules Element Language Programmer's Guide. Read prerequisite manuals before reading this manual. Read corequisite manuals for background information as explained.

Prerequisite Manuals:

Getting Started

This manual is an overview of the entire Neuron Data Elements Environment. It contains a chapter that describes the Rules Element shell, including the development graphical user interface, the inference engine, and application structures. Many of the concepts described in the Language Programmer's Guide are first introduced in this manual.

Corerequisite Manuals:

User's Guide

This manual gives general procedures for using the development graphical user interface. It explains how to use each of the editors, networks, menus, etc. It also explains the application development process, from implementation to editing to documenting to processing, and concluding with testing. Many of the structures described in the Language Programmer's Guide can be created by referring to this manual.

This manual also explains how to integrate relational and flat file databases with your application. Database interactions can have a profound impact on the agenda since each retrieval corresponds to a multiple volunteer.

Language Reference Manual

This manual is the application developer's reference guide to the Rules Element tool. It explains the operators of the Rule Language and shows the correct syntax to use. Look up topics in the Language Reference Manual when you want to know more about individual structures.

C / C++ Programmer's Guide

This manual describes how to integrate the Rules Element within an application framework using either the C or C++ programming language. It gives a complete description of the C or C++ application programming interface which allows you to, among other things, investigate working memory, volunteer values, and suggest hypotheses.

The Bibliography, located in the Getting Started manual, gives a complete list of manuals.

Users who received the Intelligent Rules Element along with other Neuron Data Elements, including the Open Interface Element and the Data Access Element, will have other documents in addition to the Intelligent Rules Element documents described above.

Conventions

When we speak of a particular data structure, such as "`employee_1`", it will appear in the Courier font. Reserved words, such as `TRUE` and `FALSE,` will be written with all caps and also appear in Courier font. The appropriate sections will introduce specific graphic representations for knowledge structures.

Intelligent Rules Element and Rules Element are synonymous and we will use them interchangeably.

Chapter 1

Representation

This chapter describes the Intelligent Rules Element structures you will use to describe or “represent” the application domain. It also describes the relationships you can create between these structures:

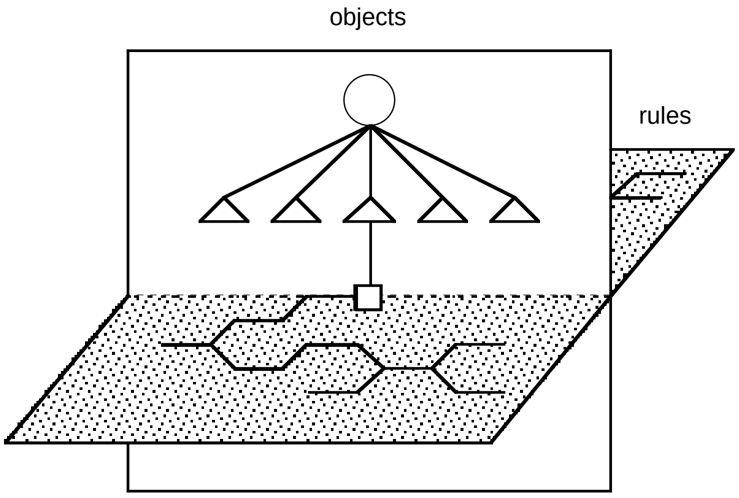


Figure 1-1 The Object Plane

Chapter Two, “Inference Engine Processing” will explain the focus of attention which is the intersection of the object and rule planes.

Introduction

The Rules Element provides you with many representational structures. There are objects and classes to describe the entities in the domain. There are properties which are characteristics of objects and classes and slots which store information about specific objects and classes. There are also meta-slots which describe how the slots behave.

Properties can be inherited from a class or object to another class or object. Values can also be inherited from a class or object to another class or object. Certain meta-slots can be inherited from a class or object to another object. Inheritance allows efficiency, as the particular attribute only needs to be declared in one place, it provides consistency as everything which inherits an attribute behaves in the same way, and it provides generality.

In addition, the Rules Element allows you to create objects dynamically during a session. These dynamic objects allow you to model a world whose exact structure isn't known a priori (for instance how many records are in a database). You can also create dynamic links between objects or classes and other objects or classes to reflect changing relationships during processing.

The Rules Element supports rules which contain all of the domain knowledge. Rules manipulate the slots as well as the object and class structures. Pattern matching and interpretations allow you to reference objects which are determined at runtime. Thus you can write generic rules which reason on a set of objects which are determined when the rule is processed.

In conjunction with rules, the Rules Element supports methods and message passing to provide heuristics that are object-oriented in nature. Methods can be triggered explicitly after receiving a message from a rule or other method, or they can be triggered automatically following a determination made by the system. Method heuristics act like routines that operate entirely in the object domain on specific slots, objects, or sets of objects.

The Rules Element supports multiple inheritance. Properties, values, methods, and some of the meta-slots can be inherited down the object hierarchy. Inheritance up the object hierarchy is supported for properties and values only. You can create dynamic objects as well as dynamically modifying the relationships between objects and classes, thus allowing objects and classes to inherit from different parents at different times.

Data Structures

This section focuses on a description of the basic object-oriented features of the Rules Element. The Rules Element describes the world in terms of objects, generalizations of those objects called classes, characteristics of objects and classes called properties, and slots which store information about particular objects and classes.

This section describes each main representational mechanism, introduces a graphical representation of it, and gives examples of each where appropriate. The subsequent sections detail how each of these mechanisms interact with each other.

Object

An object is the smallest chunk of information in the knowledge-based system. It represents any person, place, thing, or idea in the domain for this particular application. You describe your application's world in terms of various objects. For instance, in a petroleum application, each particular well is an object, as well as all the components needed to produce each well.

Objects are represented in this document by the triangular icon depicted in Figure 1-2:

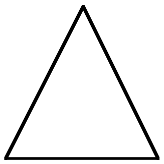


Figure 1-2 An Object

Class

You could describe the whole world in terms of objects, but before long you would realize that many objects have common features, behaviors, etc. For instance, in an insurance application, each client's application for insurance may be an object, but one begins to see lots of very similar objects (eg. each of the other applications). Thus the Rules Element has the notion of a class of objects, or, in this case, a class of insurance applications. A class is merely a grouping or *generalization* of a set of objects. Objects are specific members or *instantiations* of a class.

Classes are represented in this document by the circular icon depicted in Figure 1-3:

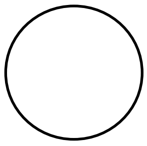


Figure 1-3 A Class

Objects may belong to several classes, such as `car_insurance_application` is both a member of the class `Documents` as well as the class `Insurance_applications`:

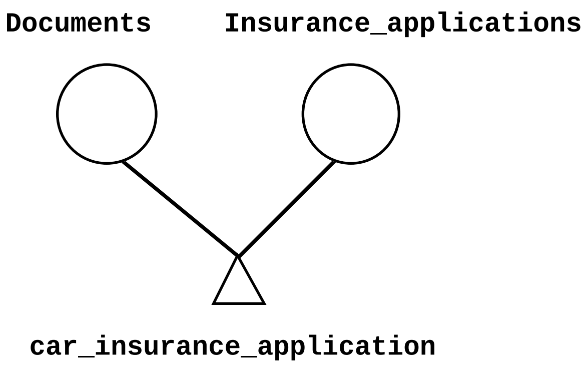


Figure 1-4 An Object Belonging to Two Classes

The classes will be referred to as the *parents*, and the object will be referred to as the *child*. We will also say that a *link* exists between the child object and the parent classes.

Since classes may also have many objects, there is the possibility for many to many relationships.

Subclass

A subclass is a class which represents a subset or *specialization* of another class. It is a class in its own right and has all the characteristics of other classes. For instance, `Math` could be one class with `Algebra`, `Geometry`, and `Calculus` subclasses and a particular course, such as `math_101`, an object which belongs to both `Algebra` and the parent class `Math`. Since a subclass is also a class, it will also be represented by the circular icon:

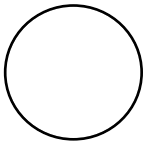


Figure 1-5 A Subclass

Classes can have any number of subclasses or parent classes or both. You can create a class hierarchy with any number of levels.

Subobject

Classes and subclasses add to your ability to describe a particular domain, but often you would like to express another type of distinct relationship between objects which aren't instantiations of each other but are neither completely disjoint. Subobjects represent a relationship of the type "is a part of."

Since a subobject is also an object, we will represent it by the same triangular icon:

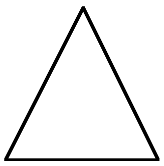


Figure 1-6 A Subobject

A subobject (which is also an object in and of itself) represents a part of another object, as `modem` is a part of `computer_x`, and `computer_x` is an instantiation of the class `Computers`:

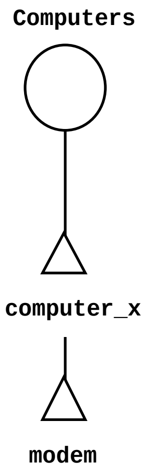


Figure 1-7 An Object Hierarchy

Two objects linked by this type of relationship usually do not share many characteristics. For instance, the characteristic of `memory_capacity` and `microprocessor` are shared by the object `computer_x` and other members of the class `Computers`, but these characteristic are irrelevant to `modem` as a subobject of `computer_x`. Since subobjects are also objects, they may belong to classes, as `modem` is both a part of `computer_x` and a member of the class `Communication_tool` and the class `External_devices`:

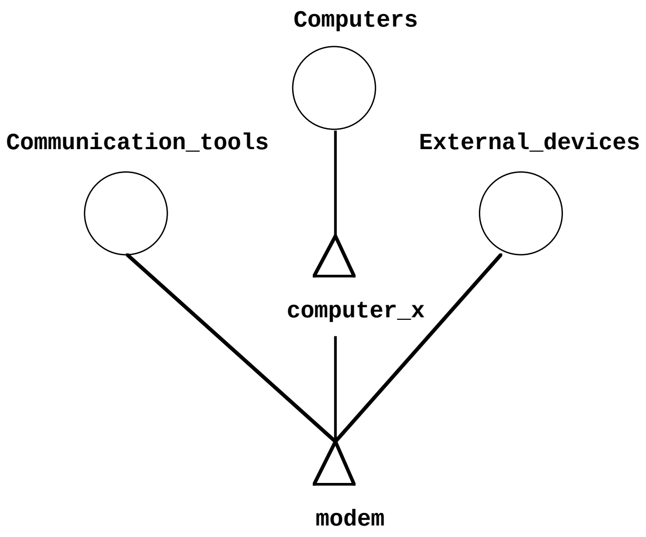


Figure 1-8 A Subobject with Parent Classes

Analogous to the other object and class relationships discussed previously, an object may have any number of subobjects, and may be a subobject of any number of other objects.

Property

Once again, it's possible to represent the domain you wish to reason on with objects and classes alone, but often you need to describe the objects and classes. Properties describe these objects and classes.

Properties have a particular data type: they can be *string*, *integer*, *.float*, *boolean*, *date*, or *time*. They can also be multi-valued. Some example properties are: weight, color, value, time, and so on. You can use any number of properties to describe an object or class. You may also have objects and classes without properties.

Properties are represented in this document by the rectangular icon depicted in Figure 1-9:

Figure 1-9 A Property

It is important to note that while objects and classes may have specific properties, these properties are not limited to any one object or class. Thus other objects and classes can have the same property. Furthermore, since the property is independent of the object or class, it will always have the same data type throughout the knowledge base. Thus if one object has a property `force` which is of the data type *float*, then any other object or class which has the property `force` must also use it as a *float*. This helps ensure consistency throughout the system.

Note: There is one notable exception to the way properties behave. The special property "Value" can have (and usually does have) different data types when it is attached to different objects or classes. Thus the Value property of the `stock` object may have a *float* data type, while the Value property of the `magna_carta` object may have a *date* value type, and other objects using the value property can have any other data type. See the next section for more details on the Value property.

Slot

Slots are used to store property values for objects and classes. Thus they hold all of the information in the application. Any information which comes into the knowledge-based application, whether it comes from a database, from the user, from any external program, or is generated internally, is stored in slots.

Slots are properties which are attached to objects. The simplest type of slot, which uses the `Value` property, is generated automatically for you by the Rules Element inference engine. For example, assume you have three objects: `assets`, `liabilities`, and `owners_equity`. When you assign a value to an object, the inference engine creates a special property of that object, called "Value", which stores that information. This information can then be referenced at any time in the future.

Assume now that a value for `liabilities` and `assets` has been determined (the value of `liabilities` and `assets` could have been determined in any manner):

```
liabilities = 2000
owners_equity = ?
assets = 3500
```

If there is an expression which assigns the difference between `assets` and `liabilities` to `owners_equity` (disregard the syntax for now),

```
Assign     assets-liabilities     owners_equity
```

The Rules Element inference engine creates a Value slot for `owners_equity` as well and assigns it the difference between `assets` and `liabilities`. A slot can also be any other property of an object. In addition to the value slot, `assets` might also have a slot which is `average_turnover`, another which is `fixed` and so on. When slots are represented in the object hierarchy in this document, they will be represented by the square icon depicted in Figure 1-10:

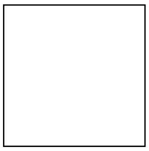


Figure 1-10 A Slot

In text, the Rules Element represents a slot as the object or class name, then a period, and finally the property name. Thus the slots described above are represented in the Rules Element as `assets.value`, `assets.average_turnover`, `assets.fixed`, `liabilities.value`, and `owners_equity.value`.

It is also important to note that in any expression, `object` and `object.value` are completely equivalent. For example,

```
Assign    assets.value - liabilities.value    owners_equity.value
```

has the exact same effect as

```
Assign    assets-liabilities    owners_equity
```

An object or class can have any number of slots. The number of slots an object or class has is precisely equal to the number of properties the object or class has. A slot also has a set of characteristics that determine how it will be used by the system, together these characteristics are know as meta-slot attributes and will be described in the next section of this chapter. For now it is useful to note that one of these meta-slot attributes controls the accessibility of the slot's data value. In this regard, slots that you create can be one of two types: "private" when data protection is desired or "public" when restricted data access is not required.

This document depicts the slots belonging to a particular object or class in a similar manner to how it depicts the relationships between objects and classes:

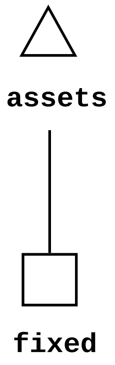


Figure 1-11 An Object/Property Slot

A slot, no matter what data type, initially has the value `UNKNOWN`. This means that the inference engine has not tried to determine a value for the slot. When the inference engine tries to determine the value for a slot, one of two things can happen:

- The inference engine finds a value ==> the slot takes whatever value has been determined.
- The inference engine does not find a value ==> the slot takes the value `NOTKNOWN.`

Summary

Application builders represent their domain in terms of objects, generalizations of objects called classes, parts of those objects called subobjects, descriptions of the classes and objects called properties, and finally properties of particular classes or objects called slots.

Objects do not have values. They have relationships to other objects, classes, and properties. Properties also do not have values, but they do have a particular data type, and this data type is constant throughout the application regardless of which objects or classes are described by them. Slots, which are a construction consisting of the particular property of an object or class, have the properties data type and hold a particular value. The data value may be protected in the case of private slots and and accessible without restrictions in the case of public slots.

We will use the term *object hierarchy* to refer to a graphic representation of the relationships between any combination of the above application structures. Here is a representative object hierarchy:

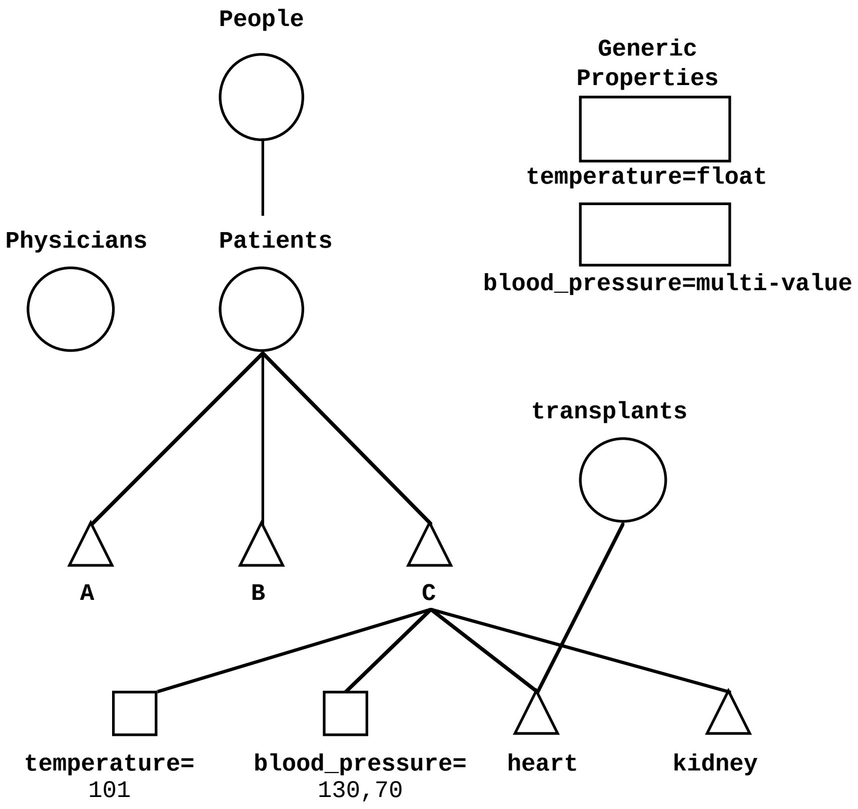


Figure 1-12 Sample Object Hierarchy

In this example, we have a parent class `People` and two subclasses named `Physicians` and `Patients`. The class `Patients` has three objects: A, B, and C. The object C has two slots, `temperature` and `blood_pressure` and two subobjects `heart` and `kidney`. Finally, the slot `C.temperature` has a value of 101 and the slot `C.blood_pressure` has a value of 130,70. Notice also that the generic properties `temperature` and `blood_pressure` have data types *float* and a *multi-value* consisting of two integers, respectively, but do not and cannot ever have a value. Particular slots have values, but properties don't have values. This hierarchy could be greatly expanded, but it should give you the general idea of how the Rules Element represents the world.

Meta-Slot Attributes

The meta-slot lets you customize certain characteristics of the individual slot. Meta-slot attributes span a broad range including:

- Whether the system will initialize a slot with a known value.
- Whether the slot will be private or public.
- How the system will ask the user for their value when it is needed.
- What values the system will accept when it is needed.
- What types of inheritance strategies to use (breadth versus depth-first, class versus object-first).
- What type of inheritability is used to obtain a property and value for the slot.
- What type of inference and inheritance priority it should have.

Most meta-slot attributes have default settings. Thus you can design a complete application without creating a meta-slot for each existing slot in your application. However, the default settings can be modified to customize system attributes of the slot for your application.

Initial Value

The initial value attribute provides an initialization value for the slot to begin knowledge processing with. A slot is initialized with the value immediately after the knowledge base file is loaded or after restarting a session. The initial value may be inherited down depending on the inheritability strategy selection of the meta-slot. If inheritability of the initial value is enabled, the system automatically propagates the value to the children at the start of the session. This lets you make use of known values that can be modified as required during knowledge processing.

Private or Public Slot

The slot that you create can be either public or private. The slot privacy attribute determines how the slot value will be accessed during application processing. When unrestricted access to the slot's value is desired, you will want to use a public slot. If you want your application to rely on object-encapsulation, however, you will want to use a private slot to restrict access to the slot's data value. By definition private slot values are accessible only by the specific method associated with the slot's object components (class, object, or property). This restriction means that private slots are only useful in applications that employ message passing in order to trigger object-associated methods. Using the object-encapsulation approach with private slots allows you to be sure that no part of the application will modify the private slot value other than the object's associated method. Public slots have no such restriction and can therefore be accessed freely from rules that test the public slot's value. Naturally rules, in contrast to methods, do not provide object encapsulation since data values may be modified throughout the rule-based application. In the Rules Element, however, it is possible to use both rules and object / methods to combine both approaches and provide as much encapsulation as desired. For more information about methods, refer to the Methods section in this chapter.

Prompt Line

The prompt line attribute specifies what question the Rules Element should ask the user when it needs to query the user for the value of the slot. The default prompt the Rules Element uses is: "What is the *property* of *object_or_class* ?" where the property of the current slot replaces *property*, and the object or class in question replaces *object_or_class*. Thus if the Rules Element needs to determine the value of the slot `car.speed`, it will prompt: "What is the speed of car?"

The prompt line attribute is a text string which replaces this default behavior. Thus you could specify a prompt line of "How fast does your car go?" and this question would be used instead of the default, "What is the speed of car?" As an option, you can also replace the standard window used to display the prompt line in the Rules Element with one that you create in the Open Editor facility of the Rules Element. To display the prompt line in a custom window, you can retrieve the string through the Rules Element Application Programming Interface (API) or the Rules Element scripting language.

The prompt line may or may not contain interpretations (see the section on interpretations for more details). Prompt Line meta-slot attributes are automatically inherited down.

Data Validation

The data validation attribute determines whether or not the system will accept the value of the slot once it is supplied. The data validation attribute is triggered by the system whether the value is supplied by the user in response to a query or the value is supplied by the system through an internal calculation. Valid entries can be specified as a range of numeric values, a list of strings, a complex boolean expression, or an external function.

For example, the expression `SELF.width > SELF.height` tests whether the value of the current object's width is greater than its height. If the value supplied, causes the expression to fail, the system can optionally reject, accept, or ask for a retry. The data validation attribute may or may not contain interpretations (see the Interpretations section for more details). Data Validation meta-slot attributes are automatically inherited down.

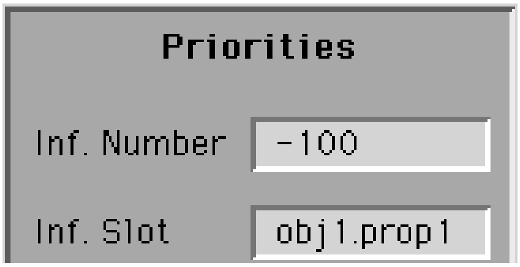
Inference Priorities

The inference mechanism as well as how and why the inference engine processes slots are explained in Chapter Two, "Inference Engine Processing." Briefly stated, when the Rules Element does need to process slots to evaluate a hypothesis or perform pattern matching in a condition for example, it will do so according to their *inference priorities* (highest first). By default, all slots have an inference priority of 1. This value can be changed to be any integer between -32000 and 32000. Inference priorities you choose will determine the order of slot evaluation for pattern matching, conditions, and hypotheses.

In addition, inference priorities can be dynamically changed. This allows slots to be processed with different priorities at different times, which allows the application to better adapt to changing conditions. This is done by assigning an *inference slot* to a slot's inference priority. The system dynamically processes the inference slot (which is, in fact, an integer slot) when it needs to obtain the inference priority.

If an inference slot has been declared and it currently has a value (the Rules Element will not prompt for the value of an inference slot), then the slot in question's *inference number* will be the value of the inference slot. If an inference slot hasn't been declared or the declared slot's value is `UNKNOWN`, then the inference engine will use the value of the inference number. If the inference number hasn't been modified, then the default inference priority is 1.

For example, assume there is a slot `a.p` which has the inference priority depicted in Figure 1-13:



Inf. Number: -100

Inf. Slot: obj1.prop1

Figure 1-13 Inference Priorities

When the inference engine needs to process the slot `a.p`, it determines its inference priority by the value of the inference slot `obj1.prop1`. If `obj1.prop1` has a value, then its value is used as the inference priority. If it doesn't have a value, then the inference number (-100) will be used.

In summary:

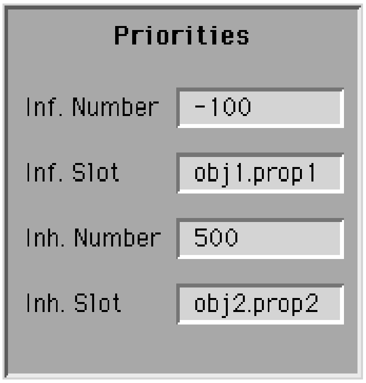
- If the inference slot has a value ==> use the value as the inference priority.
- If the inference slot does not have a value (or hasn't been declared) but an inference number has been declared ==> use the number as the inference priority.
- If the inference slot does not have a value (or hasn't been declared) and the inference number has not been declared ==> use 1 as the default value.

Inheritance Priorities

The Rules Element supports multiple inheritance (inheritance is explained in the inheritance section). This means that the target of an inheritance event (a slot) may have several different sources from which to inherit a value or method. If there are two or more parents (or children) which are at the same level in the inheritance space from which a particular slot can inherit, then the inheritance priority will determine which parent (or child) is used.

Slots with higher inheritance priorities are inherited from before slots with lower priorities. By default, all slots have an inheritance priority of 1. Similar to the inference priorities, you can change this priority to be any integer between 32000 and -32000.

In addition, there is also an inheritance slot which is completely analogous to the inference slot. This slot is an integer slot. If an inheritance slot has been declared and it currently has a value (the Rules Element will not prompt for the value of an inheritance slot), then the inference engine will use the value of the slot in question's inheritance slot for conflict resolution. If an inheritance slot hasn't been declared or its value is UNKNOWN, then the inference engine will use the value of the inheritance number. If this hasn't been modified, then the default inheritance priority is 1. The inheritance priorities are set just below the inference settings in the Meta-Slot editor:



Inf. Number -100
Inf. Slot obj1.prop1
Inh. Number 500
Inh. Slot obj2.prop2

Figure 1-14 Inheritance Priorities

In summary:

- If the inheritance slot has a value ==> use the value as the inheritance priority.
- If the inheritance slot does not have a value (or hasn't been declared) but an inheritance number has been declared ==> use the number as the inheritance priority.
- If the inheritance slot does not have a value (or hasn't been declared) and the inheritance number has not been declared ==> use 1 as the default value.

Inheritability Setting

In addition to setting inheritance priorities which determine how the slot will compete with other slots when children or parents want to inherit from it, the inheritability meta-slot also determines whether or not a slot can be inherited at all.

There are global inheritance defaults which are explained in the Inheritance section. These defaults determine what can be inherited for the vast majority of the slots. However, some slots may display a behavior which is unique to that one particular slot. This behavior can be set through those slots' inheritability meta-slot attribute:

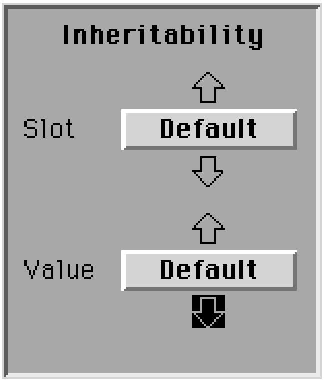


Figure 1-15 Object Inheritability

The darkened arrows show strategies which are enabled and the white arrows show strategies which are disabled. Figure 1-15 shows the default behavior for an object slot as follows:

- Inherit a value down to a subobject (when the subobject has the same property.
- Do *not* inherit a value up to a parent object.
- Do *not* inherit a property up to a parent.
- Do *not* inherit a property down to a subobject .

By default, classes display the same settings except they inherit properties down to their subclasses and objects. Thus the slot down arrow is darkened as well:

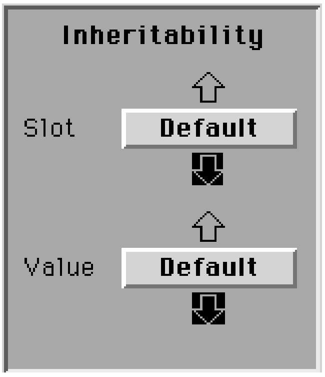


Figure 1-16 Class Inheritability

Inheritance Strategy

The inheritance strategy meta-slot determines the breadth-first depth-first and class-first object-first types of conflict resolution. The default inheritance strategy is class-first, breadth-first as depicted in Figure 1-17:

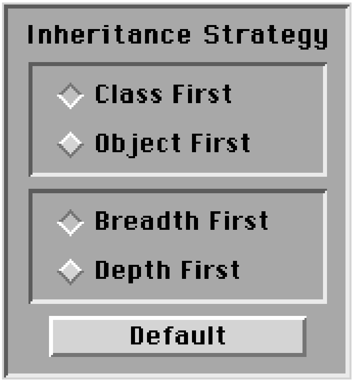


Figure 1-17 Inheritance Strategy

To change this merely click on the checkboxes of the Inheritance Strategy component corresponding to a depth-first search or an object-first search.

Methods

Methods describe the behavior of individual slots, objects, or sets of objects. Methods are composed primarily of a set of actions which when executed modify the behavior of the object upon which they act. The type of behavior specified by methods belongs to one of three categories:

- Methods that are triggered from a rule or method by the SendMessage operator (user-defined).
- Methods that find the value of a slot (Order of Sources).
- Methods that react if the value of a slot changes (If Change).

The first category lets you explicitly execute the method from a rule or other method. This category is called the *user-defined methods* because they serve whatever purpose is required by the objects upon which they act. Order of Sources and If Change methods are often referred to together as *system methods* because they both let you specify lists of actions to modify the default behavior of the Rules Element inference engine. Unlike user-defined methods, system methods are not executed explicitly; rather they are executed by the system under the appropriate circumstances.

Note: All types of methods can be inherited. Inheritance allows you to define methods at the class level and have subclasses or objects use them or define them at the object level and have subobjects use them. As with other forms of inheritance, this capability provides both consistency and genericity (more about this in the Inheritance section).

Structure

Methods have five basic parts:

- Name of the method itself
- Name of the object to which the method is attached
- Actions list
- Conditions list (optional)
- Arguments template.

The method name and the object name allow the system to bind a message sent by a rule (or method) with the specific method to be executed at runtime. The conditions represent a series of tests to determine whether or not to execute the actions. The actions specify the end-result of a method. Actions can be executed without conditions by omitting the conditions list within the rule. Unlike rules, methods are not required to have conditions. The arguments template specifies the characteristics of arguments that may be passed to the method for processing as local arguments in the list of actions and conditions.

Note: Unlike public slots, private slots have restrictions on their usage in method conditions and actions as described in the following sections.

Method Name

A name is a required part of any method. The system uses the method name in part to determine which method will be triggered. For example, an object in a mail order system, might have several methods attached: one that calculates the mailing cost, one that adjusts the inventory, and one that prints the customer's invoice. In this case, the object receives a message to trigger a method, but the message is not complete unless it includes the name of the particular method.

One helpful aspect of methods is that their names do not need to be unique within the system. In our mail order example, a variation of the method that calculates mailing cost could be created in order to use a different cost formula for each class and yet all of these methods can have the same name. The system knows which method is which because it only considers a method in the context of the object to which that method is attached. Furthermore, two methods that share the same name, may be attached to the same object, as long as one is specified as Private and the other as Public. This allows you to choose a single, well understood name for a set of related methods, rather than choose a name for each method that identifies its particular variation.

Attached Object

The system tries to trigger a method after a message that includes a method name is received by a specific object. This means that in order for the system to successfully trigger a method, the objects of the system must know about the methods. In the Rules Element the relationship between objects and methods is established in the method's "Attach To" field of the Method editor. The field can be the name of a particular class, an individual object, a specific slot, or even a property.

Because methods can be inherited down from parent objects or classes, the object named in the "Attach To" field serves another important function. The named object literally attaches the method to a certain position in the object hierarchy. Therefore, the object that receives a message to trigger a method need not have the method attached directly to it if it can inherit a method of the same name from an including class or parent object. This mechanism is more fully described in the Inheritance section.

Actions

A method is primarily a list of actions that operates on the object hierarchy. A method's actions are triggered by a message sent to the object to which the method is attached (or inherited). The triggered method's list of actions may be executed immediately or following the evaluation of test conditions. If conditions are present, the actions can be divided into two separate lists to be executed depending on the evaluation outcome of the conditions. The entire list of actions is normally executed from top to bottom, although the Order of Sources system method provides an exception to this behavior.

In its most pure form, a method might produce side-effects that change the value of a slot somewhere in the system or it might only return a value. This distinction is not absolute though since the actions of a single method might perform both simultaneously, depending on the operators chosen. Table 1-A shows the operators which are available for use in methods (see the Intelligent Rules Element Reference Manual for a precise definition and usage of each of the actions).

Method Operator	Conditions	Actions
`=,<>,>,<,>=,<=`	X	
`AskQuestion`		OS only
`Assign`	X	X
`Backward`		OS only
`CreateObject`	X	X
`DeleteObject`	X	X
`Execute`	X	X
`InhMethod`	X	X
`InhValueDown`		OS only
`InhValueUp`		OS only
`Interrupt`	X	X
`LoadKB`	X	X
`Member`	X	
`No`	X	
`NoInherit`	X	X
`NotMember`	X	
`Reset`	X	X
`Retrieve`	X	X
`RunTimeValue`		OS only

`SendMessage`	X	X
`Show`	X	X
`Strategy`	X	X
`UnloadKB`	X	X
`Write`	X	X
`Yes`	X	

Table 1-1 Method Operator Usage

Table 1-1 shows which operators can be used in method conditions and which ones can be used in method actions. It also shows that certain method operators are not available for use by every category of method. Since user-defined and If Change methods are not used to actually obtain a value for the slot in question, some of the Order of Sources operators aren't necessary (including AskQuestion, Backward, InhValueDown, InhValueUp, and RunTimeValue). Otherwise, user-defined methods and If Change methods have the same list of operators.

Because the data of a private slot is not intended to be accessed by any structure other than the method associated with the slot, private slot names cannot appear as data to be acted on in the actions of other methods. Private slots can however be used in their own method by the SELF variable as described in the Method Inheritance section.

Conditions

Since a method is primarily a set of actions to be executed when an object receives a message, the event that triggers the method usually presupposes the need for the method. However, conditions can still be of use in a method. For example, pattern matching on the conditions list might be used to pass a set of values to the actions list to act on. If present in a method, the conditions are always executed before the actions.

Method conditions permit the application developer to specify two separate lists of actions: one to be executed if all the test conditions are found to be TRUE, the other list to be executed if at least one condition evaluates to FALSE. Conditions in methods are analogous to those used in rules, with the exception that method conditions do not produce goal-generation. For more information about conditions, refer to the Rules Conditions section.

When the inference engine evaluates the conditions in a method, it evaluates the conditions in order from top to bottom. Unlike rule conditions, changing the inference priority on the data of a method condition does not effect the order of evaluation.

Because the data of a private slot is not intended to be accessed by any structure other than the method associated with the slot, private slot names cannot appear as data to be acted on in the conditions of other methods. Private slots can however be used in their own method by the SELF variable as described in the Method Inheritance section.

Local Arguments

The actions and conditions of methods can process arguments that are passed to the method at runtime. The argument the method binds with may be a slot value, an object, or a class, or a list. Because the actual value of the local argument is usually determined at runtime, your list of actions and conditions must refer to each argument that it expects to receive by a placeholder name. Argument templates stored with the method identify the characteristics of each these placeholders. Arguments that get passed to the method, must match the characteristics of one of the argument templates or they will not be processed locally. The argument template has the following syntax:

```
MethodName (@NAME=)
Argument1(@ARG1=;@NATURE=;@TYPE=;@DEFVALUE)
Argument2(@ARG1=;@NATURE=;@TYPE=;@DEFVALUE)
...
```

The @ARGx is the placeholder name which your list of actions and conditions use to refer to the local argument (the name must start with an underscore character "_"). @NATURE is how the argument is passed, either by reference or by value. @TYPE is the data type when the argument is a slot (as opposed to an object or a class). @DEFVALUE is a default value that the inference engine will adopt in the event the argument is not passed to the method.

Arguments can be passed to the method by reference or by value. If they are passed by reference, the actions or conditions that process the local argument can modify the value of the original slot. If they are passed by value, the slot value is copied locally in the local argument and modifying the local argument will not affect the original slot. The nature of the local argument is determined by the argument template stored with the method itself.

Because the data of a private slot is not intended to be accessed by any structure other than the method associated with the slot, private slot names cannot be passed to other methods by reference. Private slots can only be passed from their own method by value and the private slot name must use the SELF variable as described in the Method Inheritance section.

Order Of Sources Method

The Order of Sources method determines where a slot will get a value when it is needed. The Order of Sources method contains a list of actions to perform to determine the value of the slot. These actions are performed in order, from top to bottom. If a value is found at any point in the list, then the rest of the sources are disregarded. When the inference engine needs the value of a slot, it will perform the following series of actions.

1. Check the Order of Sources (OS) method attached to the public slot.

   If any OS is written for the slot, then the system executes the actions in the OS sequentially, from top to bottom, until a value is determined. If the slot's own OS list fails to find a value, skip to step 4.

2. If the public slot has no OS attached, then check the parent's OS.

   If parent `object.property` or `class.property` has an OS attached, the slot will inherit the method and execute it as its own. If the slot's inherited OS list fails to find a value, skip to step 4.

3. If the public slot's parent has no OS attached, then use the default OS strategy:

   3a. If the slot is a hypothesis, use backward chaining to evaluate the hypothesis. If this fails, go to step 3b.

   3b. Inherit the value *down* from a parent or inherit the value *up* from a child. If this fails, go to step 4.

4. If the public slot's value is still not determined after completing steps 1, 2, or 3, then the system prompts the user to enter the value of the slot.

Generally, private slots will not need an Order of Sources method since the system will not try to determine a value for a private slot through the course of knowledge processing. An exception to this occurs if your application explicitly triggers an Order of Sources method for a private slot.

Figure 1-18 depicts steps 1, 2, and 3 when the system tries to locate the value "English" for the slot `moby_dick.language`.

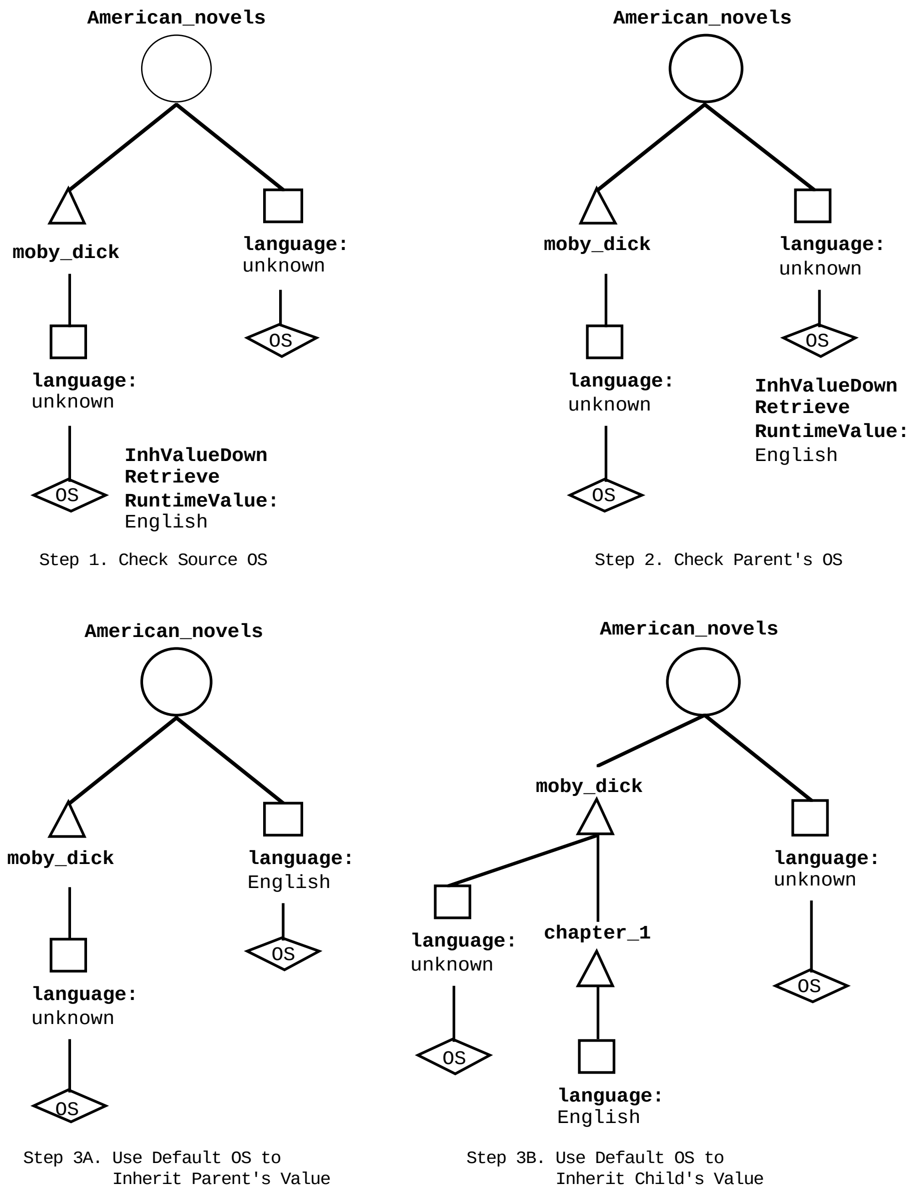


Figure 1-18 How a Slot Value is Obtained

As seen above, when the inference engine needs the value of a slot which doesn't have a value (therefore its value is `UNKNOWN`), it will use an Order of Sources method, which is a list of different possible sources, to try to get a value. It first tries to use an Order of Sources method declared locally, then one which can be inherited, and finally the default Order of Sources as follows:

- Backward chain
- Inherit value down
- Inherit value up.

The inference engine tries to execute one of the three possible lists of sources. If the list of sources all fail, whether they are a local method, an inherited method, or the default sources, then the inference engine will prompt the user for the value of the slot.

As an example, assume the inference engine needs to know the value of the slot `moby_dick.language` and no Order of Sources method has been attached to the slot `moby_dick.language` or to one of its parents. This means the slot will use the default sources list as follows:

1. The Rules Element inference engine first tries to backward chain on `moby_dick.language`. Since it is not a hypothesis, this source will fail (for more information about backward chaining, see Chapter Two, "Inference Engine Processing").
2. Then `moby_dick.language` will next try to inherit a value down from a parent object or class. If a parent has a language value, then `moby_dick.language` will use that language.
3. If not, then `moby_dick.language` will try to inherit a value up. If this succeeds, then `moby_dick.language` will use it.
4. If not, then all of the default sources have failed, so the inference engine prompts the user for the value.

As mentioned above, these are only the default Order of Sources. You can define any other sources you want in any order. The important point to remember is that as soon as the inference engine determines a value for the slot, the inference engine will exit the method and the rest of the sources in the list will be disregarded, although even this default behavior can be controlled through the Strategy dialog window by selecting the ON / CONTINUE option.

In addition to the default sources, some of the other possible sources include:

Initial Value	This meta-slot attribute initializes the slot to a particular value when the knowledge base file is first loaded and propagates the value to the

	children as specified by the value inheritability setting. This source is unlike any of the other sources because the system executes it when a session is restarted, rather than when it needs a value.
RunTimeValue	This operator sets the slot value when the inference engine needs it to be determined – thus the slot will remain `UNKNOWN` until the Order of Sources is triggered. This serves as a default value.
Assign	This operator makes a direct assignment to the slot in a condition or action.
Retrieve	This operator retrieves data from a database to determine the value of a particular slot or multiple slots.
Execute	This operator executes an external program or routine which can determine a value for the slot.

Using these sources combined with operators that deal specifically with determining the value of a particular slot (see Table 1-1) it is possible, for example, to:

- Retrieve the value from a database. If it's found there, then the Order of Sources succeeds and the remaining items are ignored. If a value isn't found in the database, then the second item in the Order of Sources may be to try to ...
- Inherit the value from a parent object. If that doesn't succeed, the next item may call for an ...
- Execute to an external routine to be executed and so on.

Order of Sources methods will be represented by the diamond with the OS title displayed in Figure 1-19:

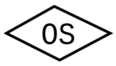


Figure 1-19 Order of Sources Method

If Change Method

The If Change method lists a series of actions to perform after the value of either a public or private slot is changed. There are several very important points about If Change actions:

- If Change actions are performed *immediately* after the value changes.

- When a series of If Change actions are defined, all of the actions are executed from top to bottom after the public or private slot's value is changed.
- By default, there are *no* If Change actions. Contrast this with the Order of Sources method which has a series of default sources.
- Similar to Order of Sources, If Change methods can be inherited down the object hierarchy.
- The If Change method will not be executed when a particular slot is reset to `UNKNOWN` using the Reset operator, unless the default strategy has been modified in the Strategy dialog window.

If Change methods will be represented by the diamond with the IC title displayed in Figure 1-20:

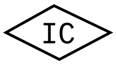


Figure 1-20 If Change Method

The method operators are particularly useful as If Change methods because they can maintain consistency throughout the system by modifying appropriate data structures whenever particular slot values change. For example, if there is a closed system which contains a particular gas, and the absolute temperature increases by 30%, then the pressure should increase by 30% as well. Obviously some changes can have very wide ranging effects. This accounts for the large number of operators as well as the capability to have a whole list of possible actions.

Rules

The Rules Element's capability of providing an intuitive way to represent our domain is a tremendous asset, but we also need to have some way of reasoning on it. Rules provide this reasoning capability. They reason over the object hierarchy. Rules capture the knowledge necessary to solve particular domain problems. Rules represent, among other things: relations, heuristics, procedural knowledge, and the temporal structure of knowledge.

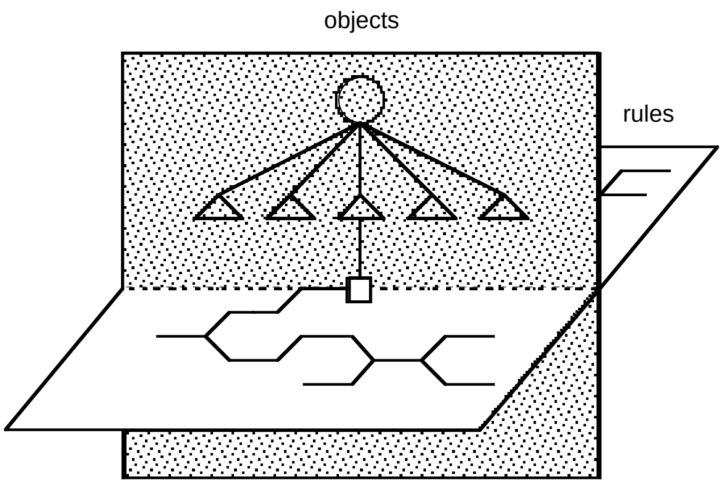


Figure 1-21 The Rule Plane

Rules have three basic parts:

- Left-hand side conditions
- The hypothesis which is a boolean slot
- The right-hand side actions.

Conditions, rules, and hypotheses are all boolean data structures. Similar to boolean slots, they may have one of four basic values: `UNKNOWN`, `TRUE`, `FALSE`, or `NOTKNOWN`. In this document a rule is represented by this icon:

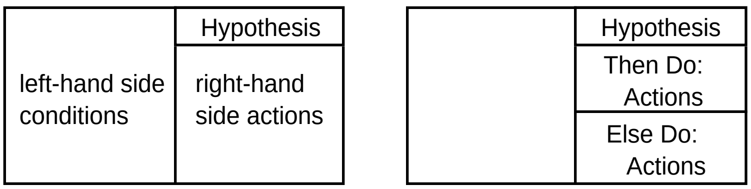


Figure 1-22 Basic Rule Structure

The conditions represent a series of tests to determine whether or not the hypothesis is TRUE. If *all* of the conditions are TRUE, then the hypothesis is set to TRUE and the right-hand side actions are all executed.

A rule's value depends on the state of its conditions:

- If no attempt has been made to evaluate the conditions, then the rule will be UNKNOWN
- If the Rules Element inference engine evaluates all of the conditions to TRUE, then the rule is set to TRUE as well
- If the inference engine has tried to evaluate the conditions, but could not determine the value of at least one condition, then the rule will be set to NOTKNOWN
- If the inference engine evaluates the conditions and one of them is FALSE, then the rule will be set to FALSE as well.

Rules Element rules are *symmetric* because they have no inherent "direction" associated with them. This means that the rule can either be processed in the forward direction by forward chaining events or in the backward direction by backward chaining events. These types of events are explained in Chapter Two, "Inference Engine Processing," but the important point to remember is that the rules are symmetric so you don't need to write one set of forward chaining rules and another set of backward chaining rules.

All slots used explicitly in the conditions or the actions of a rule are called *data*. A hypothesis, in and of itself is not a datum, but if it is used in the conditions of another rule, then it is a datum as well as a hypothesis. Hypotheses which are also data are referred to as *subgoals*. It is also possible to manipulate slots which are not data by means of interpretations or pattern matchings. These two concepts are explained in the corresponding sections of this chapter.

Note: Data that belongs to a private slot cannot be used in any of the rule structures described in this section. Unlike public slot data, the data of a private slot is not intended to be accessed by any structure other than a method associated with the slot. Refer to the Methods section for information about accessing private slot data.

Conditions

The conditions on the left-hand side of the rule have three columns. The first column contains the operator to be used in the test. The second column contains some expression, and the third column, if not empty, contains a series of constants or parameters. For example, one particular rule could look like:

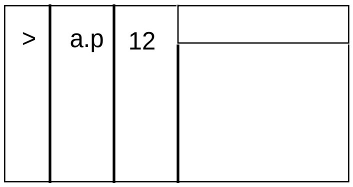


Figure 1-23 Condition in a Rule

This condition tests the value of the slot `a.p` against the constant 12, and if it is larger, then the condition is evaluated as TRUE. All rules must contain at least one condition. There is no limit to the number of conditions a rule can have.

The conditions within a rule are “anded” together, so they must all be TRUE for the rule to be TRUE . If you wish to express an “or” relationship within a rule, you must separate the values in the third column by commas:

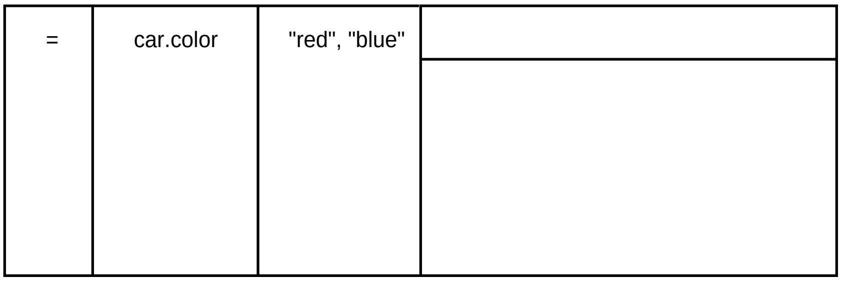


Figure 1-24 Or Condition

This condition means: “Is the car’s color red or blue?”. If the slot `car.color` is red or blue, then the condition will be TRUE; otherwise, it will be FALSE.

The operator can, among other things, test the value of an expression, perform object or set manipulations, or interact with external programs or databases.

The expression column can be any mathematical expression involving slots, constants, and pre-defined functions as well as a variety of other possibilities. In the case where the Rules Element is interacting with outside files (whether they be databases, executable files, or handlers), the expression column contains the name of the file.

The third column of the left-hand side conditions contains constants to test the expression against or parameters for many of the actions. For example, the Constant column of the Execute statement contains what information is sent to the executable file, or the Constant column of the Retrieve statement specifies exactly what information should be retrieved.

When the inference engine evaluates the conditions in a rule, it evaluates the condition which has the data with the highest *inference priority* first. By default, all data have inference priorities of 1, and when all the priorities are equal, conditions will be evaluated from the top to the bottom.

For example, consider a rule which determines whether or not you should buy a car:

yes	need_car(1)		buy_car
<	car.cost(1) + car.tax(5)	10000	
>	salary (4)- fixed_costs(3)	7000	

Figure 1-25 Data Inference Priorities

The inference priorities are listed in parentheses after the slot's name. Thus the inference priority of the slot `need_car.value` (remember if only an object is mentioned in any expression, then the inference engine uses the value property of that object) is 1, the slot `car.cost` is 1, the slot `car.tax` is 5, the slot `salary.value` is 4, and the slot `fixed_costs` is 3. The inference engine will evaluate the second condition first, then the third condition, and finally the first condition. The inference engine evaluates the condition with the highest data inference priority first. The slot `car.tax` has the highest priority so the second condition is evaluated first. After this condition is evaluated, `salary.value` has the highest inference value (4) of the unevaluated conditions, so the third condition is evaluated next. Finally, the first condition is evaluated.

Hypothesis

All rules have one and only one hypothesis. However, a hypothesis can have many different rules leading to it.

As previously stated, the hypothesis is a boolean slot. If all the conditions on the left-hand side are evaluated to TRUE, then the hypothesis is set to TRUE as well.

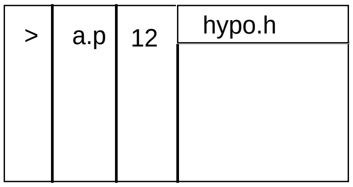


Figure 1-26 The Hypothesis

In the rule depicted in Figure 1-26, if the slot `a.p` is greater than 12, then the hypothesis `hypo.h` will be set to TRUE.

The hypothesis is a central component of the Rule Element's inferencing mechanism. We will go into this in detail in Chapter Two, "Inference Engine Processing."

Right-Hand Side Actions

The right-hand side actions include two lists of actions. The first list is only executed if the rule is evaluated to TRUE. Conversely, the second list is only executed if the rule is evaluated to FALSE. In contrast to the other two parts of a rule, actions are not required. They are a series of consequences of the rule being fired which are executed as soon as the rule is verified. There may be any number of actions.

Similar to the conditions, the right-hand side actions contain three columns: an operator column, an expression column, and a constant column:

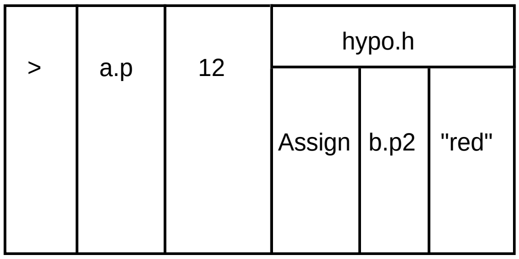


Figure 1-27 Right-Hand Side Actions

The actions which can be performed are quite similar to those of the methods described in Table 1-1 and to those contained in the conditions. Table 1-2 shows the complete list of operators for rules.

Rule Operator	Left-hand Side Conditions	Left-hand Side Actions	Right-hand Side Actions
`=,<>,>,<,>=,<=`	X		
`Assign`		X	X
`CreateObject`		X	X
`DeleteObject`		X	X
`Execute`		X	X
`LoadKB`		X	X
`Member`	X		
`No`	X		
`NotMember`	X		
`Reset`		X	X
`Retrieve`		X	X
`SendMessage`		X	X
`Show`		X	X
`Strategy`		X	X
`UnloadKB`		X	X
`Write`		X	X
`Yes`	X		

Table 1-2 Rule Operator Usage

Comparing the possible operators, one sees that all of the action operators are available on both the left-hand side and the right-hand side of a rule. Those operators which are available only on the left-hand side are specific to testing a condition’s value, such as >, =, Member, NotMember, Yes, and No.

Inheritance

These representation mechanisms are quite useful in terms of structuring a world, but inheritance is what gives the greatest utility to this form of representation. There are three fundamental types of inheritance that are under the control of the application developer:

- Property inheritance
- Value inheritance
- Method inheritance.

Note: Inheritance of meta-slot attributes is not under the control of the application developer and is therefore not covered in this section. See the Meta-Slot Attributes section for details about inheritance of prompt lines and data validation functions.

Property inheritance refers to the ability for an object to inherit the existence of a particular property from a class (or a subclass from a parent class). This means that an object, such as `b`, which belongs to a class `Patients` that has the property `temperature`, will also inherit that property:

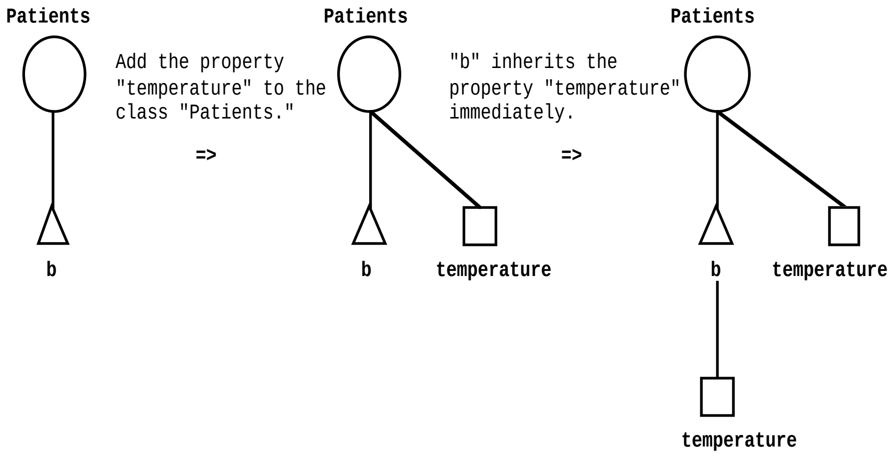

Patients
Add the property "temperature" to the class "Patients."
=>
b
Patients
"b" inherits the property "temperature" immediately.
=>
b
temperature
Patients
b
temperature
temperature

Figure 1-28 Property Inheritance

Property inheritance occurs *immediately*. This means that as soon as an object is added to a class or a property is added to a class, inheritance occurs before anything else.

For example, if there are one hundred patient objects attached to the class `Patients`, and a new property `temperature` is attached to that class, each of the objects will immediately inherit that property. Obviously inheritance is a great utility as it saves adding the same property to each child, typing errors, omitting a couple of patients, adding the property to objects which aren't patients, and so forth. As soon as a property is added at the parent level, all of the children will inherit the particular property.

The second type of inheritance, value inheritance, is the ability for a slot to assume the value of one of its parents (or children) if its own value is `UNKNOWN`. For example, there may be a situation where the inference engine doesn't know which language the novel `gone_with_the_wind` is in, but it does know the class `American_novels` has a value of English for its `languages` slot, so `gone_with_the_wind` inherits this value.

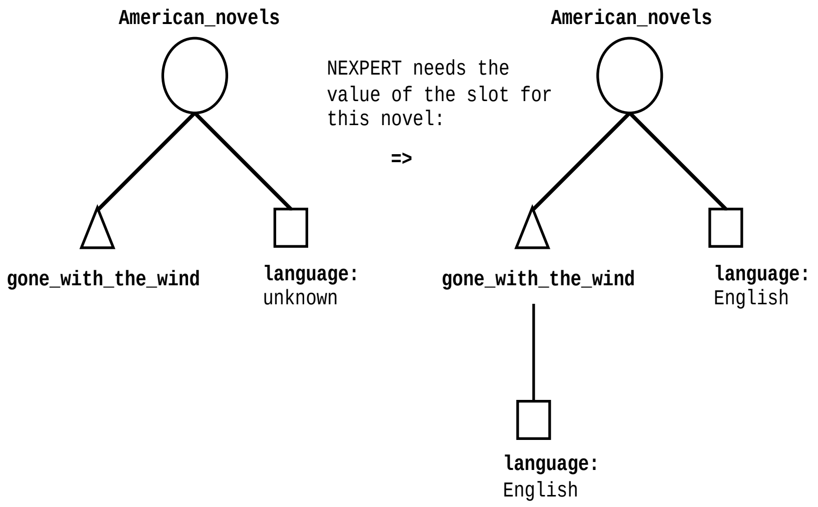


Figure 1-29 Value Inheritance

The important difference between value inheritance and property inheritance, in addition to the fact that a value is being inherited instead of a property, is that the value is only inherited when the inference engine needs the value. If the inference engine doesn't need the value, it won't needlessly propagate values around the object hierarchy.

Once again, this leads to a great utility in terms of expression. Instead of specifying each of the individual slot values, the generic value can be specified at the class level and inherited by any of its objects when the value is needed.

The third type of inheritance is inheritance of methods. Methods contribute to the behavior of a slot by providing a set of relevant actions. General behaviors can be specified at the parent object or class level, and they will be inherited when they are needed.

Assume now that the inference engine once again needs the value of the slot `gone_with_the_wind.language`, and there is no value declared at the parent level since the slot `American_novels.language` is `UNKNOWN`. There is, however, an Order of Sources method declared at the parent level:

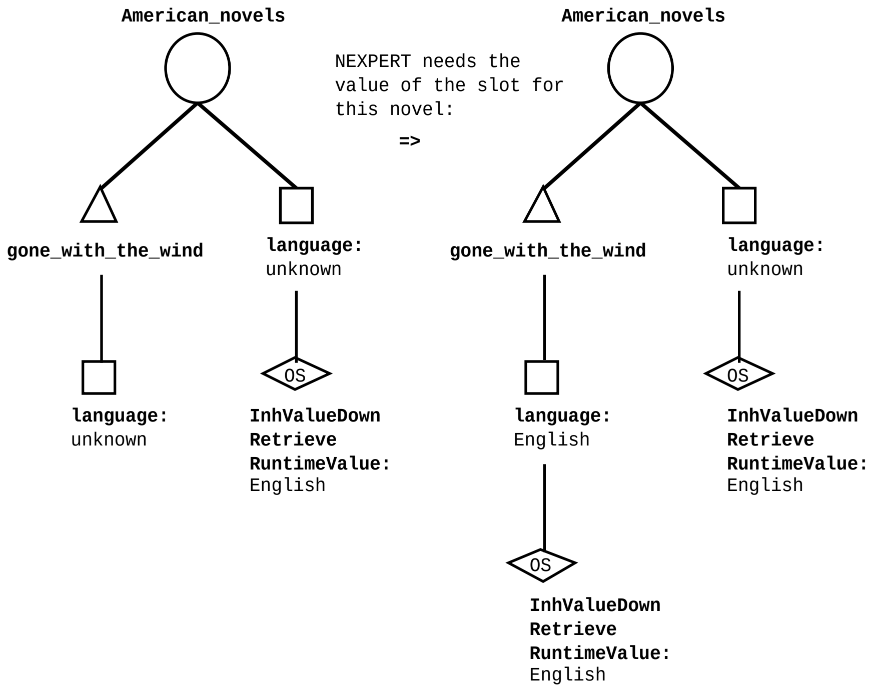


Figure 1-30 Method Inheritance

`gone_with_the_wind.language` will inherit the Order of Sources method. The system automatically executes the method's list of actions for the slot which in this case will try to:

- Inherit a value down (which will fail since `American_novels.language` is `UNKNOWN`),
- Retrieve a value from a database. If this fails, then
- `gone_with_the_wind.language` will use the runtime value of English.

Similar to value inheritance but unlike property inheritance, methods are only inherited when they are needed.

The next three sections describe each of these types of inheritance in detail.

Property Inheritance

Property inheritance refers to the ability for an object to inherit the existence of a particular property from a class (or a subclass from a parent class). Property inheritance occurs *immediately*. This means that as soon as an object is added to a class or a property is added to a class, inheritance occurs before anything else.

Default Behavior

The default strategy allows properties to be inherited down from classes to subclasses and from classes to objects, but not from objects to subobjects. Thus if there is a class `Cars` with a subclass `Make_1` and a property `top_speed` is linked to the class `Cars`, the property will immediately propagate down:

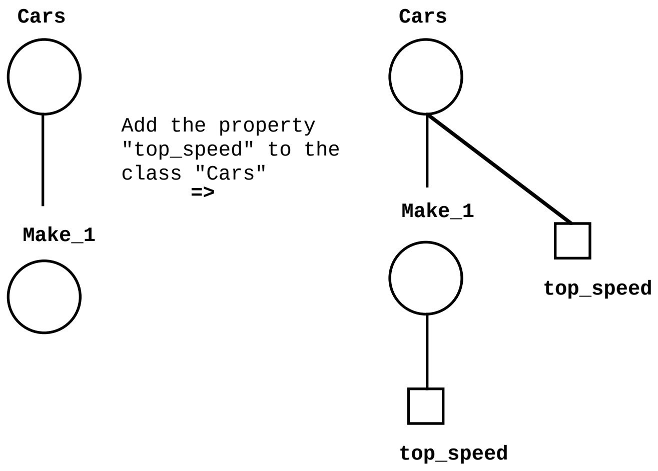


Figure 1-31 Class to Class Property Inheritance

If there is a class `Cars` which has an object `her_make_1` and the property `top_speed` is linked to `Cars`, then `her_make_1` will inherit this property:

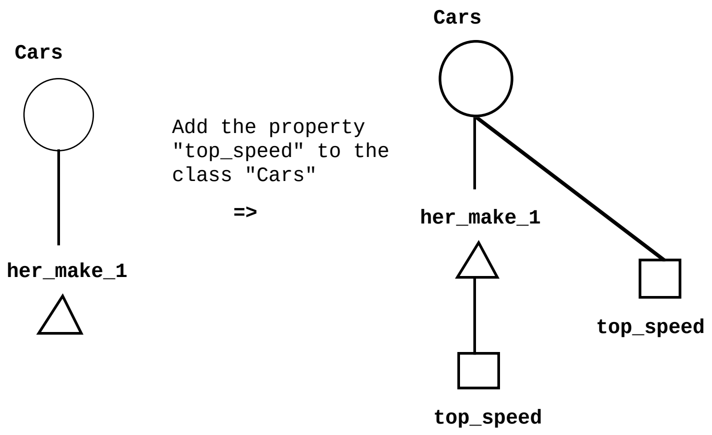


Figure 1-32 Class to Object Property Inheritance

Finally, if there is an object such as `her_make_1` which has a subobject `her_make_1_trunk`, then this subobject will *not* inherit any properties which are linked to the parent object `her_make_1`:

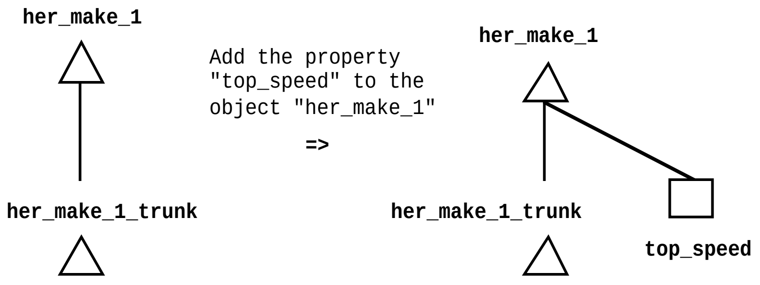


Figure 1-33 Object to Object Property Inheritance

Property inheritance is recursive. This means that if we add a property to a parent class with some arbitrary tree structure below it, then property inheritance will propagate down the hierarchy immediately:

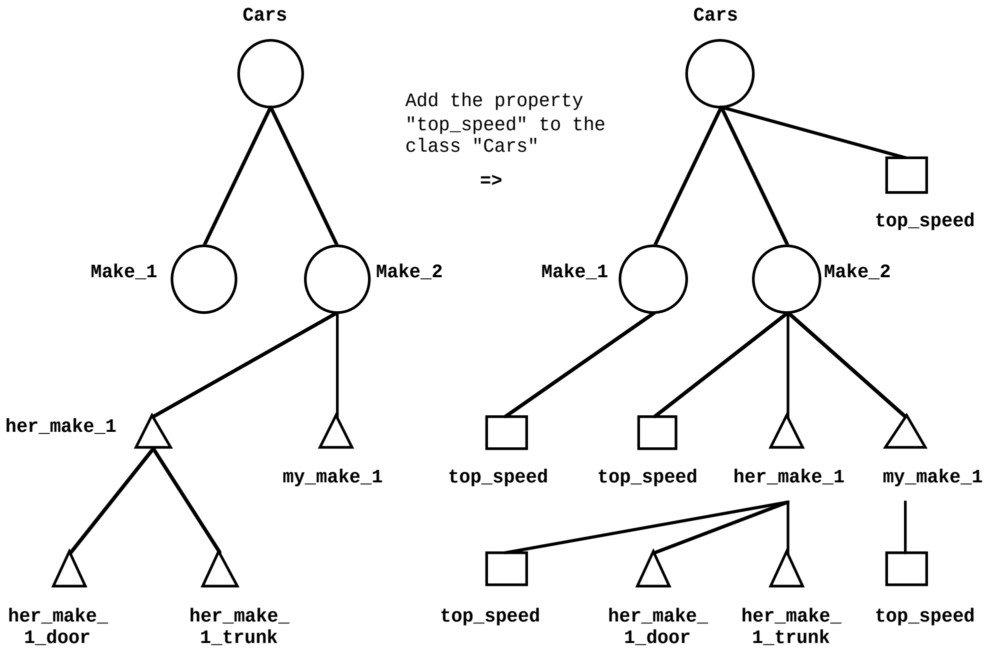


Figure 1-34 Recursive Property Inheritance

The property `Top_speed` propagated down to the child classes `Make_1` and `Make_2` as well as the objects `her_make_1` and `my_make_1`. However, it did not propagate down to the subobjects `her_make_1_door` and `her_make_1_trunk`.

Since subclasses are specializations of parent classes, they usually have the same properties. Objects are instantiations of classes so they also quite often have the same properties (as well as individual properties of their own). Finally subobjects are just parts of other objects and, as such, quite frequently have entirely different properties.

Inheriting from classes to subclasses and from classes to objects but not from objects to subobjects is the default strategy of property inheritance. By default, properties are not inherited up for any type of class to class, class to object, or object to object hierarchy. However, similar to most of the Rules Element environment, this default strategy may be altered to disable property inheritance from class to class or from class to object, or to enable property inheritance from object to object or up the object hierarchy. See the Inheritance Strategies section for more details on this subject.

The behavior of property inheritance when objects or classes are added to parent objects or classes is analogous to the case of adding properties described above. When a class is added to a class (or group of classes), it will immediately inherit any properties linked to the parent class.

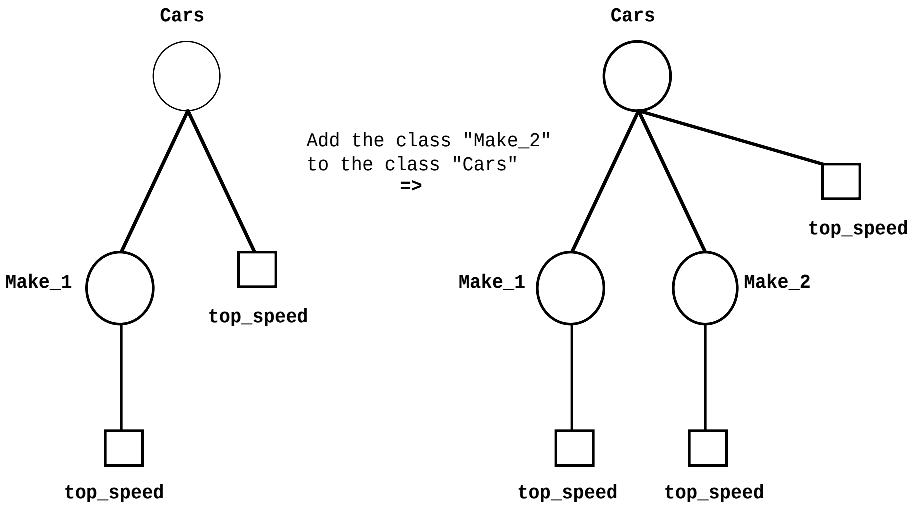


Figure 1-35 Inheriting Down After Creating a New Link

Similarly, as soon as an object is added to a class (or group of classes), it inherits all of the properties linked to the parent class.

When an object is linked to a parent object, it will not inherit any of the properties linked to the parent object.

Once again, these are the default strategies. See the section on Inheritance Strategies for details on how to modify them.

In summary:

- Whenever a property is added to an object or class, the inference engine checks the current strategy to see if it should propagate the property to any other objects or classes linked to the source object or class.
- Whenever a new link is created, the inference engine checks to see if it should propagate any properties between the two objects or classes involved with the link.

Once again, the *Value* property is treated differently. It is NEVER inherited down (or up) the object hierarchy. Thus if we add the Value property to a class or object, none of the subclasses or objects will inherit this property.

Now assume there is a subclass which has a particular property, but neither the parent class nor some objects which are members of the subclass have the property. For example, the subclass `Make_2` has the property `top_speed`, but neither the parent class `Cars` nor another subclass `Make_1` have the property:

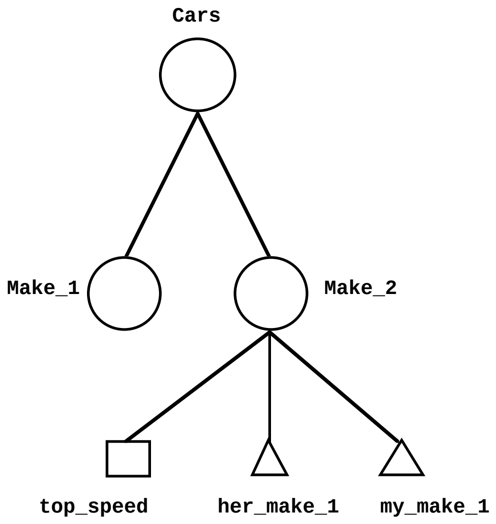


Figure 1-36 Subclass with a Unique Property

If the property `top_speed` is added to the parent class `Cars`, then the property `top_speed` will propagate down to the subclass `Make_1`, but it will *not* propagate down to the objects of the subclass `Make_2`:

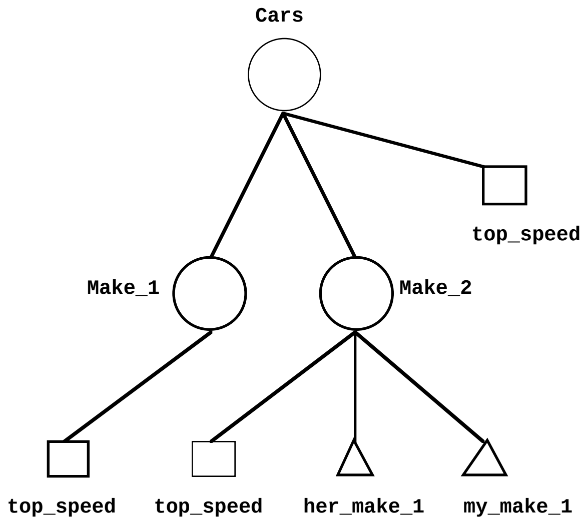


Figure 1-37 Property Propagation Blocked by the Property's Pre-existence at a Lower Level

In our example above, the property `top_speed` does propagate down to the class `Make_1`, but when it tries to propagate down to the class `Make_2`, it sees that `Make_2` already has that property. Therefore, it doesn't try to propagate the property down that particular branch of the tree. Since the class `Make_2` already has the property, it would have been propagated down that branch of the hierarchy already if it were supposed to be propagated there. To propagate down this branch is either a waste of time (if it already exists), or it violates the reason the property wasn't propagated down in the first place.

This is an important point. When a new property is added to a particular class or object, it is immediately propagated down the object hierarchy. If at any time an object or class is found to already contain the property in question, then it will *not* be propagated further down that particular branch. Of course, if an entirely new property is added to the parent class or object, then it would be propagated to all the classes and objects in that particular object hierarchy structure.

Inheritability Control

There are many situations where this default property inheritance strategy may not be appropriate. You have several choices. First of all, if you wish to change the default strategy for the whole application, you may do so by choosing the appropriate selection from the Strategy dialog window from the Expert Menu:

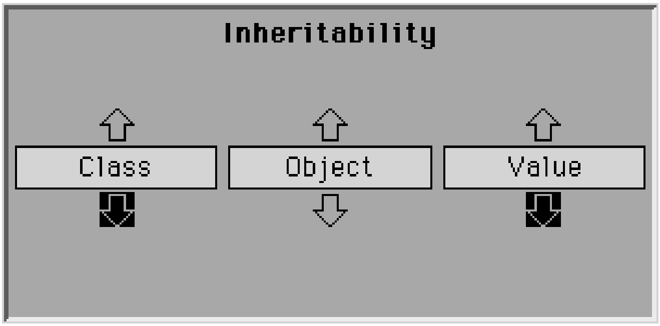


Figure 1-38 Strategy Dialog Window Inheritability Settings

The class and object inheritability settings determine whether or not properties are inherited up or down from classes and objects. As one can see, the default is to use the inheritance links down from classes but not down from objects. You merely need to click on the white arrow under Object to allow inheritance to proceed down from objects, click on the black arrow under Class to prevent inheritance down from classes. Similarly, to enable property inheritance up, click on the white arrow above Class or Object.

Disabling Inheritability Down from Classes

Preventing inheritance down from classes would be useful in the type of situation where classes are generalizations of objects but they don't share the same properties and thus there's no reason to needlessly propagate properties to all of the objects. The same case holds for the situation with classes and subclasses.

Enabling Inheritability Down from Objects

Allowing inheritance from object to subobject may be useful where the parts of an object share the same properties as the entire object itself. If there are only general properties such as color and weight, then there may be a good reason for the object to inherit these properties. Sometimes inheriting more specialized properties to subobjects may be useful, for instance an application may examine how to construct a house to minimize heat loss. Then not only would `the_house` have a property such as `thermal_conductivity`, but so would the subobjects such as `walls`, and even subobjects of those subobjects such as `sheet_rock`, `insulation`, and so on.

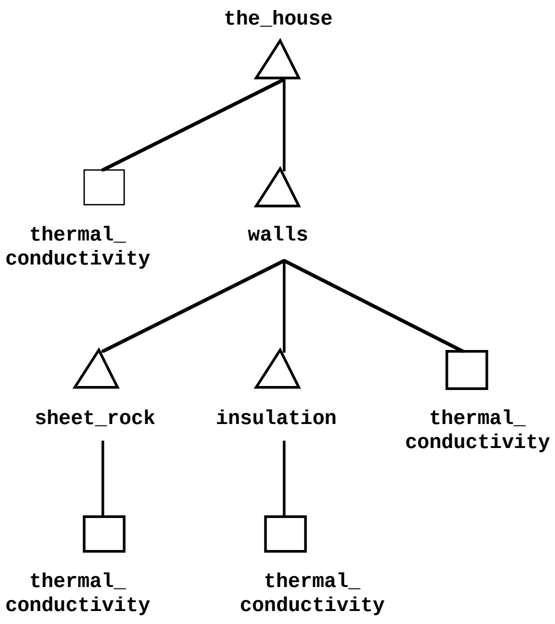


Figure 1-39 Inheriting Properties to Subobjects

This type of hierarchy can continue in this manner indefinitely.

Enabling Inheritability Up

There may be many situations where you want to enable the inheritability up strategy. For example, if the class `Transportation_vehicles` doesn't have a `speed` property, a useful strategy may be to get the property from the subclass `Cars`.

Similar to inheriting down, if inheriting up is enabled, then inheritance of properties will be done immediately. Also similar to inheriting down, neither the existence of the "Value" property nor its current value is ever inherited up.

Value Inheritance

There are three cases wherein the inference engine must find a value in order to continue processing the knowledge base. The needed value is always associated with the evaluation of a slot under one of the following circumstances:

- The slot is a hypothesis and it becomes the current evaluation
- The slot is used in a condition
- The slot is used in an assignment statement.

Default Behavior

The first case will be covered in Chapter Two, "Inference Engine Processing." The second case involves a situation like:

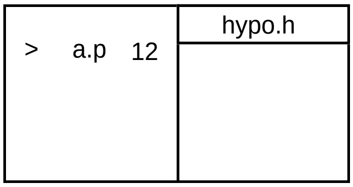


Figure 1-40 Evaluating a Slot in a Condition

When the inference engine evaluates this rule and comes to the condition bearing on slot `a.p`, it will use the value of `a.p` if it has one. If it doesn't have a value (because the value is `UNKNOWN`), then the inference engine processes an Order of Sources method (whether that be local, inherited, or the default) for `a.p`. See the Order of Sources section and the Method Inheritance section for more details. The third case involves a situation like:

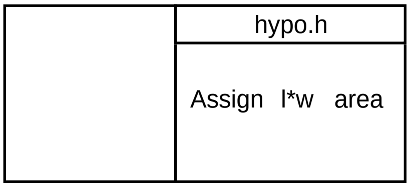


Figure 1-41 Evaluating a Slot in an Action

In this case, the inference engine evaluates the value of the slots `l` and `w` in order to assign the product to `area`. Once again, if `l` and `w` are `UNKNOWN`, the inference engine will process their Order of Sources method in order to determine their values. An interesting construction often used is:

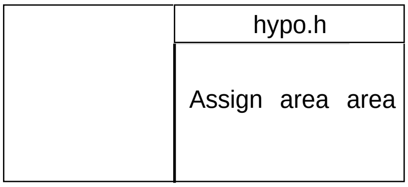


Figure 1-42 Assign Slot-Slot Syntax

Similar to the above assignment statement, the inference engine will evaluate the value of the slot `area` in order to assign it to the slot `area`. The net effect of this is to evaluate the slot `area`. Thus you can force the evaluation of any slot by using this construct.

If the inference engine needs to find the value for a slot for any of the reasons outlined above, and the slot doesn't have a value yet, then the slot may be able to inherit a value from one of its parent classes. This type of inheritance normally proceeds:

- From parent class to subclass
- From class to object
- From object to subobject.

However, unlike property inheritance, this inheritance is only triggered when the inference engine needs the value for a particular slot. For example, if there is a hierarchy where the value of the slot `Make_1.top_speed` is known to be 120, we see that this value is not propagated down the hierarchy to the slots `her_make_1.top_speed` and `my_make_1.top_speed` (they remain `UNKNOWN`):

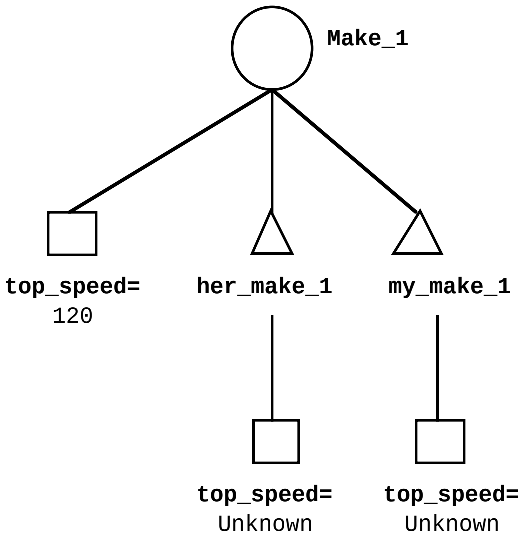


Figure 1-43 Value Inheritance

If the inference engine needs the value for the slot `her_make_1.top_speed`, it will inherit this value from the parent class `Make_1`. Thus a value of 120 would be used for `her_make_1.top_speed`:

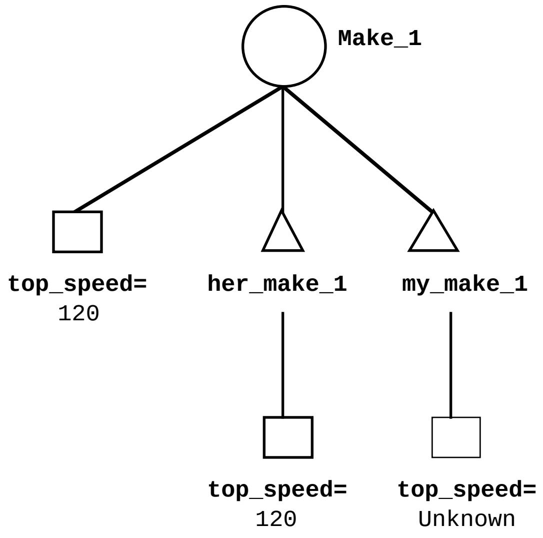


Figure 1-44 Value Inheritance

Notice also that even though `her_make_1.top_speed` has inherited the value of 120 from the parent class `Make_1`, the slot `my_make_1.top_speed` remains `UNKNOWN`.

The inference engine will never try to determine the value of a parent class so that one of its children can inherit it. If the parent class has a value, then the child can inherit it. If not, then the child must look for a value from another parent or from another source.

The *Value* property is once again an exception. If the parent class or object and the child object or class have the Value property, and the parent's slot has a value (it is *not* `UNKNOWN`) then, unlike the case with other slots, there will be no inheritance. The inference engine must get the value from some other source.

Inheritability Control

There are many situations where the default value inheritance strategy may not be appropriate. You have several choices. First of all, if you wish to change the default strategy for the whole application, you may do so by choosing the appropriate selection from the Strategy dialog window from the Expert Menu:

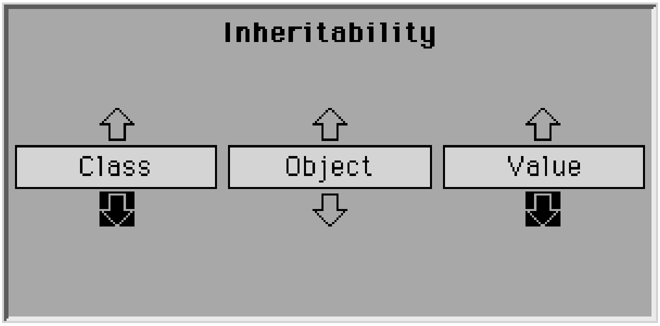


Figure 1-45 Strategy Dialog Window Inheritability Settings

The value inheritability setting determines whether or not values are inherited up or down. The Value arrow is also selected which means values are inherited down by default. You merely need to click on the black arrow under Value to prevent inheritance down of values or click on the arrow above Value to permit inheritance up of values.

Disabling Inheritability Down

Value inheritance is allowed down as a default. There are many cases where we wouldn't want this to occur. For example, the slot attached to a class `stocks.net_value` may be a certain value. We would certainly want the objects of this class (individual stocks) to have the property `net_value`, but we wouldn't want to assume all the stocks in this particular class had the same net value by inheriting down the class slot's value.

Enabling Inheritability Up

There may be many situations where you want to enable the inheritability up strategy. For example, if there is no value for a particular car's color, a good guess is to inherit up the color of the car's door. Similarly, if the class `Cars` has a `number_of_wheels` slot which hasn't been determined yet, a good strategy may be to inherit up the value from one of the objects of the class `Cars`.

Similar to inheriting down, if inheriting up is enabled, then inheritance of values will be done when a particular parent needs a value. Also similar to inheriting down, neither the existence of the "Value" property nor its current value is ever inherited up.

Method Inheritance

Methods can be inherited in a manner similar to properties and slot values. Also analogous to slot values, they are inherited only when needed and when there is no method attached at the current level.

Default Behavior

Methods are never inherited up. Inheritance of methods proceeds along the same links as inheritance of properties and values, except for the fact that they can only be inherited down, regardless of the current strategy. Depending on the type of method, the inheritance behavior is as follows:

In the case of the user-defined method, when the SendMessage operator (from a rule or method) sends a message to its list of addressees that specifies the name of the method to trigger, the inference engine:

- Looks to see if the addressee named in the message (can be a slot, object, class, or property name) has a method declared locally with the same name as the one specified in the message. If there is one, then it is executed.
- If no matching method is declared locally, then the inference engine tries to inherit a method that has the same name as the one specified in the message from a parent object or class.
- If there is none to inherit (or if inheritance down is disabled) and the method is attached to slot, then it tries the property.
- If there is no method available, then no actions are performed.

In the case of the Order of Sources system method, when the inference engine needs the value of a slot:

- It looks to see if the slot has anything in its Order of Sources method. If so, these are executed.
- If not, then the inference engine tries to inherit an Order of Sources down from one of its parents.
- If none of them have anything declared (or if inheritance down is disabled), then the inference engine resorts to the default Order of Sources: Backward Chaining, InheritValueDown, InheritValueUp.
- If all of these Order of Sources fail, the inference engine always asks a question.

In the case of the If Change system method, when the inference engine detects that the value of a slot changes:

- It looks to see if the slot has an If Change method declared locally. If there is one, then it is executed.
- If no If Change method is declared locally, then the inference engine

tries to inherit one down.

- If there is none to inherit (or if inheritance down is disabled), then no actions are performed since there are no default If Change actions.

Thus method inheritance proceeds down from parent class to subclass, from class to object, and from object to subobject. This default strategy can be modified so that individual methods are private and cannot be inherited by the children of the parent to which the method is attached.

The Self Variable

The *SELF* variable is very important. It is used in methods of any kind and within the @V syntax of the Prompt Line meta-slot attribute. It is used to refer to the object which is being processed at the time.

The main reason this is so useful has to do with inheritance. For example, consider the situation where we have a class `Fasteners` with a particular object of that class `a_bolt`. The inference engine needs to determine the total cost of all the one inch screws. The formula for total cost is price multiplied by the number. Thus you could specify a method `Compute_Total_Cost` for the object `a_bolt` that multiplies the cost and number together:

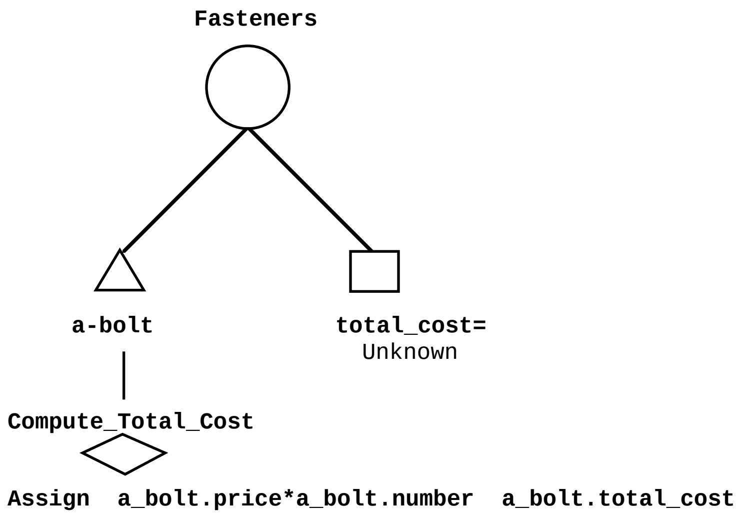


Figure 1-46 Inheriting an Explicit Method

This approach works fine. However, most likely there are many different types of fasteners. The total cost of each fastener is calculated the same way;

thus it makes sense to specify this behavior at the parent class level and have each of the children inherit it:

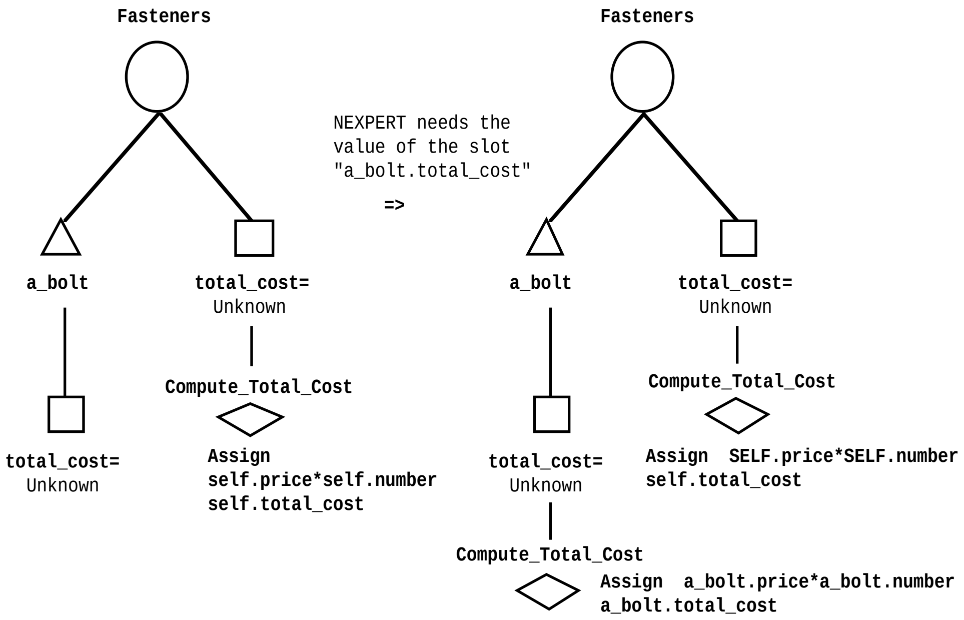

Fasteners
NEXPERT needs the value of the slot "a_bolt.total_cost"
=>
Fasteners
a_bolt
total_cost= Unknown
Compute_Total_Cost
total_cost= Unknown
Assign self.price*self.number self.total_cost
a_bolt
total_cost= Unknown
Compute_Total_Cost
Assign SELF.price*SELF.number self.total_cost
total_cost= Unknown
Compute_Total_Cost
Assign a_bolt.price*a_bolt.number a_bolt.total_cost

Figure 1-47 Inheriting a General Method

Since each object which inherits the method will be calculating its total cost based on its price and number, we cannot use explicit references to slots. Each object (or class) that inherits this method will substitute its own name for the SELF variable. This means that the explicit method for slot `a_bolt.total_cost` in Figure 1-46 behaves exactly the same as the inherited general method for `a_bolt.total_cost` displayed in Figure 1-47. When a general method is used in place of an explicit method it is known as *specialization*.

Mixing Explicit with Inherited Methods

It is also possible for a slot to use a series of actions defined locally, then try to inherit additional sources or actions from a parent, and finally try any additional actions defined at the slot level. This is accomplished by defining a list like:

Series of actions

InhMethod (specify a explicit parent name or use default)

Series of additional actions

Once again, if these are defined in an Order of Sources method, depending on the current strategy setting, the inference engine will stop executing the list as soon as a value is found for the slot in question. If they are defined in a user-defined method or an If Change method, then all actions will be executed. Note that any method triggered by a SendMessage operator is considered “user-defined” by definition.

Non-Inheritable Methods

Although no inheritance strategy exists to disable method inheritance for the entire system, it is possible to disable method inheritance on a case by case basis. This is most likely the situation for an explicit method (the SELF variable does not appear in any of the actions). In this case, the method’s actions are intended for use only by the slot to which it is attached. The Method editor provides this option when you create the method.

If you have a situation where some of the actions should be inheritable and others not, the method actions list can be divided between two different methods. Thus by specifying two methods with the same name, one can contain only the private actions and the other the public actions. Any child object or class inheriting the method from the parent will inherit only the method with the public actions list. Or, if you want to have the parent and the children execute the same actions (when there are private actions), you can duplicate the exact same actions in a method for the child using the copy Function In The Method Editor.

Conflict Resolution

A particular object or class may often have several parents or even an entire hierarchy of possible parents from which to inherit a value or a method. Each time an inheritance event occurs, there is the possibility for conflict between alternate sources of information. You specify how to search for a value and a method using general search strategies. The strategies can be applied each time there is a conflict during an inheritance event.

Note that properties inherited by objects do not give rise to conflicts since the same property of two parents will be inherited only once. In the case of methods, the conflict arises when two methods of the same name exist at the level of the parents. Downward inheritance of properties, values, and methods is known as *specialization*. The following search strategies apply to inheritance of values and methods.

Parent Search Strategies

There are several general strategies to choose from: one can either look at parent objects or parent classes first, and one can either proceed in a depth-first or a breadth-first type of search. This gives rise to four general types of search strategies (Figure 1-48 will help explain each of them):

- <u>Class-first, breadth-first:</u> This is the default strategy. This means if a slot needs to inherit a value or method, it will look at all of its immediate parent’s first. If none of them have one to inherit from, then it will look at its parent’s parents, and so on.
- <u>Class-first, depth-first:</u> This strategy means if a slot needs to inherit a value or method, it will look at one of its parent classes first. If that class doesn’t have one it can inherit, then it will look at one of that parent’s parent classes and so on until that branch of the tree is completely exhausted or until one is found. If a value or method isn’t found, then the search back up until it reaches a sub-branch which hasn’t been completely explored and then searches down it. It may even back all the way up to the top level and then proceed down another parent class branch. Finally, if all the classes have been searched without finding a value or method, then the object hierarchy will be searched in the same manner.
- <u>Object-first, breadth-first:</u> This strategy is the same as the class-first, breadth-first strategy except in this strategy the parent objects are searched first instead of the parent classes at each level.
- <u>Object-first, depth-first:</u> This strategy is the same as the class-first, depth-first strategy except in this strategy the parent objects are searched first instead of the parent classes.

If there is a tie because two or more parents exist at the same level, then the parent whose slot has the higher inheritance priority will be inherited from. In the case of methods, however, which may not be attached to a slot, an additional way to break a tie is provided: if two methods exist at the same level, the application developer can explicitly specify which parent to begin the search from through the InhMethod operator.

The following diagram shows an object hierarchy and the following paragraphs explain how the inheritance would proceed under the four distinct strategies:

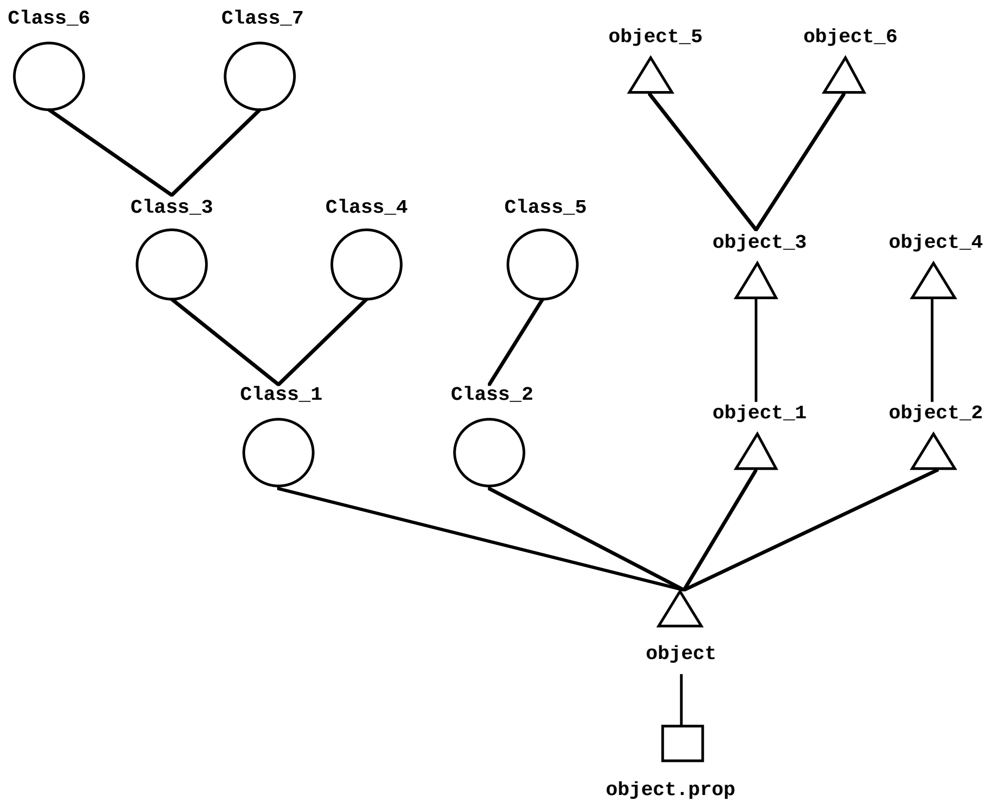


Figure 1-48 Conflict Resolution

Let's assume the system needs the value of the slot `object.property` or that a message to trigger a method for the slot or object is sent. Then, if the current strategy is:

- Class-first, breadth-first: Then the search will proceed in the following order: Class_1, Class_2, object_1, object_2, Class_3, Class_4, Class_5, object_3, object_4, Class_6, Class_7, object_5, object_6. The search will stop as soon as a value or method is found. Notice the inference engine searches the first level classes and objects until there are no more classes and objects at that level, then it searches the second level of classes and objects, and so on.
- Class-first, depth-first: Then the search will proceed in the following

order: Class_1, Class_3, Class_6, Class_7, Class_4, Class_2, Class_5, object_1, object_3, object_5, object_6, object_2, object_4. Once again, the search will stop as soon as the value or method is found. Now the search picks one branch of the class tree and moves to deeper levels until that branch is exhausted, and then it backs up just enough to find an unexplored branch, at which point it searches to the end of the branch again. After all the class branches are exhausted, the object branches are searched in a completely analogous manner.

- Object-first, breadth-first: Then the search will proceed in the following order: object_1, object_2, Class_1, Class_2, object_3, object_4, Class_3, Class_4, Class_5, object_5, object_6, Class_6, Class_7. This is the same as the class-first, breadth-first search except the object branches are searched first.
- Object-first, depth-first: Then the search will proceed in the following order: object_1, object_3, object_5, object_6, object_2, object_4, Class_1, Class_3, Class_6, Class_7, Class_4, Class_2, Class_5. This is the same as the class-first, depth-first search except the object branches are searched first.

In summary, when the inference engine searches through the object hierarchy, it uses one of the four general strategies and breaks ties using the inheritance priority and the child node each parent was expanded from. These strategies in the order of decreasing priority are breadth first (at each level), then class versus object, then the child node that each parent was expanded from, and finally the inheritance priority within each group of nodes defined at the level below.

It is very important which areas are searched first, both because different parents may have different values (or methods) so the child object will inherit different values (or methods) depending on the setting, and even if the same value is inherited, one may want to inherit from one parent rather than another due to cost, CPU time, and so forth.

If a class instead of an object is inheriting the value or method from a parent, the same principles apply for the depth-first versus breadth-first types of searches. There is no distinction between object-first and class-first since the value or method is being inherited down and objects cannot be parents of classes.

Children Search Strategies

Similar to our situation with inheriting down, a particular object or class may often have several children or even an entire hierarchy of possible children from which to inherit. Unlike inheriting down, however, methods cannot be inherited up, only the properties and the values are eligible to be inherited by the parents of a child object. This type of upward inheritance of properties and values is known as *generalization*.

In order to avoid inheritance conflicts when values are inherited up, you may need to specify how to search for a value using the general search strategies. The strategies are applied each time there is a conflict during an inheritance event. The same strategies outlined for inheriting down apply here as well: one can either look at child objects or child classes first and one can either proceed in a depth-first or a breadth-first type of search. This gives rise to the same four general types of search strategies:

- Class-first, breadth-first (the default)
- Class-first, depth-first
- Object-first, breadth-first
- Object-first, depth-first.

For details on these strategies, see the previous section on Parent Search Strategies.

If an object instead of a class is inheriting the value from a child, the same principles apply for the depth-first versus breadth-first types of searches. There is no distinction between object-first and class-first since the value is being inherited up and classes cannot be children of objects.

Dynamic Structures

The representation mechanisms described above provide a rich environment to describe a world in which all the relationships are clearly defined when you create the knowledge-based application. But what happens if there are objects and relationships whose existence isn't known a priori? This section will address this issue.

The inference engine is designed to be integrated into your computing environment. Thus it needs to interact with databases, other programs, and many other computational and mechanical devices whose exact specifications will not and cannot be known when writing the application. This presents a bit of a dilemma. To solve it, the inference engine allows applications to create dynamic objects and dynamic links. The objects and links are created at runtime rather than being compiled with the rest of the application. As the system realizes the need for new objects and new relationships, it can create them. Of course, both objects and links can also be deleted at runtime as they are no longer needed.

This gives you tremendous flexibility as you only need to hard code those objects and relationships which are always used, while the objects and relationships whose existence depends on the current state of the system and the external environment can be created as needed. This also saves both memory and disk space as only the permanent objects and relationships which are needed are created and stored in memory.

Dynamic Objects

Objects can be created with the CreateObject operator that appears in a rule or a method. A newly created object will start with no properties, parent classes or objects, or subobjects. Dynamic objects add to the Rules Element's representation of the domain, however since they have no slots or relationships, they have no way of relating with the other structures in the Rules Element. If dynamic objects were the only dynamic structures the Rules Element allowed, this new means of representation wouldn't be very useful from the computing point of view since dynamic objects wouldn't have slots, and if they didn't have slots, then they couldn't hold any information.

It is important to note that dynamic objects are full-fledged objects. They have almost all of the capabilities and characteristics of other objects. The differences are as follows:

- They are created at runtime rather than being compiled with the rest of the application.

- They are deleted when a new session is started.
- Since these objects are defined during the processing of your knowledge base, the only way they can have properties and methods is by linking them to classes and relying on inheritance mechanisms.

Dynamic Links

A Rules Element application can create new links between both compiled objects and classes as well as dynamic objects using the CreateObject operator. Newly created objects can be linked to other objects or classes (whether the other objects and classes are dynamic or not). This linking can be done as soon as the object is created or at any time during the inference process.

Links can also be created between compiled objects and other objects or classes. Links can be deleted from any objects or classes, whether they are compiled or dynamic. Thus the whole object hierarchy can be altered dynamically while the inference engine is running: new objects can be added to the network, objects can be made subobjects of other objects or instantiations of classes, and the links can all be destroyed.

Thus in an insurance knowledge-based system, there could initially be just a class network, such as:

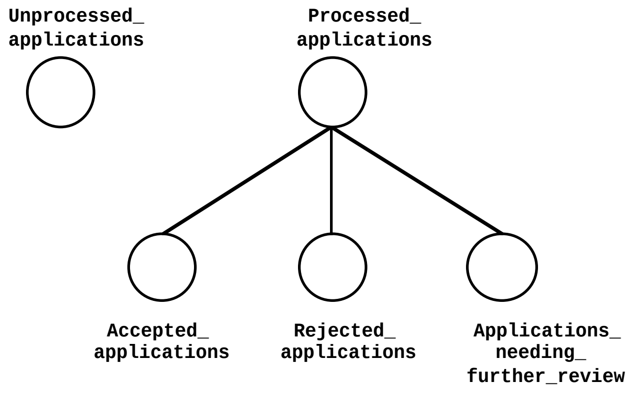


Figure 1-49 Initial Class Network

A dynamic object can be created for each client's application for insurance and linked to the class `Unprocessed_applications`. The information for these objects can come from a database, from the user interface, or anywhere else:

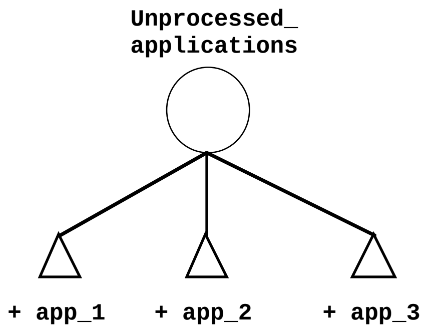


Figure 1-50 Adding Dynamic Objects

Notice that the name of dynamic objects is preceded by a plus sign “+”.

After processing each application, the link to the class `Unprocessed_applications` can be deleted, and then a link between the application and `Processed_applications` as well as either `Accepted_applications`, `Rejected_applications`, or `Applications_needing_further_review` can be created:

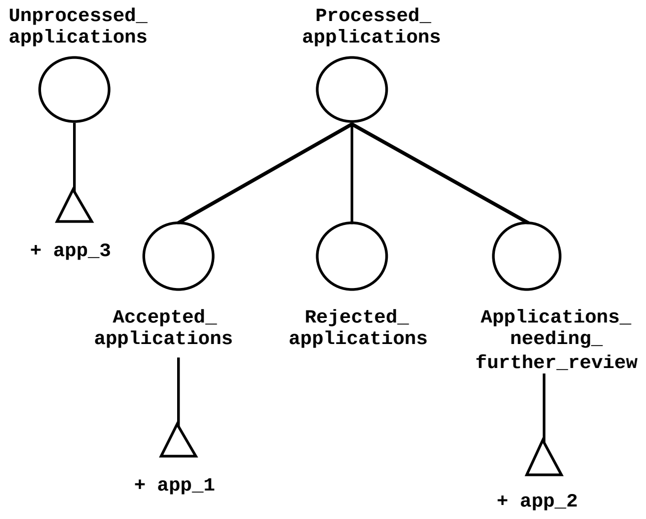


Figure 1-51 Attaching Dynamic Objects to Other Classes

Any number of dynamic links can be created and / or deleted between either dynamic objects (as in the case above) or compiled objects. In the above example, it made sense to switch the dynamic objects out of one class before adding them to some more classes, but this is certainly not a prerequisite.

Dynamic links allow the internal representation to be dynamically modified to accurately reflect the changing environment or state of the inference session. Similar to the case with dynamic objects, dynamic links are deleted when a new session is started.

In summary:

- If you use the CreateObject operator with a compiled object and a set of other objects or classes, then new links will be created between the compiled object and the other objects.
- If you use the CreateObject operator with an object name which doesn't exist yet, then a dynamic object will be created.
- If you use the CreateObject operator with an object name which doesn't exist yet and a set of other objects or classes, then a dynamic object will be created as well as dynamic links between the dynamic object and the other objects or classes.

Inheritance

One of the tremendous benefits to this restructuring of the object hierarchy is it allows for a dynamic alteration of the inheritance paths. In parallel to what we described for regular objects, dynamic links allow both dynamic objects and compiled objects and classes to inherit from parent or child objects and classes. Dynamic links are a very important capability: they allow one to change the inheritance patterns by changing which objects and classes are associated with each other. This allows objects or classes to inherit from whichever child or parent class or object is appropriate at that particular point in time.

In the insurance system detailed above, when the `+ app_1` dynamic object is linked to the `Unprocessed_applications` class, it inherits the properties and any values it needs from that class. After it is processed and linked to `Accepted_applications`, then it will inherit from this new class (and possibly any others in that branch of the network, for example `Processed_applications`) instead of the first class it was linked to.

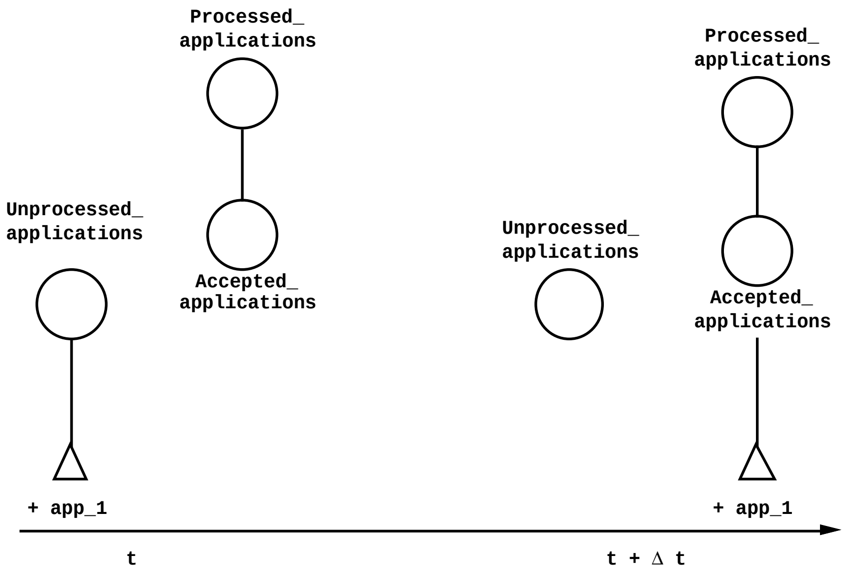


Figure 1-52 Inheriting Along Dynamic Links

Once again, this capability allows the inheritance events to proceed along the optimal path according to the current state of the inference engine.

It is also important to emphasize once again the notion of temporality. When dynamic objects and links are created, inheritance will proceed according to the strategy at the instant when the new object or link is created in the case of property inheritance, and at the instant the inference engine needs the value of a slot in the case of value and method inheritance.

Property Inheritance

Just as new properties are propagated up or down the object hierarchy as soon as they are added (depending on the strategy settings of course!), whenever a dynamic link is created between an object and either an object or a class, the properties are propagated immediately (if allowed by the current strategy). In our insurance system, if the class

`Unprocessed_applications` has the property `processed` then as soon as the dynamic objects are created they will inherit this property:

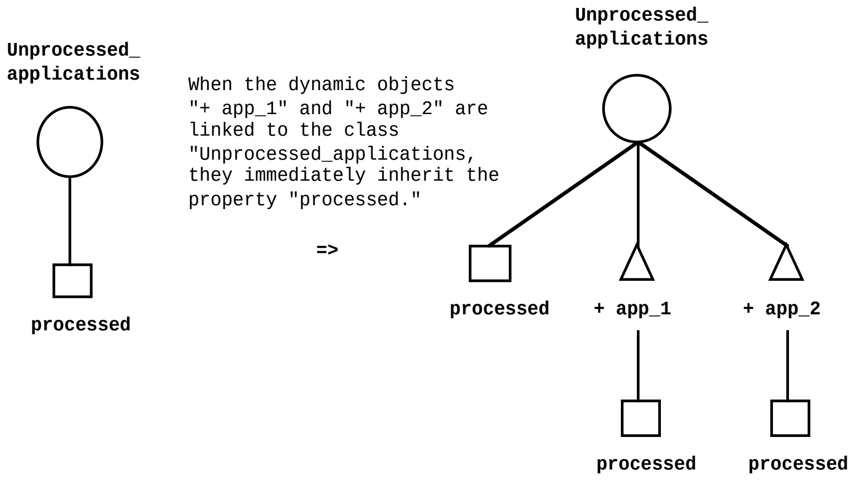

Unprocessed_ applications
processed
When the dynamic objects "+ app_1" and "+ app_2" are linked to the class "Unprocessed_applications, they immediately inherit the property "processed."
=>
Unprocessed_ applications
processed
+ app_1
+ app_2
processed
processed

Figure 1-53 Property Inheritance with Dynamic Objects

Property inheritance allows an object to immediately obtain the properties of its parents (or children) so reasoning can be performed on its slots.

When links are deleted between objects and other objects or classes, the properties remain with the object. Thus when the link between the dynamic object `+ app_1` and the `Unprocessed_applications` class is deleted, the slot `+ app_1.processed` will still exist. It does not matter when or how the link is destroyed – the properties will not be deleted from an object or class. Of course, after a link is destroyed, the child objects or classes will no longer be able to inherit new properties which are added to the parent class.

Value Inheritance

Analogous to the case with regular links in the object hierarchy, values will only be inherited down or up when the current strategy allows it and the value is needed. Thus in our insurance system, if the value of the `Unprocessed_applications.processed` slot is `FALSE`, and we need the value of a slot of one particular object in that class, say `app_1.processed` then this child slot will inherit the value FALSE from its parent class:

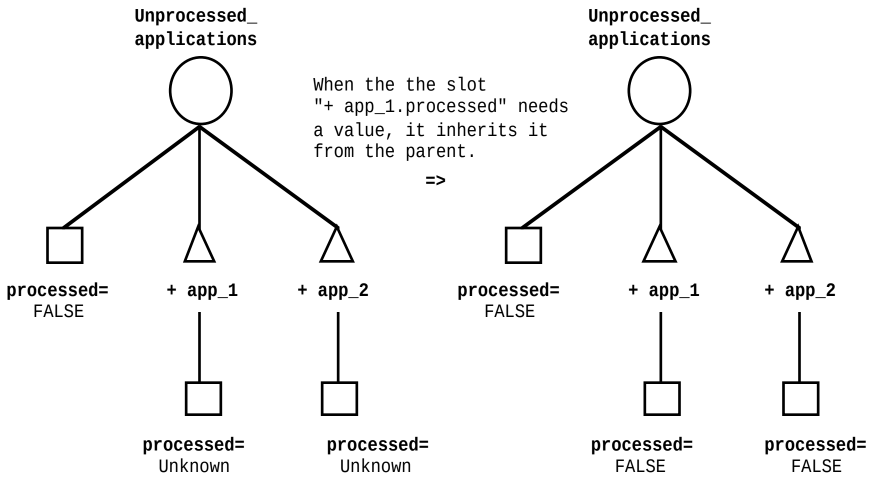

Unprocessed_ applications
When the the slot "+ app_1.processed" needs a value, it inherits it from the parent.
=>
Unprocessed_ applications
processed= FALSE
+ app_1
+ app_2
processed= Unknown
processed= Unknown
processed= FALSE
+ app_1
+ app_2
processed= FALSE
processed= FALSE

Figure 1-54 Value Inheritance with Dynamic Objects

Once again, similar to the case for value inheritance along compiled links, inheritance will only occur when the slot needs a value. Thus the `app_2.processed` slot remains UNKNOWN until the inference engine needs the value.

However, once the applications are processed they will be removed from the `Unprocessed_applications` class and linked to their new classes. Now when the inference engine needs the value of one of their slots, it will inherit the value from a different parent, leading to a completely different value. Inheritance will never be attempted on the class from which the link was deleted.

Inheritance will follow whatever the current strategy is. The fact that the link or object is dynamic has no influence on whether or not the value will be inherited.

Method Inheritance

Dynamic objects cannot have any methods defined locally since they are created at runtime rather than being compiled. This means that they must either inherit their methods or use the default values. Thus inheritance plays a very big role when determining the behavior of dynamic objects.

For example, the inference engine may need to calculate the insurance premium for each application. The method used to calculate it can be stored in the parent class `Unprocessed_applications` and inherited by each dynamic object which is linked to the class:

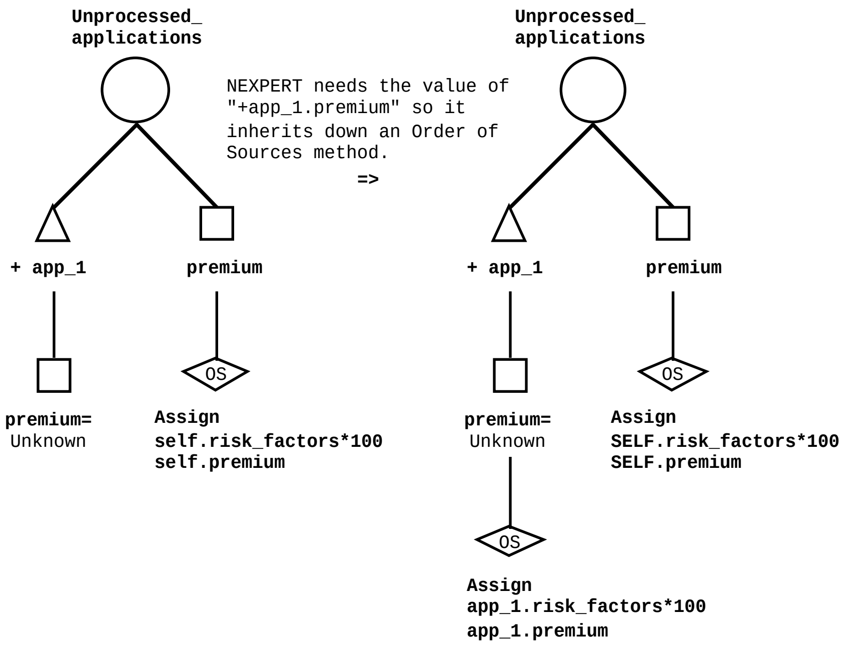

Unprocessed_ applications
NEXPERT needs the value of "+app_1.premium" so it inherits down an Order of Sources method.
=>
+ app_1
premium
OS
premium= Unknown
Assign self.risk_factors*100 self.premium
Unprocessed_ applications
+ app_1
premium
OS
premium= Unknown
Assign SELF.risk_factors*100 SELF.premium
OS
Assign app_1.risk_factors*100 app_1.premium

Figure 1-55 Dynamic Objects Inheriting Methods

Each dynamic object figures out what its associated risk factors are, and then calculates the premium based on this value. The behavior for inheritance of methods by dynamic objects or through dynamic links is exactly the same as it is for the compiled case.

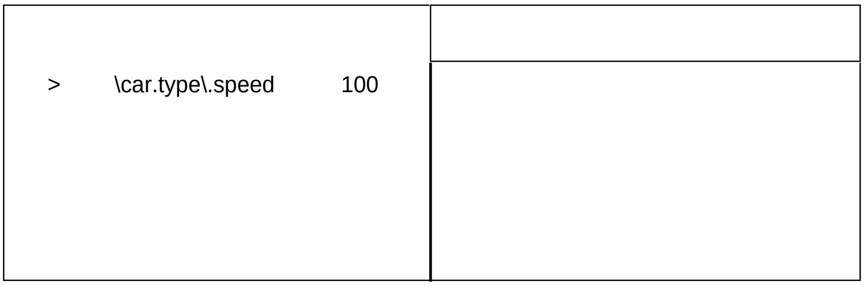
Interpretations

Often the exact slots you want to test in rule or method conditions, send to a particular routine, or send to a particular function are not known before the inferencing session. In this situation, it is necessary to generate the objects, classes, and their properties at runtime rather than explicitly naming them in the rules. The Rules Element allows you to use interpretations to implement this strategy.

Interpretations can be made on either public or private slots, but in the case of private slots certain restrictions apply. Because an interpretation will cause the system to determine a value for a slot, private slots may only be interpreted in the method attached to the private slot or its object components (object, class, or property). When referencing the private slot name in the interpretation, the SELF variable must be used.

Interpreting Slots

An interpretation is a slot value which is interpreted to be the name of an object, class, or property. Thus if the value of the slot `car.type` is “porsche”, then the condition:

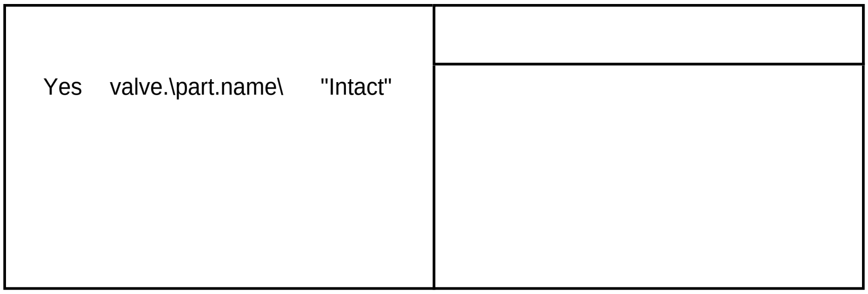


```
>    \car.type\.speed    100
```

Figure 1-56 Simple Object Name Interpretation

would first determine the value of `car.type`, conclude that it is “porsche”, and then test the value of `porsche.speed` to see if it is greater than 100. This capability adds a great deal of flexibility as you only need to write one rule to test this value rather than writing a different rule for every possible value of the interpreted slot.

Object name interpretations yield one object from a group of objects that probably share the property. The same technique can be applied to properties when an object whose property is not known in advance needs to be tested. Thus if the value of the slot `part.name` is “seal”, then the condition:

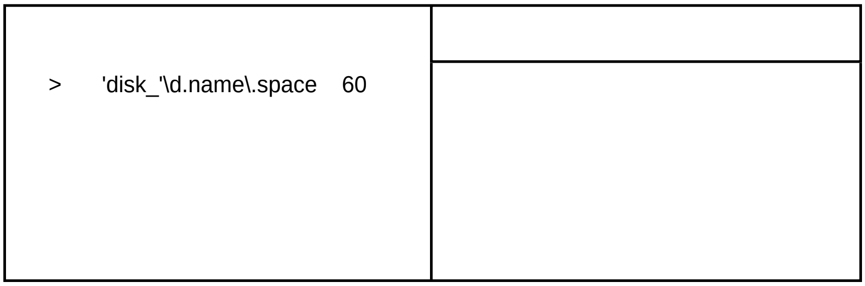


Yes valve.\part.name\ "Intact"

Figure 1-57 Simple Property Name Interpretation

would first determine the value of `part.name`, conclude that it is “seal”, and then test the value of `valve.seal` to see if it is “intact”. This capability can be used to yield data that is specific to a particular object.

In addition to the interpreted part of an interpretation, there can also be a root which is always included with that particular interpretation. In this case, the interpretation acts as the dynamic part of the object or class construction, and the root acts as a constant part. Thus if the value of the slot `d.name` is “brand_x”, then the condition in Figure 1-58 means: “Is the value of the slot `disk_brand_x.space` larger than 60?”.



> 'disk_'\d.name\.space 60

Figure 1-58 Interpretation with Root

Whenever this rule is used, it will always use the prefix “disk_” in the condition displayed above, and then it will add whatever the current value of `d.name` is to the root to test in the condition. So if the value of the slot `d.name` becomes “brand_y” and the rule is re-evaluated, then the condition will test the value of the slot `disk_brand_y.space` against the constant 60.

Interpretations can also be used in the right-hand side actions. In this case, the exact same behavior we have described for the conditions applies. For instance, if the slot `a.p` evaluates to “acid”, then the action displayed in Figure 1-59 would set the value of the slot `nucleic_acid.type` to guanine:

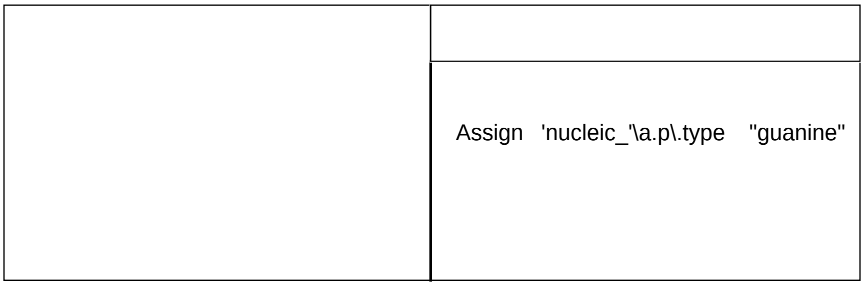


Figure 1-59 Interpretation in Action

Interpreting Strings

Interpretations can also be used in several places within Write and Retrieve statements, as well as in the Prompt Line meta-slot attribute, and the Data Validation meta-slot attribute. In this case, we are evaluating the slot to be a string rather than an object or class name. The syntax is slightly different than the first type of interpretation: `@V(slot)`.

The name of the data file which is retrieved from or written to can be an interpretation as displayed in Figure 1-60:

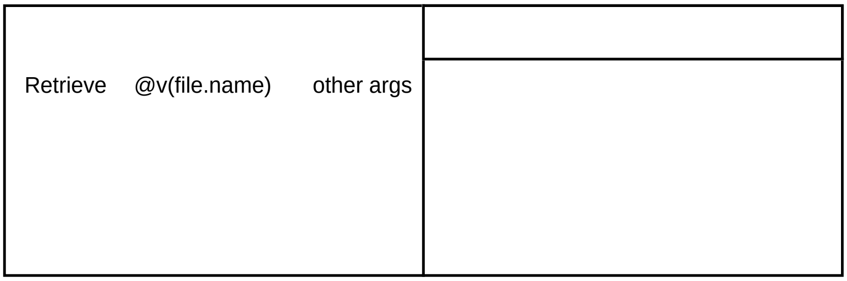


Figure 1-60 Interpreting the Data File Name

When this condition is evaluated, the inference engine determines the value of the slot `file.name`, and then it retrieves whatever information is specified in “other args” from this file. This gives you the ability to retrieve information from or write information to the particular database which is appropriate given the current state of the inference process. This capability, called a *parameterized query* or *variablized query*, adds a great deal of flexibility as well as allowing you to code far fewer rules.

In addition, interpretations can be used within the arguments (third column of the conditions) to the database Retrieve or Write statements. They can be used in the Begin statement to determine dynamically where in a particular database to begin retrieving or writing information then can be used in the query statement or in the query arguments to build a dynamic query, and they can be used in the End statement to tell the database what to do when the interaction is complete. For more information on using interpretations with databases, see the Rules Element Database Integration Guide.

The external file or routine named in the Execute operator can also be interpreted. The syntax is exactly the same as it is for the Retrieve and Write operators:

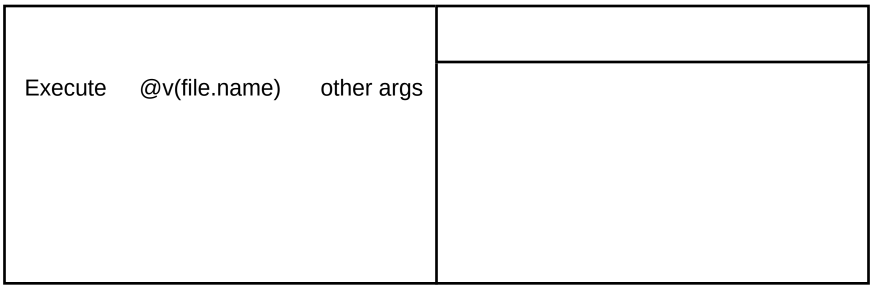


Figure 1-61 Interpreting the Execute File Name

Once again, the inference engine determines the value of the slot `file.name`, and then it executes this function or routine.

Interpretations can also be used in the Prompt Line meta-slot attribute. Recall that the Prompt Line meta-slot attribute specifies how to query the user for the value for a particular slot. Interpretations allow you to build a dynamic query for the user, giving him some information about the current state of the session.

For example, in a network configuration example, there may be currently twenty-five nodes on the network. Rather than just asking the user if he wants to add another one, you can specify a prompt line such as:

```
There are currently @V(nodes.number) nodes on the network.  Would
you like to add another?
```

Listing 1-1 Using Interpretations in the Prompt Line Meta-Slot Attribute

If the slot `nodes.number` currently has a value of 25, the inference engine will query the user: "There are currently 25 nodes on the network. Would you like to add another?".

String interpretations can also be nested. If the value of the slot `component.value` is "nodes", then the above prompt line could also be:

```
There are currently @V(@V(component).number) @V(component).  Would
you like to add another?
```

Listing 1-2 Nested Interpretations in a Prompt Line

The inference engine interprets the value of `component.value` to "nodes", and then the value of `nodes.number` to 25 or whatever the value is. Note that if the value of `component.value` is servers, and the value of `servers.number` is 3, then the prompt will be: "There are currently 3 servers. Would you like to add another?".

The `SELF` variable can also be nested inside a prompt line with the `@V()` interpretation.

The property of a slot can also be interpreted. The interpretation either evaluates to a class or object to build a dynamic slot, or to some other string to send to a database, an execute routine, or the screen.

In summary, interpretations give you the power to write concise, flexible, generic rules, while the inference engine decides at runtime which slots to process with the rules. This is an important point: the slots the rules process are not known when the knowledge base is compiled. Only when the inference engine processes the rule is the exact slot determined.

Pattern Matching

Another method of allowing you to test the values of slots without mentioning them explicitly is through the use of pattern matching. Pattern matching creates a list of objects which belong to a parent class or object. There are two basic types of pattern matching:

1. The first type, called a universal qualifier, allows you to test conditions like: do all members of this class (or do all subobjects of this object) meet this condition, and
2. The second type, called an existential qualifier, allows the test: are there any members of this class (or any subobjects of this object) which meet this condition.

An example of the first type (universal qualifier) is:

>	{valves}.p	100	

Figure 1-62 Universal Pattern Matching (are all)

A pattern matching of the first type is delimited with curly braces. This example condition means: “Do all objects of the class `Valves` (or do all subobjects of the object `valves` depending on whether `valves` is an object or a class) have a slot `p` whose value is larger than 100?”.

Since all of the slots involved in a universal pattern matching must meet the specified test for the condition to be TRUE, the inference engine will stop evaluating the pattern matching as soon as one of them doesn’t meet the specified test. The condition and the rule are set to FALSE.

Pattern matching always ignores private slots when determining the results of the test or generating a list of objects.

The second type of pattern matching (existential) is delimited by the following symbols: < >. An example of the second type of pattern matching is:

>	<valves>.p	100	

Figure 1-63 Existential Pattern Matching (are any)

This condition means: “Are there any objects of the class `Valves` (or are there any subobjects of the object `valves`) which have a slot `p` whose value is larger than 100?”.

The inference engine will always exhaustively test all slots involved in an existential pattern matching, since even if the first “n” don’t meet the specified test, the “n+1st” might meet it. If a pattern match occurs on a set of objects that contains only private slots (no public slots), the condition will be set to FALSE.

A pattern matching always generates a list of objects to test or use, whether it is a pattern matching on a class or an object. Furthermore, this list is always the first level of objects reached on each branch of the object hierarchy. When it reaches an object on one branch it doesn’t search down that branch further. When it reaches classes, it continues to search down the different branches until it reaches some objects. Thus if we have the following object hierarchy:

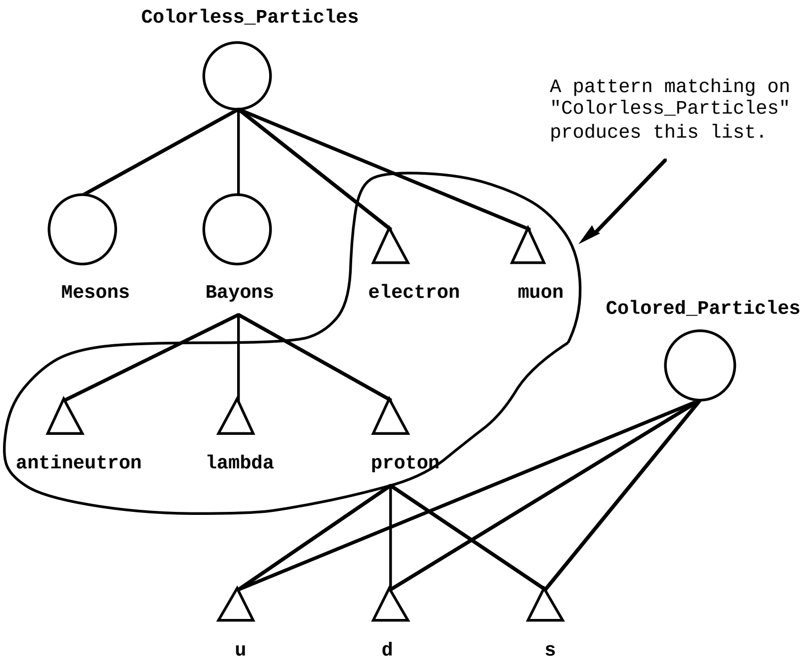


Figure 1-64 Pattern Matching on a Class

A pattern matching on `<|Colorless_Particles|>` (vertical bars are used in a pattern matching, and in conditions and actions in general, to explicitly state that you are using a class) would test the values of slots associated with the objects: `electron`, `muon`, `antineutron`, `lambda`, and `proton`. The inference engine searches down the object hierarchy branches associated with `Mesons` and `Bayons`, but it will not search down the hierarchy branches associated with `Colored_Particles` or any of the objects belonging to the class of `Bayons`.

A pattern matching on the object `body` would test the values of slots associated with the objects `heart` and `left_arm`:

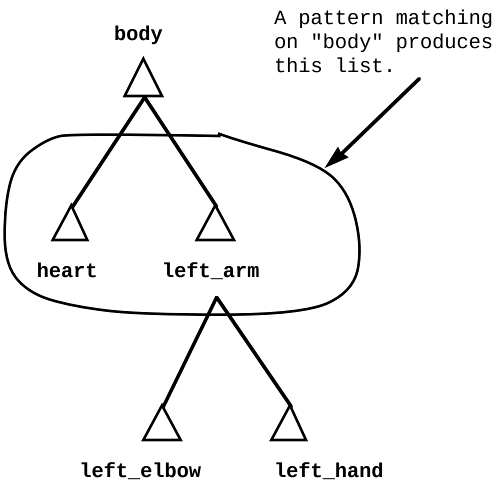
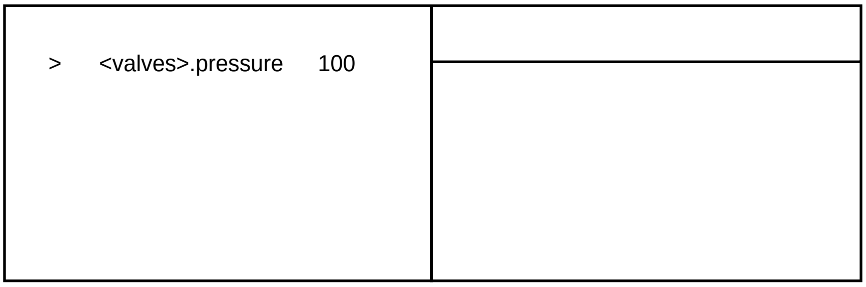


Figure 1-65 Pattern Matching on an Object

Once again, since these are both objects, the inference engine will not search further down the object hierarchy to test the values of slots associated with subobjects of `left_arm`.

Lists Created By Pattern Matchings

An existential pattern matching has a very important function besides being a test: it keeps the list of objects which meet the given criteria. So, in the example below, the list of valves which have a slot `pressure` with a value greater than 100 would be kept for use later within the same rule.

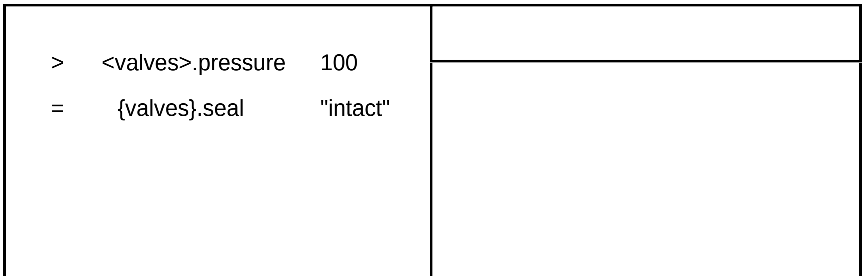


```
>     <valves>.pressure     100
```

Figure 1-66 Pattern Matching in a Condition

This local list can be used in either subsequent conditions or on the right-hand side actions list, but after the rule is completely evaluated then the list will be lost (of course if you wish to save the list, the CreateObject operator can be used to attach it to another class, and subsequent pattern matchings can operate on this class).

If there is a rule which has more than one pattern matching, then subsequent pattern matchings work on the result of the first pattern matching:



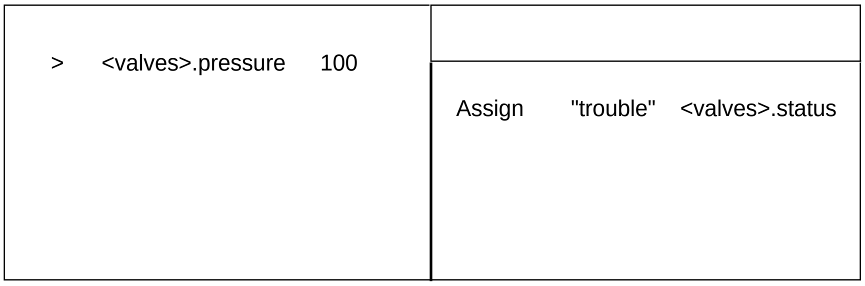
```
>    <valves>.pressure    100
=     {valves}.seal       "intact"
```

Figure 1-67 Pattern Matching Using Previous Result

These two conditions mean: “Are there any `valves` whose slot `pressure` is greater than 100, and of those `valves` for which this condition is TRUE, do all of them have a slot `seal` whose value is “intact”?”. The second condition operates on the result of the first one. It is also important to note that these two conditions (and universal and existential qualifiers in general) are not commutative. For instance, in this example, if we reversed the two conditions, it would mean: “Do all `valves` have a slot `seal` with a value “intact”, and then, since the sub-list created by a universal qualifier must contain all of the objects in the parent class or object (since everything must match the condition for it to be evaluated as TRUE), the next condition finds all valves which have a slot `pressure` with a value greater than 100?”.

A series of pure existential qualifiers is commutative, and a series of pure universal qualifiers is commutative, but a mix of the two is not commutative. Evaluating universal qualifiers first is more restrictive than using existential qualifiers first.

As mentioned above, pattern matching lists can also be used in the right-hand side actions. This serves several general purposes: it can assign a value to the slots of an entire list, it can create or delete links between the list and other classes or objects, and it can send the list to a database or external routine to be processed. An important point about processing lists in the right-hand side actions is that only the objects which satisfied the conditions are processed in the right-hand side actions. For example, assume the class `Valves` has four objects, and two of these four objects have `pressure` slot values which are greater than 100. Thus the rule condition shown in Figure 1-70:



```
>     <valves>.pressure     100
Assign     "trouble"    <valves>.status
```

Figure 1-68 Action on Result of Pattern Matching

creates a list of the two objects whose `pressure` slot values are greater than 100. This sets the rule itself and its hypothesis to `TRUE`, and sends the two objects in the sub-list specified by `<valves>` to the right-hand side actions for further processing. Finally, the rule right-hand side action assigns the value “trouble” to the `status` slots of the two objects specified by `<valves>` sub-list.

There are several important things to note here. First of all, only the objects which pass the left-hand side pattern matching conditions are processed in further conditions or in the right-hand side actions. Secondly, since the right-hand side actions do *not* test the values of any slots, the actions are executed on all of the objects in the list. This means the universal ({ }) and the existential (< >) qualifiers are identical in right-hand side actions.

Multiple Pattern Matchings In One Rule

Sometimes you may want to process several pattern matching conditions in the same rule against the entire list of objects belonging to a parent object or class rather than just the sub-lists which have fulfilled earlier conditions. To do this, you need to use a different number of qualifying brackets around the object or class name.

For instance, the first time you refer to the list, use one set of < > to enclose the parent object or class, and the second time, use two sets of << >>. If two sets of qualifiers are used in later conditions or actions of that particular rule, it will refer to the second pattern matching list, and if one set is used, it will refer to the first list. Any number of qualifying brackets may be used, but they must be balanced on each side of the parent object or class. There does not need to be any order to when the varying numbers of qualifying brackets are used, for example, three can be used in the first condition, one in the second and two possibly never. Thus the rule shown in Figure 1-71:

>	<<A>>.p1	11			
Yes	<A>.p2		CreateObject	{{A}}	|Class|
=	<<A>>.p3	"red"	Assign	5+<A>.p5	<A>.p5

Figure 1-69 Varying Lists of Pattern Matching within one Rule

creates a list of all members of the class A (or all subobjects of the object A depending on whether A is an object or class) whose slot `p1` is larger than 11 and whose slot `p3` is red, and creates another list of all members of the class A whose `p2` slot is `TRUE`. If both of these lists contain at least one object, then the right-hand side actions link the first list to the class `Class`, and increment the `p5` slot of the second list by 5. Note that if these two sets intersect, then some objects will be linked to `Class` and have their `p5` slot modified.

It is also important to remember that a pattern matching construction such as <A> creates a temporary list of all objects meeting the specified criteria. Once this list is created, it is completely independent of the parent class or object A. Thus if we have a rule which creates a sub-list, then deletes an object which belongs to the sub-list from the parent class or object, it will not affect the pattern matching list.

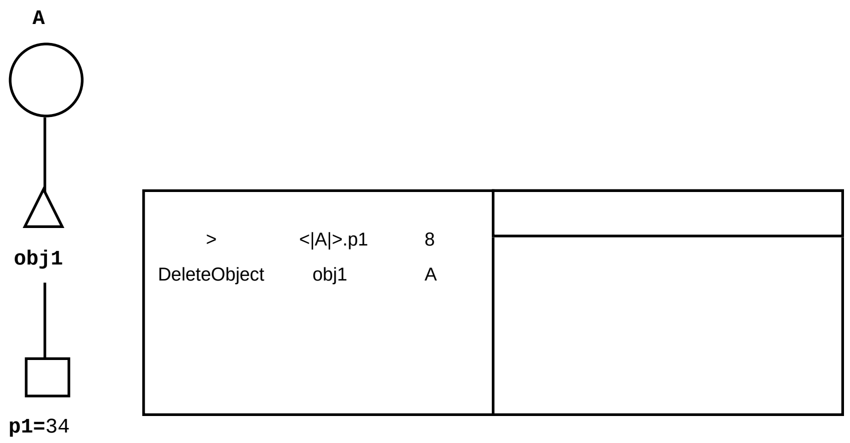


Figure 1-70 Deleting Objects Using Pattern Matching

If `obj1` is a member of `A` before any of the conditions in Figure 1-72 are evaluated, and if `obj1.p1` has a value of 34, then it will be a member of the pattern matching list `<|A|>`. The second condition deletes the link between `obj1` and `A`, but `obj1` will still be linked to `<|A|>`. Of course, pattern matching lists are local to a rule, so at the end of the rule, the pattern matching list will be lost unless a link is created between it and some other object or class.

The dual case is completely analogous. If a pattern matching list is created, and then a subsequent condition links another object to the parent class or object, it will not be included in the pattern matching list even if its slots meet previously evaluated conditions.

Interpretations And Pattern Matching

Interpretations can be combined with pattern matching, providing you with even more flexibility. Interpretations are always nested within a pattern matching – never the other way around. At runtime, the inference engine first evaluates the interpretation, and then it evaluates the pattern matching condition:

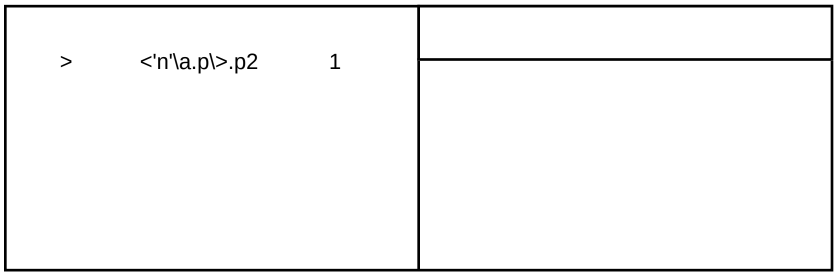


Figure 1-71 Interpretations with Pattern Matching

Thus in the rule above, the inference engine evaluates the slot `a.p`. If it evaluates to 3, then the inference engine does a normal pattern matching on the object or class `n3`, testing each of the objects `p2` slots to see if they are larger than 1. Combining both interpretations and pattern matching allow you to test the objects of any class or object, with the parent class or object being generated at runtime.

Thus we see that pattern matching provides several very important capabilities. It allows you to perform the following tasks.

- Manipulate lists of objects rather than manipulating each of the objects one by one.
- Test lists against conditions to see which slots pass certain criteria and to save those lists.
- Dynamically manipulate sets of objects which aren't known at runtime.

These are very important capabilities for dynamic objects. Since dynamic objects aren't known until runtime, they are never explicitly named in rules or methods. Thus the only way a rule or method can access them is through either a pattern matching or through an interpretation. In addition, as new links are created and destroyed, the same rule will manipulate different sets of objects by using pattern matching conditions. Generic rules created in this manner are more flexible and much easier to maintain.

Refer to the Rules Element Reference Manual for a complete guide to pattern matching syntax.

Pattern Matching With Data Validation

Pattern matching lists can also be used in the Data Validation meta-slot attribute. You can use either universal or existential pattern matching in order to match acceptable values against the list. Thus a data validation function with pattern matching that uses the existential qualifier:

```
        SELF.Value = <Class>.prop
```

requires the current object to match at least one of the `object.property` slots. A data validation function with pattern matching that uses the universal qualifier:

```
        SELF.Value > {Class}.prop
```

requires the current object to exceed every `object.property` slot in the set. The SELF variable is useful when the child of a parent slot that has a data validation function defined inherits the parent's function. Data validation inheritance is automatic and is not under the control of the application developer.

Pattern matching lists generated in the data validation attribute cannot be reused to "reduce the list" through further pattern matching. In other words, the pattern matching is considered local to the data validation expression.

Pattern Matching With Methods & External Routines

Pattern matching lists can also be used in the conditions and actions of methods. If pattern matching is not begun in a conditions list or no conditions are present in the method, pattern matching on the actions side is always performed on the entire list. In this case, there is no difference between universal and existential qualifiers, since there is no testing of slot values. Thus a method action using the existential qualifier:

```
Assign        "trouble"     <|valves|>.status
```

is identical to one using a universal qualifier:

```
Assign        "trouble"     {|valves|}.status
```

Only when pattern matching occurs in method conditions does the type of qualifier matter. When a reduced list is passed to the actions side to be processed, the list refered to by the pattern match in the action must have the same number of sets of brackets as the corresponding condition pattern match. Refer to the early discussion of pattern matching in rule conditions for more information about existential and universal qualifiers. The use of pattern matching in method conditions is analogous to rule conditions, with the exception that method pattern matching does not produce goal-generation.

Pattern matching lists may also be passed to external routines and methods. Once again, existential and universal pattern matching delimiters perform the exact same operation.

Knowledge Islands

A *knowledge island* is a group of related rules. Rules within a Knowledge Island share hypotheses or left-hand side data with hypotheses, left-hand side data, or right-hand side data from other rules. This definition is transitive, ie. if one rule is in the same knowledge island as a second rule (according to the above definition) and that second rule is in the same knowledge island as yet a third rule, then the first and third rules are also in the same knowledge island.

Two rules which are in different knowledge islands have nothing in common except possibly the same right-hand side data. You don't state explicitly which rules you want to be in which knowledge island; it is determined implicitly by shared data and hypotheses.

The importance of knowledge islands has to do with the inference engine's focus of attention. Basically, the inference engine will focus on one knowledge island at a time. Only after everything relevant in one knowledge island is processed will the inference engine move to the next one.

This capability allows you to modularize your knowledge, separating appropriate chunks into different knowledge islands and processing them accordingly. For more information on how the inference engine processes knowledge islands, see the Contexts section in Chapter Two, "Inference Engine Processing."

Knowledge Bases

A *knowledge base* consists of everything we have defined here: objects, classes, properties, rules, and knowledge islands. A knowledge base may contain any number of the above mentioned concepts. This allows you one more level of structure in your application. Just as one entity in your domain is an object, one heuristic a rule, and a set of related objects and rules a knowledge island, a set of related knowledge islands is stored in a knowledge base.

Unlike knowledge islands, you specify explicitly what you want to be in each knowledge base. Of course, at any time, any or all of the data structures can be moved from one knowledge base to another, but this is an explicit action you take rather than a function of how the objects and rules relate to each other.

Knowledge bases can be loaded and unloaded from memory as the system desires using the LoadKB and UnloadKB operators from the conditions and actions of a rule or method. This modularity provides both a structure to the knowledge allowing for easier creation and debugging, as well as freeing memory as only the pertinent objects, classes, and rules are stored in memory.

An application may have just one knowledge base or there may be many that are processed at the same or different times.

Summary

There are two main representational paradigms in the Rules Element: objects and rules. The knowledge base designer describes the world in terms of physical symbols called objects, generalizations of those objects called classes, parts of the objects called subobjects, properties which describe the objects and classes, and slots which are specific properties of objects or classes. Objects and classes do *not* have values, it is the slots of those objects and classes which have values and thus store information:

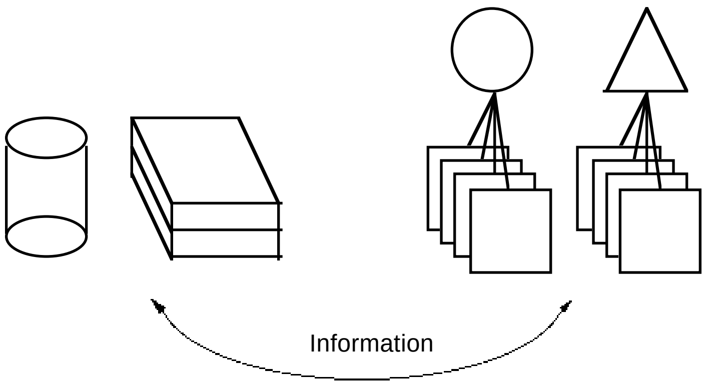


Figure 1-72 Transferring Information

Meta-slot attributes lend more flexibility to slots by providing customizable options for the individual slot. Meta-slot attributes that you can define for a slot include whether the slot value will be accessible by a method only (private slot) or by rules and methods (public slot), a Prompt Line to query the user for a value, Inference and Inheritance Priorities and Settings to determine how the slot will be used when objects want to inherit from it or rules need to reason on it, Data Validation to specify an acceptable range of values or more complex constraint, and an Initial Value option to specify an initialization value. Customization of meta-slot attributes is optional since a default is provided by the system where appropriate.

Methods describe the behavior of slots or modify the default behavior of the system. They can be either user-defined and explicitly executed from a rule or method, or they can be executed by the system. The system methods, called the Order of Sources and If Change, execute a script when they are triggered . The Order of Sources method modifies how the system normally gets a value for the slot, and the If Change method details what should happen whenever the slot's value changes. They are called system methods because, unlike user-defined methods, they do not require the SendMessage operator to be triggered.

The Rules Element supports several important object-oriented features. Objects (and classes) can inherit properties and values from parents or children, and they can inherit methods from parents. The Rules Element supports multiple inheritance, which means an object or class can have many parents and inherit from any or all of them. The Rules Element also supports dynamic objects and dynamic links between objects and classes. Together these object-oriented features allow the representational environment to change as the world they describe changes.

There are three main types of inheritance:

- Inheritance of properties
- Inheritance of values
- Inheritance of methods.

Properties are inherited down (or up) the object hierarchy as soon as a change in the hierarchy is made, whether that is a new object, new property, or a new link. Thus if an object is added to a class and inheritance down is enabled, any properties which the object doesn't have are added immediately. If inheritance up is enabled, then any properties the class doesn't have are added immediately.

If a property is added to a parent class or object, then it will also be propagated up or down the object hierarchy, according to the current strategy and the current links, whether they are dynamic or static. This propagation stops when a particular child class or object already has the new property. Deleting a slot or link never affects property inheritance, ie. a property which was inherited is not taken away.

Value inheritance is only performed when a particular slot needs a value. When different slots change values, this value is not inherited by child or parent objects and classes. When a slot does need a value, it looks at who its current parents (or children depending on the strategy) are, and tries to inherit from them. Once again, if the links are destroyed, the object or class will not lose the value it inherited. Its value will remain unchanged. However, if the Rules Element needs its value in the future, it will not be able to inherit from the object or class from which the link was destroyed.

Finally, method inheritance allows you to specify methods at the parent level and have child objects or classes inherit them. All types of methods can be inherited down the object hierarchy but are never inherited up. Similar to the inheritance of values, they are only inherited when they are needed. The SELF variable allows you to refer to whichever object inherits the method. The inheritance of methods is very important for dynamic objects since they cannot have their own methods.

Rules allow you to describe all of the knowledge in the domain, including the application logic and procedural information:

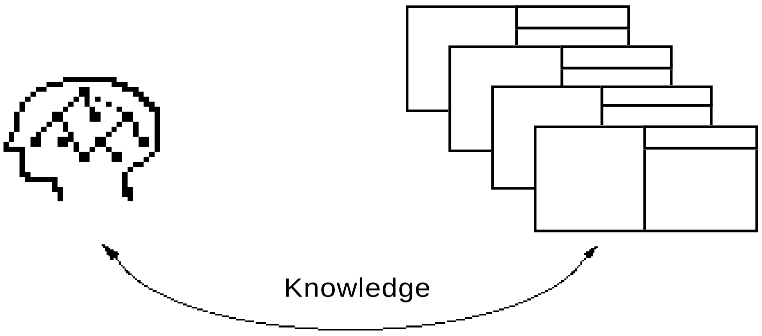


Figure 1-73 Transferring Knowledge

Rules have three basic components, the left-hand side conditions, the hypothesis, and optional right-hand side actions. For the rule to be TRUE, all of the conditions must be TRUE. If one condition is FALSE, then the rule is FALSE. If the conditions are all TRUE, the hypothesis, which is a boolean slot, is set to TRUE as well. One of two sets of actions can be executed depending on the outcome of the conditions. Actions are always executed in order from top to bottom.

Slots which are mentioned explicitly in the conditions or actions of rules and methods are called data. The Rules Element rules and methods operate on data, but they can also operate on slots which are not mentioned implicitly by using interpretations or pattern matching. Interpretations and pattern matching allow the inference engine to reason on whichever objects, classes, and slots are appropriate at the given time.

Interpretations are string slot values which are interpreted to be either the name of another object or class, or the name of some file, database, or external routine. Pattern matching allows one to manipulate all the objects which belong to a parent class or object, whether it is to process the whole list or test the list against the conditions of a rule or method. The lists generated in pattern matching conditions are local data structures which are lost when the rule or method finishes executing. However, within the rule or method these lists can be manipulated in any way.

The application logic which is encoded in the rules then operates on the objects, classes, and slots. Methods lend more object-oriented support by allowing behaviors to be grouped together with the object definition and stored for a particular slot, object, or set of objects (classes). Methods offer an alternative to using multiple rules to perform similar actions on the same set of objects.

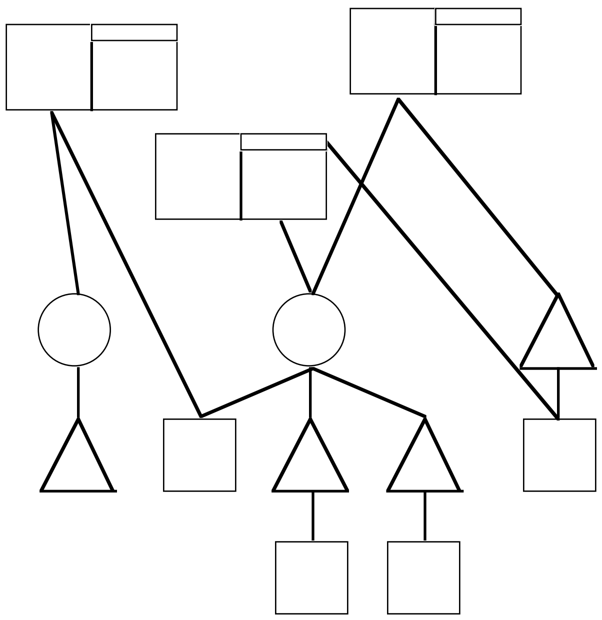


Figure 1-74 Rule and Object Relations

Chapter 2

Inference Engine Processing

This chapter describes how the Intelligent Rules Element inference engine processes the representational structures described in the previous chapter. The manner in which the inference engine processes all of this information is just as important and unique if not more so than the representation capabilities. The inference engine processes events according to an agenda. We will first describe the basic agenda mechanism, and then we will describe how events are prioritized and finally processed.

Introduction

The Agenda is the mechanism by which events are scheduled to happen in the Rules Element during application processing.

One difference between classic programming and agenda-based programming is that in classic programming the stack is executed with a first-in / first-out or last-in / first-out algorithm while agenda-based programming allows you to dynamically modify a list of events by the insertion of new events with varying priorities.

Agenda-based programming also incorporates the notions of conflict resolution, which is a decision between different possible inference paths, and nonmonotonic reasoning, which allows one to change previous conclusions which have been reached.

The Agenda is a *dynamic* mechanism. It is the engine of the Rules Element that provides the central transformation between the perception of events and the actions the system will take. It is modelled after the notion of *focus of attention*. At any time, the complexity of the real world can be reduced to a limited set of parameters and possible decisions. In turn they will affect the world, and perhaps the very next events or actions that were planned. The Agenda accounts for the *adaptability* of the system to real world, real time situations.

The focus of attention is the intersection of the rule and object planes:

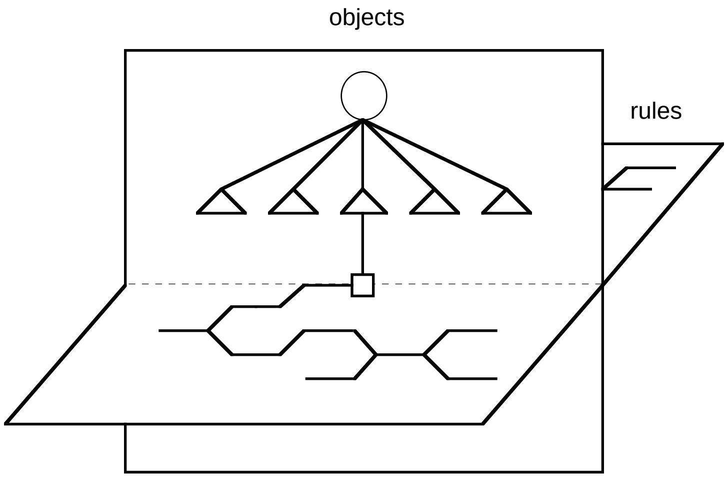


Figure 2-1 Focus of Attention

The agenda determines how these planes move in relationship to each other, in other words where the focus of attention is.

One has to view the Rules Element inference engine as an agent applying knowledge to events of any type. Indeed, inference engine reasons upon objects and slot values, but those objects and values can *come from anywhere*. They can be injected into the system from either Rules Element operators or from outside calls. Except in special cases, we will assume that we are working in this general framework of any object / any value.

The Rules Element is built to provide a comprehensive, powerful set of heuristics for scheduling events. The number of actual agenda operators is a concise set rather than a bewildering group, but it is the large number of arrangements possible that provides the wide-spectrum applicability of the environment.

Finally, the simplicity of its design and basic heuristics accounts for its power, usability, and learnability.

We will approach the Agenda's description by using a priority criteria. In order words we will define the placement of events in the agenda by their relative priority (or importance, level of granularity, and so forth). We begin by describing events at the rule-level, then the object-level, and finally an integration of the two.

Evaluation Concepts

The Rules Element agenda contains a prioritized list of hypotheses, not rules. It is important to realize the inference engine tries to evaluate the state of a hypothesis, not that of a particular rule. In its quest to find the value of a hypothesis, one or many rules are often executed, but they are evaluated in order to find the state of the hypothesis.

In order for the inference engine to reason, there must be at least one hypothesis on the agenda. You can explicitly place a hypothesis on the agenda by *suggesting* it, or it can be put on the agenda as a relevant goal by one of the inference search mechanisms. If *knowcess* is triggered while there is at least one hypothesis on the agenda, the inference engine will begin evaluating each of the hypotheses on the agenda as well as any other relevant events. The application processing session ends when all relevant events have been completely processed.

Rule Evaluation

The evaluation of a rule is in fact the most basic event in application processing. The hypothesis of a rule corresponds to its name, topic, goal, or whichever interpretation can be given. Evaluating a rule in the Rules Element always consists, in the end, of attempting to find the state of its hypothesis.

When evaluating a rule, the default strategy is to evaluate the conditions from top to bottom. However, this default strategy can be modified by using the slot's inference priorities. The condition which has the slot with the highest inference priority is processed first, then the condition which has the second highest inference priority, and so on. Let's assume we have a rule with the following conditions:

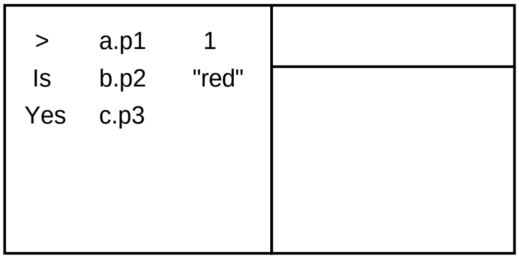


Figure 2-2 Rule Evaluation

By default, the inference engine will evaluate the first condition bearing on the slot `a.p1` first, then the second condition bearing on the slot `b.p2`, and finally the third condition bearing on the slot `c.p3`.

However, if we modify the inference priority of `b.p2` to 10 and `c.p3` to 5 while leaving `a.p1` at the default value of 1, then the second condition will be evaluated first, then the third condition, and finally the first condition.

If a condition has several slots in it, then the highest inference priority of any slot is used for the conflict resolution. If the first condition has a slot with the default priority of 1, the second condition has two slots with priorities of 15 and 20, and the third condition has two slots with priorities of 1 and 25, then the third condition is evaluated first since the slot with the highest inference priority in it is 25, followed by the second condition since it has a slot with priority 20, and finally the first condition since its highest priority slot has a value of 1.

As explained in the Inference Priority part of the Meta-Slot section, inference priorities can also be dynamic. This means we can attach an inference atom to any slot. When the inference engine evaluates the rule, the inference atom's current value becomes the slot's inference priority.

A hypothesis is a boolean slot. As such, it must take one of the four possible values for a boolean slot: `UNKNOWN`, `TRUE`, `FALSE`, or `NOTKNOWN`. The value of a hypothesis depends upon the value of the conditions in the rule leading to it as shown in Table 2-A.

State of Rule Conditions	State of Hypothesis
not investigated	UNKNOWN
all verified	TRUE
at least one rejected	FALSE
at least one condition not determined (and the others verified)	NOTKNOWN

Table 2-1 Determining the State of a Hypothesis

The evaluation of a rule as `FALSE` based on the lack of evidence that all conditions are `TRUE` corresponds to a mode of reasoning with a closed-world assumption, an important characteristic of our reasoning capabilities. The closed-world assumption means that not knowing can generate actions or decisions.

Multiple Rules Evaluation

Several rules leading to the same hypothesis generate an *or* graph using the logical states of the rules described above:

- All rules must be FALSE for the hypothesis to be FALSE
- At least one rule must be TRUE in order for the hypothesis to be TRUE
- At least one rule must be NOTKNOWN and no rules TRUE for the hypothesis to be NOTKNOWN

As stated above, the inference engine reasons according to the closed-world assumption. This means that if the rules leading to a particular hypothesis are evaluated as FALSE, then the inference engine will conclude that hypothesis is FALSE as well and will use that determination in further reasonings. Thus in the diagram in Figure 2-3 where both rules leading to `Hypo.h` are FALSE, `Hypo.h` is concluded as being FALSE as well, and this information is available for further use:

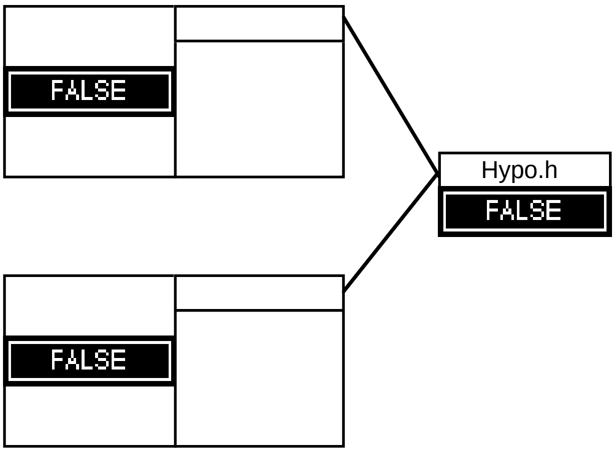


Figure 2-3 FALSE Rules

While the conditions within a particular rule are "ANDed" together (so that all of them must be TRUE for the rule to be TRUE), multiple rules pointing to the same hypothesis are connected by "ORs". Thus if any number of rules pointing to a particular hypothesis are evaluated as FALSE, but one is evaluated as NOTKNOWN then the hypothesis will be NOTKNOWN:

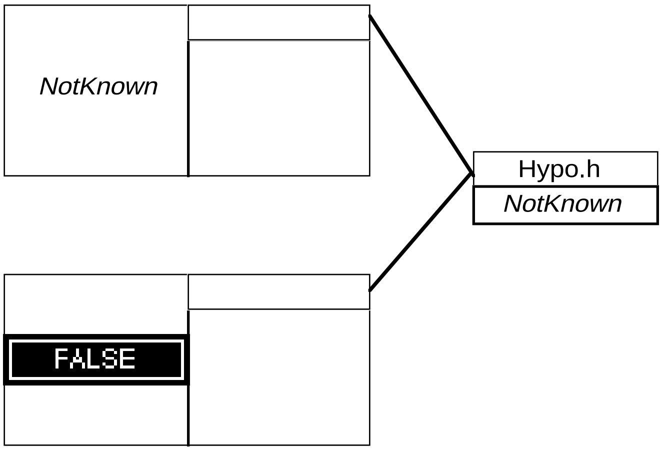


Figure 2-4 FALSE and NOTKNOWN Rules

If a rule pointing to a particular hypothesis is evaluated as TRUE, then the hypothesis will be evaluated as TRUE regardless of the values of the other rules leading to the hypothesis in question:

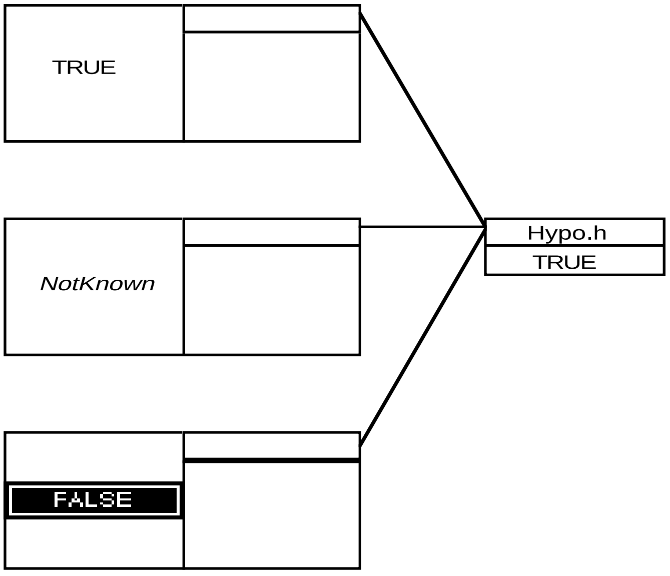


Figure 2-5 FALSE, NOTKNOWN, and TRUE Rules

In summary, we see *rules* are considered to be

- UNKNOWN until the inference engine tries to evaluate them (or when they are reset)
- TRUE if the inference engine evaluates the all conditions of a rule to TRUE
- FALSE if any condition is FALSE, and
- NOTKNOWN if one of more conditions are NOTKNOWN and all of the others are TRUE

Hypotheses are:

- UNKNOWN until the inference engine tries to evaluate them (or when they are reset)
- TRUE if any rule leading to a hypothesis is evaluated as TRUE
- NOTKNOWN if no rules leading to a hypothesis are TRUE, but at least one is NOTKNOWN, and
- FALSE if all rules leading to a hypothesis are FALSE

However, the actions list is actually linked to the conditions and **not** to the hypothesis. If several rules lead to the same hypothesis and some of the rules are TRUE and some are FALSE, it is possible that each rule can still execute a list of actions. For example, assume we have three rules leading to a hypothesis, with one of them evaluated TRUE, another NOTKNOWN, and the third FALSE:

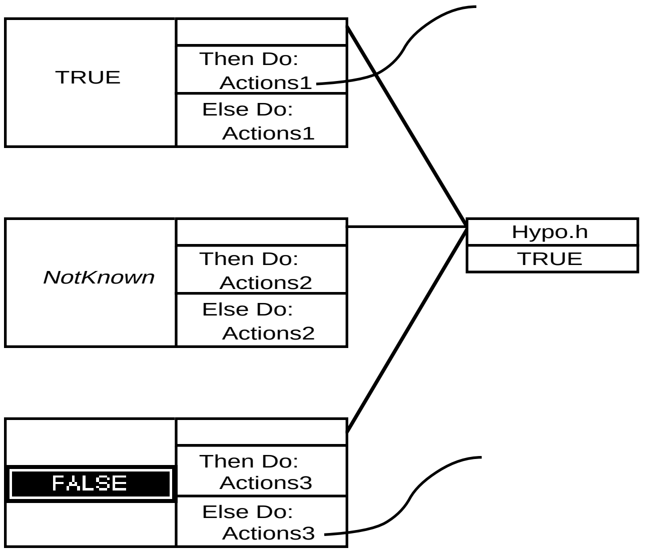


Figure 2-6 Multiple Rules Leading to One Hypothesis

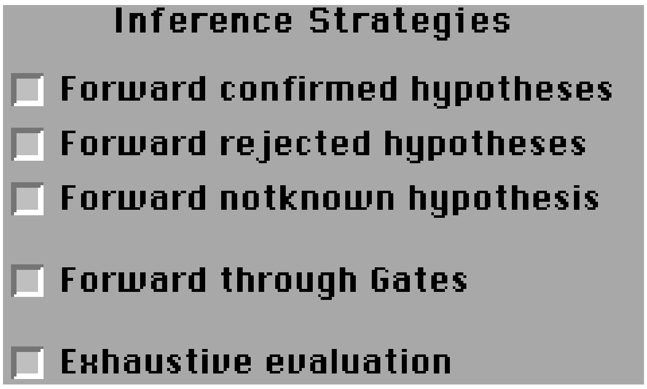
As Figure 2-6 shows, Hypo.h is set to TRUE since one of the rules leading to it is set to TRUE. In this case “Then Do Actions1” are executed, but we also see that “Else Do Actions3” are executed. Since the conditions list of the second rule is NOTKNOWN, neither of the actions lists for this rule is evaluated. To distinquish between the two separate sets of actions that each rule may have, we refer to the actions list as true or false actions because they depend on the evaluation outcome of the rule conditions.

Thus we see that the agenda consists of a prioritized list of hypotheses rather than rules to evaluate. When the inference engine needs to find the value of these hypotheses, it uses the rules leading to them. The actions are linked to the conditions list and one list of actions may be executed whether the outcome is TRUE or FALSE.

Exhaustive Evaluation

The fact that multiple rules leading to the same hypothesis generate an “or graph” brings up an interesting question. If there are several rules leading to the same hypothesis and the first rule is evaluated as TRUE, then we know the hypothesis will be TRUE independent of the state of the other rules. The question is, should we still evaluate the other rules? This is a question of exhaustivity and the correct answer depends on both the application in question as well as the particular section of the application.

The default strategy is to exhaustively evaluate all the rules leading to a hypothesis. Thus if the first rule is evaluated as TRUE, the inference engine will continue to evaluate the other rules. This strategy can be changed globally for an application by unchecking the Exhaustive evaluation checkbox in the Rules Element’s Strategy dialog box:


Inference Strategies

- [ ] Forward confirmed hypotheses
- [ ] Forward rejected hypotheses
- [ ] Forward notknown hypothesis
- [ ] Forward through Gates
- [ ] Exhaustive evaluation

Figure 2-7 Selecting Exhaustive Evaluation

In this case, as soon as the inference engine found a rule TRUE, it would stop evaluating the other rules leading to this hypothesis.

Exhaustive evaluation can also be turned off locally using the Strategy operator from the conditions and actions of rules or methods.

No matter what the strategy is, rules will continue to be evaluated until at least one is found to be TRUE or all enabled rules leading to a hypothesis have been evaluated.

Conflict Resolution

In the above example, one may want to impose an order on the rules to be evaluated. This can be done by assigning a *rule priority* to each one. Similar to inference and inheritance priorities, when the inference engine evaluates a hypothesis with multiple rules leading to it, the rule with the highest rule priority will be evaluated first. Also similar to inference and inheritance priorities, rule priorities can be either static or dynamic. Both types of rule priorities can be set in the rule editor:

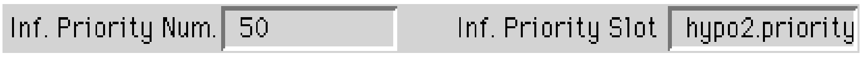

Inf. Priority Num. 50 Inf. Priority Slot hypo2.priority

Figure 2-8 Rule Conflict Resolution

If a rule has an Inference Priority Slot declared and its value is KNOWN, then the value of this integral slot (`hypo2.priority` in the above example) will be used in conflict resolution to determine when this rule is evaluated. If a Priority Slot hasn't been declared, then the value of the Priority Number will be used (50 in the above example). Finally, if neither have been declared, then the default value of 1 is used.

If several rules leading to the same hypothesis have the same inference priority, regardless of whether it is determined by a static or dynamic priority, then the rule which contains the condition with the highest inference priority on one of its slots will be evaluated first.

The order in which rules are evaluated can be important in many situations, since each rule can trigger lots of other events and the order in which they happen can be very important. In addition, if a non-exhaustive search is being used, then evaluation will stop as soon as one rule leading to a particular hypothesis has been verified. This means that many of the rules may not even be evaluated, hence their actions will never be used.

Inference Search Disabling

There are also several special categories of rule priorities. Any rule which has a priority:

- Less than -20,000 will be disabled from all forward and backward processes
- -20,001 < priority < -10,000 will be disabled from forward processes but will function normally with backward processes
- -10,001 < priority < -5,000 will be disabled from the gate mechanism
- -5,001 < priority < -1,000 will be disabled from rule and method actions
- Priority > -1,001 is enabled for all forward and backward processes

The exact meaning of what these categories disable will be explained in the appropriate section.

Method Evaluation

The evaluation of a method is an application processing event analogous to rule evaluation. When a method is triggered, like a rule, the conditions and actions of the method are evaluated and executed. Unlike rule evaluations, however, the evaluation of methods does not involve a hypothesis. In fact methods should be more closely identified with objects than rules because they serve as a procedure or routine that acts on a specific object.

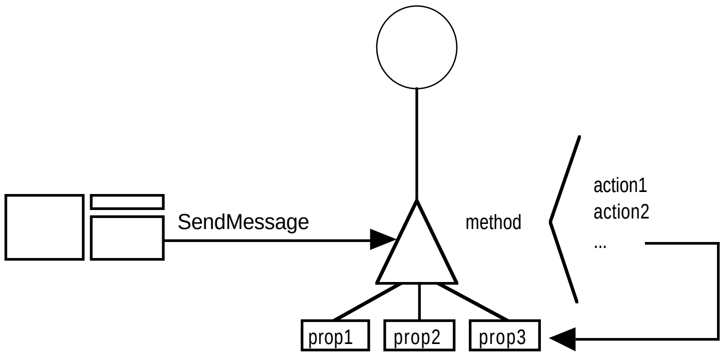


Figure 2-9 Method Attached to Object Upon which It Acts

Message Passing

Although methods act in general upon the object for which they are specifically designed, they do have the capability to trigger other methods from their actions list. This mechanism is an important features of object-oriented systems. Conventional systems always take all the data available and then select a particular subset of the data based on the desired operators. A method, on the other hand, in general deals with the data associated with the object to which it is attached (or through inheritance, the parent object's data). If a method needs other data to be processed, it sends a message to another object. Methods associated with the other object actually manipulate any data associated with the other object and arguments can be passed back to the calling method or not. Data passed by reference act as global variables since the original arguments can be modified by the actions on the local argument in the target method. Data passed by value has no effect on the original arguments.

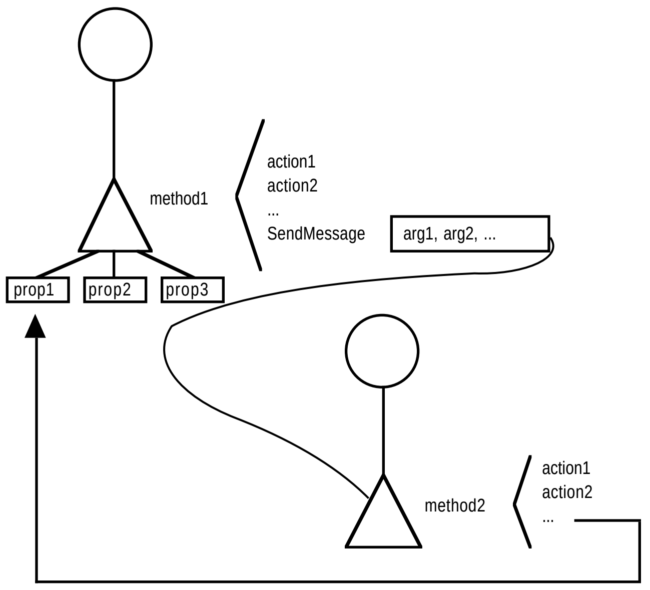


Figure 2-10 Message Passing Between Methods

The process which triggers a method attached to an object is called *message passing*. More specifically, message passing is the process by which the message name is bound to a specific method. A message is made up of three parts: 1) the name of the object or objects to receive the message (called the list of addressees), 2) the name of the method to be triggered, and 3) an optional arguments list. In the Rules Element the application developer has the choice to initiate message passing from a rule or method through a single SendMessage operator.

Execution

The SendMessage operator allows the application developer to specify which methods to trigger and when. When the developer determines the circumstances to trigger a method and execute its actions list, it is called a *user-defined* method. Another category of methods differentiated by the way they are triggered is the *system methods*. There are two system methods, which as the name implies, are triggered by the inference engine under special circumstances:

- The *order of sources* method is triggered automatically when the value of the slot is needed in the course of inference processing and was found to be UNKNOWN.
- The *if change* method is triggered automatically when the value of the slot is changed in the course of processing.

User-defined methods are not limited to slots; however, they must be explicitly triggered through a SendMessage operator that appears in a condition or action of a rule or method. Whenever a method is triggered by the SendMessage operator, the system executes the complete list of actions. For details about the SendMessage operator see the Intelligent Rules Element Reference Manual.

The list of conditions is optional for all methods. If no conditions are present, the system automatically executes the “Then Do” actions list when the method itself is triggered. If method conditions are present, the system executes one of two different lists of consequent actions (“Then Do” and “Else Do”) depending on whether the method is satisfied or not.

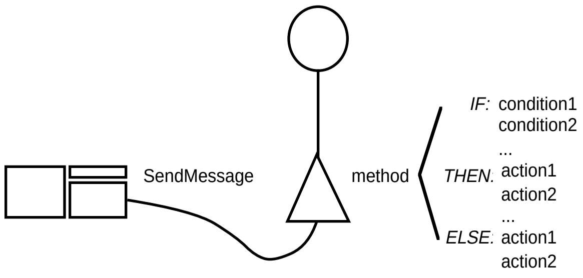


Figure 2-11 Conditional Methods

For the list of Then Do actions to be executed, all of the method’s conditions must evaluate to TRUE. The conditions are thus implicitly linked by the logical “and” operator. If you want to achieve the effect of a logical “or,” you can use the OR boolean operator within one condition.

If present, conditions within a method are always evaluated sequentially, in the order they appear in the method definition; unlike rule conditions this evaluation order is <u>not</u> altered by the inference priorities of the data involved.

Information Hiding

The idea that an object should be a self-contained unit that includes data and methods to process that data, supports another object-oriented feature called *information hiding*. If the application is developed with this approach enforced, it guarantees that, if the developer needs to change the object hierarchy (including the objects, slots, and properties), then the methods that access that data can be easily located within the application. In contrast to modifying data acted upon by rules, or in the case of conventional systems by procedural code, the consequences are typically widespread and less easy to locate.

To help enforce the purely object-oriented approach to triggering functions, the developer may want to enable the slot's meta-slot attribute for privacy. Unlike public slots, the Rules Element will not allow private slots to be used in rule conditions and actions. Any attempt to develop rules with private slots will generate an error message when the rule is compiled. The system allows only one means of accessing the data of a private slot: through a method associated with the slot or its object components (object, class, or property). This requirement helps the developer be sure that no part of their application will modify the stored value other than the slot's associated method.

Inferencing Mechanisms

There are seven types of inference search mechanisms:

- Backward chaining
- Suggesting
- Hypothesis forward
- Semantic gates (or simply gates)
- Forward action-effects (from rule or method action lists)
- Volunteering
- Context links.

Each of these search mechanisms helps the inference engine expand the search for relevant conclusions without exhaustively evaluating all of the rules in the knowledge base. Each of these search mechanisms will be described according to their priority in the inference engine.

Backward

Backward chaining is the highest priority event in the inference engine. It is based solely on the evaluation of the hypotheses. If a condition (or an action with the `Assign` operator) bears on an `UNKNOWN` boolean slot which is in fact a hypothesis, then the rules pointing to that hypothesis will be evaluated *immediately*. Thus the evaluation of these rules has been inserted in the agenda, and they will be evaluated before finishing the evaluation of the conditions or actions which caused the backward chaining to occur.

If we have the rules depicted in Figure 2-12, and are currently evaluating the rule leading to the hypothesis `Hypo2.h`, then when the inference engine comes to the condition “`Yes Hypo.h`”, the inference engine will evaluate the two rules leading to `Hypo.h` before it finishes evaluating the rules leading to `Hypo2.h`:

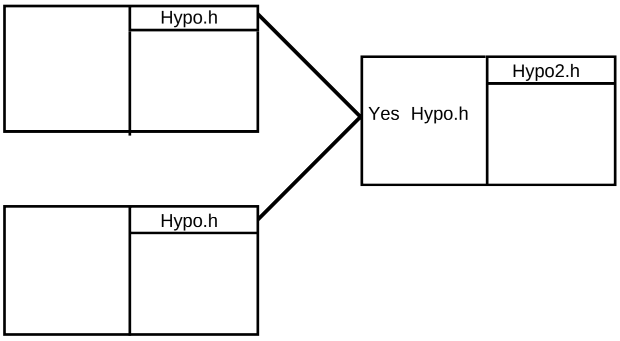


Figure 2-12 Backward Chaining

The determination of the value of the condition is done according to the principles described previously in the cases of one or more rules.

From the program execution standpoint, we have *inserted* the evaluation of those rules in the evaluation stack of the first rule. The conditions following "`Yes Hypo.h`" in the first rule are conditions which will be evaluated *after* the evaluation of the two rules leading to `Hypo.h`.

Slots which are both a data and a hypothesis are called *subgoals*. Hypotheses which are not subgoals are called *terminal hypotheses*.

The next illustration shows a deeper structure. The darkest conditions and hypotheses are executed before the lighter conditions and hypotheses:

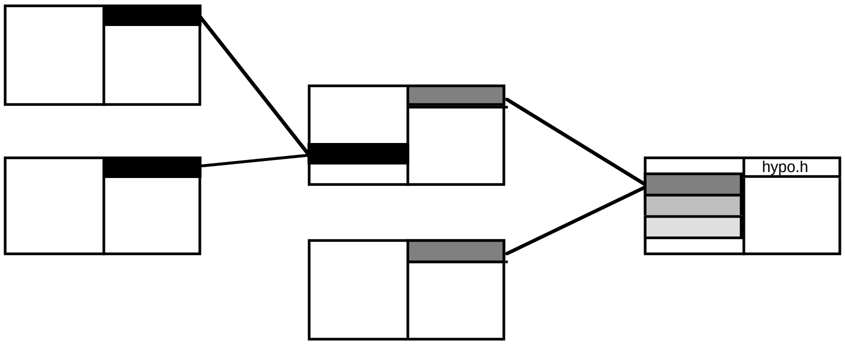


Figure 2-13 Multiple Level Backward Chaining

The same principles described above apply recursively to the evaluation of the hypothesis `hypo.h`. Namely, when the inference engine begins evaluating `hypo.h`, it comes to another hypothesis in the conditions whose value is needed to continue processing the rule leading to `hypo.h`. So the inference engine backward chains on this hypothesis, which has two rules leading to it. While evaluating the first of these rules, the inference engine needs to backward chain again, inserting yet another hypothesis onto the agenda. All the rules leading to this hypothesis are evaluated (the rules with solid black hypotheses), then the inference engine finishes evaluating the rules leading to the second hypothesis (middle color shading), and finally the inference engine finishes evaluating the rule leading to `hypo.h`.

As mentioned earlier, there are many different ways to cause a backward chaining event to occur. The conditions:

```
Yes slot
No slot
Assign slot slot
```

where `slot` is an `UNKNOWN` hypothesis will backward chain on any rules pointing to slot (subject to the control mechanisms). The state of the first two types of backward chaining conditions depend on the value of the hypothesis in question, whereas the result of the last condition is `TRUE` regardless of the final value of the slot since the inference engine will always be successful at assigning the value of slot to slot.

Backward chaining can be initiated from any rule or method action using the "`Assign slot slot`" syntax where `slot` is an `UNKNOWN` boolean hypothesis. Similar to the conditions list behavior noted above, when the inference engine evaluates an action of this kind, it will insert this backward chaining event into the agenda, and it will be evaluated immediately. Any rules leading to this hypothesis will be evaluated before any further actions (or their consequences) are executed.

Suggest

Suggesting a hypothesis from the development interface puts it on the agenda for immediate evaluation. When you suggest a hypothesis, you are in essence telling the inference engine that the hypothesis is an important goal, and, as such, it should be investigated as soon as possible.

Suggested hypotheses have priority over hypotheses generated by any other inference search mechanism except backward chaining:

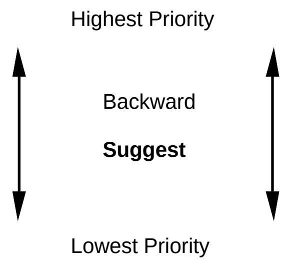


Figure 2-14 Agenda Priorities

Thus the inference engine will evaluate all relevant backward chainings, and then, as soon as it has evaluated these, the suggested hypothesis with highest priority will be evaluated.

Unlike all of the other search mechanisms (except Volunteering), suggesting hypotheses is always an explicit action taken by the user. Thus the user is giving the inference engine a relevant goal to investigate, whereas the other inference search mechanisms generate relevant hypotheses based on the current state of the inference engine.

Hypothesis Forward

The next highest priority after backward chaining and suggested hypotheses are hypotheses put on the agenda due to hypothesis forward events. Hypothesis forward is a consequence of investigating subgoals as opposed to a terminal hypothesis. It consists of not only exploring the backward chaining associated with the subgoal, but *immediately* thereafter placing on the agenda the hypotheses of the rules in which the subgoal is involved as a data. Thus, it is a *forward* propagation from the hypothesis:

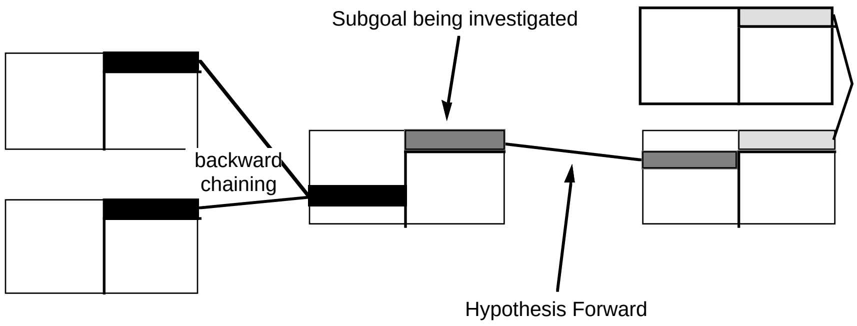


Figure 2-15 Hypothesis Forward

Figure 2-15 illustrates such a series of events. The lighter the pattern, the later the event takes place in the evaluation.

When it occurs, hypothesis forward has the next level of priority in the agenda after backward chaining and suggested hypotheses. Thus the order of evaluation of the events we have discussed so far is:

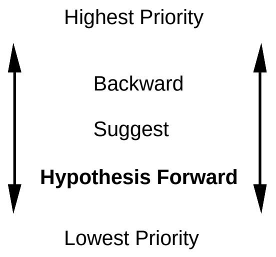


Figure 2-16 Agenda Priorities

An important difference, though, is the fact that backward chainings are evaluated immediately (even before the current rule or action proceeds another step), while suggested and hypothesis forward events are queued and evaluated as soon as the current rule has been completely evaluated.

While the standard backward chaining is performed to establish the value of the hypothesis, the fact that it is a subgoal is then exploited by the inference engine. The inference engine follows backward structure in both directions.

Hypothesis forward does not depend upon the value of the subgoal originally placed on the agenda. Propagation is recursive until one or more terminal hypotheses are reached.

Gates

Gates are structure-based inference mechanisms which account for the opportunistic insertion of hypotheses on the agenda. It is a mechanism designed to expand the search space in a selective, relevant fashion while always reducing the exhaustivity of the search. Gates are the basic mechanism for the automated goal generation and opportunistic reasoning.

The effect of gates is to place new hypotheses on the agenda. Gates are generated during the evaluation of the conditions list of rules. They are based on a structural analysis of rules which identifies whether any two rules share a public slot in their conditions. Private slots cannot generate gates because they cannot appear in rule conditions. Also, the gates mechanism is completely disabled for method conditions; so that a slot that is shared between a rule and a method or between two methods has no effect on the agenda. Gates depend upon a pre-testing of rule conditions that share the public slot.

When any new slot is evaluated in the conditions of a rule, the inference engine checks to see if any other rules also have this slot in their conditions. If any target rules have this slot in their conditions, the inference engine tests the value of the particular conditions which include this value. If the condition is `TRUE`, the associated hypothesis is put on the agenda. If the condition is `FALSE`, then the hypothesis is not put on the agenda. Notice the hypothesis will remain `UNKNOWN` rather than being evaluated to `FALSE`, an important difference.

The following diagram shows two rules with a common slot:

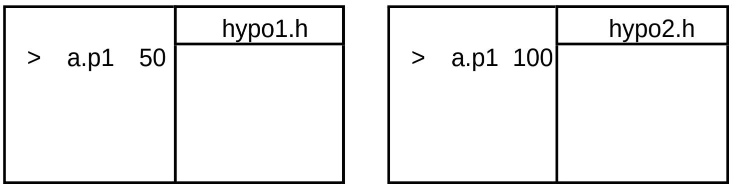


Figure 2-17 Gates

Both rules contain the slot `a.p1` in their conditions. Sharing a particular slot or pattern matching is a necessary condition for the gate mechanism to occur.

The two rules in the above example are independent (different hypotheses). Suppose `hypo1.h` is suggested (the source rule) and its rule is evaluated. A value will be given for the slot `a.p1`.

If `a.p1` has a value of “75”, then the condition bearing on `a.p1` in the top rule is `TRUE`. The condition bearing on `a.p1` in the second rule is `FALSE`. Thus `hypo2.h` will not be put on the agenda (the hypothesis will remain `UNKNOWN`).

If `a.p1` has a value of “200”, then the condition bearing on `a.p1` in the first rule is `TRUE` *and* the condition bearing on `a.p1` in the second rule (the target rule) is `TRUE` as well. In this case, `hypo2.h` will be put on the agenda for later evaluation. What happened was the “passage of a gate” due to the value of the condition on the target rule.

It is important to note that the value of the target condition is only computed for the purpose of knowing whether to put the associated hypothesis on the agenda. In other words, by the time this hypothesis and its rules are evaluated, the value of `a.p1` might have changed. However, making the *assumption* that the target hypothesis is relevant was a valid action at the time the gate is generated.

Gates-generated hypotheses have the next highest priority level after backward chaining, suggested, and hypothesis forward generated hypotheses. Thus they will be processed as soon as all of these higher priority events have been evaluated:

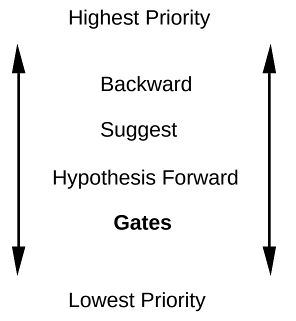


Figure 2-18 Agenda Priorities

Gates place hypotheses on the agenda whose evaluation can in turn generate any other type of agenda events, such as backward chainings and other gates. The notions described above are applicable in such cases.

As stated above, for a gate to occur, the condition which involves the associated gate must evaluate to `TRUE`. If the condition in question has an expression involving several slots, a gate will only occur if all the slots in the expression are `KNOWN`, and, with their current values, the condition is `TRUE`. The inference engine does not ever “force” a gate by evaluating an `UNKNOWN` slot. Gates put currently relevant hypotheses on the agenda.

Assume we have a condition bearing on the slot `a.p1` in the source rule, and a condition bearing on both `a.p1` and `b.p2` in the target rule:

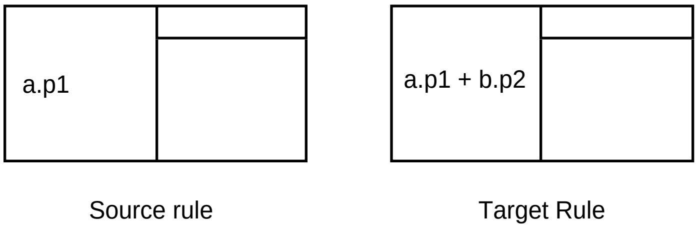


Figure 2-19 Gates with Expressions

If the value of `b.p2` is `UNKNOWN`, then a gate will not occur regardless of the value of `a.p1`. If the value of `b.p2` is `KNOWN`, and using the values of both `a.p1` and `b.p2` the condition is `TRUE`, then a gate will occur.

Forward Action-Effects

The actions lists of rules and methods can be triggered in two possible situations, depending upon the evaluation outcome of its conditions list. If the conditions list has been found `TRUE`, one set of actions is triggered; if the conditions list has been found `FALSE`, a separate list of actions is triggered. These actions are maintained in the rule or method as two separate lists and are known as TRUE actions and FALSE actions to signify their relationship with the evaluation of the conditions list.

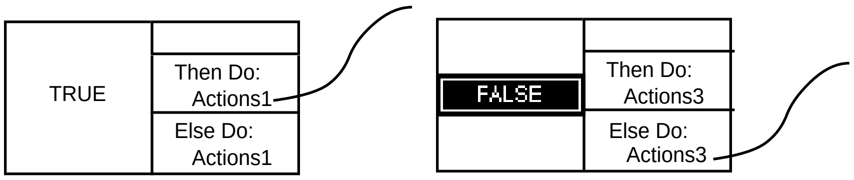


Figure 2-20 TRUE and FALSE Actions Lists

The actions lists of rules and methods are a forward propagation mechanism, similar to gates (except gates allow conditional forwarding). The ultimate result in terms of the agenda is to place new hypotheses on the agenda and schedule new events. In the case of a rule, the hypothesis becomes `TRUE` and then the actions are triggered in order from top to bottom. In the case of a method, no hypothesis exists, so the actions are triggered immediately in order from top to bottom.

Actions which may alter the agenda are triggered by three different operators:

- Assign
- Execute (if the routine affects slot values and / or suggests hypotheses through the Rules Element Application Programming Interface or Rules Element Execute Library)
- Retrieve which consists of multiple volunteers from a database.

None of these actions, when performed on a private slot within a method, may alter the agenda. Private slots cannot propagate data because they must appear exclusively in the method associated with the private slot.

Figure 2-21 shows the basic mechanism underlying the action-based effects on the agenda using rules (methods could be substituted). An initial rule with the black hypothesis is triggered and verified.

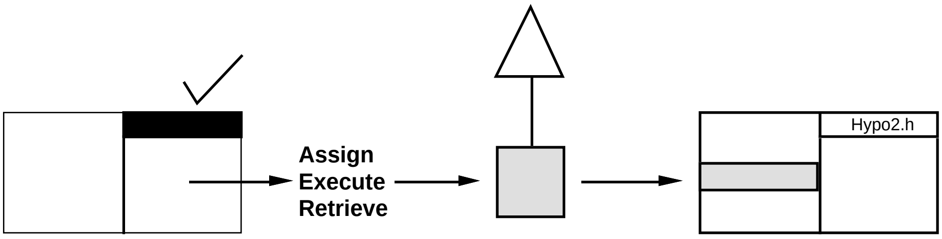


Figure 2-21 Forward-Action Effects between Two Rules

As Figure 2-21 shows at least one of the actions that appears on the right-hand side of a rule modifies the slot of an object which happens to be involved in the conditions list of another rule pointing to `Hypo2.h`. The latter hypothesis will be placed on the agenda. In the case of a method, the action must still involve the conditions of a *rule* (not another method), since the forward propagation mechanism only places hypotheses on the agenda for evaluation.

There is an important difference between forward action-effects and gates. Gates pre-evaluate the target condition, and only place the hypothesis on the agenda if the condition is TRUE. Forward action-effects, on the other hand, are non-selective. This means they place on the agenda any hypothesis whose conditions have a slot that has been affected by the actions list. Thus it may queue hypotheses even though the condition involving the modified slot might be FALSE. This is an important distinction.

The results of the actions list are evaluated regardless of whether these actions lead to concluding rules and hypotheses that are TRUE or FALSE.

There is symmetry between TRUE and FALSE: if a hypothesis is FALSE, the inference engine uses that information to further explore rules and methods.

Gates are a structure-based mechanism which expands the breadth of the search based on rules with similar conditions. Thus it is only appropriate to investigate hypotheses which have conditions which are similar to the ones currently being evaluated. Hence the distinction between TRUE and FALSE conditions with gates.

Like gates, hypotheses generated as goals due to forward action-effects will be evaluated only after all necessary backward chainings, suggested, and hypothesis forward events have been performed. Action-generated hypotheses are in competition with the gate-generated hypotheses by means of the priorities.

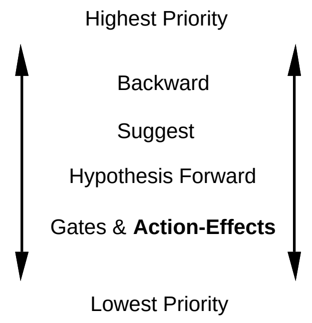


Figure 2-22 Agenda Priority

Gates and Action-Effects are Forward Propagation Mechanisms.

Volunteer

Volunteering from the development interface sets certain slots in the inference engine to particular values. Thus the effect it has on the inference engine agenda is very similar to what happens with actions. Any hypotheses which use this value in one or more of the conditions of its rules will be queued on the agenda. Furthermore, these hypotheses will be in competition with those generated by the gates and forward action-effects.

There are three differences between slots which are volunteered and slots which are set by actions:

- Similar to Suggesting hypotheses, Volunteering values is an explicit action by the user. The user is in essence saying that he knows a particular piece of information and the inference engine should use this information to evaluate all relevant hypotheses.

- There is no strategy setting to disable forwarding. Thus hypotheses which use the volunteered slot will be queued no matter what the current global strategy settings are. This topic will be fully explained in the Controlling Inference Strategies section.
- If the slot is used in pattern matchings in the conditions lists of some hypotheses, the associated hypotheses will be queued for evaluation. This is a different behavior than the other agenda mechanisms. Thus if we have a situation like:

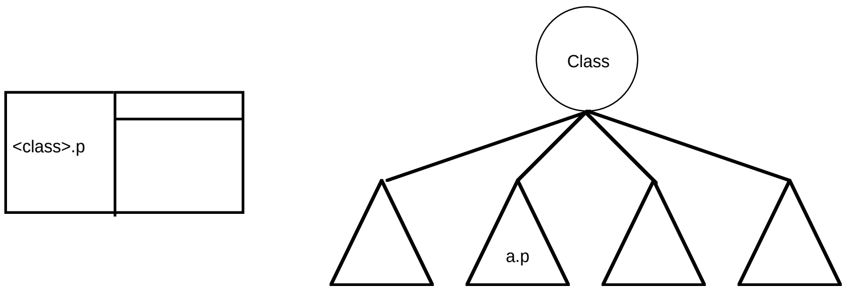


Figure 2-23 Volunteering to a Pattern Matching

and we volunteer the value of the slot `a.p`, the rule on the left which contains a pattern matching involving a parent class of `a.p` will be queued for evaluation. This topic will be fully dealt with in the Pattern Matching section.

Contexts

Contexts are “weak” forward links. They are the lowest level event on the agenda and are investigated only after all the hypotheses generated by the other inference search mechanisms have been completely evaluated:

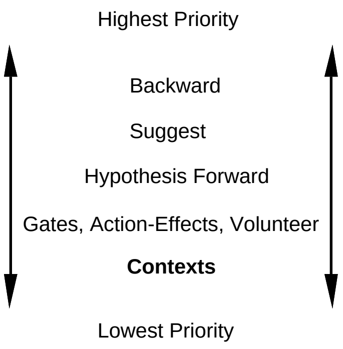


Figure 2-24 Agenda Priorities

Remember that a knowledge island is a group of rules and objects whose set of conditions and hypotheses intersect each others set of conditions, hypotheses, and actions. But now that we have investigated most of the agenda events, we can draw another definition of a knowledge island. Namely, a group of rules and objects which are linked at runtime by any of the following events: backward chaining, hypothesis forward, gates, or actions. Rules within one particular knowledge island are said to be strongly linked.

Since there is no way to propagate control from one knowledge island to another using backward chaining, hypothesis forward, gates, or actions, another mechanism was devised to propagate control between knowledge islands. This is called the context link. Context links are also called weak links.

Context links connect one knowledge island to another knowledge island. They are not defined at runtime as are the strong links, but in the knowledge base architecture. They link two or more *hypotheses* together. It is important to realize that in order for the context link to work properly, the two knowledge islands connected by the link must indeed be separate. If any data is shared between the rules of the two knowledge islands, forward chaining may take place before the context link.

Contexts are a unidirectional forward propagation mechanism. There is no backward chaining along context links.

Figure 2-25 shows two hypotheses linked by a context link.

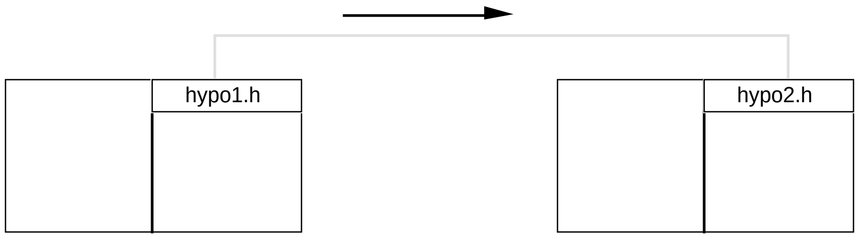


Figure 2-25 Context Link

The arrow indicates the flow of reasoning. After evaluating the rule pointing to `hypo1.h`, the inference engine “jumps” to `hypo2.h` and evaluates its rule. In the absence of any other relationship between these hypotheses’ rules (backward chaining, hypothesis forward, gates, or actions), this is the only way the context linked hypothesis could be placed on the agenda.

It is important to remember that all of the possible backward chaining, hypothesis forward, gates, and actions are performed before any context links. Thus the inference engine evaluates all of the pertinent hypotheses within a knowledge island before jumping to the next knowledge island. This behavior reflects the notion of focus of attention. The inference engine will focus on the current knowledge island and investigate everything pertinent before moving on rather than haphazardly jumping all over the place investigating one hypothesis from one island then another from a different island.

Figure 2-26 shows an example knowledge island structure where all the possible events have been laid out.

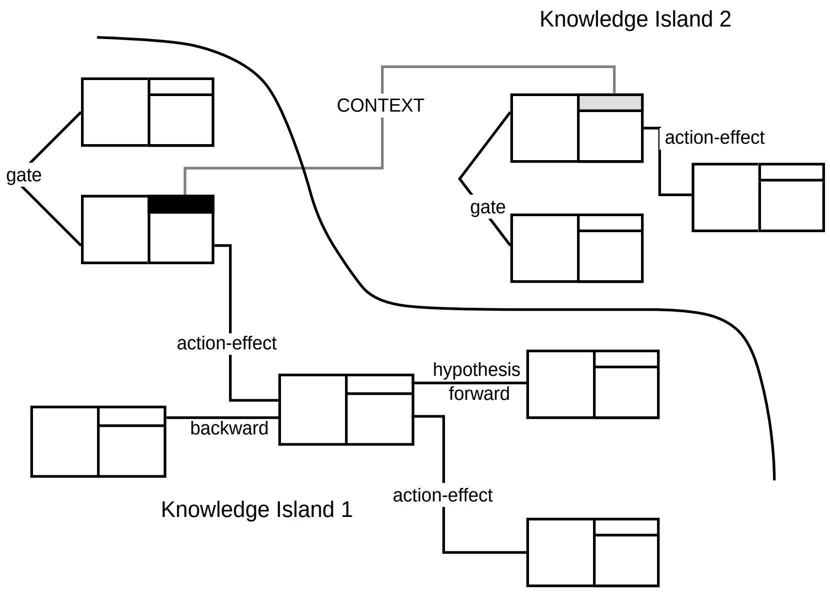


Figure 2-26 Knowledge Islands

The black hypothesis is supported by a single rule. Other rules are linked to this one by means of various events. However, this set is independent from the set of rules linked to the gray hypothesis. Both sets constitute knowledge islands. They can be linked by a *weak* link: the context. Before the context link takes place, all of the other lower level events must be explored.

A hypothesis can be in the context of any number of other hypotheses, and it can have any number of hypotheses in the context of itself.

Two hypotheses within the same knowledge island can also be put in context with each other. A hypothesis can even be put in context with itself. Since the context is the lowest priority event, the meaning of this construction is to investigate the target hypothesis when everything else which is pertinent in the knowledge island has already been investigated. One very important use of this is with the Reset operator to implement nonmonotonic behavior. See the Non-Monotonicity section for more information.

Interpretations

Interpretations are slot values which are interpreted to the name of an object, class, or property in a condition or action. Interpretations are only performed when the condition or action is evaluated.

The slot which is evaluated and whose value is the name of another object, class, or property is called the *interpreted slot*. The new slot, which is formed by the value of the interpreted slot, along with either an object or class name or a property name, is called the *resolved slot*. Consider the rule displayed in Figure 2-27 and assume the slot `a.p` has a value of “hypo”:

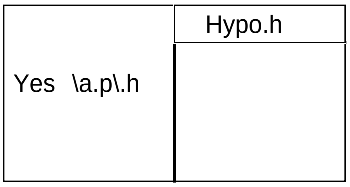


Figure 2-27 . Interpretation in a Condition

`a.p` is the interpreted slot while `hypo.h` is the resolved slot. There are two important concepts to remember when considering how interpretations affect the inference process:

- The interpretation is resolved first and then the condition or action works on the resolved slot. All forward and backward mechanisms work exactly as explained in the appropriate section on the resolved slot. There is never any forwarding from the interpreted slot.
- There will never be any forwarding to an interpretation. This is because the condition really bears on the resolved slot which is undetermined until the condition is evaluated, and not on the interpreted slot itself.

With Backward Chaining

Interpretations can be used in the conditions or actions of rules or methods to backward chain from any hypothesis. When the appropriate condition or action is evaluated, the interpretation is evaluated first and then the condition or action is processed exactly as it would be if an interpretation wasn't involved. Let's assume we have an interpretation in the conditions list of the rule shown in Figure 2-28,

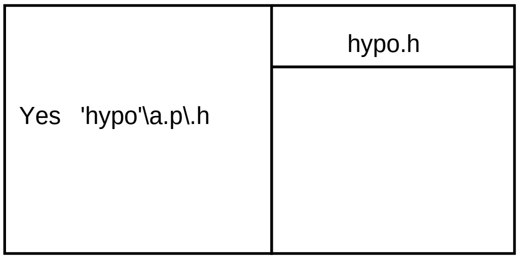


Figure 2-28 Interpretations with Backward Chaining

and we also have the following set of rules:

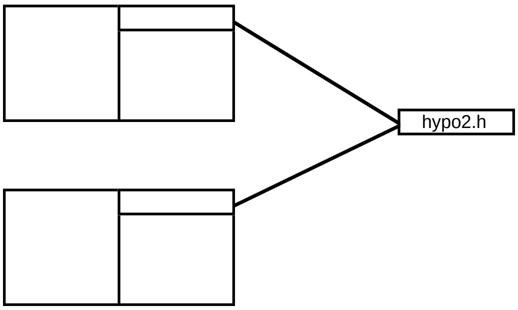


Figure 2-29 Interpretations with Backward Chaining

When evaluating the rule leading to the hypothesis `hypo.h`, the inference engine comes to the interpretation: `'hypo'\a.p\.h`. It evaluates the value of the slot `a.p`. Let's say it evaluates to the integer 2. The condition in the original rule becomes "`Yes hypo2.h`" after the interpretation is evaluated. To test this condition, the inference engine backward chains on the hypothesis `hypo2.h`. After evaluating `hypo2.h`, the inference engine finishes processing the rule leading to `hypo.h`.

Interpretations can also be used in the actions list of rules and methods. This follows the same syntax outlined above and is similar to the situation without interpretations. In other words, the interpretation is resolved first, and if the action then bears on a hypothesis, then the appropriate rules will be evaluated immediately.

The ability to have backward chaining based on interpretations allows a great deal of flexibility to the knowledge-based system. The inference engine can decide at runtime which hypotheses to backward chain on based upon the current situation, various external events, or where it is in the inference processing process.

With Suggest

Interpretations cannot be used with the Suggest command. The Suggest command requires an explicit hypothesis to suggest.

With Hypothesis Forward

Interpretations also cannot be used with the hypothesis forward inference search strategy. This is because:

- The source hypothesis cannot be an interpretation since the hypothesis of a rule is always an explicit slot.
- There will never be any forwarding to an interpretation. Thus an interpretation cannot be the target of the hypothesis forward inference search mechanism.

With Gates

Interpretations with gates work in the standard manner outlined above. Recall that a gate occurs when The inference engine evaluates a slot in a rule condition, that slot is also in another rule condition, and the target condition is TRUE given the evaluated slot’s value. There are two places where the interpretation could occur:

- The interpretation is in the source rule’s conditions list, and the target rule has a regular, compiled slot. Then, there will be a gate on the resolved slot but not the interpreted slot (assuming a TRUE evaluation of the target condition).
- The interpretation is in the target rule’s conditions while the source is either another interpretation or a regular, compiled slot. A gate will never occur in this situation.

In the following diagram, a gate will take place if the value of `a.p1` is “c” (and, of course, the rule condition involving `c.p2` is TRUE):

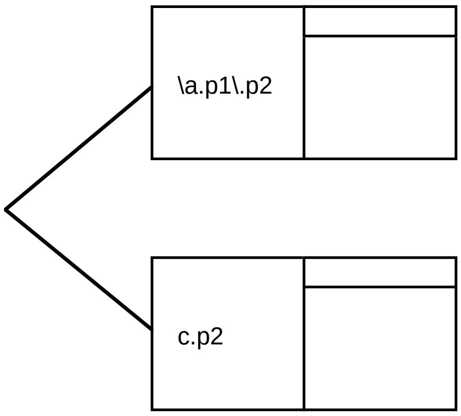


Figure 2-30 Gates with Interpretations

The gate occurs here, because the inference engine interprets the value of `a.p1`, and then it treats the expression as if it had the value of `c.p2` without considering the special fact that an interpretation was used in the condition. Basically an interpretation is evaluated, and then the expression is treated the same as any other expression.

With Action-Effects

The actions lists of rules and methods can contain interpretations. The same principles we have seen previously will apply once the interpretation has been resolved. Namely:

- If the source rule or method has an interpretation in its actions list, the interpretation is resolved, and then it is treated as any other action. It will forward to the resolved slot but it will not forward to the slot named in the interpretation:

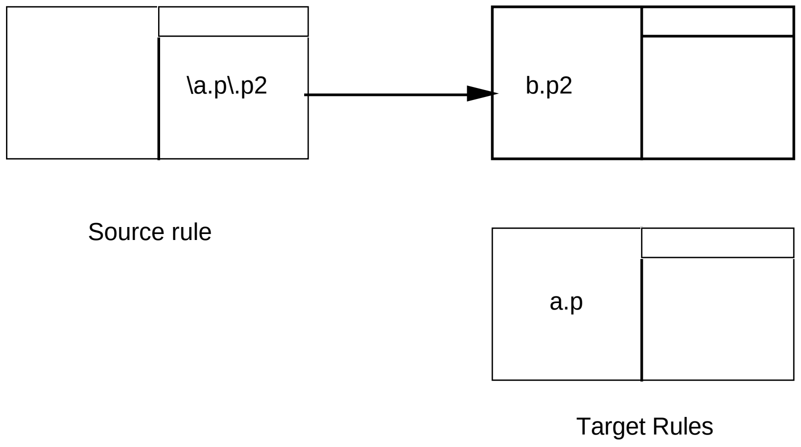


Figure 2-31 Forward Action-Effects on Interpreted Slots

As Figure 2-31 shows, if the value of the slot `a.p` is “b”, then the inference engine will put the hypothesis which has a condition on `b.p2` on the agenda. If the value of the slot `a.p` is not “b”, then the hypothesis which has a condition on `b.p2` will NOT be put on the agenda. The second target rule, which has a condition on the interpreted slot `a.p` will NEVER be put on the agenda no matter what the value of the interpretation is.

- If the source rule has an action bearing on any slot, then the inference engine will not forward chain to another rule which contains either the same slot in an interpretation, or an interpretation which evaluates to the same slot.

With Contexts

Context links are created by explicitly declaring a source hypothesis and any relevant target hypotheses. Since these must be explicitly listed in the context editor, interpretations are never used in conjunction with this forward inference search mechanism.

Pattern Matching

There are three general cases which involve pattern matching and the inference process:

- The source condition or action has a pattern matching but the target condition does not have a pattern matching. In this case, the inference engine treats the pattern matching as if the condition or action acted on each slot which is a member of the pattern matching. Any of the inference search mechanisms which would be triggered by explicitly listing one of the members of the pattern matching would be triggered in this case as well.

- The source condition or action has a pattern matching and the target condition has a pattern matching. In this case, the inference engine will forward only to conditions which have the exact same pattern matching.
- The source condition or action has a regular compiled slot while the target condition has a pattern matching. Forward chaining will only occur in this case if the source slot has been volunteered.

Because private slots are ignored by pattern match conditions they have no affect on inferencing, therefore, the following pattern matching discussions apply only to public slots.

With Backward Chaining

The hypothesis of a rule can be a boolean slot of any object or class. There are no restrictions on this object as far as the object hierarchy is concerned just because it has a boolean slot which is a hypothesis. Thus the object can have subobjects, any other slots, and any number of parent classes or objects.

The ability to have parent classes (the case with parent objects is almost completely analogous) of objects with hypothesis slots allows one to group similar hypotheses within the same class. These hypotheses will then have all the benefits of the class structure, including both inheritance and the ability to use pattern matching with them.

For example, there could be a rule like:

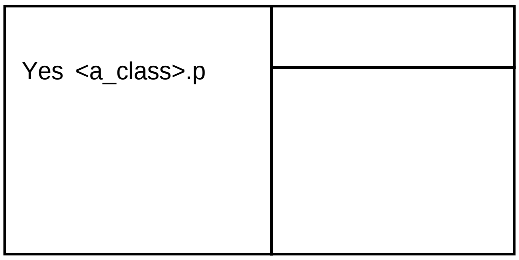


Figure 2-32 Pattern Matching and Backward Chaining

Furthermore, the class `a_class` contains two objects, `a` and `b`. The `p` slots of these two objects are, in fact, hypotheses of other rules:

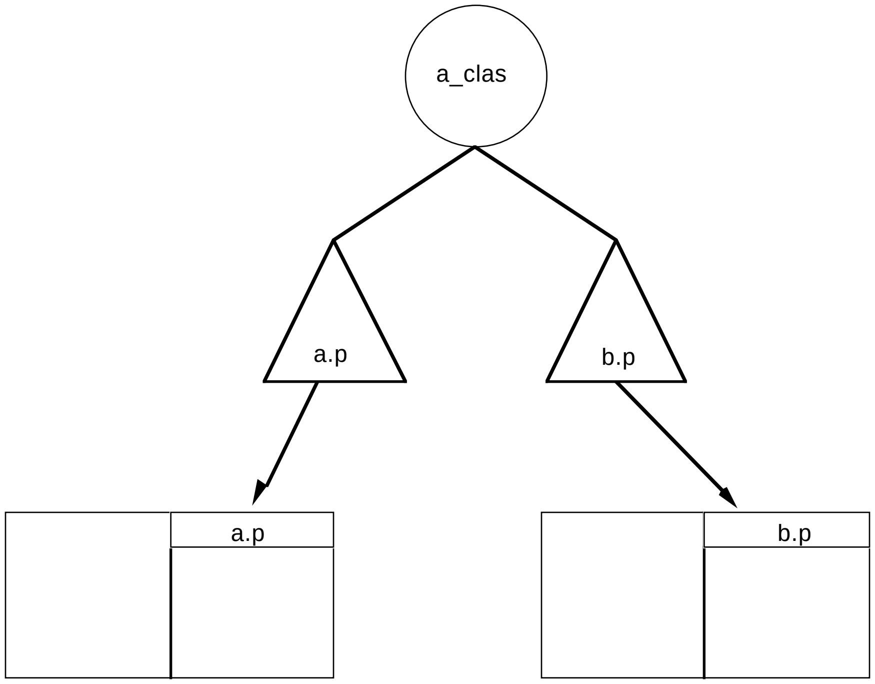


Figure 2-33 Pattern Matching and Backward Chaining

When the inference engine processes the first rule containing the pattern matching on the class `a_class`, it will look for the values of all of the object slots of that class. To find those values, the inference engine will backward chain on the appropriate hypotheses, inserting these rules on the agenda for evaluation before it finishes processing the original rule.

Of course there could be many rules leading to these hypotheses, and if the current strategy is exhaustive evaluation, many rules could be evaluated before the original rule is finished being evaluated.

Pattern matchings from any rule or method action can trigger multiple rule backward chaining as well. For example, the action "`Assign <a_class>.prop <a_class>.prop`" would trigger the exact same backward chaining illustrated above. It is important to remember that backward chaining is a way of determining the value of a boolean slot. Thus if the slot already has a value instead of being `UNKNOWN`, the inference engine will not backward chain on it. Similar to how pattern matchings work everywhere else, interpretations can be embedded inside pattern matchings. Thus we could have a rule like:

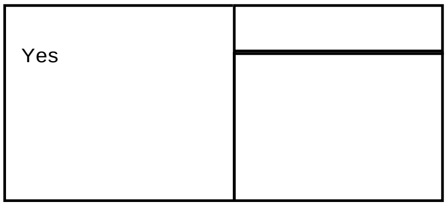


Figure 2-34 Interpretations with Pattern Matching

The inference engine will evaluate the interpretation `\a.p\` first. If the value of the slot `a.p` is a class (or object) whose child objects have a boolean slot `p` which is a hypothesis, then the rules leading to these hypotheses would be evaluated as described above for the case with regular conditions and actions. The objects in the parent class (or object) can be either dynamic objects or compiled objects, thus giving you a lot of flexibility over which rules are queued for evaluation when.

With Hypothesis Forward

Since the source of the hypothesis forward search mechanism is a hypothesis, and there can NEVER be a pattern matching as the hypothesis of a rule, this search mechanism will never have a pattern as its source.

The only time the inference engine forwards to a pattern matching is when the source pattern is exactly the same as the target pattern (volunteering is an exception). Since this inference search mechanism cannot have a pattern matching as its source, it will not forward to any pattern matchings.

With Gates

When the inference engine encounters a pattern in rule conditions (gates are not allowed from methods), it does two things each time it evaluates one particular slot in the pattern matching:

- It tries to gate to any other rule condition which has the slot which was just evaluated.
- It tries to gate to any other rule condition which contains the exact same pattern.

Both of these require a pre-evaluation of the rule's conditions list and the associated hypothesis will only be put on the agenda if the conditions are all evaluated TRUE. Figure 2-35 shows the first case, where we have a pattern matching in the source rule and explicit slots in the target rules; a gate will occur on each slot as soon as the slot is evaluated (assuming, once again, that the rule's condition is evaluated to TRUE):

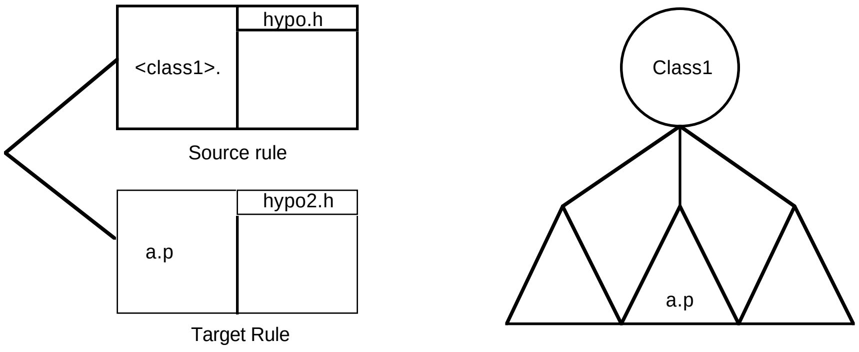


Figure 2-35 Gate on Pattern Matching Condition and Explicit Slots

When the inference engine evaluates the slot `a.p` in the pattern matching condition leading to the hypothesis `hypo.h` (see Figure 2-35), it immediately checks to see if a gate can occur, so that the condition involving `a.p` in the rule leading to `hypo2.h` is TRUE. If so, `hypo2.h` is put on the agenda.

As Figure 2-36 shows, the second situation occurs when there are pattern matching conditions in both the source rule and the target rule. In this case, if the pattern matching is on the exact same parent class or object, then a gate will occur:

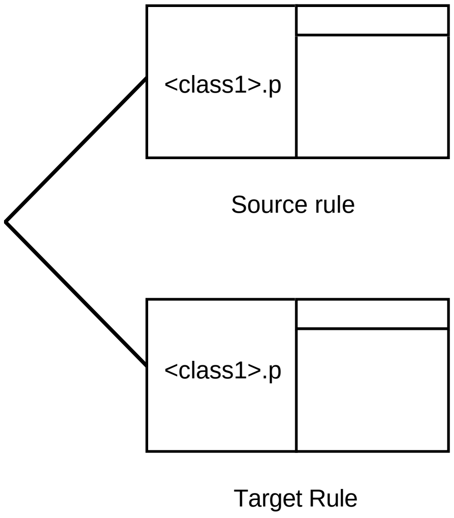


Figure 2-36 Gate Occurs on Same Patterns

However, the target condition must have the exact same pattern matching in its conditions list. As Figure 2-37 shows, a gate will not occur between two conditions with different patterns, even if the two patterns share some child objects:

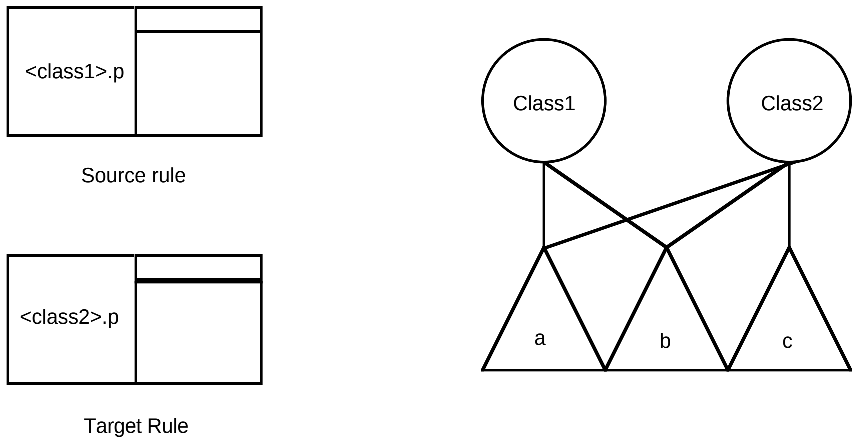


Figure 2-37 No Gate Occurs on Dissimilar Patterns

The target rule must have an explicit reference to one or more slots which are referenced in the source rule or have the exact same pattern matching for a gate to occur.

With Action-Effects

The principles to apply to forward action-effects are the same as those described for the gates. When the inference engine encounters a pattern in the actions list of rules or methods, it does two things each time it evaluates one particular slot in the pattern matching:

- It tries to forward to any other rule condition which has the slot which was just set.
- It tries to forward to any other rule condition which contains the exact same pattern.

Remember that actions lists of rules or methods will put hypotheses on the agenda without pre-evaluating the target rule conditions.

In the first case, the actions list of the source rule or method that contains a pattern matching is applied to a list of explicit slots defined in the left-hand side conditions of the rule. Such an action is equivalent to a set of individual actions bearing on each slot affected. Figure 2-38 shows four objects which are members of the class `a_class`:

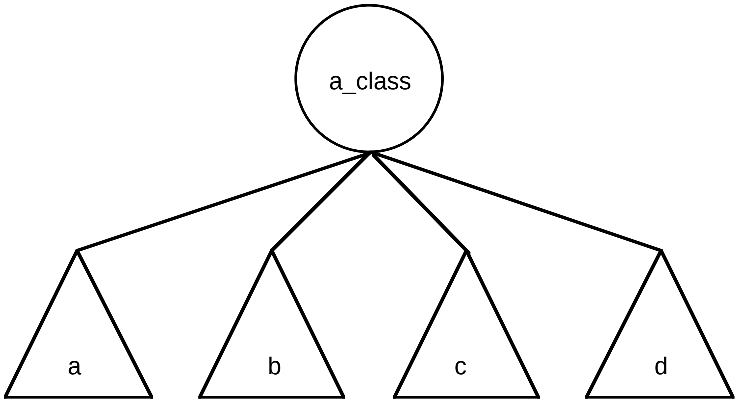


Figure 2-38 Class/Object Structure

Figure 2-39 shows a rule structure in which propagation will occur to all four hypotheses whose conditions involve the slots of `a_class`:

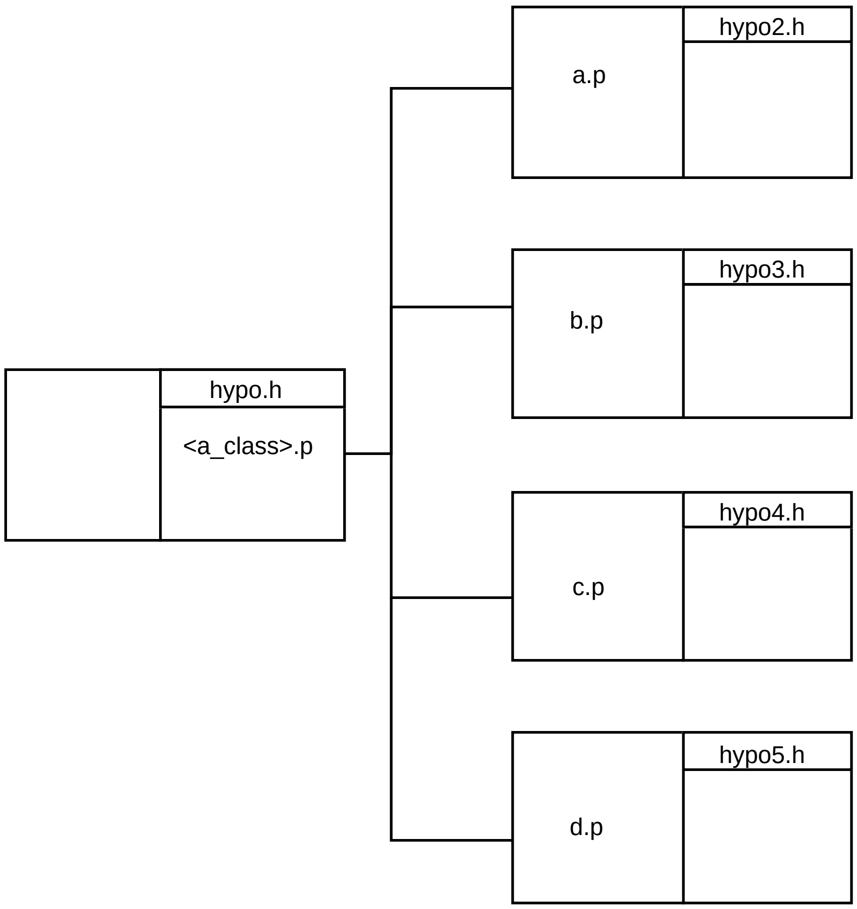


Figure 2-39 Propagating Forward Actions from Class Patterns

In Figure 2-39, `hypo2.h`, `hypo3.h`, `hypo4.h`, and `hypo5.h` will all be put on the agenda for evaluation as the system evaluates each object in the pattern <`a_class`>.

Figure 2-40 shows that the action of the source rule is propagated to a rule condition where a pattern matching exists on the same class. This is called *class-selectivity*.

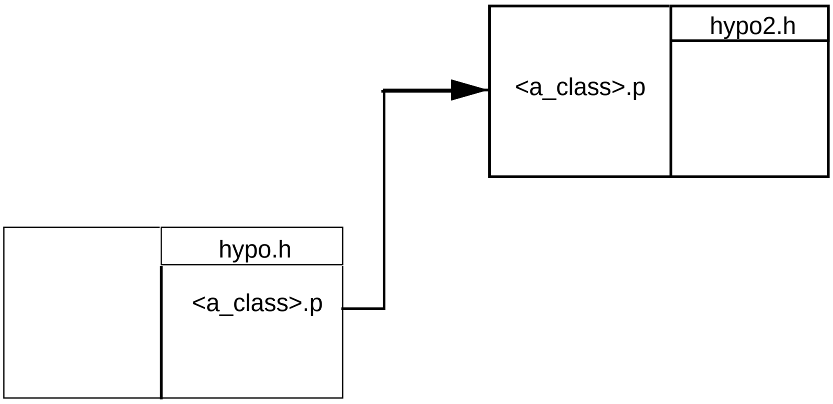


Figure 2-40 Forward Action-Effects and Class-Selectivity in Patterns

However, as Figure 2-41 shows, if the exact same class names are not used in the source action and target condition, then no propagation will take place even if their sets of objects intersect. Notice also that the properties must be the same for propagation to occur.

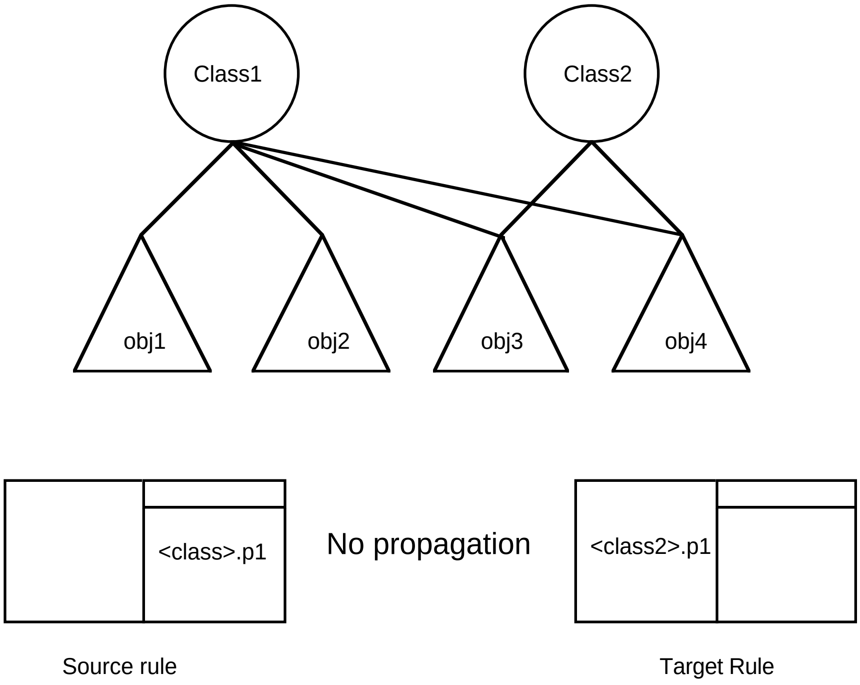


Figure 2-41 No Forward Action-Effects Between Dissimilar Class Patterns

With Volunteer

Volunteering is very similar to a forward actions-effect as it gives a slot a particular value. Similar to forward action effects, if you volunteer a slot, any hypotheses which contain this slot in their conditions list will be put on the agenda for future evaluation.

However, volunteering with pattern matching displays two behaviors which are quite different from those displayed with actions:

- You must volunteer an explicit slot to a specific value – you cannot volunteer a pattern matching (whereas forward action-effects can set a number of values using a pattern).
- If you volunteer a slot, then any condition which has a pattern matching containing that slot will be put on the agenda. This is the only case where an explicit slot can forward to a condition with a pattern matching.

Thus, if you have the rule and object structure pictured in Figure 2-42, and you volunteer the value of the slot `a.p`, then `hypo.h` will be put on the agenda:

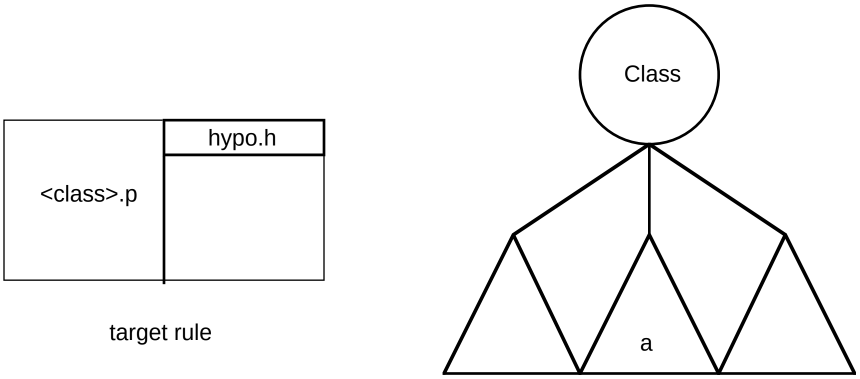


Figure 2-42 Forwarding to a Pattern Matching Using Volunteer

Similar to forward action effects, volunteering does not require a pre-evaluation of the target rule's conditions.

With Context Links

The case for pattern matching is much the same as for interpretations. Since both the source hypothesis and all of the target hypotheses must be explicitly declared, pattern matching cannot be used in conjunction with context links.

Conflict Resolution

There are usually many relevant hypotheses on the agenda at any one time. Whenever the inference engine finishes evaluating one hypothesis, it needs to determine which hypothesis to evaluate next. This process is called conflict resolution.

As previously described, there are five basic categories of hypotheses to be evaluated:

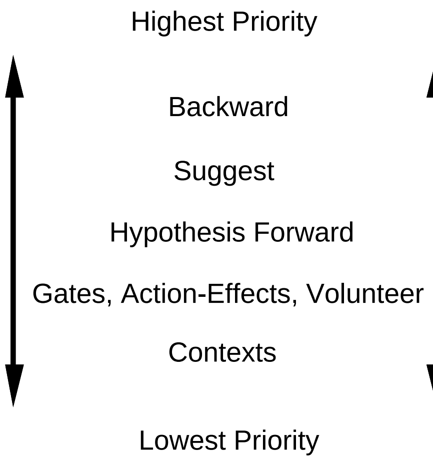


Figure 2-43 Agenda Priorities

Hypotheses within any higher priority category are evaluated before hypotheses within a lower priority category. Within any particular category, the hypotheses inference priority is the determining factor. Thus if three hypotheses have been put on the agenda due to a Volunteer, a gate, and an forward action-effect, the hypothesis with the highest inference priority will be evaluated first, then the hypothesis with the second highest priority, and finally the hypothesis with the lowest priority.

All hypotheses which are relevant for a backward chaining are evaluated *immediately*. If during the course of evaluation of one hypothesis another backward chaining becomes relevant, then the inference engine suspends processing of the original hypothesis to evaluate the new hypothesis. Once the new hypothesis has been evaluated, the inference engine continues processing the old hypothesis. Thus the inference engine exhibits a LIFO queue with respect to several concurrent backward chaining events.

When the inference engine finishes evaluating the current hypothesis, it looks for hypotheses from the list shown in Figure 2-44 and evaluates them in the order shown.

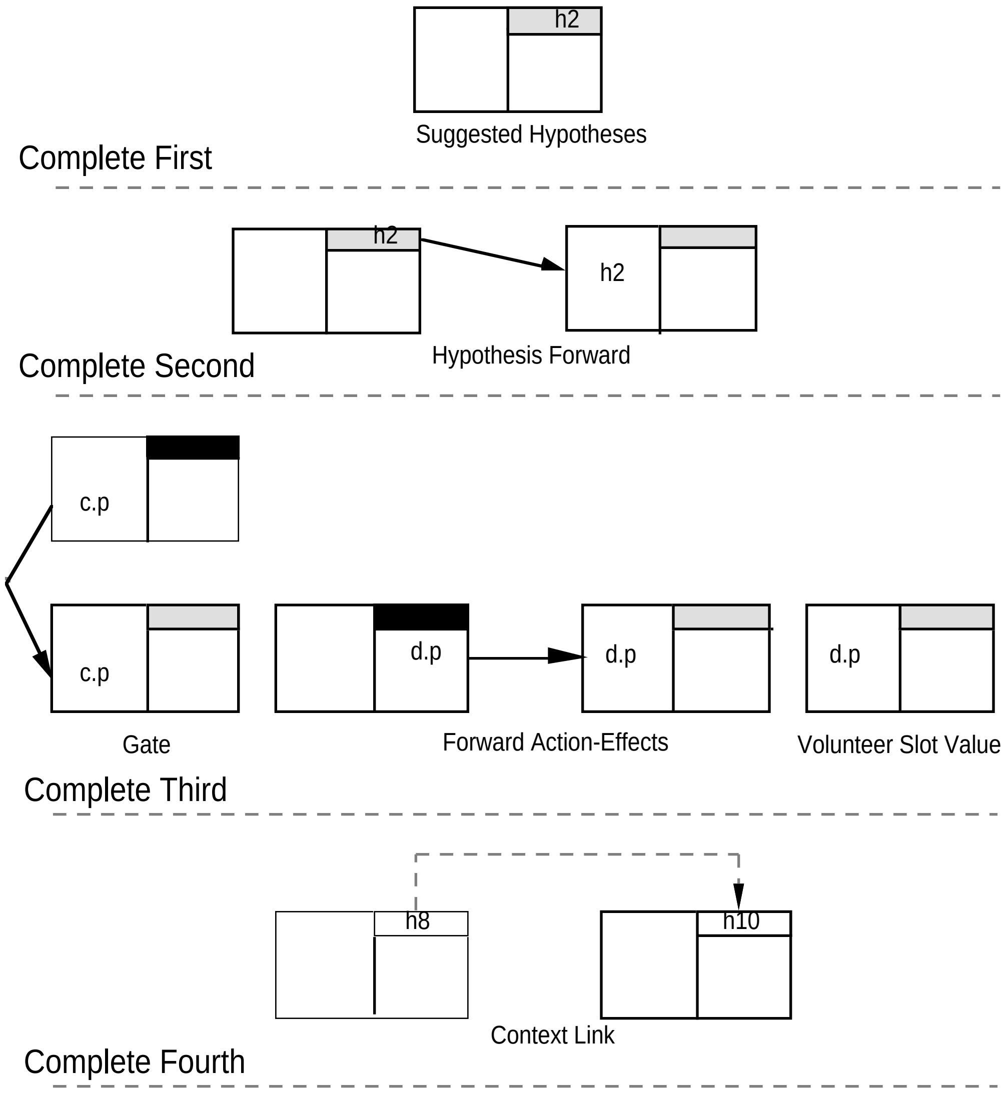


Figure 2-44 Conflict Resolution

For example, assume the inference engine is evaluating the following rule (the hypothesis inference priority is listed in parentheses after the hypothesis' name):

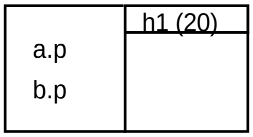


Figure 2-45 Sample Rule

When the inference engine begins evaluating this rule, there is nothing else waiting on the agenda. However, while evaluating the conditions list, the slot `a.prop` produces two gates and the slot `b.prop` produces one gate (the current hypothesis is black):

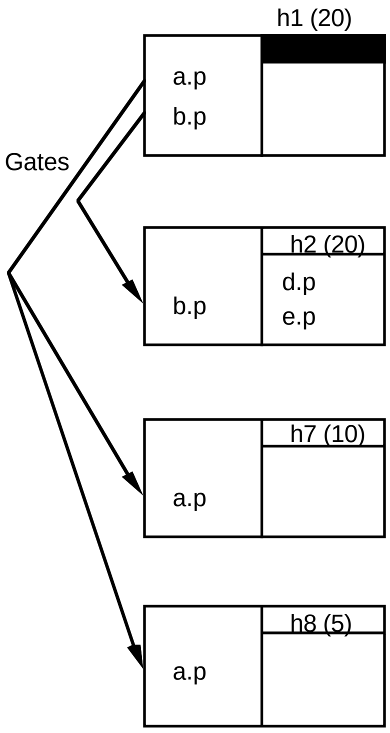


Figure 2-46 Gates Generated Hypotheses

The inference engine finishes evaluating the rule leading to `h1`. After finishing evaluating the hypothesis, the inference engine begins evaluating the hypothesis with the highest inference priority from the highest category. There are three relevant hypotheses, h2, h7, and h8. Since all of them are part of the Gates, Actions, Volunteer category, the inference priorities determine the order. Thus h2 is evaluated next (h1 is grayed out since it has been completely evaluated):

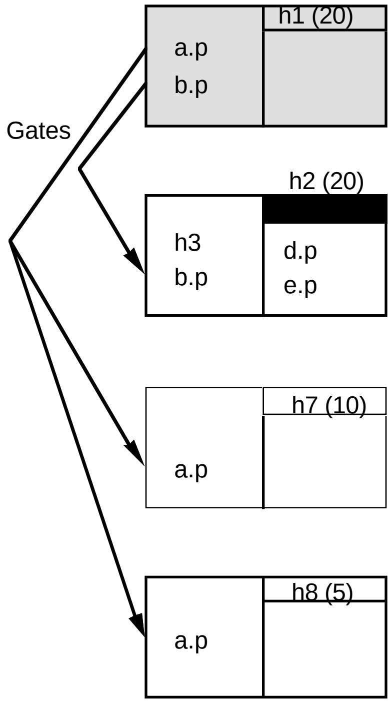


Figure 2-47 Gates Generated Hypotheses

However, while evaluating the rule leading to h2, the inference engine encounters a condition on h3. h3 is actually a hypothesis of another rule, so the inference engine will immediately evaluate this rule:

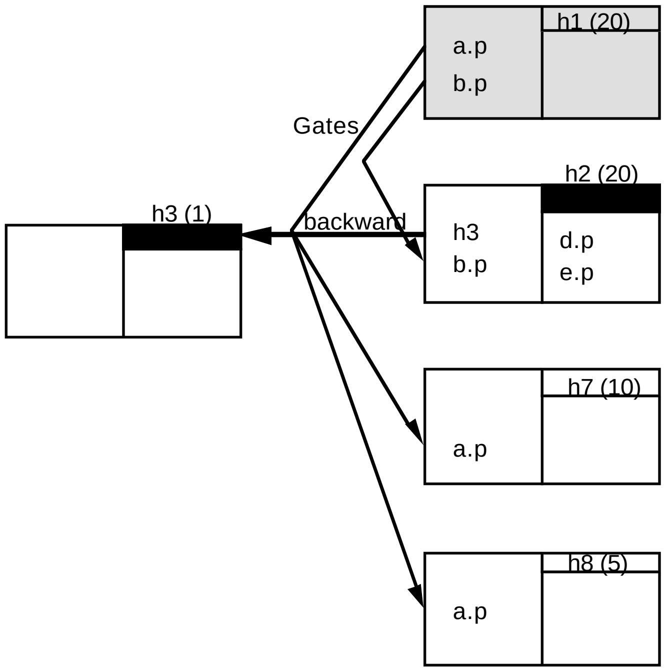


Figure 2-48 Backward Generated Hypotheses

While the inference engine evaluates h3, another gate makes the hypothesis h5 relevant. h5 is put on the agenda in competition with the other gate, action, and volunteer generated hypotheses (h7 and h8). the inference engine finishes evaluating h3 then it returns to h2 and finishes evaluating this rule:

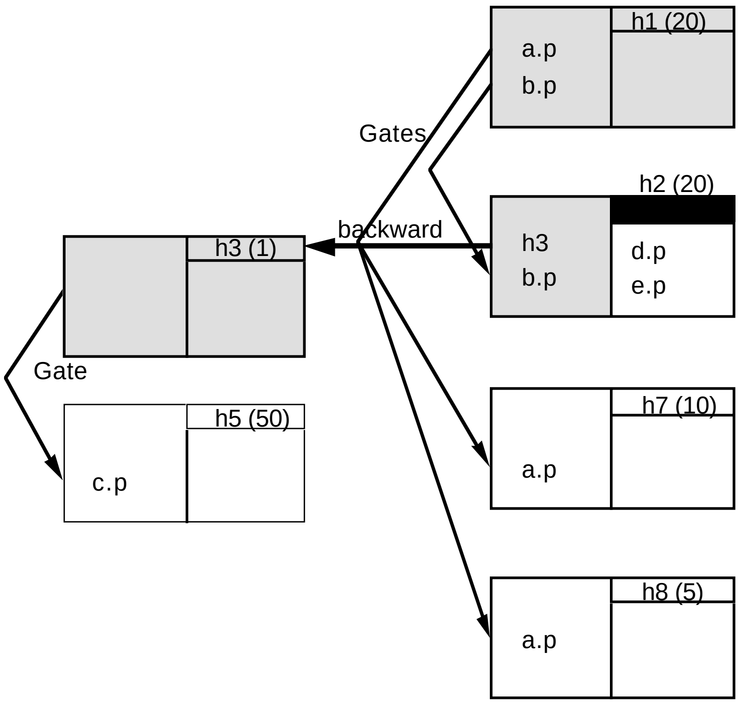


Figure 2-49 Another Gates Generated Hypothesis

After evaluating the conditions list of h2 in Figure 2-50, h4 becomes relevant because it has h2 in its conditions. The inference engine continues to evaluate the rule leading to h2. Forward action-effects cause two other hypotheses to become relevant due to the actions modifying data in the target rule's conditions. These two hypotheses, h6 and h9 are put on the agenda in competition with the other gates, action, and volunteer generated hypotheses:

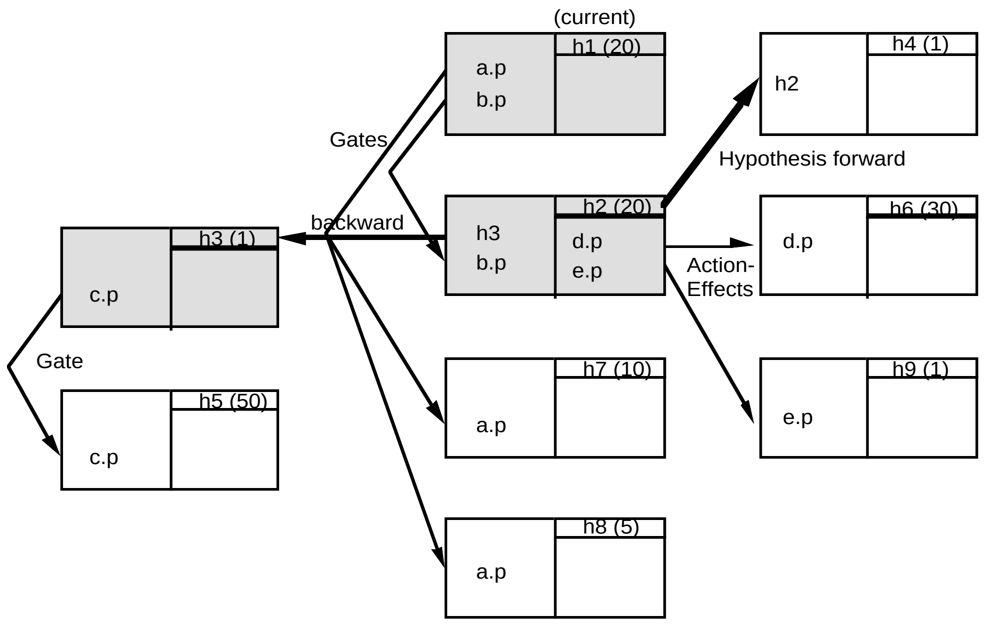


Figure 2-50 Action and Hypo Forward Generated Hypotheses

When h2 and h3 are finished, there is nothing left in the current evaluation queue, so the inference engine once again chooses the highest priority hypothesis from the highest category of event. Currently we have:

Suggest	Hypothesis Forward	Gate, Action, and Volunteer	Context
none	h4 (1)	h5 (50)	none
		h6 (30)	
		h7 (10)	
		h8 (5)	
		h9 (1)	

Table 2-2 Current Hypotheses on the Agenda

Thus the inference engine evaluates the rule leading to h4 since it is in the highest category. This hypothesis generates no further relevant hypotheses, so the inference engine now chooses the hypothesis with the highest inference priority from the Gate, Action, and Volunteer category:

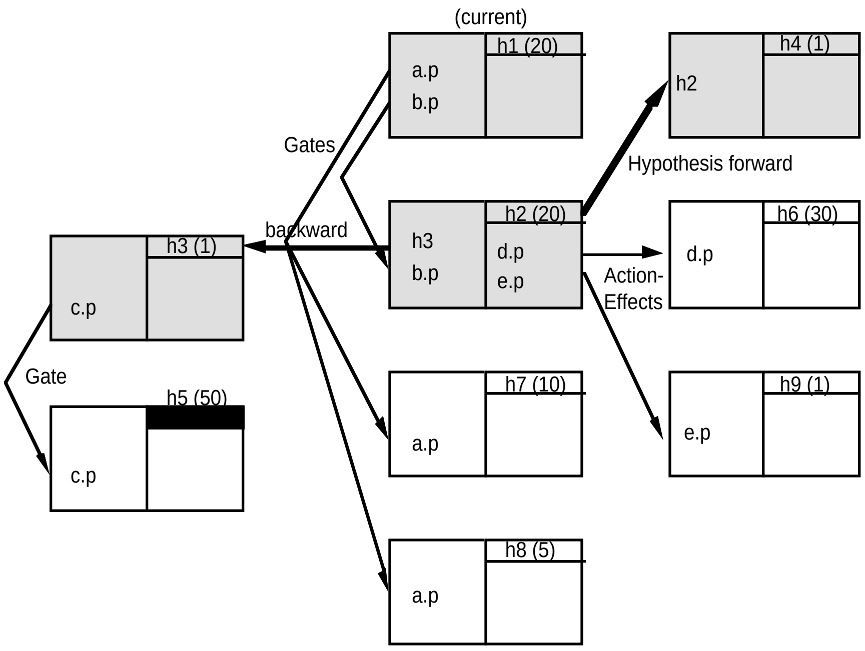


Figure 2-51 Refocusing on the Highest Priority Hypothesis

After evaluating h5, the inference engine focuses on h6, h7, and then h8 since h6 has the highest inference priority and h8 has the lowest inference priority. However, h8 generates a new type of event, namely a context:

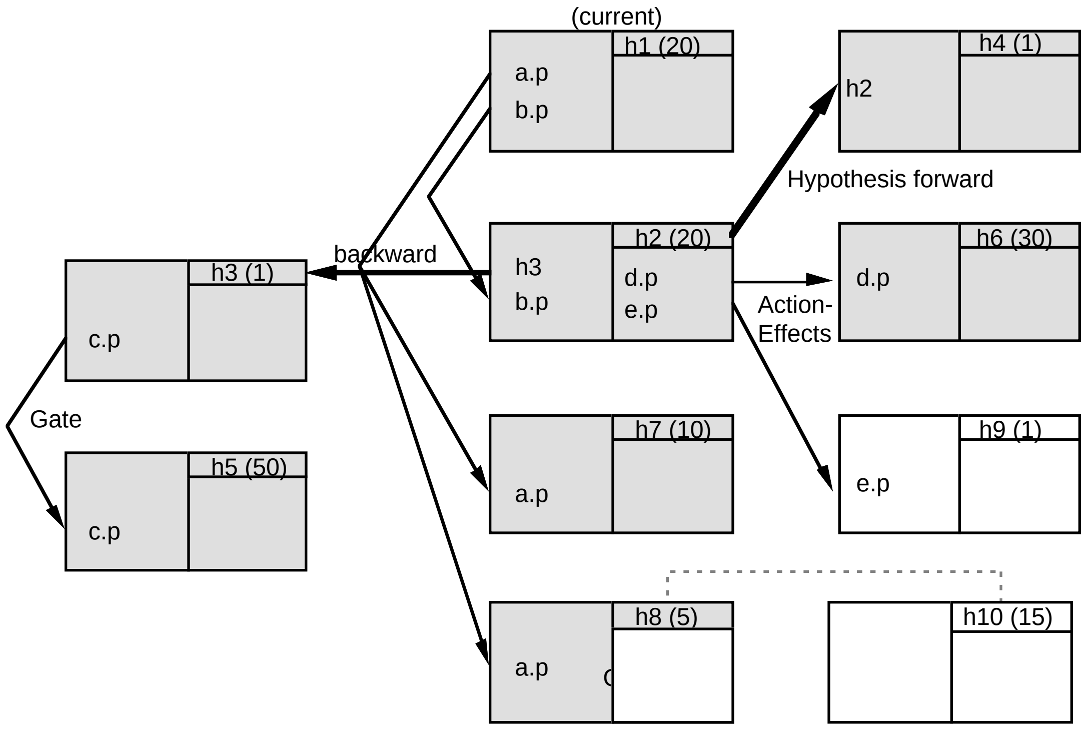


Figure 2-52 Context Generated Hypothesis

The inference engine finishes evaluating the rule leading to h8. Then it evaluates h9 since action generated hypotheses have priority over context generated hypotheses. Finally, the inference engine evaluates h10, after which the session is over since there are no more hypotheses on the agenda.

In summary, the inference engine evaluates any possible backward chainings immediately. When the inference engine has finished any relevant backward chainings, it focuses on the hypothesis with the highest inference priority from the highest category. While evaluating this hypothesis, many more hypotheses may become relevant. If so, then they are in competition according to how they were generated with any other hypotheses generated in a similar manner.

Summary

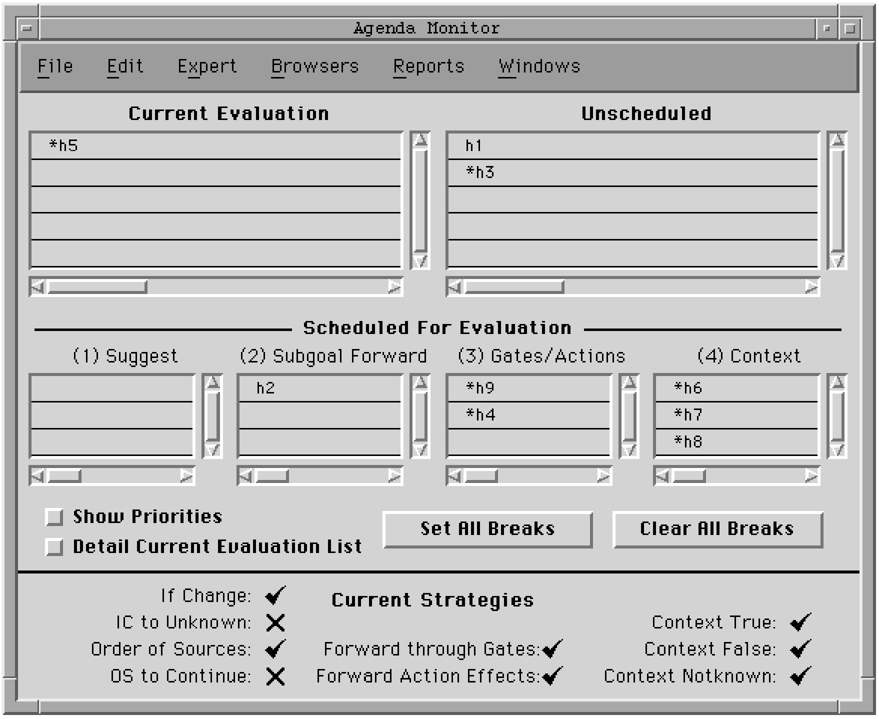
This concludes our discussion of the Rules Element agenda. We have seen that the agenda differs from classical FIFO or LIFO programming due to the insertion of events in queues with different priorities rather than just evaluating things based on when they became relevant. The non-exhaustive nature of the Rules Element inference engine gives it tremendous advantages over exhaustive paradigms which require absolute knowledge about everything and are thus very information intensive.

We have now described the basic events in the agenda. The agenda keeps several prioritized lists of hypotheses. As new hypotheses become relevant they are inserted in the appropriate queue according to why they are relevant, and they are put in the appropriate place in the queue according to their inference priority.

We see that the queues on the agenda are evaluated in the following order:

- Backward chaining is evaluated immediately
- Suggested hypotheses are queued with the highest priority
- Hypothesis forward have the next highest priority
- Gates and actions compete for priority after the above events
- Context links are the lowest priority event on the agenda.

This structure is also reflected in the Agenda Monitor window:



Current Evaluation

*h5

Unscheduled

h1
*h3

Scheduled For Evaluation

(1) Suggest	(2) Subgoal Forward	(3) Gates/Actions	(4) Context
	h2	*h9	*h6
		*h4	*h7
			*h8

☐ Show Priorities
☐ Detail Current Evaluation List

Current Strategies

If Change:	✓				
IC to Unknown:	✗			Context True:	✓
Order of Sources:	✓	Forward through Gates:	✓	Context False:	✓
OS to Continue:	✗	Forward Action Effects:	✓	Context Notknown:	✓

Figure 2-53 Agenda Monitor

The inference engine processes everything in the Current Evaluation first. The Current Evaluation contains a LIFO list: if the inference engine is evaluating one rule, and then it needs to evaluate a backward chaining, the backward chaining is inserted at the top of the Current Evaluation and it is evaluated. When the inference engine finishes evaluating this new event, it processes the rest of the Current Evaluation list.

When everything in the Current Evaluation has been evaluated, the inference engine finds the highest priority list which has at least one hypothesis in it. Then it finds the hypothesis with the highest inference priority within that list and that hypothesis becomes the new Current Evaluation which is evaluated as described above. The session ends when all of the relevant hypotheses have been evaluated.

Controlling Inference Strategies

The Rules Element allows you to control how the inference search mechanisms behave. Controlling an inference search mechanism refers to the ability to either disable it or somehow limit it in its scope. Globally disabling or limiting a particular inference strategy means that strategy is disabled or limited when Rules Element loads the knowledge base and remains that way until specifically changed (see below for details). These inference search strategy settings are saved with the knowledge base.

Many inference search mechanisms can also be locally disabled or limited. This means a particular inference search mechanism is changed from the conditions or actions of a rule or method. While the term “local” is used, the new strategy will be in effect until overridden by another strategy change or the session is restarted.

The global strategy settings normally determine the general types of search strategy used throughout the knowledge base, while the local strategy settings are often executed in opposing pairs. For example, one could have an action which disables a particular strategy, then an action which would normally affect the agenda but whose consequences aren’t taken into account due to the strategy change, and then a final action which re-enables the strategy. Thus that particular type of search strategy is disabled only for the one particular action or a small set of actions.

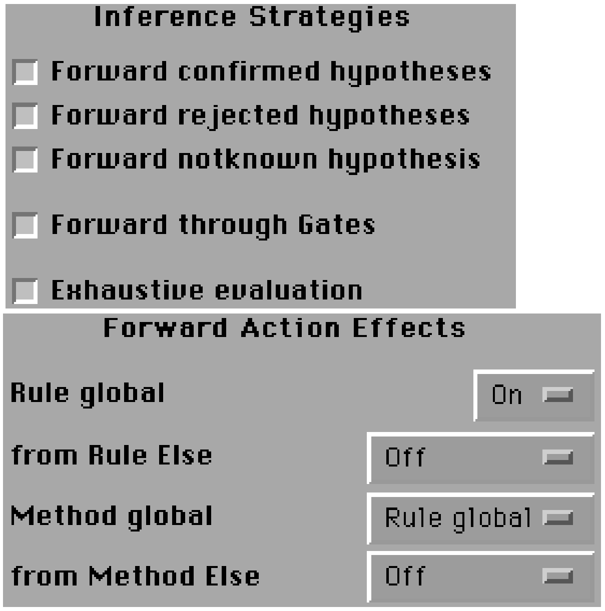
Note: :Inference strategies are associated with how hypotheses are put on the agenda and are not connected to any one hypothesis in particular.

Rules

All types of inferencing ultimately involve rule evaluation, and rule evaluation can be disabled by unloading the appropriate knowledge base. Thus if you have a knowledge base which has been disabled at a level of “DisableStrong” using the UnloadKB operator, and the inference engine needs to determine the value of a hypothesis in that KB, it will prompt for the value rather than backward chaining. For more information on loading and unloading knowledge bases, see the Multiple Knowledge Base section of this chapter or the UnloadKB and LoadKB operator descriptions in the Intelligent Rules Element Reference Manual.

Globally

Forwarding through gates and forwarding action effects can be globally disabled by using the Strategy Monitoring window that you select from the Expert menu:



Inference Strategies

- [ ] Forward confirmed hypotheses
- [ ] Forward rejected hypotheses
- [ ] Forward notknown hypothesis
- [ ] Forward through Gates
- [ ] Exhaustive evaluation

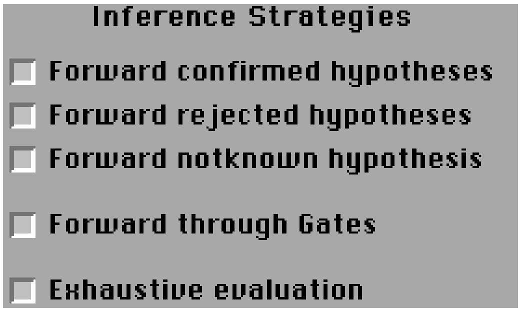
Forward Action Effects

Rule global	On
from Rule Else	Off
Method global	Rule global
from Method Else	Off

Figure 2-54 Globally Controlling Gates and Action Effects

Similar to the Exhaustive Evaluation strategy we saw earlier, forwarding through gates and forwarding action effects is enabled by default. If these checkboxes are unselected, then there will be no forwarding through gates or actions under any conditions, unless forwarding through gates or action effects are enabled using a local strategy change.

Context links can be made conditional upon the value of the source hypothesis. In such cases, the propagation depends upon either the verification or the rejection of the source hypothesis. Context links can be globally disabled from any state of source hypothesis using the Strategy Monitoring window:



Inference Strategies

- ☐ Forward confirmed hypotheses
- ☐ Forward rejected hypotheses
- ☐ Forward notknown hypothesis
- ☐ Forward through Gates
- ☐ Exhaustive evaluation

Figure 2-55 Globally Controlling Context Links

By default, as can be seen above, the inference engine will forward along TRUE, NOTKNOWN, and FALSE hypotheses. By unselecting any of these checkboxes, the corresponding strategy will be disabled throughout the knowledge base unless a local strategy change overrides this global default.

For example, if the Forward Rejected Hypothesis checkbox is unselected while everything else is left as the default settings, then any context links associated with hypotheses which are evaluated as TRUE or NOTKNOWN will be queued on the agenda, while contexts associated with hypotheses which are evaluated as FALSE will not be put on the agenda.

Locally

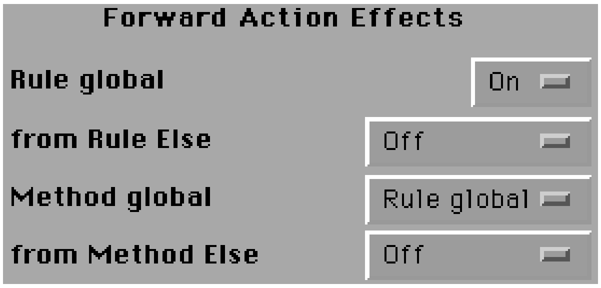
There are many different ways to control the inference search mechanisms locally. First of all, there are five categories of rule priorities. Rules in specific categories are protected from various inferencing mechanisms:

- Less than -20,000 will be disabled from all forward and backward processes
- -20,000 < priority < -10,000 will be disabled from forward processes but will function normally with backward processes
- -10,000 < priority < -5,000 will be disabled from the gate mechanism
- 5,000 < priority < -1,000 will be disabled from rule or method actions
- Priority > -1,000 is enabled for all forward and backward processes

Remember that this rule priority can be either fixed or a priority atom. This allows a rule to dynamically change how it can be used in the inference process.

Backward chaining is a mechanism for obtaining the value of a hypothesis. It consists of the evaluation of one or more rules. Thus, it is a default mechanism to obtain the value of such slots. However, the inference engine considers this as one option among others, and if the application requires it, it is possible to override backward chaining by first proposing other sources of information with the Order of Sources method. For more details, refer to the Order of Sources method section.

Forwarding Action Effects and Forwarding Through Gates can be disabled locally by using the strategy operator from the conditions or actions of a rule or method:



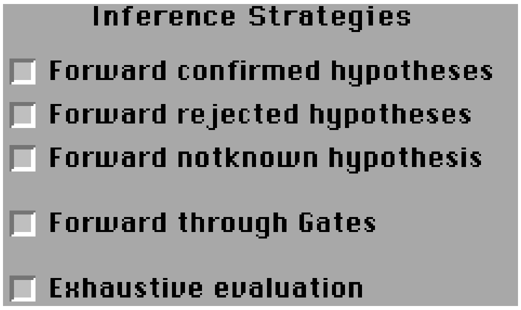
Forward Action Effects

Rule global	On
from Rule Else	Off
Method global	Rule global
from Method Else	Off

Figure 2-56 Locally Controlling Forward Action Effects

If Forward Action Effects or Forwarding Through Gates is turned off with this strategy operator, then they will remain disabled until another strategy change turns them on or the session is restarted.

Forwarding through context links from any type of source hypothesis can be disabled locally by using the strategy operator from the conditions or actions of a rule or method:



Inference Strategies

- ☐ Forward confirmed hypotheses
- ☐ Forward rejected hypotheses
- ☐ Forward notknown hypothesis
- ☐ Forward through Gates
- ☐ Exhaustive evaluation

Figure 2-57 Locally Controlling Context Links

If forwarding through any type of context link is turned off with this strategy operator, then it will remain disabled until another strategy change turns it on or the session is restarted.

Similar to backward chaining and unlike the other agenda mechanisms, context links only involve hypotheses, not actions or conditions. Due to this fact, there is no way a context link can be disabled for particular rules or hypotheses. Of course, if the rule priority is less than -20,000, it will be completely disabled. In this case, the inference engine would still use the context link to propagate to the new hypothesis, but then, since the rule is disabled, it would prompt for the value of the hypothesis (assuming there are no other rules or sources).

Methods

As we have already discussed in the section on methods, the user-defined method can trigger any actions the application developer needs, the Order of Sources method is used to establish possible sources for finding the value of a slot, and the If Change method is used to trigger actions as soon as any change occurs in the value of a slot. Each category of method can trigger other actions since the menu of operators available entails all of the rule action and test operators. From the point of view of the agenda, the significant action operators are the:

- Assign operator
- Execute operator
- Retrieve operator.

None of these actions, when performed on a private slot within a method, may alter the agenda. Private slots cannot propagate data because they must appear exclusively in the method associated with the private slot, therefore, the following discussion applies only to public slots.

If a method is triggered during the inference process, it will be processed *immediately*. Thus it is immediately inserted onto the top of the Current Evaluation stack and everything else is suspended until it has been evaluated.

Should these methods have any consequence on the agenda via the Assign, Execute, or Retrieve operators, they will take place following the same principles as for rule forward action-effects. Thus if any conditions' data are modified, the associated hypotheses will compete with the gates and forward action-effects propagation strategies.

In addition, if any backward chainings become relevant due to a method using a construction such as "`Assign hypo.h hypo.h`", it will be evaluated in the same manner as if it occurred in a rule, namely it will be put on the top of the Current Evaluation list and executed immediately.

Controlling methods refers to the ability to either disable them or somehow limit them in their scope. Globally disabling or limiting a particular inference strategy means that strategy is disabled or limited when Rules Element loads the knowledge base and remains that way until specifically changed (see below for details). These inference search strategy settings are saved with the knowledge base.

All global inference strategies can also be locally disabled or limited. This means a particular inference search mechanism is changed from the conditions and actions of a rule or method. While the term “local” is used, the new strategy will be in effect until overridden by another strategy change or the session is restarted.

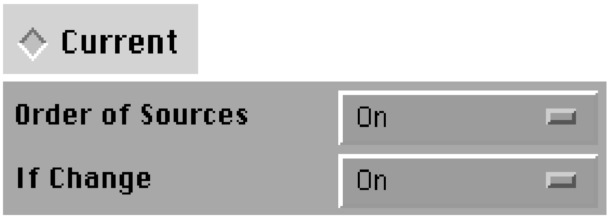
There are three basic ways to control methods. The first is by disabling them entirely. In the case of the Order of Sources system method, this means the default sources are executed instead of whatever actions are declared in the system methods. The second way is to execute the methods in their entirety, but to disable or limit the effect of their actions. Thus values could be changed, but they wouldn’t cause new hypotheses to be put on the agenda. The third way is to limit their scope to the object to which the method is directly attached by making individual methods private, thereby preventing downward inheritance.

In addition to affecting the agenda by means of the Assign, Execute, and Retrieve operators, methods can also change the strategy settings through the Strategy operator. While this particular operator doesn’t affect the agenda, it will affect how future hypotheses are put on the agenda. For example, an If Change method can disable gates. This won’t affect any gates-generated hypotheses currently on the agenda, but it will affect future gates, so that no more gates-generated hypotheses will be put on the agenda until the strategy has been re-enabled.

Note: Inference strategies are associated with how hypotheses are put on the agenda and are not connected to any one hypothesis in particular.

Globally

Both the Order of Sources and the If Change methods can be globally disabled from the Strategy Monitoring window that you select from the Expert menu. By default, both methods are enabled:



Order of Sources: On
If Change: On

Figure 2-58 Globally Controlling System Methods

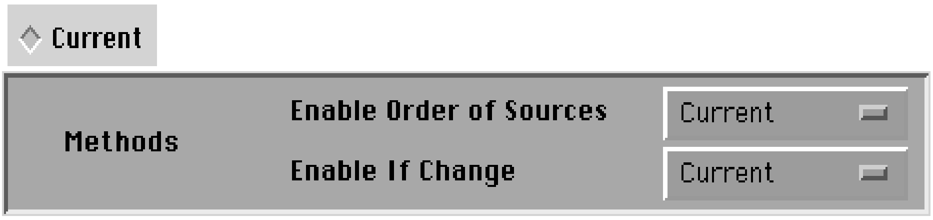
Note: You can trigger individual methods using the SendMessage operator from a rule or method even though system methods have been disabled in the Strategy Monitor window.

Unselecting either of these checkboxes will disable the associated method throughout the knowledge base unless it is overridden by a local strategy change. This means that the default methods will execute instead of whatever is declared in the object or classes methods.

If one wishes to execute the methods, but to disable the effects of their actions, one would set Method Global to Off for the Forward Action Effects strategy (see Figure 2-56). Note that this disables the effects of all actions, whether the actions come from the actions lists of a rule or a method.

Locally

Once again, only the Order of Sources and If Change methods can be locally disabled using the Strategy operator from the conditions and actions of rules or methods. In this sense a local strategy takes effect only at runtime, whereas global changes are done statically during development. Figure 2-59 shows the dialog window that lets you specify local strategy changes in rules and methods.


◇ Current

Methods

Enable Order of Sources: Current

Enable If Change: Current

Figure 2-59 Locally Controlling System Methods during Runtime

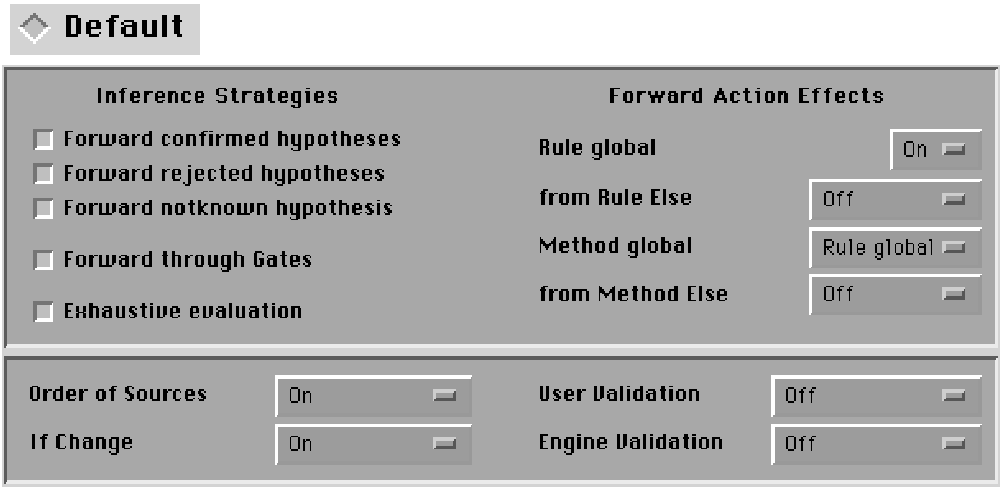
Note: You can trigger individual methods using the SendMessage operator from a rule or method even though the Strategy operator has been used to disable system methods.

Disabling or enabling the system methods from the Strategy operator will change whether or not the associated method is used until a new local change occurs, the session is restarted (in which case the default is used again), or a new knowledge base is loaded.

Similar to the above situation, globally changing the action-effects, one can disable the method's action-effects by setting Method Global to Off (see Figure 2-56) . Once again, this disables all actions from putting hypotheses on the agenda, whether the actions are in a rule or method. With Forward Action Effects unselected, the methods will still be executed, but no hypothesis will be brought on the agenda due to one of its rule conditions' data being changed.

It is also possible to "wrap actions" using two Strategy operators. Let's say you wanted to change the value of a particular datum, but you didn't want the change to put any additional hypotheses on the agenda. Then one could disable the Forward Action Effects, then modify the datum or several data, and finally one could re-enable the Forward Action Effects. Thus the strategy is the same as it was before the series of actions, the data were changed, and no new hypotheses were put on the agenda.

For example, assume the default strategies are in effect:


◇ Default

Inference Strategies

- ☐ Forward confirmed hypotheses
- ☐ Forward rejected hypotheses
- ☐ Forward notknown hypothesis
- ☐ Forward through Gates
- ☐ Exhaustive evaluation

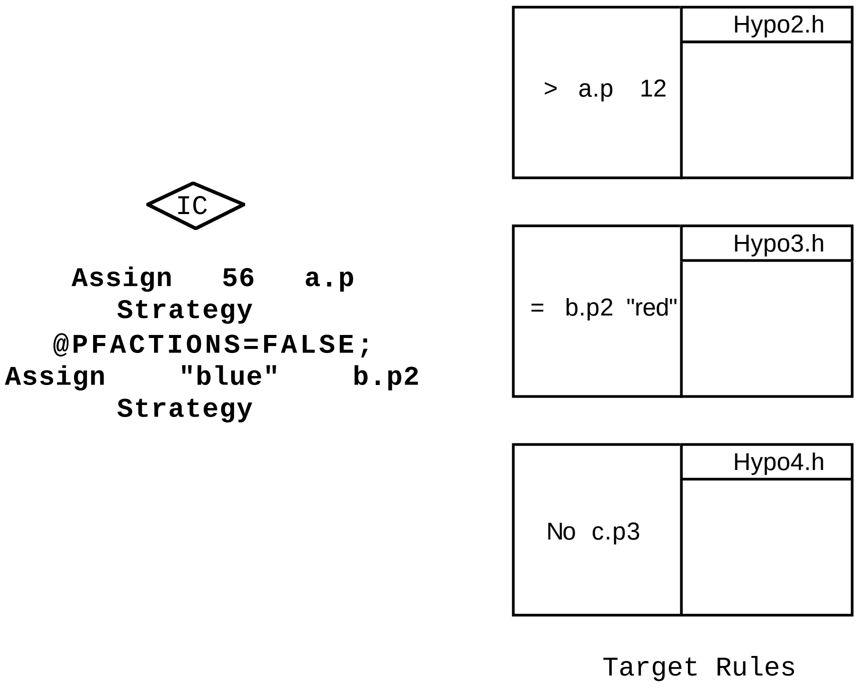
Forward Action Effects

Rule global	On
from Rule Else	Off
Method global	Rule global
from Method Else	Off

Order of Sources	On	User Validation	Off
If Change	On	Engine Validation	Off

Figure 2-60 Rules Element Default Strategies

In addition, assume that the following If Change method in Figure 2-61 has been triggered and the Target Rules are currently loaded and enabled:



```
Assign   56   a.p
   Strategy
@PFACTIONS=FALSE;
Assign    "blue"    b.p2
   Strategy
```

Figure 2-61 If Change Method Propagating to Target Rules

The following events occur:

- The first If Change action puts the value 56 in the slot `a.p`. Since action effects are enabled, the inference engine queues the hypothesis `hypo2.h` on the agenda for future evaluation.
- The inference engine then evaluates the second action, which disables Forward Action Effects. This does not affect any hypotheses on the agenda. It merely alters which hypotheses will be put on the agenda in the future.
- The third action modifies the value of the slot `b.p2` to blue. However, since Forward Action Effects are not enabled, the hypothesis `hypo3.h` is not put on the agenda for evaluation.
- The fourth action re-enables Forward Action Effects.
- The fifth and final action puts the value TRUE into the slot `c.p3`. Since Forward Action Effects are enabled, the hypothesis `hypo4.h` is put on the agenda. Notice that even though the condition in the rule leading to

`hypo4.h` would be evaluated as `FALSE` due to the value of `c.p3`, the hypothesis is still queued for evaluation since `c.p3` was involved in an action and not a gate.

Thus, after evaluating this If Change method, three slot values have changed (`a.p`, `b.p2`, and `c.p3`), two hypotheses have been put on the agenda (`hypo2.h` and `hypo4.h`) to compete with other gate- and action-generated hypotheses, and the strategy settings are exactly the same as before.

Note: The strategies for Forward Action Effects are the only strategies that can be wrapped around an action.

The current strategy of the six inheritability settings can be changed with the Strategy operator just like the other settings of the Strategy operator. However, inheritance settings that have been specifically selected for the meta-slot of individual slots always override those of the current strategy.

Application Programming Interface

The Rules Element has an Application Programming Interface (API) through which it is possible to access all of the information in the system while it runs. Calls in general will allow for the investigation of the working memory, the setting of handlers and controls, the control of the reasoning process, and the editing of knowledge. We focus here on those calls affecting the inference mechanism.

The three calls from the C Library which can directly affect the agenda are:

- `NXP_Suggest`
- `NXP_Volunteer` and
- NXP_Control (NXP_CTRL_RESTART)

In addition, one other call affects how hypotheses are put on the agenda in the future:

- NXP_Strategy

We will describe all of these calls below.

Suggest

Suggesting a hypothesis tells the inference engine to evaluate it. Of course, we've seen that the inference engine has several lists of hypotheses to evaluate, so just telling the inference engine to evaluate a hypothesis is not enough. One must also state the priority to give it. These priorities correspond with the lists described previously. So, the suggest call determines which hypothesis to suggest and with what priority:

```
NXP_Suggest (theHypothesis, priority)
```

The possible priorities are:

- `NXP_SPRIO_UNSUG`: unsuggests (removes) `theHypothesis` from the Agenda. Thus, through the Application Programming Interface, it is possible to remove as well as add hypotheses to the existing agenda (unless the hypothesis is currently under evaluation)
- `NXP_SPRIO_SUG`: places `theHypothesis` in competition with hypotheses suggested from the development interface. These have the highest priority on the agenda
- `NXP_SPRIO_HYPISL`: places `theHypothesis` on the agenda in competition with hypotheses generated from the hypothesis forward search mechanism
- `NXP_SPRIO_DATAISL`: places `theHypothesis` on the agenda in competition with hypotheses generated by gates and actions
- `NXP_SPRIO_CNTX`: places `theHypothesis` on the agenda in

competition with hypotheses generated by context links

Thus it is possible to completely control which hypotheses are queued and with what priority from an external program. There is no difference between suggesting the hypothesis from an external program and generating it by an internal inference search mechanism.

Volunteer

Volunteering a slot sets a slot to a particular value. As we have seen elsewhere, when data change values, hypotheses which use those data values can be put on the agenda. Thus volunteering a value not only changes the value, but it also puts hypotheses on the agenda. The syntax for the Volunteer call is:

```
NXP_Volunteer (theSlot, desc, thePtr, priority)
```

The priority argument in the Volunteer call tells the inference engine how to forward the new slot value. For example, you may want to change the value but have no forwarding, or possibly treat it as if it was volunteered from the development interface. The possible priorities are:

- `NXP_VSTRAT_VOLFWRD`: forwards the value in the strongest fashion. This priority is the same as volunteering from the user interface, which means that any hypothesis which uses the associated slot in its conditions list will be queued for evaluation in competition with gates and forward action-effects. Note that, as described earlier for volunteering from the development interface, hypotheses whose conditions have pattern matchings bearing on the volunteered slot will also be queued for evaluation.
- `NXP_VSTRAT_RHSFWRD`: forwards the value as if it were an forward action-effect, regardless of the current forward strategy.
- `NXP_VSTRAT_CURFWRD`: forwards the value as if it were an forward action-effect. If forward action-effect are turned off, this will have no influence on the inference process.
- `NXP_VSTRAT_QFWRD`: forwards the value as if it were asked in the session control panel of the main window.
- `NXP_VSTRAT_NOFWRD`: pastes the value in the slot but will not influence the inference process, so that the new value will not be forwarded during inferencing.
- `NXP_VSTRAT_RESET`: used for resetting a hypothesis and all of its associated conditions and subgoals (by setting each of these to `UNKNOWN`).

Once again, we see that values can be modified from an external program, and you can completely control how the forwarding works on these modified values.

Restart Session

The Restart Session command issued from the API is the same as if it was issued from the development interface. Restarting a session sets all of the slots to the value `UNKNOWN`. All hypotheses which are on the agenda will be taken off. All dynamic links and dynamic objects will be cleared from memory. All of the conclusions reached during the session will also be purged.

However, any knowledge bases which were loaded during the session will remain loaded, and any effects that the session had on the outside world (writing to databases, sounding alarms, and so on) will remain in effect.

The syntax of the Restart Session command is:

```
NXP_Control (NXP_CTRL_RESTART)
```

Strategy

In addition to Suggest, Volunteer, and Restart which directly affect the agenda, one can also change the current strategy using the `NXP_Strategy` call. Changing the strategy does not affect what's currently on the agenda. Nothing new will be put on the agenda due to a strategy change, nothing will be taken off, and none of the priorities will change. However, changing the strategy will have tremendous influence on what is put on the agenda in the future.

All of the strategies described previously can be modified from any external program. The syntax is:

```
NXP_Strategy( code, bool )
```

where code is one of the following:

- `NXP_AINFO_PWTRUE`: this setting enables / disables context propagation on `TRUE` hypotheses
- `NXP_AINFO_PWNOTKNOWN`: this setting enables / disables context propagation on `NOTKNOWN` hypotheses.
- `NXP_AINFO_PWFALSE`: this setting enables / disables context propagation on `FALSE` hypotheses.
- `NXP_AINFO_EXHBWRD`: this setting enables / disables exhaustive backward chaining.
- `NXP_AINFO_PFACTIONS`: this setting enables / disables forwarding on righthand-side "then" actions.

- NXP_AINFO_PFELSEACTIONS: this setting enables / disables forwarding on righthand-side "else" (false) actions.
- NXP_AINFO_PFMETHODELSEACTIONS: this setting enables / disables forwarding on righthand-side "else" (false) actions for methods.
- NXP_AINFO_PTGATES: this setting enables / disables forwarding through gates.
- NXP_AINFO_INHOBJUP: this setting enables / disables upward inheritability of object slots.
- NXP_AINFO_INHOBJDOWN: this setting enables / disables downward inheritability of object slots.
- NXP_AINFO_INHCLASSUP: this setting enables / disables upward inheritability of class slots.
- NXP_AINFO_INHCLASSDOWN: this setting enables / disables downward inheritability of class slots.
- NXP_AINFO_INHVALUP: this setting enables / disables upward inheritability of the value of a slot.
- NXP_AINFO_INHVALDOWN: this setting enables / disables downward inheritability of the value of a slot.
- NXP_AINFO_PARENTFIRST: this setting determines whether the inheritance search should proceed in a class-first versus object-first manner.
- NXP_AINFO_BREADTHFIRST: this setting determines whether the inheritance search should proceed in a breadth first or depth first direction
- NXP_AINFO_SOURCESON: this setting enables / disables Order of Sources methods.
- NXP_AINFO_SOURCESCONTINUE: this setting enables / disables the full execution of Order of Sources methods.
- NXP_AINFO_CACTIONSON: this setting enables / disables If Change methods.
- NXP_AINFO_CACTIONSUNKNOWN: this setting enables / disables If Change methods specifically when the slot is set to UNKNOWN.
- NXP_AINFO_VALIDENGINE_ON: this setting enables / disables validation of value set by the inference engine.
- NXP_AINFO_VALIDUSER_ON: this setting enables / disables validation of value entered by the end user.

Non-Monotonicity

The Rules Element reasons in a non-monotonic fashion. Non-monotonicity adds several advantages to reasoning. It allows the inference engine to:

- Make decisions and take actions based on incomplete information.
- Make revisions as the environment changes and things it had assumed were TRUE, FALSE, or NOTKNOWN no longer are.

The ability to make decisions and take actions based on incomplete information is very important, as you don't need to absolutely prove that some condition is TRUE before taking an action (in other words, you don't need to prove that there's a road from your home to your work every single day before driving there).

This capability is handled by the fact that the inference engine reasons according to the closed world principle. This principle allows the Rules Element to draw conclusions based on a lack of evidence. If the inference engine has investigated a particular hypothesis and couldn't find anything to show that it is TRUE (because all rules leading to it were FALSE), then the inference engine concludes the hypothesis is FALSE.

The ability to make revisions as the environment changes and things it had assumed were valid no longer are (an earthquake ripped a crevasse between your house and work, therefore you can't drive there) allows the Rules Element to quickly adapt to the environment as it changes.

This capability is handled by the Rules Element's revision mechanism. There are two basic ways a hypothesis can be reinvestigated: either some of the conditions leading to it have data that have been modified in such a way that the conditions containing them change value (from TRUE to FALSE or the reverse), or a hypothesis can be completely reevaluated by resetting it.

Revisions

Revisions are caused by the slots in rules changing value. Whenever an action or a Volunteer (from the user interface or an external program) changes the value of a slot, the hypothesis of any rule which uses that slot will come on the agenda (according to the strategies). Hypotheses generated in this manner will compete with the gates and other action generated hypotheses.

However, if the state of the modified slot's condition does not change due to the changed value of the slot, then the hypothesis will not be re-evaluated. Thus if we have two rules such as:

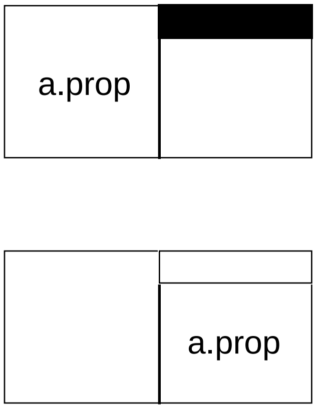


Figure 2-62 Revisions

The top rule is evaluated first, and then, some time later the bottom rule is evaluated. If the bottom rule is `TRUE`, then its actions list will fire. The action on the slot `a.prop` will put all rules which have `a.prop` in their conditions list onto the agenda to compete with other forward action-effects generated hypotheses. However, if the condition involving `a.prop` in the target rule has not changed state, then the rule will not be evaluated again.

Reset

The case may often arise when you wish to evaluate a hypothesis again regardless of whether or not the state of its conditions have changed. This will occur if one wants to implement loops to monitor an activity. The Rules Element provides a special operator for this situation, namely the Reset operator. The Reset operator will return the state of any slot to `UNKNOWN`. In addition, if a hypothesis is Reset, it will not only set the state of the hypothesis to `UNKNOWN`, but any rules and the conditions of those rules to `UNKNOWN` as well.

If there are any hypothesis subgoals leading to a hypothesis which is Reset, then they will be Reset as well along with their rules, conditions, and subgoals. Thus resetting a hypothesis is a recursive action along the backward links. Note that Resetting a hypothesis does NOT reset all the slot values (or the data) used in the rules and methods. The only other slots which will be set to `UNKNOWN` aside from the original hypothesis are hypotheses which are subgoals of the original hypothesis. It merely sets the conditions to UNKNOWN so that they can come on the agenda again without having to modify the value of a condition.

The Reset operator allows some very complicated behavior. For instance, one could have a network such as:

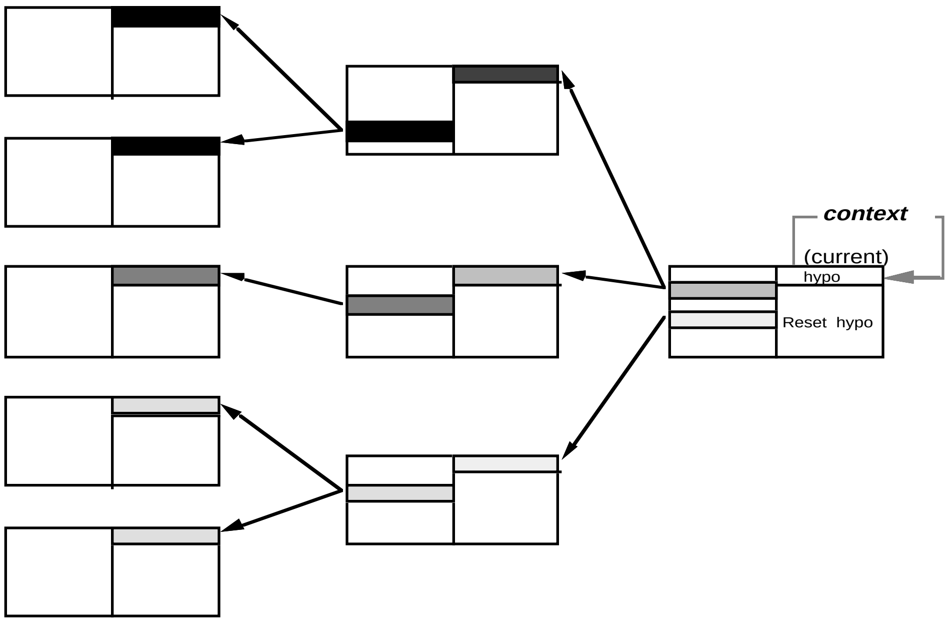


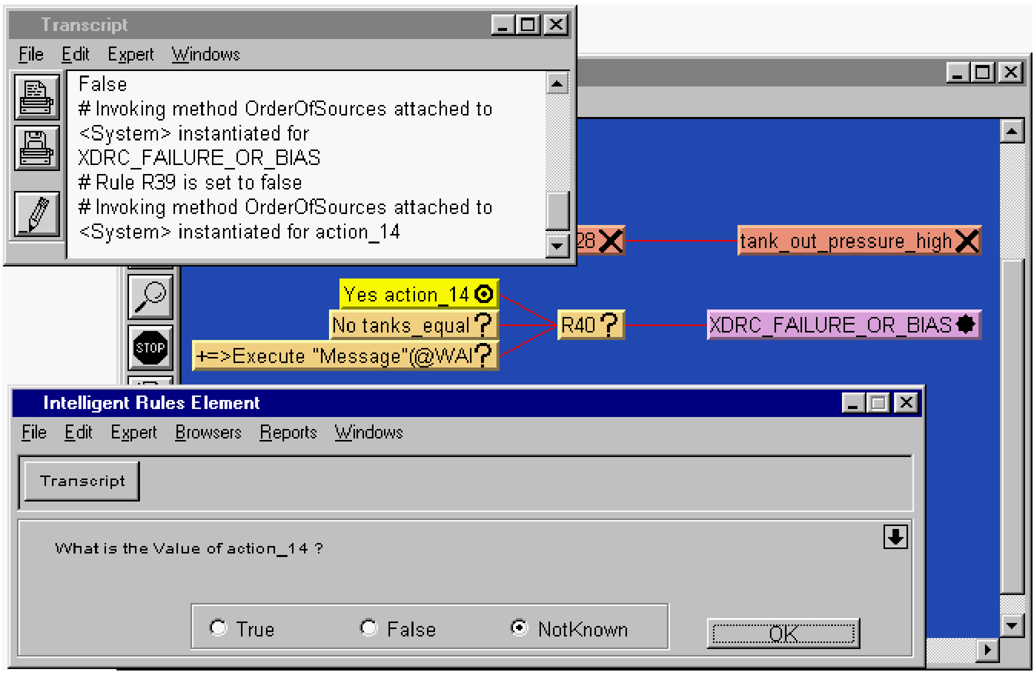
Figure 2-63 Multiple Level Reset

In this example, the inference engine is investigating the hypothesis labeled current. This hypothesis causes a series of other rules to be put on the agenda for evaluation. It also has a context link to itself. Since context links are the lowest priority, everything else will be evaluated first. After all of the rules are evaluated, the current hypothesis is reset by a right-hand side action. This is recursive along the backward links, so all of the hypotheses and rules are Reset as well (the other left-hand side data keep their values). Since everything else in the knowledge island has been evaluated, the original hypothesis is evaluated again due to the context link and the process starts once again.

Once again, note that when you Reset a hypothesis, the hypothesis and all of the conditions leading to the hypothesis are set to UNKNOWN, but all of the data (aside from sub-goals) retain their original values. If you want to Reset all of the left-hand side data as well, you can use the ResetFrame operator.

Interpretations

Interpretations work with revisions in the same manner as they work with the inference engine agenda in general. Namely, the interpretation is resolved first, and after it is resolved, the action takes place on the interpreted value. Revisions will occur on the slot which was modified by the action, rather than the slot which was interpreted.



```
False
# Invoking method OrderOfSources attached to
<System> instantiated for
XDRC_FAILURE_OR_BIAS
# Rule R39 is set to false
# Invoking method OrderOfSources attached to
<System> instantiated for action_14
```

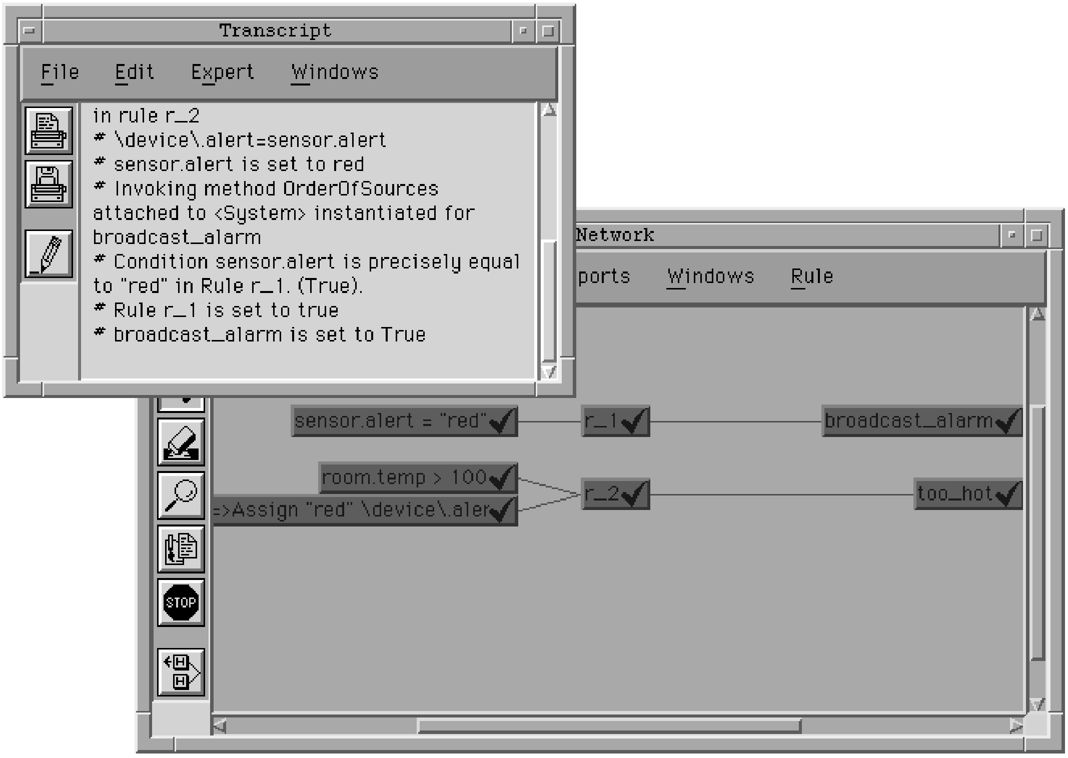
What is the Value of action_14 ?

True False [✓ NotKnown]

Figure 2-64 Interpretations with Revisions

In this example, the `broadcast_alarm` hypothesis has been evaluated as FALSE. However, the rule with `too_hot` as hypothesis has been verified. If the value of device is “`sensor`”, then the action “`Assign “red”`

`\device\.alert`" bears on the slot `sensor.alert` hence a revision in the first rule is triggered:



```
in rule r_2
* \device\.alert=sensor.alert
* sensor.alert is set to red
* Invoking method OrderOfSources
attached to <System> instantiated for
broadcast_alarm
* Condition sensor.alert is precisely equal
to "red" in Rule r_1. (True).
* Rule r_1 is set to true
* broadcast_alarm is set to True
```

Figure 2-65 Interpretations with Revisions

Pattern Matching

For a revision to work with pattern matching, one of two conditions must occur:

- The exact same pattern matching must be included in the source action as in the target rule's condition list and the target rule must be Reset.
- The target hypothesis' conditions list must explicitly mention one of the slots in the pattern matching. In this case, the rule does not need to be Reset.

As we've seen elsewhere, if there is a different pattern matching which contains the same objects in the conditions of a rule, no revision will occur. Either the exact same pattern matching or an explicit mention of the slots must occur for the target hypothesis to be put on the agenda for evaluation.

Conflict Resolution

Hypotheses brought on the agenda due to revisions are in competition with other actions as well as gates. The sole determining factor is the hypotheses inference priority.

Control

The two versions of revisions must be handled separately. Revisions which are initiated by modified data in rule's conditions can be controlled in the same manner as the other actions. Namely, they can be disabled by turning off action effects from either the Global Strategy window or from any condition's or action's strategy operator.

Revisions which are caused by the Reset operator are not quite as easy to control. If a hypothesis is reset, then it's value is `UNKNOWN`. Thus it can be put on the agenda by any of the agenda search mechanisms we have seen previously. To control or disable the re-evaluation of hypotheses which have been Reset, one needs to control or disable all of these other types of inference search strategies.

Multiple Knowledge Bases

There can be any number of different knowledge bases (KBs) loaded in Rules Element at the same time. Each particular structure, whether it is an object, a rule, a method, or anything else, is associated with one particular knowledge base. The Rules Element keeps track of which atom is associated with which knowledge base, though unique identifiers must be assigned to each atom across knowledge bases when loaded into memory. When you modify or delete existing data structures, the changes affect the respective knowledge base. You can change the knowledge base an atom belongs to using the appropriate editor (see the Intelligent Rules Element User's Guide for more details).

There are three ways to load and unload Rules Element knowledge bases:

- From the development interface
- From the conditions and actions of a rule or method, or
- From the Application Programming Interface (API).

You can use any or all of these methods in any particular application, though loading from the development interface is geared more for development and loading from the application programming interface is usually done with the runtime.

In addition to the many user-defined knowledge bases, there are three knowledge base names reserved for the Rules Element:

- untitled.kb
- temporary.kb
- undefined.kb

Untitled.kb

When the Rules Element is initially launched, it creates a default knowledge base for the user with the name `untitled.kb`. All data structures which are created are put in this knowledge base. When you eventually want to save your work, you must choose another name for the knowledge base.

Temporary.kb

You can create dynamic objects as well as dynamic links between objects and classes in the Rules Element. All of these new data structures are stored in the special knowledge base `temporary.kb`. Everything in this knowledge base is deleted when you restart a session or when you exit the Rules Element.

Note: You cannot explicitly transfer any structures into temporary.kb. However, you can transfer any structures to another knowledge base using the ChangeKB command.

If you wish to save the dynamic structures, you can save them using the Save Knowledge Base command from the Expert menu. However, you must first rename the knowledge base.

Undefined.kb

If you reference an atom without explicitly defining it, then the implicit definition will be stored here. For example, assume you have divided your application into several knowledge bases. One of them contains all of the object definitions, while the others contain rule definitions. This will work fine as long as the object definitions are loaded before the rule definitions. If you load a rule such as:

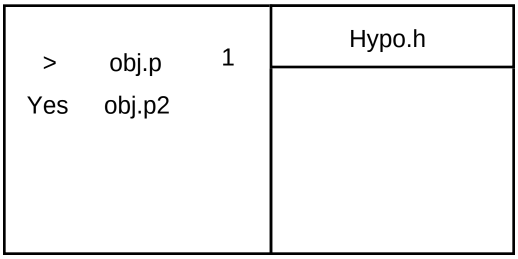


Figure 2-66 Loading a Rule Without Defining the Objects

and you have not defined the objects `obj` and `Hypo` and the properties `p`, `p2`, and `h`, then the Rules Element will put their implicit definitions in the knowledge base “undefined.kb”. Thus undefined.kb would contain an integer property `p`, and boolean properties `p2` and `h`, and object `obj` with two properties `p` and `p2`, and an object `Hypo` with a property `h`.

If some or all of the undefined objects and properties are then loaded, the Rules Element will remove their respective definitions from the undefined.kb knowledge base.

Note: You cannot explicitly transfer any structures into or out of the undefined.kb knowledge base.

Current Knowledge Base

There is one and only one knowledge base which is considered the current knowledge base. Whenever you explicitly create new data structures using the Rules Element editors, they are added to this knowledge base. By default, the most recently loaded knowledge base is the current knowledge base. If you wish to change this, you can use the “Set Knowledge Base” command from the Expert menu.

Modular Knowledge Base Architecture

Dividing your application into several knowledge bases allows you to modularize your knowledge. This has several benefits:

- Knowledge bases can be associated with particular functions.
- Different people can work on different knowledge bases.
- One method of inhibiting unwanted interactions between rules (there may be rules that you want to separate but cannot be separated in one knowledge base).
- Better performance since only a subset of all the rules and objects are stored in memory.
- Better control over the inference process as only those rules and objects which are pertinent are stored in memory.
- Different knowledge base’s can represent different viewpoints or methods of performing some functionality (kind of like having multiple opinions of the same diagnosis).
- Software engineering control.

The following paragraphs describe these benefits of the modular architecture.

Knowledge Bases Associated with Particular Functions

Having knowledge bases associated with particular functions allows you to load a knowledge base and inference on it only when it is appropriate. For example, you could have an engine diagnostic application. If it seems apparent that there is a problem with the air / gas mixture, then a knowledge base devoted to Carburetors and Fuel Injection Systems could be loaded, if initial indications point to a clutch problem, then another knowledge base could be loaded and so on.

Different Knowledge Bases for Different People

Multiple knowledge bases also allow different people to work on each specific part. Thus in our example above, a Carburetor specialist could write the first knowledge base, while a Clutch specialist could write the second. Each of them can work completely independently of each other without worrying about some of their rules interfering with each other’s knowledge bases.

Dividing Rules from Objects

Rules and objects can also be separated into different knowledge bases. Thus in the above example, there could be one knowledge base which contains the entire object structure of the car. Then, each developer can write rules which reason on these common data structures. The object structure would always remain loaded, while the different rule KBs could be loaded and unloaded as appropriate. This insures that all of the object information is available to each of the reasoning knowledge bases while preserving the modularity.

Performance

Another benefit of dividing your application into several knowledge bases is the fact that performance will improve as every time the Rules Element needs to search for an atom, whether it is an object, a slot, or a rule, there will be many less atoms to search through, hence the search time will be shorter.

Enhancing Control of Inference Engine

Finally, and analogous to the previous discussion, creating several knowledge bases will enhance your control over your application. It restricts the search space thereby reducing the number of extraneous events.

Software Engineering Control

This final advantage combines some of the advantages described earlier. Dividing your application into several knowledge bases allows you to:

1. Define in one central knowledge base all the classes, objects, and properties which are common to all of the other knowledge bases
2. Turn off Auto-Creation of Atoms using the SetUp Environment command.
3. All of the application developers receive the “globals” knowledge base and create their applications bases around it. This insures that all of the developers are working with common data structures rather than having similar names for the same thing. If an application developer creates a new atom whether intentionally or not, then he will be prompted by the Rules Element whether or not he actually wants to create it. If it’s a mistake, then creation can be cancelled. If not, then it can be merged into the global knowledge base if other developers could use it or kept local if it is specific to that one knowledge base. In all cases, atoms must be assigned unique names.

Inferencing With Multiple Knowledge Bases

For the purposes of inferencing, there is no distinction between currently loaded and enabled knowledge bases. The Rules Element processes the objects and rules from many knowledge bases just as if they were all part of the same knowledge base.

Passing control from one knowledge base to another can proceed in one of several different fashions. If both of them are loaded at the same time, then propagation can proceed along any of the general inference search strategies. If you have a general control knowledge base which dynamically loads the appropriate knowledge base when needed, then unloads it when it is finished and loads the next one, then context links work well. When a hypothesis is evaluated which has other hypotheses in its context, then these hypotheses are put on the agenda. It doesn't matter that there aren't any rules supporting the hypothesis when it is put on the agenda.

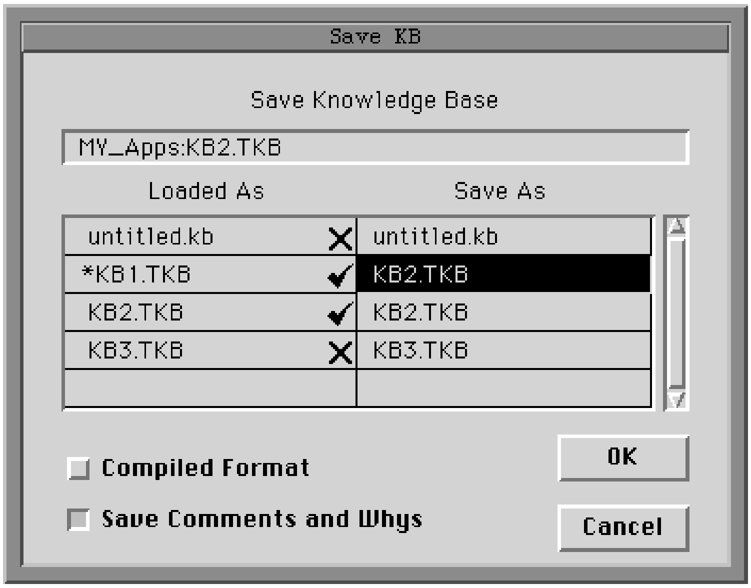
Since context links have the lowest priority, the source knowledge base will be evaluated first and then control will return to the control knowledge base which will unload the source KB, load the new KB, and when the context generated hypothesis is evaluated, the new rules will have been loaded.

Merging Multiple Knowledge Bases

There are several different levels of merging knowledge bases:

- If you merely wish to transfer a couple of structures from one knowledge base to another, the easiest route is to use the Change KB command from any of the Rules Element's editors. For example, if the object `obj` belongs to KB2.tkb and you want to store it in KB1.tkb, merely find `obj` in the Object Editor, choose "Change KB" from the popup menu, and finally choose KB1.tkb as the new parent knowledge base.
- If you wish to completely merge two knowledge bases, then choose Save Knowledge Base from the Expert menu. There is a checkmark or "X" icon in the "Loaded As" column indicating whether the file will be saved or not. Whenever the knowledge base filename appears with a checkmark icon, the file will be saved to the knowledge base name shown in the "Save As" column. For example, assume there are three knowledge bases currently loaded, KB1.tkb, KB2.tkb, and KB3.tkb. Assume also that we wish to save the contents of the first two into the contents of the second one (thus completely disregarding KB3.tkb). Display the checkmark icons for KB1.tkb and KB2.tkb to say you want the contents of those two knowledge bases, then edit the name of

KB1.tkb to match the name of the merge to file KB2.tkb:



Save KB

Save Knowledge Base

MY_Apps:KB2.TKB

Loaded As		Save As
untitled.kb	✗	untitled.kb
*KB1.TKB	✓	KB2.TKB
KB2.TKB	✓	KB2.TKB
KB3.TKB	✗	KB3.TKB

Figure 2-67 Merging Knowledge Bases

- If you wish to merge large portions of several knowledge bases but not the entire contents, the easiest method is to save the knowledge bases in text format, and then modify the knowledge bases with whichever text editor you prefer.

Effects When Loading

When you load a knowledge base, everything defined in the knowledge base is loaded into memory. Several things may happen when you load a knowledge base:

- If there is an atom in one of the currently loaded KBs which is also defined in the newly loaded KB, then a warning will be issued in the transcript, and the old definition of the atom will be replaced by the definition from the new KB. Note that an error will not occur, just a warning message.
- If any global strategies are declared in the newly loaded knowledge base, they will overwrite the existing strategies. Once again, a warning message will be issued.
- If there are some atoms which are referenced, but not defined in the newly loaded knowledge base and they also are not defined in any of

the other knowledge bases which are loaded, then their implicit definition will be contained in a special knowledge base entitled "undefined.tkb"

Since new definitions will overwrite old definitions, the order in which knowledge bases are loaded is important. Additionally, if some knowledge bases contain the object definitions, while others contain only rules, the object definitions should be loaded first so that all objects referenced by the rules are defined.

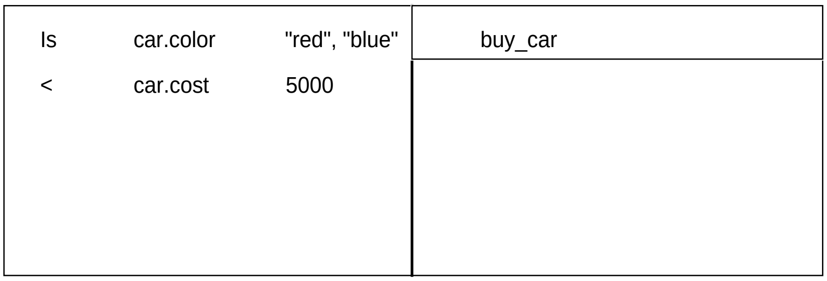
No hypotheses will be put on the agenda as a result of loading a knowledge base. Hypotheses are put on the agenda due to changes in the value of conditions' data or when the inference engine needs to evaluate a particular slot. Since loading a knowledge base doesn't cause either of these two events to occur, no hypotheses will be put on the agenda.

Even if all of the data in a newly loaded hypotheses' conditions are `KNOWN` because of a previously loaded KB and would cause the hypotheses evaluation to be `TRUE`, the hypothesis will not be evaluated. One of the inference search mechanisms must make the hypothesis relevant after it is loaded.

For example, assume we know the values of the following slots:

```
car.color = "red"
car.cost = 3995
```

Now assume a knowledge base is loaded which contains the following rule:



Is	car.color	"red", "blue"	buy_car
<	car.cost	5000	

Figure 2-68 Newly Loaded Rule

The `buy_car` hypothesis and the rule displayed in Figure 2-68 will not be evaluated even though both of the conditions would evaluate to `TRUE` (unless an independent event occurs, such as the evaluation of another rule which contains `car.color`, `car.cost`, or `buy_car`).

There are several different levels at which a knowledge base can be loaded:

■ Enable: all definitions in the knowledge base are fully effective and operational, including objects, classes, properties, rules, and methods.

- DisableWeak: object, class, and property definitions in the knowledge base are in effect. Rules and methods are defined, but are temporarily disabled and unavailable for inference processing; they can later be reenabled by specifying load level “Enable”. Any such disabled rules or methods already on the agenda remain there and will be processed normally.
- DisableStrong: object, class, and property definitions in the knowledge base are in effect. Rules and methods are defined, but are temporarily disabled and unavailable for inference processing; they can later be reenabled by specifying load level “Enable”. Any such disabled rules or methods already on the agenda are removed from the agenda and will not be processed.

When you load a knowledge base from the development interface, it is always loaded at the “Enable” level.

The Initial Value meta-slot attribute is not evaluated when a knowledge base is dynamically loaded. Initial values only take place when the knowledge base is loaded from the development interface or a restart session command is given. Hence always use RuntimeValue for dynamically loaded KBs.

Effects When Unloading

There are several different levels at which to unload knowledge bases. The effects depend upon what level the unloading is performed at:

- Enable: all definitions in the knowledge base are fully effective and operational, including objects, classes, properties, rules, and methods.
- DisableWeak: object, class, and property definitions from the knowledge base remain in effect. Rules and methods remain defined, but become temporarily disabled and unavailable for inference processing; they can later be reenabled with `LoadKB`. Any such disabled rules or methods already on the agenda remain there and will be processed normally.
- DisableStrong: Object, class, and property definitions from the knowledge base remain in effect. Rules and methods remain defined, but become temporarily disabled and unavailable for inference processing; they can later be reenabled with `LoadKB`. Any such disabled rules or methods already on the agenda are removed from the agenda and will not be processed.
- Delete: Object, class, and property definitions from the knowledge base remain in effect. Rules and methods are permanently deleted from memory and no longer available for inference processing; they can be reenabled only by reloading the knowledge base with `LoadKB`.

- Wipeout: All definitions from the knowledge base are permanently deleted from memory, including objects, classes, properties, rules, and methods; they can be reenabled only by reloading the knowledge base with LoadKB. If there are object definitions which are defined in the unloaded knowledge base but referenced by other knowledge bases, the definitions will be stored in the special KB "undefined.kb".

When you unload a knowledge base from the development interface, it is always unloaded at the "Wipeout" level.

Summary

This concludes our discussion of the Rules Element and the inference engine. We have seen that the Rules Element represents the world in terms of objects, generalizations of those objects called classes, and parts of those objects called subobjects. Objects and classes are described by properties. Specific properties of objects or classes are called slots. Slots store all of the information in the Rules Element. Slots can be either public or private.

Methods describe how a slot should behave. They give information such as how the slot should determine its value (Order of Sources), what it should do if its value changes (If Change), or any custom-defined operation needed for the application. Methods attached to private slots ensure data is protected from change by the application. The private slot's behavior is considered encapsulated in the single method associated with the slot.

Inheritance allows for genericity. It allows you to define a value or behavior at a parent level and have all of the child objects or classes inherit it. The Rules Element also supports both upward inheritance (from object to class) and multiple inheritance (a particular object or class may have many different parents from which to inherit). Properties, values, and methods can be inherited.

Dynamic objects and dynamic links between objects and classes allow the representational paradigm to accurately model a changing world. It also allows objects to inherit from one parent at one time and a different parent at a later time. Dynamic objects allow the system to be more flexible as the Rules Element can create new objects whenever it needs them.

Rules provide the heuristics and relations in the knowledge base. They reason upon the object representation. Rules are symmetric so they can be processed in either a forward or backward direction. There are three fundamental parts to a rule: the conditions list, the hypothesis, and two separate actions lists. If all of the conditions are TRUE, then the hypothesis is set to TRUE and one of the actions lists is executed (if present). If any of the conditions are FALSE, then the hypothesis will be FALSE and the other actions list is executed (if present).

The Rules Element is an agenda-based system. This means that it processes events according to how they were generated rather than merely in a LIFO or FIFO algorithm. The Rules Element keeps a prioritized list of hypotheses to evaluate. It is important to note that the Rules Element agenda lists hypotheses and not rules.

The basic order of event evaluation is

- Backward chainings, method actions are processed immediately
- Suggested hypotheses have the highest queued priority
- Hypothesis forward generated hypotheses have the next highest priority
- Gates and action (rules and methods) generated hypotheses are next
- Contexts are last.

Within each of these categories, the hypotheses inference priorities determine which hypothesis will be evaluated first. It is important to note that method *actions* are evaluated immediately, however the hypotheses they generate as relevant goals are in competition with gates and forward action-effects generated hypotheses.

The Rules Element can be embedded within your application. It can be controlled through the application programming interface. Slot values can be volunteered, hypotheses can be suggested, knowledge bases loaded, and sessions started and restarted. You also have access to working memory to investigate the values of any slots, what's being processed, and so on.

The Rules Element also supports nonmonotonic reasoning. There are two basic forms of nonmonotonic reasoning:

- Making inferences based on a lack of evidence
- Making revisions on previous conclusions.

Both of these allow the Rules Element to better deal with a constantly changing world.

The Rules Element allows you to modularize your knowledge by breaking it up into several different knowledge bases. This allows you to separate the inferencing parts (rules) from the representation parts (objects), or allow different knowledge bases to perform different functions.

Finally, the central idea to remember about the Rules Element is the notion of temporality. The Rules Element performs actions according to the current state of the environment and then proceeds accordingly. Thus if a slot needs to inherit something, it looks at who its parents are when it needs to inherit the property, value, or behavior. Previous or future parents or children have no influence on this inheritance event.

Similarly, when Rules Element inference engine evaluates a slot in a condition, it looks for possible gates as soon as the slot is evaluated if gates are enabled. Hypotheses whose conditions would evaluate to TRUE are considered relevant and put on the agenda. It doesn't matter if the rules and hypotheses are from the same or different knowledge bases. If at some future time gates are disabled, any hypotheses already on the agenda will still be evaluated since they were relevant at the time they were queued for evaluation.

Chapter 3

Primer

In this chapter we explore several features by executing two small knowledge bases developed especially for this purpose. This chapter contains sessions for you to perform using the Intelligent Rules Element.

Introduction

Understanding the features of the Rules Element will help you develop knowledge-based applications that operate efficiently. To assist you in this process three sessions were developed around several small knowledge bases that act as a getting started primer. You will use these simple nine rule KBs to conduct inferencing sessions while using the Rules Element and GUI builder facilities to investigate consequences.

In each of the three sessions presented in this chapter some knowledge of the Rules Element facilities and operations is assumed. The graphics that accompany each action description depict the appearance of the Network windows following the completed operation.

Note: The knowledge processing sessions in this chapter require an understanding of the Rules Element facilities and operations. Refer to the Elements Environment Getting Started manual for an overview of the facilities. Familiarity with the Intelligent Rules Element User's Guide is also recommended.

Loading the Primer Knowledge Base

The primer is composed of several small knowledge base files. The files are as follows:

`primer.tkb`	The file needed to run either primer1 or primer2.
`primer1.tkb`	Completes primer.tkb for session one and two.
`primer2.tkb`	Completes primer.tkb for session three (with a script).

Note: When you want to load the primer files, do not load `primer1` and `primer2` at the same time. Compilation errors will result since these two KB files define structures differently. The primer file `primer.tkb` must be loaded to run `primer1` or `primer2`.

Start the Rules Element on your system. Use the following procedure to load the primer knowledge base.

1. Launch the Rules Element application. The system displays the runtime window, select OK. The system displays the Main Window.
2. Move your mouse cursor over the Expert menu and display the menu options.
3. Select the Load Knowledge Base option from the list. The system displays a dialog window that gives you access to system files.
4. If your system requires a pathname to locate the file, type the pathname with the filename in the text edit field and press Return. For example, the complete pathname might be:

   ```
   /ee/c/examples/rules/primer/primer.tkb
   ```

   The IBM PC and Apple Macintosh computers let you browse the directories to locate the file. Display the Examples directory and double-click on the filename `primer.tkb` from the list.
5. The system loads the file into memory.
6. Repeat the procedure to load the second primer file needed for the first session:

   ```
   /ee/c/examples/rules/primer/primer1.tkb
   ```

Starting a Session Over

The actions in the session must be performed exactly as given in order for knowledge processing to proceed in step with the graphics. If for example you enter unspecified data, the order of the rule evaluation may change. If at any point you get off track and want to start over you can perform the following actions:

1. Select the Clear option from the global popup menus of the Rule Network window and the Object Network window. The system will remove any previously displayed rules and objects.
2. Select the Restart option from the Expert menu. This action returns all previously entered data to its initial state thus enabling knowledge processing to start over.

Note: If a Resource Browser window has the focus, the Restart command from the keyboard will be ignored. You must first click on any Rules Element window in order to restart the inference engine.

Exiting the Session

Whenever you wish to terminate the session and close all the Rules Element windows select the Quit option from the File menu. The system will display a dialog box asking whether you want to quit; select the OK button. If the knowledge base was altered during a session another dialog box asks whether you want to save the changes to the file; always select the NO button. If you want to practice editing the primer knowledge base, be sure to use a renamed copy of the original file for your own editing sessions.

Conducting Your Own Sessions

The three sessions presented here demonstrate only a few of the valid reasoning pathways for inferencing to proceed. The exact path is determined by how you begin knowledge processing (suggesting hypotheses, volunteering data, or a combination of the two) and also on the data you provide during the session. You may want to vary these parameters to conduct your own sessions after completing the ones in this chapter. For more information about the windowing environment, refer to the User's Guide

Knowledge Processing - Session 1

You will conduct your first session using two primer knowledge bases, `primer.tkb` and `primer1.tkb` to gain familiarity with several important features of the Rules Element. Furthermore, session one demonstrates that even with the addition of the Resource Browser interface builder, the Rules Element shell lets you process your knowledge base without the GUI engine. To accomplish this, the session control panel of the Main Window displays the question to solicit data for processing.

Additional, important points of session one include the following.

- Visualizing the rule structure (IF-THEN-ELSE) in the rule network.
- Visualizing important operators that you can use in rules (SendMessage and Assign).
- Using data validation parameters (from a meta-slot) to validate end-user input.

Use the following procedure to conduct session one:

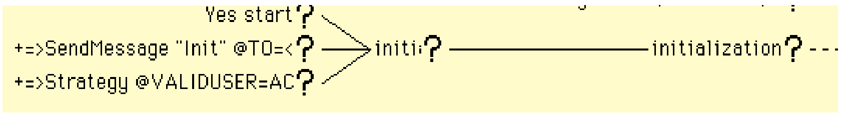
1. Select the Rule option from the Browsers menu in the menu bar. The system displays the empty Rule Network window. Click inside the

Rule Network window and display the local popup menu, select the Focus on Hypothesis option from the list.

2. The system displays a selection window that lists every knowledge base hypothesis available for display in the rule network. Double click on the hypothesis `pump_breakdown`. This hypothesis shows a context link to another hypothesis which we will use to initialize the knowledge base.

   ☞ Scroll the network window to view the `initialization` hypothesis. Scrolling of the network diagram can be accomplished by positioning the mouse cursor inside the window (not on top of the network diagram) and then clicking and dragging.

   ☞ Position the mouse cursor over the `initialization` hypothesis and extend the network to the left. This rule shows two actions which we use to set-up the environment. The `SendMessage` operator is particularly useful since it can initiate actions directly on the desired list of knowledge base objects.



3. Now let's place a hypothesis on the Rules Element agenda for evaluation.

   ☞ Display the local popup menu for the `initialization` hypothesis.

   ☞ Select the Suggest option from the list.

4. To begin knowledge processing with the suggested hypothesis:

   ☞ Position the mouse cursor over an inactive area of the Rule Network window and display the windows popup menu.

   Note: Single-button mouse users must press the Command key and mouse button together to display the windows popup menu.

   Double-button mouse users must press the CTRL key and right mouse button together to display the windows popup menu.

   Triple-button mouse users must press middle mouse button to display the windows popup menu.

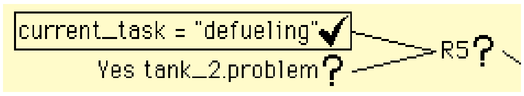
☞ Select the Knowcess option from the list.

5. The system displays the question in the session control panel of the Rules Element main window to solicit data for the single condition of the `initialization` rule.

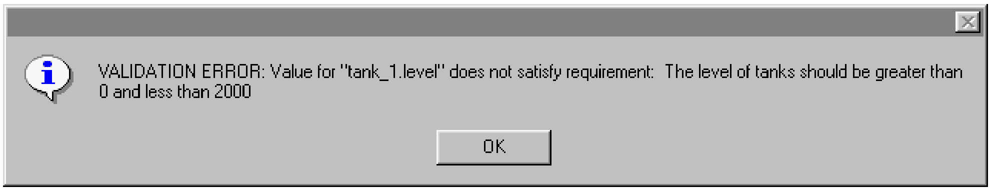
    ☞ Click on the option “True” and then select the Ok button or press the Return key to make the single condition TRUE.

    ☞ Return to the Rule Network and you will see that both the actions of the rule were triggered, including the `Strategy` operator that activates the data validation function. If desired, you can display the Rule Editor from the network to visualize the `Strategy` operator in the editor by selecting the Edit... option from the local popup menu you display on the rule name node.

6. Return to the Rules Element main window; the system expects a value for `current_task`. Display the list of options by clicking on the choice box arrow button next to the right of the highlighted field. Select the “Defueling” option for the current task and press Return to make the first condition of the `pump_breakdown` hypothesis TRUE. Notice that in the rule network, the equal sign (=) syntax is used to test data in a condition.

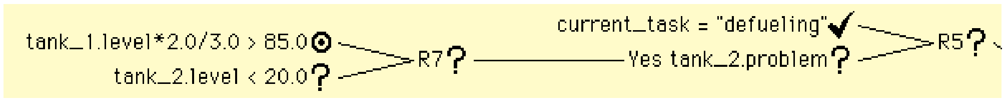


7. Next the system asks for the value of `tank_1`. Enter a value of “3000” into the highlighted input field and press Return. The system displays a message window demonstrating data validation, because in this case a data validation meta-slot for `tank_1` will only accept a value between 0 and 2000.

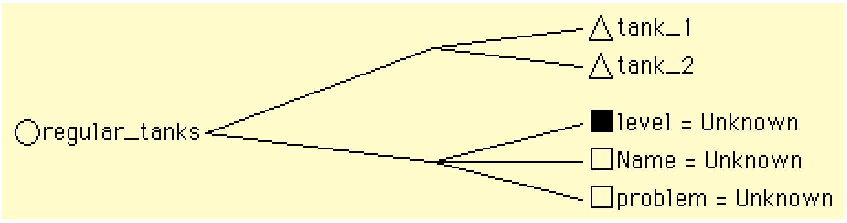

VALIDATION ERROR: Value for "tank_1.level" does not satisfy requirement: The level of tanks should be greater than 0 and less than 2000

☞ Click on the alert window OK button.

☞ Instead of typing an acceptable value for `tank_1`, let’s identify the data validation expression in the Meta-Slot Editor. First return to the rule network and expand the diagram to the left of the `tank_2.problem` hypothesis. The "target" icon identifies the current condition in the rule network:



☞ Now display the local popup menu for the current condition and select the Focus Object Network option. The system displays the Object Network with the object `tank_1` and its two classes `regular_tanks` and `tanks`. To view the meta-slot, expand the object network diagram for the class `regular_tanks`. You will see that `tank_1` inherits the meta-slot on the property `level` (solid square) from this class.



8. Now display the local popup menu for the property `level` and select the Edit Meta-Slot option. The system displays the meta-slot for `Regular_tanks.level`. Notice that it is here that data validation is defined. This shows that objects of a class can inherit this important feature like other meta-slot fields.

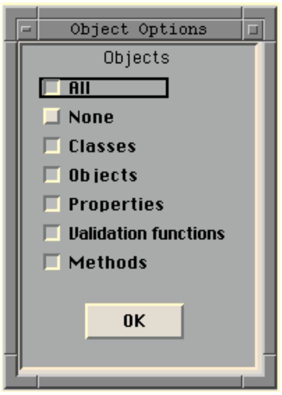
☞ To display the data validation definition, place the Meta-Slot Editor in edit mode by selecting the "pencil and paper" icon, then click on the data validation field to view the string in the text edit line of the editor. The field has the following user-defined validation expression:

```
SELF.level > 0 AND SELF.level < 2000
```

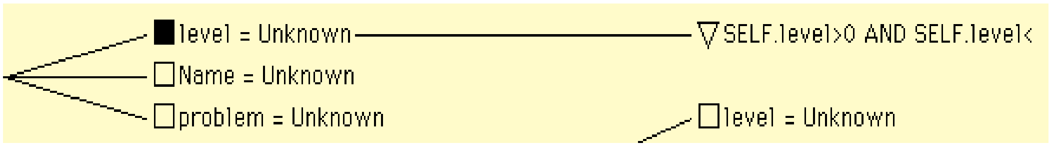
Notice that we have used the `SELF` and `AND` keywords to simplify the definition and make it generic for each object that inherits the meta-slot. See the Intelligent Rules Element Language Reference for a description of these and other keywords.

☞ Select the "stop sign" icon to cancel edit mode in the Meta-Slot Editor, then close the editor to return to the object network.

9. Let's visualize the data validation function in the object network by using the display filter window. Display the Object menu from the Object Network window menu bar and select Options.... The system displays a dialog window that lets you filter what appears in the Object Network. Select the "All" check box to include "Validation functions" and click on the Ok button to return to the Object Network.



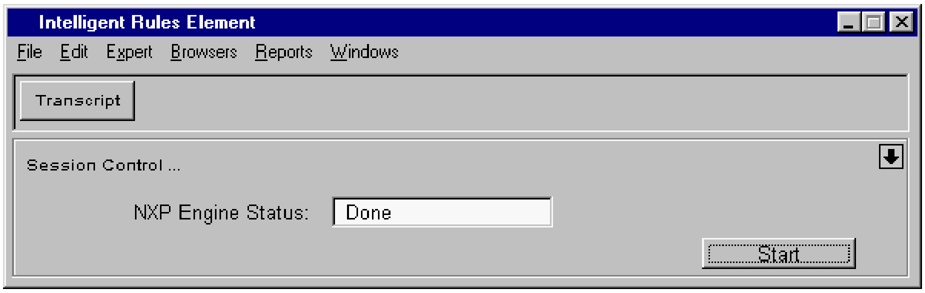
☞ Expand the object network diagram to the right by clicking on the property `level`. The system expands the diagram to show the validation function which appears with an inverted triangle:



10. Close the Object Network and return to the Rules Element main window. The system is still waiting for you to supply a value for `tank_1`.

☞ Enter a value finally for `tank_1` of "150". Notice the system accepts the value because it is in range.

☞ To complete the session, enter a value of “10”. The system displays the engine status “Done” in the Engine Status field.


Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
Session Control ...
Start

NXP Engine Status: Done

☞ Proceed to the next section without making further changes.

By reusing the currently displayed rule network diagram the next knowledge processing session can be completed in twenty minutes or less. You can also conclude the current knowledge processing session and exit the application.

Knowledge Processing - Session 2

You will conduct your second session in a similar fashion to the first session. You should still have `primer.tkb` and `primer1.tkb` loaded in memory. Both sessions use the same rules for knowledge processing. In this session, however, you will see how to pause the inference engine by placing a breakpoint filter on a method.

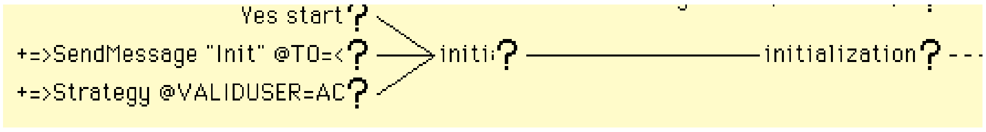
Additional, important points of Session Two include the following.

- Examining the way a method and its actions are triggered.
- How to define a method filter (breakpoint) in the Object Network window.

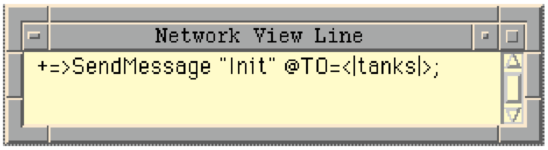
Use the following procedure to conduct session two:

1. Before beginning a new session, you must select the Restart option on the Expert menu.
2. If the Rule Network window does **not** currently show the final rule network diagram from session one, display the global popup menu and select the Focus on Hypothesis option from the list. Double-click on the `pump_breakdown` hypothesis from the selection window. Display the local popup menu for the `initialization` hypothesis and select the Full Left Extent option from the list.

☞ Scroll the network window to view rule `initialization`. In the first session we saw that the initialization rule contains a `SendMessage` operator in the list of actions. We can see from the rule displayed that the method being triggered is named `Init`, but it would help to have a View Line to see the entire definition.



☞ Display the local popup menu for the Rule Network window and select the View Line option. The system displays the View Line window. Position your mouse cursor over the `SendMessage` action in the rule network diagram to see the full line of text. (Macintosh users must first click on the View Line window.)



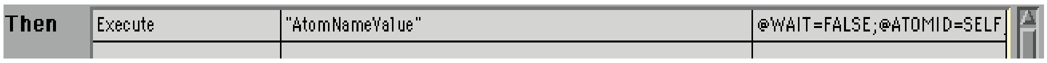
```
+=>SendMessage "Init" @TO=<|tanks|>;
```

Notice that the `Init` method actions will be sent to the members of the class `tanks` since the pattern-matching syntax (`<|tanks|>`) is used.

3. Before we proceed let's view the method action in the Method Editor by selecting the Method option from the Edit menu on the main menu bar. The system displays the Method Editor.

   ☞ Browse the Method Editor by selecting the "I-J" index to view the `Init` method.

   ☞ Since this method's action is not triggered conditionally, only the THEN portion of the IF-THEN-ELSE template is completed. You will need to scroll the template to bring the THEN section into view. The action that appears in this example is an "Execute" statement that assigns a name to the target atom (members of the class `tanks`). See the Language Reference manual for complete information about the Execute library routines.



Then	Execute	"AtomNameValue"	@WAIT=FALSE;@ATOMID=SELF

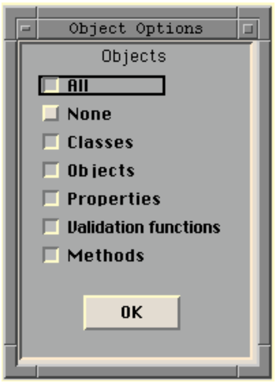
   ☞ Select the Edit mode and click on the third field to view the Execute properties of the AtomNameValue routine. Click on Cancel to return to the editor. Close the Method Editor.

4. Let's return to the Rule Network to visualize the method in the Object Network. Since we already know from the View Line that Init acts on the class `tanks`, we could focus the Object Network on that class:

   ☞ Select "Object" from the Browsers menu on the main menu bar. The system displays the empty Object Network window.

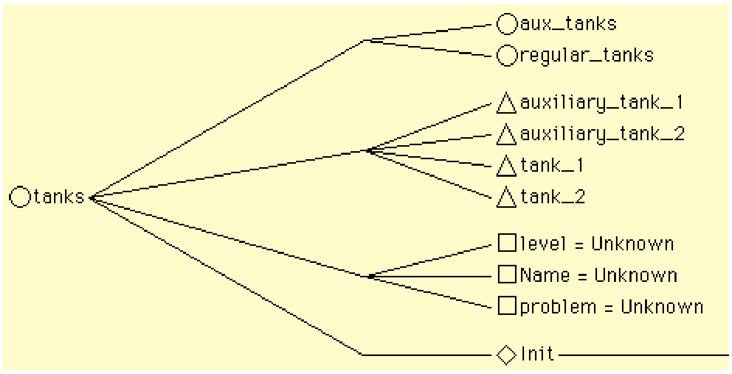
   ☞ Before we can display methods, you may need to choose display options in the Options window for the Object Network. Display the window-specific "Object" menu from the main menu bar and select the menu item "Options...". The system displays a small window that lets you filter what appears in the Object Network.

Select the “All” check box to include “Methods”. Close the Object Options window.



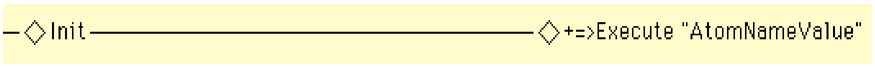
☞ Click inside the Object Network window and display the global popup menu, select the Focus on Class option from the list. A selection window lists every knowledge base class available for display in the object network. Double click on the class `tanks`.

☞ Scroll the object network to view the tanks class. The diagram displays the method `Init` which appears with a diamond.



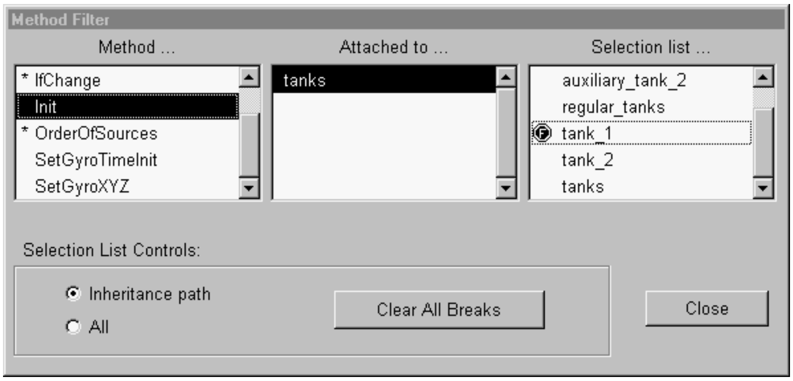
5. In the first session we processed the knowledge base by suggesting the `Initialization` hypothesis. This time, let’s first place a breakpoint on the method “Init” to see how the system behaves during processing.

☞ To view the method in the Object Network expand the diagram to the right of the Init name. The system shows the “Execute” statement that we previously displayed in the “THEN” template section of the Method Editor.



```
—◇Init——————————————————◇+=>Execute "AtomNameValue"
```

6. To place the filter on the method, display the local popup menu for the Init method in the Object Network window. Select the “Method Filter...” option. The system displays the Filter dialog window.



Method Filter

Method ...	Attached to ...	Selection list ...
* IfChange	tanks	auxiliary_tank_2
Init		regular_tanks
* OrderOfSources		tank_1
SetGyroTimeInit		tank_2
SetGyroXYZ		tanks

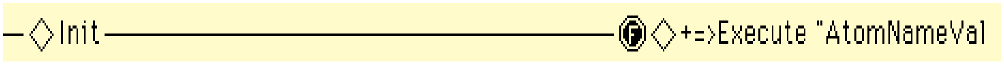
Selection List Controls:

[✓ Inheritance path] All

☞ Since the “Init” method is already selected and it is already attached to the class “tanks”, you need only specify where to place the filter. Select the tanks class member `tank_1` in the righthand list. The system places a small stop sign icon with an “F” inside beside tank_1.

☞ To validate your filter selection click on the Close button.

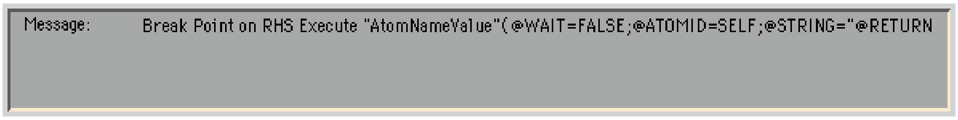
7. Return to the Object Network and select the “stop sign” icon from the window icon bar, then click directly on the method action “Execute...”. The system places a small breakpoint icon in the object network diagram with an “F” inside to indicate that it is a selective breakpoint that you have defined in the Filter dialog window (as opposed to a regular breakpoint that acts on all members of “tanks”).



```
—◇Init——————————————————Ⓕ◇+=>Execute "AtomNameVal
```

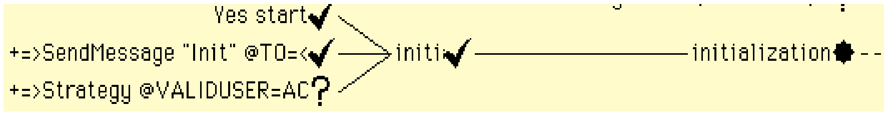
8. With the filter in place, let’s place the `initialization` hypothesis on the Rules Element agenda. Suggest and then Knowcess.

☞ Click on the option “True” and then select the Ok button to make the condition TRUE. The system pauses immediately as the method action is triggered and displays the reason in the Message field at the bottom of the Main Window.



```
Message:        Break Point on RHS Execute "AtomNameValue"(@WAIT=FALSE;@ATOMID=SELF;@STRING="@RETURN
```

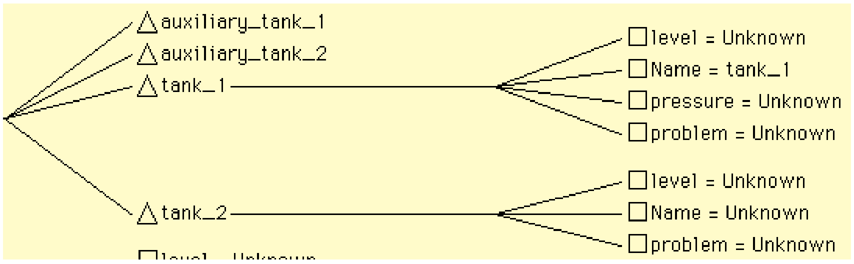
☞ Return to the Rule Network window before selecting the Continue button. The system displays the status of the initialization rule. Notice that the system has not completed rule evaluation and that the SendMessage action has been triggered.



```
                 Yes start✓
+=>SendMessage "Init" @TO=<✓ ——> initi✓ ———————— initialization● --
+=>Strategy @VALIDUSER=AC?
```

☞ Return to the Object Network window to visualize the effect of the method action (from the Execute statement). Expand the object network diagram to the right from tank_1 and tank_2.

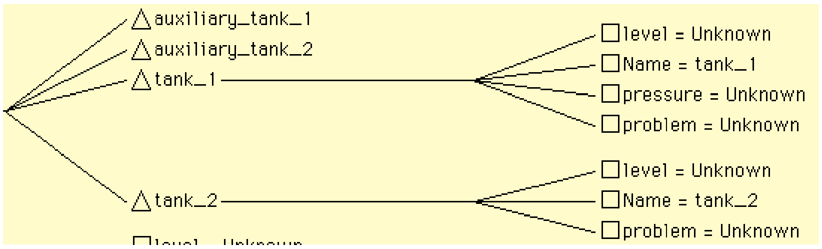
☞ Select the Change Settings option from the Options menu. Select the Show Values option to reveal the current values of data in the Object Network. Click on the Ok button to return to the Object Network. Notice that slot `tank_1.name` has been defined, but `tank_2.name` is still UNKNOWN. This is the result of the method filter which has temporarily halted the inference engine after sending the action to `tank_1`, but before `tank_2`.



```
△auxiliary_tank_1
△auxiliary_tank_2
△tank_1                        □level = Unknown
                               □Name = tank_1
                               □pressure = Unknown
                               □problem = Unknown
△tank_2                        □level = Unknown
                               □Name = Unknown
                               □problem = Unknown
```

9. Return to the Main Window and finally select the Continue button. The inference engine resumes processing and the next condition is evaluated in the already familiar `pump_breakdown` hypothesis (see session one).

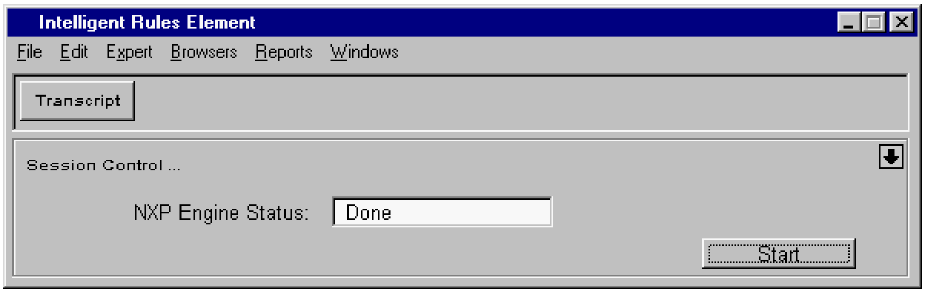
☞ Return to the Object Network window to see the status of the slot `tank_2.name`. It has been defined by the action of the "Init" method as we would expect.



10. Close the Object Network and return to the Main Window. The system is still waiting for you to supply a value for `current_task`.

☞ Double click on the option "defueling" to make the single condition TRUE.

☞ To complete the session, enter a value now for `tank_1` of "35". The system accepts the value without complaint because it is in range (see session one for more information about data validation). The system display "Done" in the Engine Status field of the Main Window session control panel.


Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
Session Control ...
Start

NXP Engine Status: Done

11. Examine the final evaluation status of the Rule Network diagram. Notice that the `pump_breakdown` hypothesis is now FALSE indicating that it failed. The Object Network diagram also shows a similar result.

☞ **Proceed to the next section after unloading primer1.tkb.**

This concludes the demonstration of knowledge processing using methods, data validation, filters, and other features available to you for knowledge base design. You may want to experiment with these two knowledge bases by restarting the data and placing additional filters on the members of the tanks class before processing again.

Session Three begins with an all new knowledge base in order to demonstrate how the Resource Browser and the GUI engine can be used to replace the standard session control panel provided by the Main Window.

Knowledge Processing - Session 3

Conduct your third session after unloading `primer1.tkb`. This session requires that you load `primer.tkb` and `primer2.tkb` into memory.

Note: Before loading the primer files, we recommend that you first unload both files from the previous session (`primer1.tkb` and `primer.tkb`). Compilation errors may result if changes were made during the last session.

This session demonstrates how scripts can initiate actions in the Rules Element. These scripts allow two-way communication between Rules Element atoms and graphical user interface (GUI) elements. Scripts are not only responsible for receiving and sending GUI output, they also receive and display input from the Rules Element. This capability of the Rules Element provides a convenient way for you to design your own knowledge-based application front-end.

In this session, we will put many scripts to use, process knowledge bases, and even view the Script Editor that you can use to create / modify these scripts on the fly (during knowledge processing).

Additional, important points of Session Three include the following.

- Using check box element scripts to initialize knowledge processing by triggering a Suggest hypothesis action from a method (the method is defined in the Rules Element knowledge base).
- Using input field scripts to display Rules Element prompt lines.
- Using several input field scripts to accept "forms-type" input and output, thus combining several questions into one window.
- Note the distinction between "immediate validation" and "deferred validation" (immediate validation can trigger actions immediately, while deferred validation requires the end user to initiate validation to execute runtime actions).
- Observe the effect the GUI engine has on the Rules Element inference engine (Main Window shows engine status is "STOPPED" while the GUI engine handles processing).
- Observe how the links to GUI objects can be inherited down from an application class to each of its objects through an Order of Sources method.

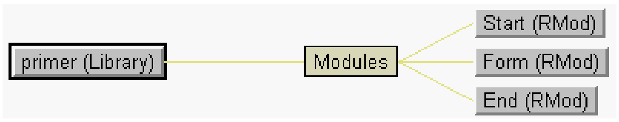
Use the following procedure to conduct Session Three:

1. We want to load the compiled resource file (.dat) into system memory and visualize the already built GUI windows.

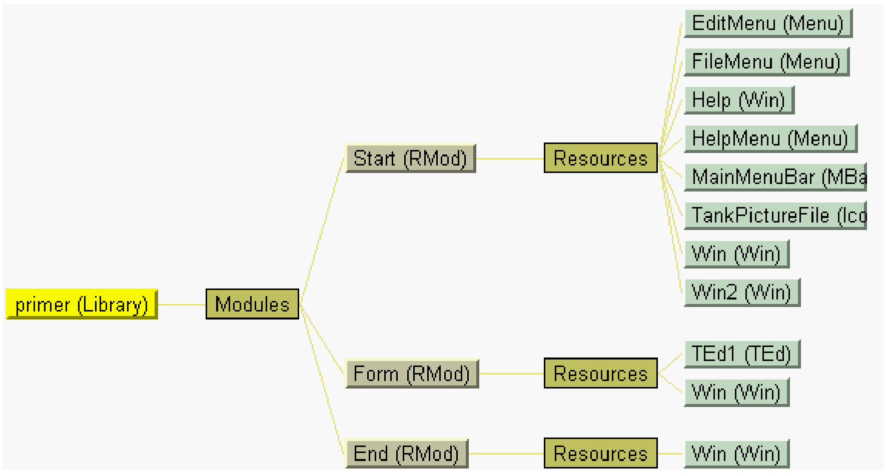
☞ Click on the main window and display the local popup menu in the scrollable area labeled GUI Libraries. Select the Open Application option from the menu. The system displays the file selection dialog window.

☞ In the `EE/c/examples/rules/primer` or `EE/cpp/examples/rules/primer` directory double-click on the `primer.dat` file that defines the GUI windows. The system loads the resources and displays the Resource Browser.

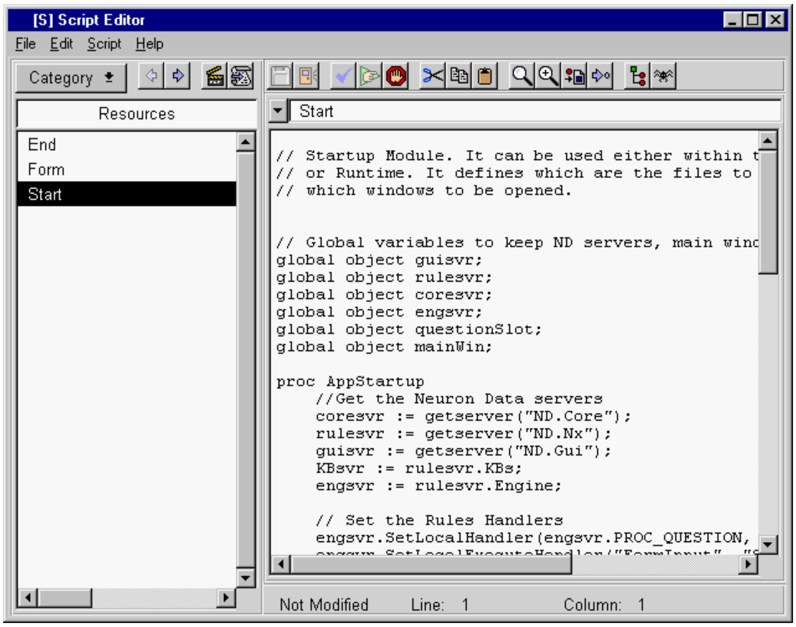
☞ To view the Primer library's modules display the local popup menu for the Library node and select Extend... Modules.



☞ Move the mouse cursor over the browser node "End" until the righthand arrow appears, then click the mouse button. The system expands the node to the right to show the components of the resource module called "End". Repeat this procedure for the remaining two nodes: "Form" and "Start".



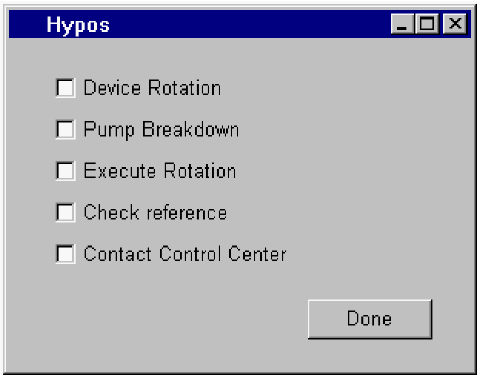
☞ Before we begin the actual knowledge processing session, let's take a diversion to examine several `primer.dat` elements in the Resource Browser. Scroll the resource browser until the "Start" module appears. Position the mouse cursor over the Start module node and display the local popup menu. Select the Edit Application Script option from the menu. The system displays the Script Editor with the script that tells the system how to begin a session with a GUI. The session in this exercise uses this application script currently displayed.



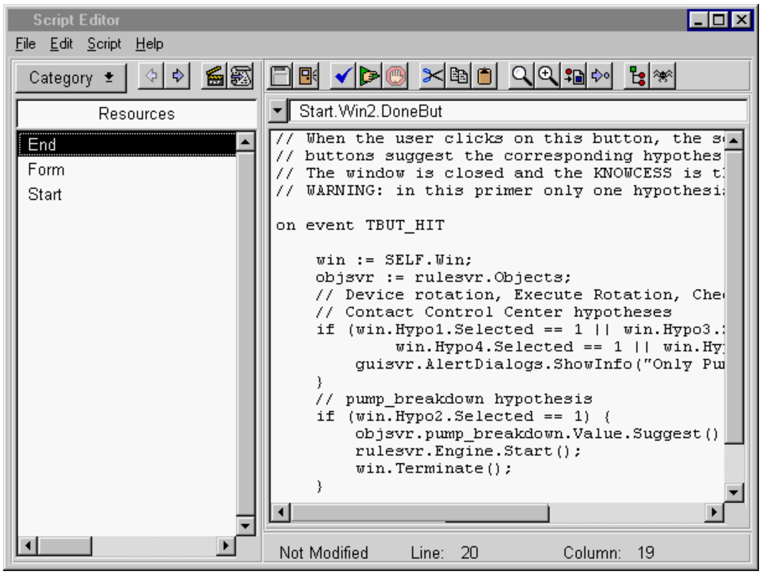
```
// Startup Module. It can be used either within t
// or Runtime. It defines which are the files to
// which windows to be opened.


// Global variables to keep ND servers, main winc
global object guisvr;
global object rulesvr;
global object coresvr;
global object engsvr;
global object questionSlot;
global object mainWin;

proc AppStartup
    //Get the Neuron Data servers
    coresvr := getserver("ND.Core");
    rulesvr := getserver("ND.Nx");
    guisvr := getserver("ND.Gui");
    KBsvr := rulesvr.KBs;
    engsvr := rulesvr.Engine;

    // Set the Rules Handlers
    engsvr.SetLocalHandler(engsvr.PROC_QUESTION,
```

2. Return to the Resource Browser and scroll until the last node of the "Start" module appears, it is labeled "Win2". Double click on the "Win2" node. The system displays the Window Editor of the Resource Browser. This special editor lets you layout and edit the contents of an entire window. In this case, we see the already created window that

contains check boxes corresponding to the familiar `primer.tkb` hypotheses.



☞ Let's view the script that defines the check boxes' behavior during a knowledge processing session. Inside the Hypos window click on the window element labeled "Done" so this item appears selected. Then go to the Window Editor menu bar and choose the "Edit Script " option from the Edit menu. The system displays the Script Editor.



```
// When the user clicks on this button, the s
// buttons suggest the corresponding hypothes
// The window is closed and the KNOWCESS is t
// WARNING: in this primer only one hypothesi

on event TBUT_HIT

     win := SELF.Win;
     objsvr := rulesvr.Objects;
     // Device rotation, Execute Rotation, Che
     // Contact Control Center hypotheses
     if (win.Hypo1.Selected == 1 || win.Hypo3.
             win.Hypo4.Selected == 1 || win.Hy
          guisvr.AlertDialogs.ShowInfo("Only Pu
     }
     // pump_breakdown hypothesis
     if (win.Hypo2.Selected == 1) {
          objsvr.pump_breakdown.Value.Suggest()
          rulesvr.Engine.Start();
          win.Terminate();
     }
```

```
Not Modified      Line:  20        Column:  19
```

3. To try out the script template, we'll recreate the line which suggests the `pump_breakdown` hypothesis. In this exercise, we won't save the changes so don't be too concerned about correctness.

```
objsvr.pump_breakdown.Value.Suggest();
```

☞ To write a script you can either type directly into the script editor text area, or you can assemble the script from the already defined system objects which appear in the scrollable list of Categories on the left side of the Script Editor. In this example we will combine these approaches. Begin by creating an empty line below the script text `objsvr.pump_breakdown.Value.Suggest();`. To do so, insert your text cursor at the end of the script line and press the Return key.

☞ With your cursor at the beginning of the empty line, press the tab key twice so the text to be inserted has the proper indentation. Type the variable name `objsvr` to identify the server needed to process a method. In this case, a variable name substitutes for the object server used to process the method `Suggest`. Type a period (.) after the word `objsvr` to indicate the end of this script element.

4. To get the hypothesis we want for this line we could type `pump_breakdown` but lets use the Script Editor instead. Select the IRE Slots item from the Category drop-down menu. The left hand side of the Script Editor displays all the slots that belong to the currently loaded knowledge base files. You can also view other types of information in this list by changing the selection on the Category menu.

☞ Highlight the `pump_breakdown.Value` hypothesis slot in the list on the left of the Script Editor. To add the item into the script text area at the current text insertion point, select the Paste button on the toolbar along the top of the Script Editor (it is the forth button from the left). It is also possible to drag items from the scrollable list into the script text area by moving the mouse cursor to the right of the highlighted item until the arrow cursor changes to the "drag" cursor. If you want to try drag and drop, with the drag cursor displayed hold the left mouse button down and drag the highlighted `pump_breakdown` item into the script text area where you want the item to appear. Release the mouse button when you have positioned the text cursor at the desired insertion point. Remember to type a period (.) after `pump_breakdown.Value` to indicate the end of this script element.

5. To insert the method used to act on the `pump_breakdown` hypothesis lets use the Script Editor again. This time select the Repositories item from the Category drop-down menu. The left hand side of the Script

Editor displays the list of registered script servers. On PC platforms you will see Microsoft applications that are registered OLE servers alongside Neuron Data Elements servers. These servers allow you to incorporate the script commands from any registered application into our own script. In this case, you need to double-click on the Neuron Data Rules Server item from the list to complete our script line.

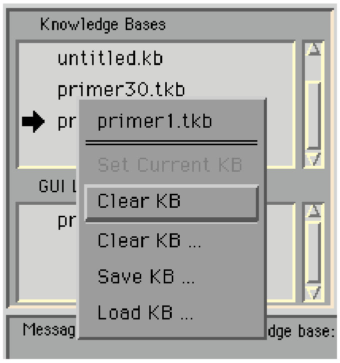
☞ The scrollable list changes to display the components of the Rules Server. Notice the convention of ending the "meta-classes" in "s" for a given class. Object-oriented languages define the meta-class as those methods and constants that apply to all instances of the class. In this example, we are using the slot class functions and not its meta-class. Double-click on the Slot item in the list.

☞ Scroll the list of displayed Slot instance methods and highlight `integer Suggest()`. The Suggest method can be added to our script line using the same techniques described above. Finally, terminate the line with a semi-colon (;) and you will have duplicated the script line:

```
objsvr.pump_breakdown.Value.Suggest();
```

6. Now let's return to the Rules Element main window, but first close these last two editor windows. Select the Close option on the Script Editor File menu without saving your changes. Select the Cancel button on the Window Editor.

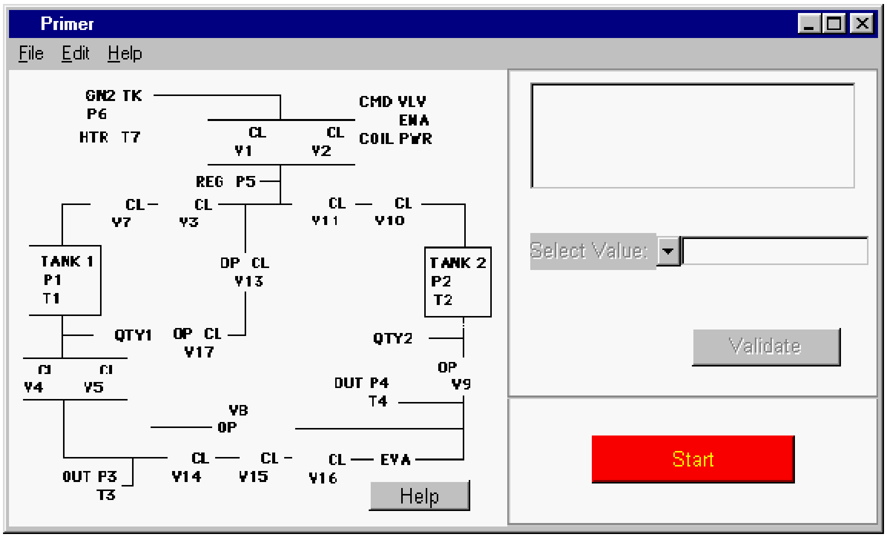
☞ From the Rules Element main window be sure to unload `primer1.tkb` and `primer.tkb` if not previously done. Display the local popup menu inside the lefthand side of the Main Window and choose the "Clear KB..." option from the menu. The system displays the dialog window for you to make the Clear All selection.



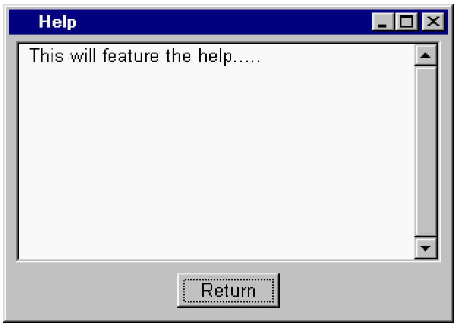
☞ Redisplay the local popup menu and this time select the “Load KB...” option from the menu. You want to load the files `primer.tkb` and `primer2.tkb` into system memory in that order. (Loading file `primer2.tkb` before `primer.tkb` will result in compilation errors; if necessary unload and start again.)

7. Let’s begin a Rules processing session this time using the elements of the graphical user interface and scripting language that we’ve just finished exploring. Move your mouse cursor to the Expert menu of the Rules Element main menu bar and select the Run with Application Script option. This option tells the system that you want to execute the script attached to the Start module that we displayed in the Script Editor. It is unnecessary to specify the name of the script since only one “application script” may be defined for the application.

8. The application script displays the Start window which contains several GUI resources, including a prompt line, a choice box, and two push buttons.



☞ Select the Help push button to view a sample help window. The script attached to the Help button triggered a script method that opens the Help window. To close the Help window click on the Return button.



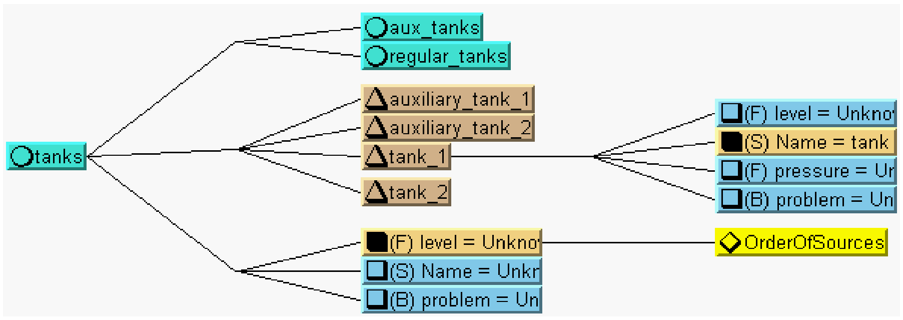
This will feature the help.....

☞ Return to the Start window and select the Start button. The system now displays the hypothesis selection window we viewed previously in the Window Editor. At this time do not make a hypothesis selection.

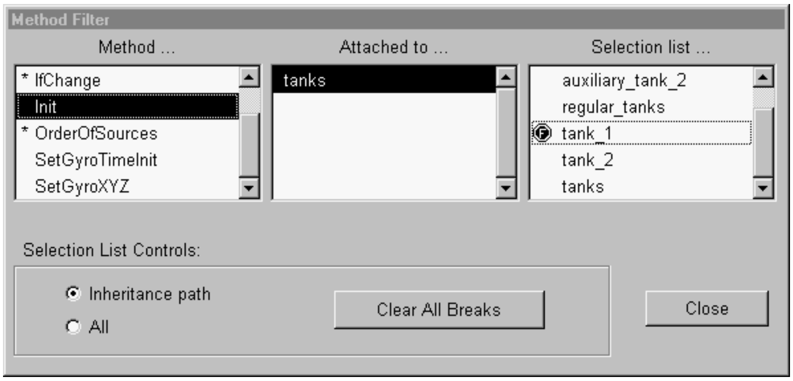
9. Before selecting a hypothesis to suggest, let's set-up debugging tools that we can use to track the rules and GUI-script processing.

☞ Select the Object option from the Browsers menu in the Rules Element menu bar. The system displays the empty Object Network window. Select the Focus on Class option and choose Tanks from the list of classes. The system displays the `Tanks` class in the Object Network.

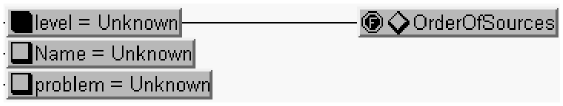
☞ To view the method defined at the `Tanks` class level, open the Options dialog and select methods from the list. When you expand the diagram to the right of the `level` property for the class `Tanks`, the system shows the OrderOfSources system method used to define the value of `tank_1.level` and `tank_2.level`.



☞ We want to place filter on the method to halt the rules processing when this method is triggered. Select the Method Filter... option from the local popup menu on the OrderofSources method. The system displays the Filter dialog window. Select the slot `tank_1.level` from the list on the right side of the dialog. The system places the Filter stop sign beside the selection. Close the Filter dialog window to validate the selection.



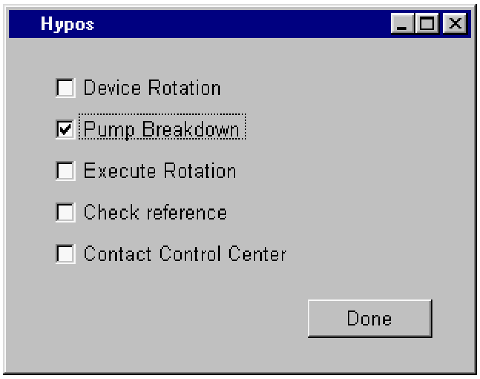
☞ Return to the Object Network and select the “stop sign” icon from the window icon bar, then click directly on the OrderOfSources method. The system places a small breakpoint icon in the object network diagram with an “F” inside to indicate that it is a selective breakpoint that you have defined in the Filter dialog window (as opposed to a regular breakpoint that acts on all members of “`Tanks`”).



☞ To view script processing, select the Script option from the Edit menu of the Rules Element menu bar. When the Script Editor appears select the Script Trace button on the toolbar along the top of the Script Editor (it is the last button on the right). The system displays the empty Trace window.

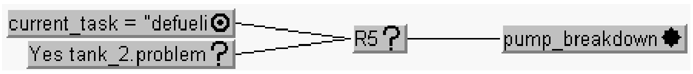
10. You can close the Object Network window and the Script Editor to clear your screen, but keep the Trace window open. Return to the Hypos selection window and click on the Pump Breakdown hypothesis check box; the GUI engine will trigger the script actions for this atom in the Rules Element, but as you might suspect it requires you to press the

Done button to validate the entry. This demonstrates the use of “deferred validation” to hold information without processing it immediately.

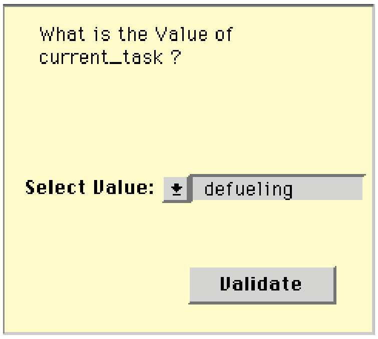


- [ ] Device Rotation
- [x] Pump Breakdown
- [ ] Execute Rotation
- [ ] Check reference
- [ ] Contact Control Center

☞ To visualize the effect of the just suggested hypothesis, we can use the Rule Network or the Agenda Monitor. Here’s what it looks like when you focus the network on the Current Rule.



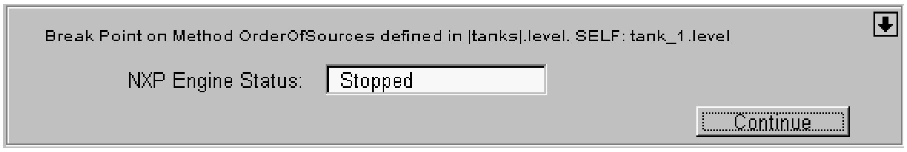
☞ After suggesting the hypothesis and triggering a knowledge processing session, the script engine immediately displays the first request for data from the inference engine in the GUI Start window. Return to the GUI Start window (only a portion is shown below) and click on the drop down menu to select the “Defueling” option. You will need to click on the Validate button to process the selection.



What is the Value of current_task ?

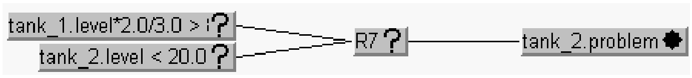
Select Value: defueling

11. Since we are interested in visualizing the effect the GUI engine has on Rules Element processing, observe the status of the inference engine after this last action. The Main Window shows the engine is "Stopped" and states that the breakpoint on the method was triggered.

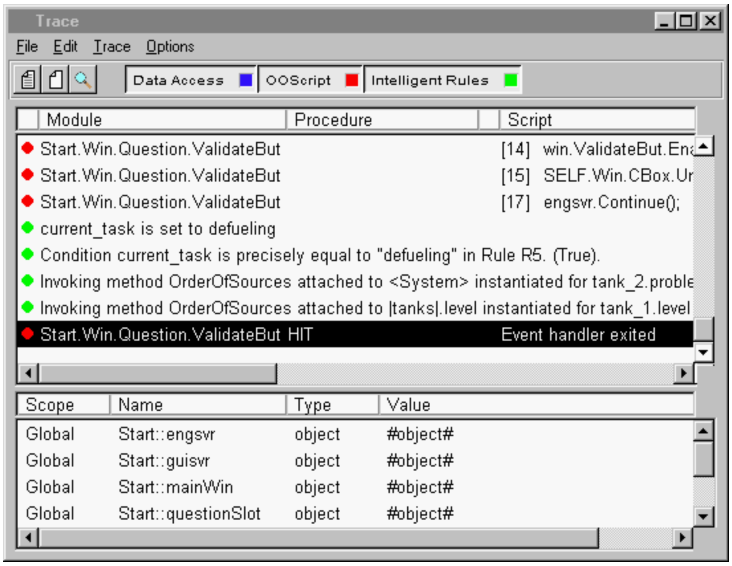

Break Point on Method OrderOfSources defined in |tanks|.level. SELF: tank_1.level

NXP Engine Status: Stopped

☞ Because the inference engine has stopped we might wish to locate the current evaluation. This time focusing on the current rule in the Rule Network does not reveal the current evaluation.



☞ As we already know, the OrderOfSources method is responsible for determining the value of `tank_1.level`. To confirm that it was inherited from the Tanks class, we can return to the Script Trace window and view the results.



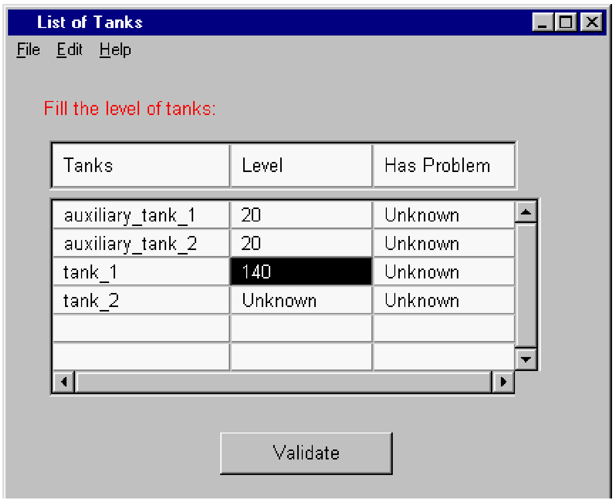
Module	Procedure	Script
Start.Win.Question.ValidateBut		[14] win.ValidateBut.Ena
Start.Win.Question.ValidateBut		[15] SELF.Win.CBox.Ur
Start.Win.Question.ValidateBut		[17] engsvr.Continue();
current_task is set to defueling		
Condition current_task is precisely equal to "defueling" in Rule R5. (True).		
Invoking method OrderOfSources attached to <System> instantiated for tank_2.proble		
Invoking method OrderOfSources attached to |tanks|.level instantiated for tank_1.level		
Start.Win.Question.ValidateBut HIT		Event handler exited

Scope	Name	Type	Value
Global	Start::engsvr	object	#object#
Global	Start::guisvr	object	#object#
Global	Start::mainWin	object	#object#
Global	Start::questionSlot	object	#object#

12. Return to the Rules Element main window and click on the Continue button. The system continues processing. This time we have a very

interesting new window that uses an input table to accept the values of the child object of class tanks. This GUI List of Tanks window is particularly useful for combining several question panels into one. To enter a value into one of the input fields, click on the field and press the Ctrl+E keys on your keyboard. Entries must be validated by pressing the Return key.

☞ Enter a value of “20” for `auxiliary_tank_1`, press Return. Enter a value of “20” for `auxiliary_tank_2`, press Return. Enter a value of “140” for `tank_1`, and press Return. Leave `tank_2` Unknown as shown below.



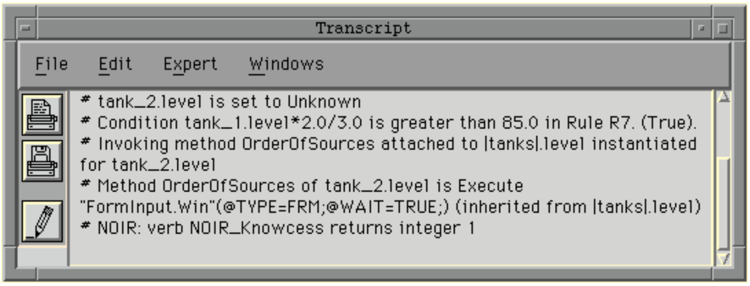
Tanks	Level	Has Problem
auxiliary_tank_1	20	Unknown
auxiliary_tank_2	20	Unknown
tank_1	140	Unknown
tank_2	Unknown	Unknown

☞ After you have entered three of the four tank values, return to the Main Window and select the Transcript button. The Transcript window will appear on your screen but must be enabled to display a detailed transcript of the processing events. Select the Pencil icon button to write enable the Transcript.

☞ Now return to the GUI List of Tanks window and position it so that the Transcript window remains visible. You are now ready to send the form-input to the system for processing. Click on the Validate button.

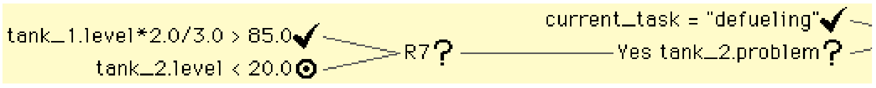
13. The first thing we will observe after sending the data to the system for processing is that the same window appears to solicit data for the intentionally withheld `tank_2.level`. Examining the previously enabled Transcript reveals the system triggered a series of events

resulting from the just processed data. Scroll down to the end of the text displayed in your Transcript window; it will contain the messages shown below if you have entered the data correctly.



```
* tank_2.level is set to Unknown
* Condition tank_1.level*2.0/3.0 is greater than 85.0 in Rule R7. (True).
* Invoking method OrderOfSources attached to |tanks|.level instantiated
for tank_2.level
* Method OrderOfSources of tank_2.level is Execute
"FormInput.Win"(@TYPE=FRM;@WAIT=TRUE;) (inherited from |tanks|.level)
* NOIR: verb NOIR_Knowcess returns integer 1
```

☞ Let's return to the Rule Network to visualize the effect of the just processed data on the system. Here's what it looks like when you focus the network on the previously displayed rule.

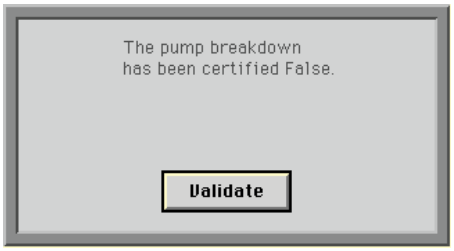


☞ Notice the target symbol has moved from the first to the second condition of the rule. Because we intentionally withheld the data for `tank_2.level` in the GUI List of Tanks window, the system was not able to complete the evaluation of this rule.

☞ The reason the same List of Tanks window appeared to request the data for `tank_2.level` demonstrates that a single GUI input window can be defined at the class level for use by multiple child objects. Examine the Transcript window and verify for yourself that `FormInput.Win` resource was inherited by `Tank_2.level` from the class `tanks`.

14. Return to the GUI List of Tanks window and enter a value of “35” for `tank_2`, press Return key (to validate the data), and press the Validate button. This time the system displays the End window with the conclusion about the hypothesis `Pump_breakdown` which was “suggested” when this session began. This last action concludes our

tour of a knowledge-based application with a graphical user interface as a front-end.


The pump breakdown
has been certified False.

This concludes our demonstration of important features offered by the Rules Element. We also investigated how the script language works across Neuron Data Elements to increase the power of your applications. The Neuron Data Elements Environment provides full intra-operability across the Elements with its own object-oriented script language, as well as C and C++ programming languages.

For additional information about the Primer components, refer to the appendices found in this manual.

Appendix

A Primer Decomposition

About this Appendix

Chapter Three, “Primer” demonstrated how the Intelligent Rules Element behaves when it processes the rules and objects of the primer knowledge-base. This appendix uses the same knowledge base to explain how the Rules Element’s unique processing behavior can more closely simulate the expert’s way of solving problems.

The rules in the following sections describe a system diagnosis problem. The diagnosis is made using data supplied by the end user. Once the diagnosis is complete and the appropriate conditions are met, the system performs some calculations useful to the end user. Please follow the discussion closely to benefit from rule and object structure diagrams of this appendix.

Rules as Building Blocks

Chapter One, “Representation” described the structure of rules in general terms. Now we will see how these IF / THEN / DO statements let you define many useful structures. The entire rule includes one or more conditions and a single hypothesis that the conditions prove. Figure A-1 shows these rule elements in a typical rule graph diagram as described in Chapter Three, “Facilities.” In the diagram the left side (with two branches) are the rule conditions and the right side (with a single branch) the hypothesis.

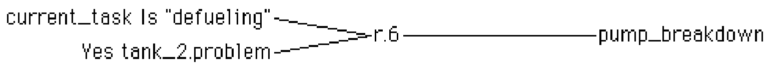


Figure A-1 One Rule Describes a Specific Situation

The rule shown in Figure A-1 comes from our knowledge base. The single rule stands on its own as a complete thought or “situation.” If we want to describe the situation represented by this single rule, we would say:

A pump breakdown exists when:

- Current task is defueling *and* a problem in tank two exists.

In this case we see that the hypothesis `pump_breakdown` is a consequence of two conditions. The hypothesis will only be true when both conditions are proven true because conditions in a single rule are always logically "added." Proving the rule's conditions enhances the concept of the rule as a situation with a single consequence. It means that the conditions you assign to a specific rule must all contribute to the same outcome when actual data is applied.

Data can come from within the knowledge base or from an outside source such as a database or end user. Data lets the Rules Element evaluate the condition and assign the condition one of three values: `TRUE`, `FALSE`, or `NOTKNOWN`. The evaluation of conditions in turn determines the value of the hypothesis which can be `TRUE`, `FALSE`, or `NOTKNOWN`. Table A-2 summarizes the evaluation relationship between a single rule's conditions and its hypothesis.

Conditions	**Hypothesis Status**
All are TRUE	TRUE
Any one is FALSE	FALSE
Any one is NOTKNOWN and none are FALSE	NOTKNOWN

Table A-2 Single Rule Evaluation

The single rule is the building-block of the knowledge base. Let's say we want to create another rule with the hypothesis used in Figure A-1 to describe another situation of a pump breakdown. In this case the hypothesis `pump_breakdown` will be a consequence of two rules, one with two conditions and the other with three conditions (and one action statement) as shown in Figure A-3.

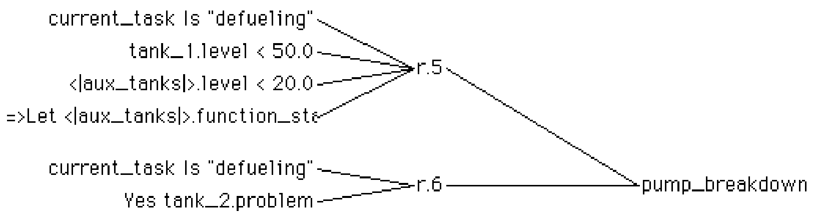

current_task Is "defueling"
tank_1.level < 50.0
<|aux_tanks|>.level < 20.0
=>Let <|aux_tanks|>.function_st
r.5
current_task Is "defueling"
Yes tank_2.problem
r.6
pump_breakdown

Figure A-3Typical And / Or Rule Diagram

The structure shown in Figure A-3 represents a typical rule graph diagram wherein multiple rules share the same hypothesis. This arrangement of rules is expandable and appears repeatedly in Rules Element applications. The first ply from the hypothesis is always an "or" decision while the second ply is an “and” decision. Putting it another way, the hypothesis `pump_breakdown` is validated by the conditions in either rule r.5 OR r.6 when at least one rules’ conditions are ALL proven true. If we want to describe the situation represented by these two rules, we would say:

A pump breakdown exists when:

- Current task is defueling, and a problem in tank two exists.

  or

- Current task is defueling, and level of tank one is less than 50, and level of any of the auxiliary tanks is less than 20.

Table A-4 summarizes the evaluation relationships between multiple rules leading to the same hypothesis.

Rules	Hypothesis Status
Any are TRUE	TRUE
All are FALSE	FALSE
Any one is NOTKNOWN and none are TRUE	NOTKNOWN

Table A-4 Multiple Rule Evaluation

The rules r.5 and r.6 thus far address the diagnostic task of our small knowledge base. Now let’s add a rule to collect data that identifies a problem in tank two. So let’s assume that a rule must be created for the following situation.

A problem in tank two exists when:

- Level in tank two is less than 20.
- Level in tank one multiplied by 2/3 is greater than 85.

Figure A-5 shows the new rule and its relationship to the original rule r.6.

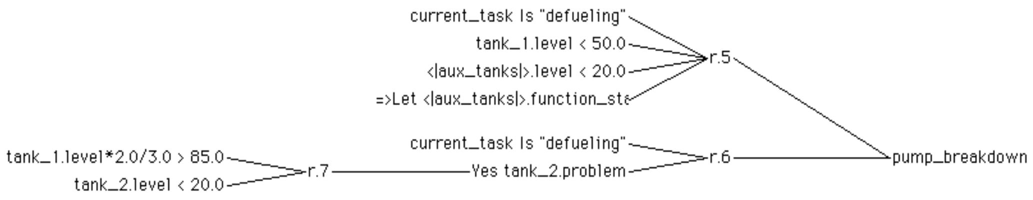

current_task Is "defueling"
tank_1.level < 50.0
<|aux_tanks|>.level < 20.0
=>Let <|aux_tanks|>.function_st
r.5
tank_1.level*2.0/3.0 > 85.0
tank_2.level < 20.0
r.7
current_task Is "defueling"
Yes tank_2.problem
r.6
pump_breakdown

Figure A-5 Rules Linked by Subgoal Hypotheses

The new rule has conditions which test diagnostic parameters. This rule’s hypothesis appears as part of the condition of rule r.6 as follows:

```
Yes    tank_2.problem       (condition in rule r.6)
```

This special condition uses the Rules Element “Yes” operator to test the value of the hypothesis `tank_2.problem` which is itself the consequence of the conditions in the new rule r.7. A hypothesis test condition translates into, “Is the value of the hypothesis, as determined by its conditions, TRUE?”

Hypothesis test conditions are termed subgoal hypotheses to distinguish them from a hypothesis that is a final outcome. Hypotheses that are themselves evaluated are considered subgoals of the system because they always contribute to the outcome of a terminal hypothesis (such as `pump_breakdown`).

Inferencing with Multiple Rules

Inferencing is the process the Rules Element uses to reach conclusions in a knowledge base. To begin inferencing on the rule base shown in Figure A-5 we might start with known data to force the evaluate of conditions, or we might suggest a hypothesis for evaluation. Either way the system will try to establish a value for the relevant hypotheses and will cease inferencing only after one or more terminal hypotheses have a value.

Suggesting a terminal hypothesis forces the system to use deductive reasoning. This means in our example, that suggesting the terminal hypothesis `pump_breakdown` would cause evaluation of the rule base to proceed in the following order: r.6, r.7, and then r.5. This type of processing is specifically known as backward chaining due to the direction of rule evaluation and the fact that the rules form a chain of reasoning. The rule diagram in Figure A-6 depicts this backward chaining path from the terminal hypothesis `pump_breakdown` .

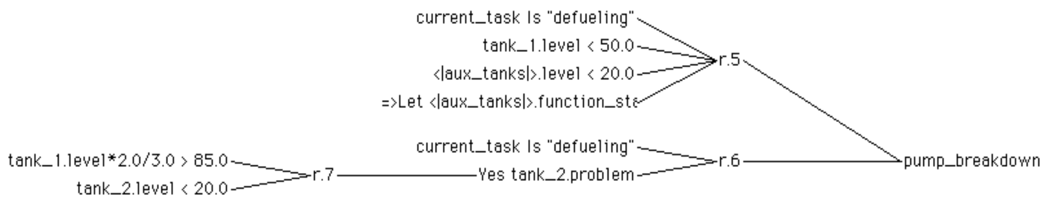

current_task Is "defueling"
tank_1.level < 50.0
<|aux_tanks|>.level < 20.0
=>Let <|aux_tanks|>.function_sta
r.5
tank_1.level*2.0/3.0 > 85.0
tank_2.level < 20.0
r.7
current_task Is "defueling"
Yes tank_2.problem
r.6
pump_breakdown

Figure A-6 Deductive Reasoning (Backward Chaining)

If the rule set has only one terminal hypothesis, as shown in Figure A-6, then inferencing can go no further once its value has been established. But what happens in the case where the knowledge base includes more than one terminal hypothesis? For example, our primer knowledge base still requires rules to handle the calculations once the diagnostic task is completed. This could involve another set of rules that seems independent from the first. So let’s assume that a rule must be created for the following situation.

Rotation can occur when:

- The current task is refueling *and* the device orientation is outward.

This situation forms the single rule shown in Figure A-7, and it shares no hypotheses with the rules shown in Figure A-6. Therefore, initiating backward chaining on this rule would seem very simple indeed.

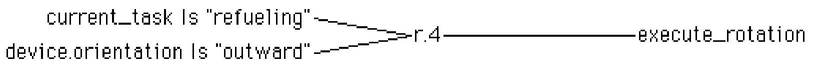


Figure A-7Terminal Hypothesis

Although the rule in Figure A-7 has a terminal hypothesis with no subgoal hypotheses, it does have something in common with the first set of rules: conditions or, to be more exact, conditions which test the same data. Notice that `current_task` is found also in rules r.5 and r.6. This relationship is called inductive because the data which the system obtains forces the evaluation of all hypotheses whose conditions can test the data.

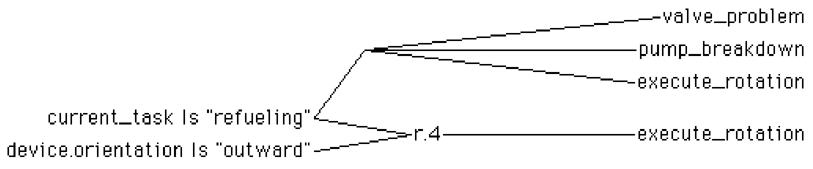


Figure A-8 Rules Linked by Data

Figure A-8 shows how the graphical interface represents the inductive reasoning path between hypotheses. In this example, the first condition of rule r.4 uses data shared by the subgoal hypotheses `valve_problem` and `pump_breakdown`. This means that suggesting the terminal hypothesis `execute_rotation` would cause evaluation of the rule base to proceed in the following order: r.4, r.6, r.7, r.5 and then r.8. This type of processing is specifically known as forward chaining due to the direction of rule evaluation (across rules). The rule diagram in Figure A-9 depicts this forward chaining path from the terminal hypothesis `execute_rotation`.

Note: In the Rules Element, data that conditions share act as gates for more efficient inductive reasoning. The “gate” permits rule evaluation to proceed only when the data makes the condition TRUE. If the data makes the condition FALSE or NOTKNOWN, the rule’s hypothesis is bypassed for evaluation.

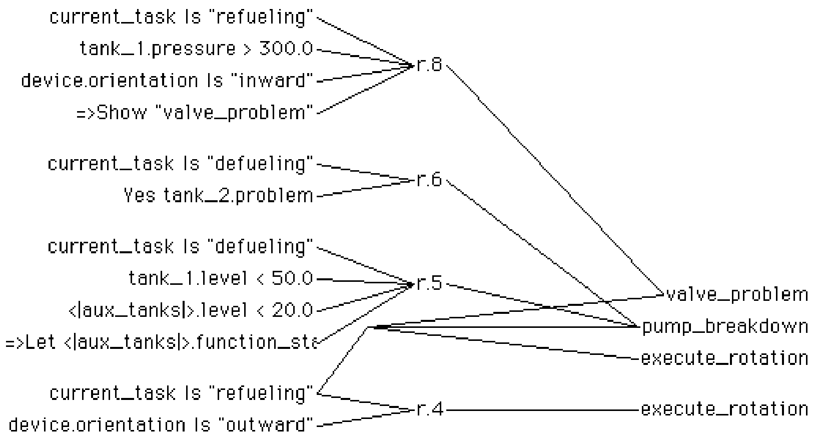

current_task Is "refueling"
tank_1.pressure > 300.0
device.orientation Is "inward"
=>Show "valve_problem"
r.8
current_task Is "defueling"
Yes tank_2.problem
r.6
current_task Is "defueling"
tank_1.level < 50.0
<|aux_tanks|>.level < 20.0
=>Let <|aux_tanks|>.function_st
r.5
current_task Is "refueling"
device.orientation Is "outward"
r.4
valve_problem
pump_breakdown
execute_rotation
execute_rotation

Figure A-9Inductive Reasoning from Conditions (Forward Chaining)

In order to finish designing the knowledge base, we have to create the rules that perform the calculations. Let’s assume this means adding two new rules r.3 and r.1 as shown in Figure A-10. Rule r.1 brings out another aspect of inductive reasoning since we see that rule actions, as well as conditions, can produce forwarding chaining of data. Figure A-10 shows how the graphical interface represents the inductive reasoning path starting from rule actions.

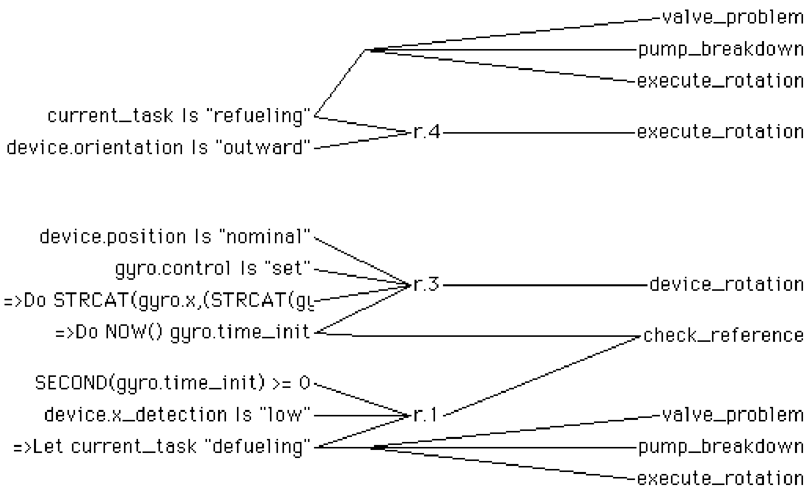

valve_problem
pump_breakdown
execute_rotation
current_task Is "refueling"
device.orientation Is "outward"
r.4
execute_rotation
device.position Is "nominal"
gyro.control Is "set"
=>Do STRCAT(gyro.x,(STRCAT(gy
=>Do NOW() gyro.time_init
r.3
device_rotation
check_reference
SECOND(gyro.time_init) >= 0
device.x_detection Is "low"
=>Let current_task "defueling"
r.1
valve_problem
pump_breakdown
execute_rotation

Figure A-10 Inductive Reasoning from Actions (Forward Chaining)

To the inference engine there is a difference between inductive reasoning from rule actions and inductive reasoning from conditions. Forwarding data from rule actions proceeds to other conditions only, while conditions forward only to other conditions through “gates.” This means that rule actions that share data with conditions can cause other hypotheses to be evaluated, but conditions that share data with actions cannot. Figure A-10 shows the slot `current_task` is found in the condition of rule `execute_rotation` and the action of rule `check_reference` (as well as rules `pump_breakdown` and `valve_problem`). Accordingly, the system will forward the slot `current_task` from rule `check_reference` to rule `execute_rotation` but not the reverse.

Inferencing Flow Control

The two methods of reasoning, inductive and deductive, described in the previous section demonstrate the versatility of Rules Element rules. Rules Element rules can operate in either mode. Whether inferencing proceeds along the rules in a backward or a forward chaining fashion depends merely on whether you start knowledge processing by suggesting a terminal hypothesis or volunteering known data.

But what would happen if you started knowledge processing with known data and an unknown hypothesis simultaneously? How would the system proceed to use the rules first, inductively or deductively? This is where the inference engine, acting as the master of the game, becomes important. The Rules Element inference engine has guidelines that establish reasoning priorities as the next paragraphs show.

Let's examine another type of inferencing mechanism that will let us connect the two sets of rules shown in Figure A-10 in the desired direction of evaluation. Recall that our knowledge base is to perform calculations only after the diagnostic is complete and the conditions of rule `execute_rotation` are satisfied. The situation could be summarized as follows.

Device rotation calculations can occur when:

- The current task is refueling *and* the device orientation is outward *and* no valve or pump problems are detected.

We need an inferencing link between the hypothesis `execute_rotation` and the hypothesis `device_rotation` that is neither forward or backward chaining. In effect the link should allow `device_rotation` to occur only after the terminal hypothesis `execute_rotation` evaluates to TRUE. This type of link is available through the Rules Element and is termed a context link. In effect the context link is an inferencing flow control mechanism because it lets you establish relationships between rules that would otherwise have no logical connection.

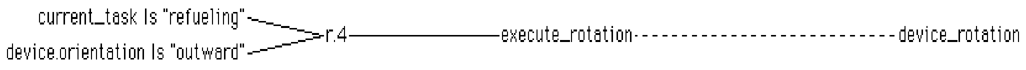


Figure A-11 Context Link Between Hypotheses

Figure A-11 shows how the graphical interface represents the context link between hypotheses. In our example, this dashed line means suggesting the terminal hypothesis `execute_rotation` will not immediately cause the evaluation of the hypothesis `device_rotation`.

The evaluation of the terminal hypothesis `device_rotation` proceeds only after the evaluation of all hypotheses related to `execute_rotation` concludes. This means that suggesting the terminal hypothesis `execute_rotation` in our now complete knowledge base would cause evaluation of the rule base to proceed in two rule groups or "knowledge islands." The rule diagram in Figure A-12 depicts these two knowledge islands connected by a context link.

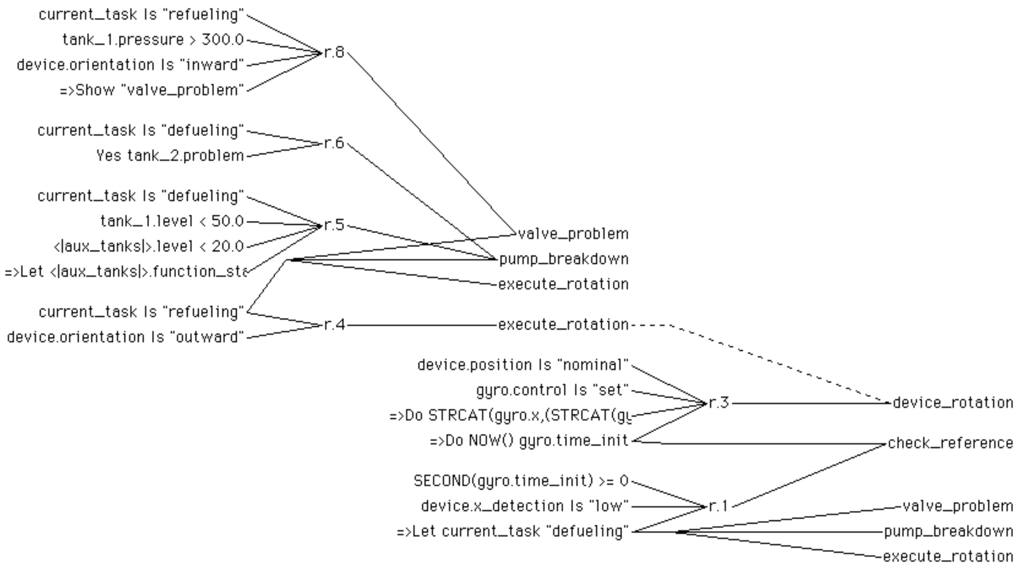

current_task Is "refueling"
tank_1.pressure > 300.0
device.orientation Is "inward"
=>Show "valve_problem"
r.8
current_task Is "defueling"
Yes tank_2.problem
r.6
current_task Is "defueling"
tank_1.level < 50.0
<|aux_tanks|>.level < 20.0
=>Let <|aux_tanks|>.function_st
r.5
valve_problem
pump_breakdown
execute_rotation
current_task Is "refueling"
device.orientation Is "outward"
r.4
execute_rotation
device.position Is "nominal"
gyro.control Is "set"
=>Do STRCAT(gyro.x,(STRCAT(gy
=>Do NOW() gyro.time_init
r.3
device_rotation
check_reference
SECOND(gyro.time_init) >= 0
device.x_detection Is "low"
=>Let current_task "defueling"
r.1
valve_problem
pump_breakdown
execute_rotation

Figure A-12 Knowledge Islands Connected by Context Link

The context link is also known as a weak link to distinguish it from the "stronger" links formed by backward and forward chaining. It is a weaker link because it imparts a lower rule evaluation priority during inferencing. Table A-13 summarizes the rule evaluation priorities each of the various inferencing mechanisms receive from the Rules Element inference engine.

Note: In actual practice the priority the inference engine assigns each mechanism is more complex. For a more complete description of these mechanisms and their priorities, refer to the Functional Description Manual.

Type of Inferencing	Evaluation Priority
Backward Chaining:	
from suggested hypos	First
from sub-goal hypos	Second
Forward Chaining:	
from conditions	Third
from actions	Fourth
Context Links	Fifth

Table A-13 Rules Element Inferencing Mechanism Priorities

Storing Data in Objects

Thus far our discussion of rule evaluation has centered on finding data to complete the condition. Once the inference engine begins evaluating a rule, the system must obtain the value of relevant data to conclude whether the rule's conditions are TRUE, FALSE, or NOTKNOWN (in the case where the specific value is "notknown"). In this discussion it has not become apparent that our data are actually objects that may belong to larger classes which may in turn store values.

The rule side of the application hides the object structure because the two integrate so easily. This is due in part to the ease with which the rule syntax handles data. To actually view the application's object structure we can use the counterpart to the rule diagrams created with the help of the Rules Element graphical interface. The following figures are examples of object structure diagrams that reveal the class / object relationships in our sample knowledge base.

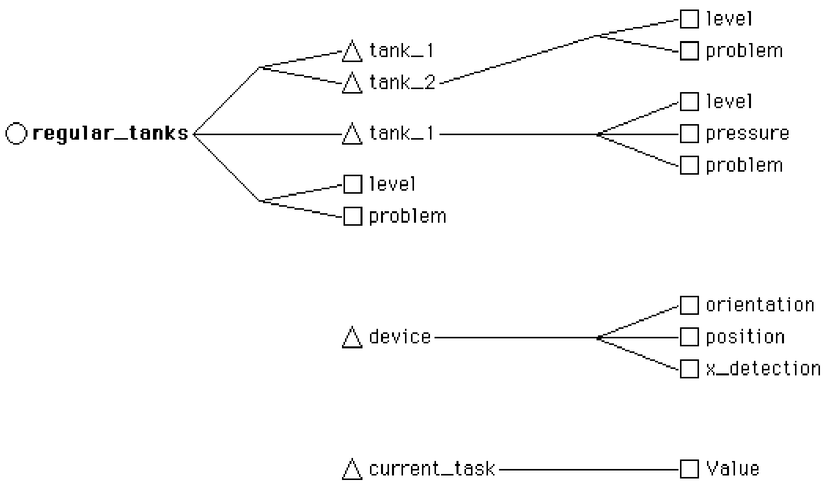


Figure A-14 Declared and Undeclared Object Structures

Figure A-14 depicts a range of declared and undeclared object structures that can be found in our rules. The structure at the bottom reveals that the slot `current_task` is actually an object (graphically represented by a triangle) and possesses the property (represented by a box) `Value`. In this case the rule used data that was not defined to be an object with a specific property, and the system automatically created an object and property to store the value. Figure A-14 also shows two declared objects that were created with specific properties attached such as `tank_1.level` or `device.orientation`, where `level` and `orientation` are the properties of `tank_1` and `device`. The structure at the top reveals another aspect of the object structure, since the object `tank_1` actually belongs to the class (represented by a circle) `regular_tanks`. In this case, `tank_1` (and `tank_2`) automatically inherit the properties level and problem from their parent class.

Figure A-15 depicts the inheritance of properties by the objects `auxiliary_tank_1` and `auxiliary_tank_2` down from two classes: `tanks` and `aux_tanks`. Notice how the classes `aux_tanks` and `regular_tanks` are actually sub-classes of the comprehensive class `tanks`.

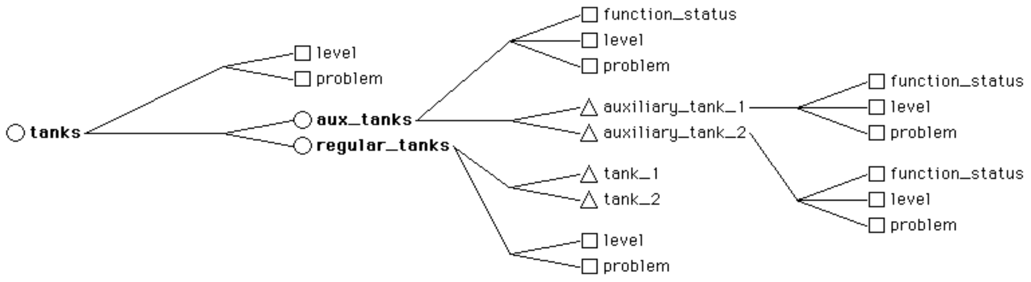


Figure A-15Inheritance of Properties from Class to Objects

Inheritance in Rules Element applications is not limited to properties. It is also possible for objects to inherit user-defined methods called meta-slots. The designer of an application can use methods when they want to attach procedural information to the object structure. For instance, Figure A-16 shows the only instance in our application where meta-slots are used.

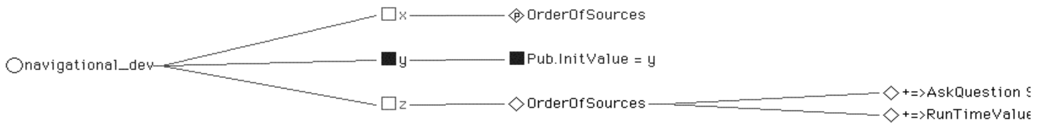


Figure A-16 Methods and Meta-Slots

Reinitiating Inferencing

To complete our understanding of the primer knowledge base let's examine the Rules Element's ability to reuse previously evaluated rules. Let's assume the diagnostic task must be performed once before the calculations and then once after the calculations. This implies reusing the same rules that already established a value for the terminal hypotheses `valve_problem` and `pump_breakdown`, but the Rules Element will in fact permit these rules to fire again if it finds reason to do so. For instance, if new data arrives, the Rules Element applies it to the rules' conditions to determine whether it changes the evaluation outcome of their hypotheses. The action statement in the check_reference rule shown in Figure A-17 demonstrates this situation by assigning the value "defueling" to current_task through the Rules Element "Let" operator. This action statement translates into, "After the conditions in the rule are all found to be TRUE, let the specified slot (current_task) equal the value given (defueling)."

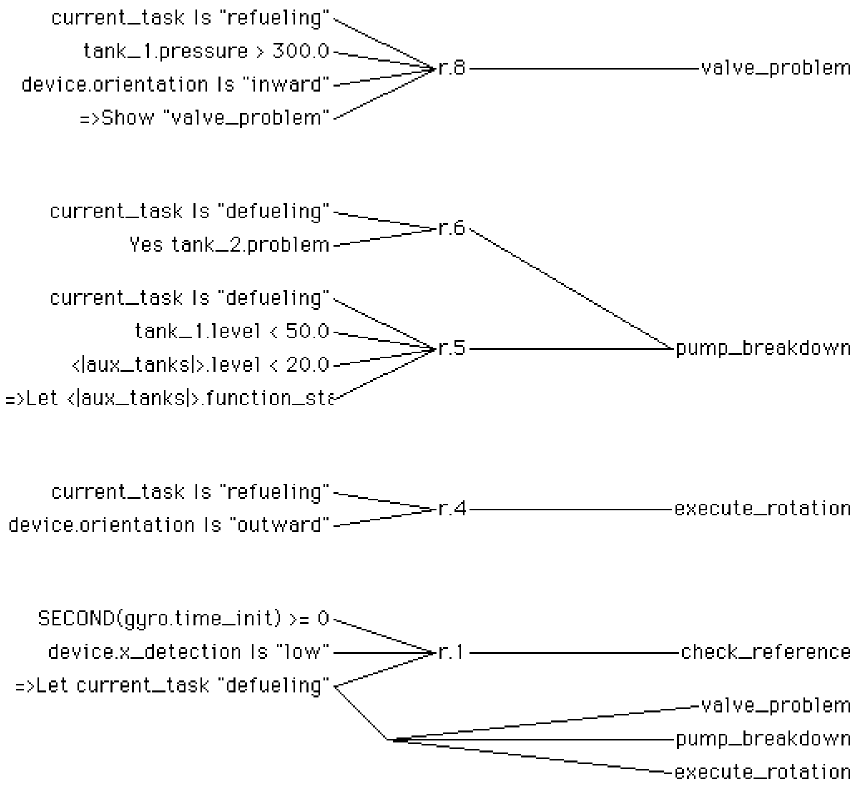

current_task Is "refueling"
tank_1.pressure > 300.0
device.orientation Is "inward"
=>Show "valve_problem"
r.8
valve_problem
current_task Is "defueling"
Yes tank_2.problem
r.6
current_task Is "defueling"
tank_1.level < 50.0
<|aux_tanks|>.level < 20.0
=>Let <|aux_tanks|>.function_sta
r.5
pump_breakdown
current_task Is "refueling"
device.orientation Is "outward"
r.4
execute_rotation
SECOND(gyro.time_init) >= 0
device.x_detection Is "low"
=>Let current_task "defueling"
r.1
check_reference
valve_problem
pump_breakdown
execute_rotation

Figure A-17Forwarding Data Can Cause Revisions

As Figure A-17 shows, the slot current_task in the check_reference rule is also shared with the rules leading to three other hypotheses. Therefore, if the action of rule r.1 fires (when the two conditions are TRUE), the Rules Element will forward the value of current_task to those rules which share that slot. In this case it is shared by rules r.8, r.6, r.5 and r.4, but notice that only the hypothesis pump_breakdown has a chance to evaluate to TRUE. The hypotheses valve_problem and execute_rotation will automatically evaluate to FALSE because their first conditions require the slot current_task to equal "refueling."

This concludes our discussion of the primer knowledge base. For more information about the order of rule evaluation in this application, refer to Chapter Two, “Inference Engine Processing.” Chapter Three, “Primer” uses the primer knowledge base to demonstrate how processing occurs in the Rules Element. Refer also to Appendix B, “Primer Text File” for a commented file listing of the knowledge base as it appears in the Rules Element’s own text file format. Appendix C, “Primer.Dat Scripts” gives a listing of the scripts created for the graphical user interface portion of the primer knowledge base.

Appendix

B Primer KB Text Format

About this Appendix

This appendix gives the listing of the three primer knowledge base files designed for use with this manual. The listing is the Intelligent Rules Element code generated by the system when a knowledge base file is saved in the Rules Element text format. The Rules Element can also save files in a compiled format that is compatible with the development platform only. The developer specifies the file format in the Save Knowledge Base dialog window.

The text format is compatible with the Rules Element running on all platforms. Saving a knowledge base in this format also lets the developer familiar with the format make modifications directly to the file using any text editor. For details about the text format itself, refer to Appendix E, "Text KB Syntax" in the User's Guide.

Data Type Listing

The following definitions of data types used in the primer knowledge base appear at the beginning of the text format file. Data types are defined in the Object Editor window for individual properties of objects or classes.

From Primer.tkb

```
(@VERSION=040)
(@COMMENTS="@(#)primer.tkb6.6")
(@PROPERTY=control          @TYPE=String;)
(@PROPERTY=definition       @TYPE=String;)
(@PROPERTY=function_status  @TYPE=String;)
(@PROPERTY=hypo             @TYPE=Boolean;)
(@PROPERTY=level            @TYPE=Float;)
(@PROPERTY=Name             @TYPE=String;)
(@PROPERTY=orientation      @TYPE=String;)
(@PROPERTY=position         @TYPE=String;)
(@PROPERTY=pressure         @TYPE=Float;)
(@PROPERTY=problem          @TYPE=Boolean;)
(@PROPERTY=prompt           @TYPE=String;)
(@PROPERTY=time_init        @TYPE=Date;)
(@PROPERTY=x                @TYPE=String;)
(@PROPERTY=x_detection      @TYPE=String;)
(@PROPERTY=y                @TYPE=String;)
(@PROPERTY=z                @TYPE=String;)
```

Rule Listing

The following rule definitions show the eight rules that make up the entire primer knowledge base. Rules are comprised of left-hand side (LHS) conditions, a hypothesis (HYPO), and an optional right-hand side (RHS) actions list that may comprise else actions (ELS). Rules are defined in the Rule Editor window.

From Primer.tkb

```
(@RULE=R1
       (@LHS=
              (SendMessage("GetGyroTimeInit")(@TO=gyro;\
                            @ARG1=InitTime.Value;))
              (>=    (InitTime)(0))
              (=     (device.x_detection)("low"))
       )
       (@HYPO=check_reference)
       (@RHS=
              (Assign("defueling")(current_task))
       )
       (@EHS=
              (Assign("refueling")(current_task))
       )
)

(@RULE=R2
       (@LHS=
              (Yes   (valve_problem))
              (>     (time_elapsed_since_problem)(45.0))
       )
       (@HYPO=contact_control_center)
)

(@RULE=R3
       (@LHS=
              (=     (device.position)("nominal"))
              (=     (gyro.control)("set"))
       )
       (@HYPO=device_rotation)
       (@RHS=
              (SendMessage("GetGyroXYZ")
(@TO=gyro;@ARG1=xyz.Value;))
              (Assign(xyz)   (gyro.definition))
              (Assign(NOW()) (theTime))
              (SendMessage("SetGyroTimeInit")(@TO=gyro;\
                            @ARG1=theTime.Value;))
       )
)

(@RULE=R4
       (@LHS=
              (=     (current_task)("refueling"))
              (=     (device.orientation)("outward"))
       )
```

```
        (@HYPO=execute_rotation)
)

(@RULE=R5
        @INFCAT=2;
        (@LHS=
                (=      (current_task)("defueling"))
                (Yes    (tank_2.problem))
        )
        (@HYPO=pump_breakdown)
)

(@RULE=R6
        (@LHS=
                (=      (current_task)("defueling"))
                (<      (tank_1.level)(50.0))
                (SendMessage("CheckAuxTanksLevelLessThan20")
                                (@TO=<|aux_tanks|>;\
                                @ARG1=auxTanksLevel.Value;))
                (Yes    (auxTanksLevel))
        )
        (@HYPO=pump_breakdown)
        (@RHS=
                (Assign("on") (<|aux_tanks|>.function_status))
        )
)

(@RULE=R7
        (@LHS=
                (>      (tank_1.level*2.0/3.0)(85.0))
                (<      (tank_2.level)(20.0))
        )
        (@HYPO=tank_2.problem)
)

(@RULE=R8
        (@LHS=
                (=      (current_task)("refueling"))
                (>      (tank_1.pressure)(300.0))
                (=      (device.orientation)("inward"))
        )
        (@HYPO=valve_problem)
        (@RHS=
                (Show  ("valve_pb.nbm"))
        )
)
```

From Primer1.tkb

```
(@RULE=initialization
        (@LHS=
                (Yes    (start))
        )
        (@HYPO=initialization)
        (@RHS=
                (SendMessage("Init")(@TO=<|tanks|>;))
                (Strategy(@VALIDUSER=ACCEPT;))
        )
)
```

Class and Object Listing

The following class and object definitions show the three classes and various objects of the primer knowledge base. Classes have properties which their component objects automatically inherit. Objects can have unique properties and may or may not belong to a particular class. Classes and Objects are defined in their respective editor windows.

Note: Hypotheses are automatically compiled by the system as objects with the default property Value.

From Primer.tkb

```
(@CLASS=aux_tanks
        (@PUBLICPROPS=
                function_status
                level
                Name
                problem
        )
)

(@CLASS=navigational_devices
        (@PUBLICPROPS=
                x
                y
                z
        )
)

(@CLASS=regular_tanks
        (@PUBLICPROPS=
                level
                Name
                problem
        )
)

(@CLASS=tanks
        (@SUBCLASSES=
                aux_tanks
                regular_tanks
        )
        (@PUBLICPROPS=
                level
                Name
                problem
        )
)
```

```
(@OBJECT=auxiliary_tank_1
       (@CLASSES=
              aux_tanks
              tanks
       )
       (@PUBLICPROPS=
              function_status
              level
              Name
              problem
       )
)

(@OBJECT=auxiliary_tank_2
       (@CLASSES=
              tanks
              aux_tanks
       )
       (@PUBLICPROPS=
              function_status
              level
              Name
              problem
       )
)

(@OBJECT=auxTanksLevel
       (@PUBLICPROPS=
              Value  @TYPE=Boolean;
       )
)

(@OBJECT=check_reference
       (@PUBLICPROPS=
              Value  @TYPE=Boolean;
       )
)

(@OBJECT=contact_control_center
       (@PUBLICPROPS=
              Value  @TYPE=Boolean;
       )
)

(@OBJECT=current_task
       (@PUBLICPROPS=
              prompt
              Value  @TYPE=String;
       )
)

(@OBJECT=device
       (@PUBLICPROPS=
              orientation
              position
              x_detection
       )
)
```

```
(@OBJECT=device_rotation
       (@PUBLICPROPS=
              Value  @TYPE=Boolean;
       )
)

(@OBJECT=execute_rotation
       (@PUBLICPROPS=
              Value  @TYPE=Boolean;
       )
)

(@OBJECT=gyro
       (@PUBLICPROPS=
              control
              definition
       )
       (@PRIVATEPROPS=
              time_init
              x
              y
              z
       )
)

(@OBJECT=InitTime
       (@PUBLICPROPS=
              Value  @TYPE=Float;
       )
)

(@OBJECT=pump_breakdown
       (@PUBLICPROPS=
              Value  @TYPE=Boolean;
       )
)

(@OBJECT=tank_1
       (@CLASSES=
              tanks
              regular_tanks
       )
       (@PUBLICPROPS=
              level
              Name
              pressure
              problem
       )
)

(@OBJECT=tank_2
       (@CLASSES=
              tanks
              regular_tanks
       )
       (@PUBLICPROPS=
              level
```

```
                Name
                problem
        )
)

(@OBJECT=theTime
        (@PUBLICPROPS=
                Value  @TYPE=Date;
        )
)

(@OBJECT=time_elapsed_since_problem
        (@PUBLICPROPS=
                Value  @TYPE=Float;
        )
)

(@OBJECT=valve_problem
        (@PUBLICPROPS=
                Value  @TYPE=Boolean;
        )
)

(@OBJECT=xyz
        (@PUBLICPROPS=
                Value  @TYPE=String;
        )
)
```

From Primer1.tkb

```
(@OBJECT=initialization
        (@PUBLICPROPS=
                Value  @TYPE=Boolean;
        )
)

(@OBJECT=start
        (@PUBLICPROPS=
                Value  @TYPE=Boolean;
        )
)
```

Meta-Slot Listing

The following listing shows the slots of the primer knowledge base which have meta-slots defined. In this case, the meta-slots include initial values (both inheritable and private values), a question window to display, context links, a data entry format, and a data validation function. Meta-slots for particular slots are defined in the Meta-Slot Editor window.

From Primer.tkb

```
(@META=time_init
       @FORMAT="hh:mm:ss";
)

(@META=|aux_tanks|.level
       @COMMENTS="private since maybe needs to be deduced from other
values and this technique might change";
)

(@META=|navigational_devices|.y
       (@INITVAL="y4")
)

(@META=|tanks|.level
       @COMMENTS="private but one of its subclass (regular_tanks)
has this slot public";
)

(@META=auxTanksLevel.Value
       (@INITVAL=FALSE)
)

(@META=execute_rotation.Value
       (@CONTEXTS=
              device_rotation
       )
)

(@META=gyro.x
       @COMMENTS="private since units might change";
       (@INITVAL="0")
)

(@META=gyro.y
       @COMMENTS="private since units might change";
       (@INITVAL="0")
)

(@META=gyro.z
       @COMMENTS="private since units might change";
       (@INITVAL="0")
)
```

From Primer1.tkb

```
(@META=|regular_tanks|.level
       @FUNC=(SELF.level>0 AND SELF.level<2000);
       @HELP="The level of tanks should be greater than 0 and less
than 2000";
)

(@META=initialization.Value
       (@CONTEXTS=
              pump_breakdown
       )
)
```

From Primer2.tkb

```
(@META=auxiliary_tank_1.Name
       (@PRIVINITVAL="auxiliary tank 1")
)

(@META=auxiliary_tank_2.Name
       (@PRIVINITVAL="auxiliary tank 2")
)

(@META=current_task.prompt
       (@PRIVINITVAL="What is the value of current task
(defueling/refueling)?")
)

(@META=current_task.Value
       @QUESTWIN="Start.Win";
)

(@META=device.orientation
       @QUESTWIN="Start.Win";
)

(@META=tank_1.Name
       (@PRIVINITVAL="tank 1")
)
(@META=tank_2.Name
       (@PRIVINITVAL="tank 2")
)
```

Method Listing

The following listing shows the slots of the primer knowledge base which have methods defined. In this case, the methods are Order of Sources actions, If Change actions, Initialization actions, and Hypothesis Suggest actions. Methods for particular slots, objects, or classes are defined in the Method Editor window.

From Primer.tkb

```
(@METHOD=CheckAuxTankLevelLessThan20
       (@ATOMID=aux_tanks;@TYPE=CLASS;)
       (@ARG1=_result;@NATURE=SlotRef;@TYPE=Boolean;)
       (@FLAGS=PUBLIC;)
       (@LHS=
              (<     (SELF.level)(20))
       )
       (@RHS=
              (Assign(TRUE) (_result))
       )
)
(@METHOD=GetGyroTimeInit
       (@ATOMID=gyro;@TYPE=OBJECT;)
       (@ARG1=_resultInSec;@NATURE=SlotRef;@TYPE=Float;@DEFVAL=0;)
```

```
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Assign(SECOND(SELF.time_init))(_resultInSec))
        )
)
(@METHOD=GetGyroXYZ
        (@ATOMID=gyro;@TYPE=OBJECT;)

(@ARG1=_resultCombo;@NATURE=SlotRef;@TYPE=String;@DEFVAL="";)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Assign(STRCAT(SELF.x,(STRCAT(SELF.y,SELF.z))))
(_resultCombo))
        )
)
(@METHOD=OrderOfSources
        (@ATOMID=gyro.time_init;@TYPE=SLOT;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Assign(NOW())(SELF.time_init))
        )
)
(@METHOD=OrderOfSources
        (@ATOMID=navigational_devices.z;@TYPE=SLOT;)
        (@FLAGS=PUBLIC;)
        @COMMENTS="comments...";
        @WHY="why...";
        (@RHS=
                (AskQuestion(SELF.z)(NOTKNOWN))
                (RunTimeValue("FALSE"))
        )
)
(@METHOD=OrderOfSources
        (@ATOMID=navigational_devices.x;@TYPE=SLOT;)
        (@FLAGS=PRIVATE;)
        (@RHS=
                (Assign("x1") (SELF.x))
        )
)
(@METHOD=SetGyroTimeInit
        (@ATOMID=gyro;@TYPE=OBJECT;)
        (@ARG1=_timeInit;@NATURE=Slot;@TYPE=Date;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Assign(_timeInit)(SELF.time_init))
        )
)
(@METHOD=SetGyroXYZ
        (@ATOMID=gyro;@TYPE=OBJECT;)
        (@ARG1=_x;@NATURE=Slot;@TYPE=String;@DEFVAL="";)
        (@ARG2=_y;@NATURE=Slot;@TYPE=String;@DEFVAL="";)
        (@ARG3=_z;@NATURE=Slot;@TYPE=String;@DEFVAL="";)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Assign(_x)   (SELF.x))
                (Assign(_y)   (SELF.y))
                (Assign(_z)   (SELF.z))
        )
)
```

From Primer1.tkb

```
(@METHOD=Init
        (@ATOMID=tanks;@TYPE=CLASS;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Execute("AtomNameValue")
(@ATOMID=SELF;@STRING="@RETURN=@self.Name,\
@NAMES,@STRAT=SET";))
        )
)
```

From Primer2.tkb

```
(@METHOD=IfChange
        (@ATOMID=pump_breakdown;@TYPE=SLOT;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Execute("FormInput")(@WAIT=TRUE;@STRING="End";))
        )
)
(@METHOD=Init
        (@ATOMID=tanks;@TYPE=CLASS;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Execute("AtomNameValue")(@WAIT=TRUE;\
@ATOMID=SELF;@STRING="@RETURN=@self.Name,\
@NAMES,@STRAT=SET";))
        )
)
(@METHOD=OrderOfSources
        (@ATOMID=pump_breakdown;@TYPE=SLOT;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (SendMessage("Init")(@TO=<|tanks|>;))
                (Strategy(@VALIDUSER=ACCEPT;))
                (Backward(TRUE))
        )
)
(@METHOD=OrderOfSources
        (@ATOMID=pump_breakdown;@TYPE=OBJECT;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (SendMessage("Init")(@TO=<|tanks|>;))
        )
)
(@METHOD=OrderOfSources
        (@ATOMID=tanks.level;@TYPE=SLOT;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Execute("FormInput")(@WAIT=TRUE;@STRING="Form";))
        )
)
(@METHOD=Suggest
        (@ATOMID=contact_control_center;@TYPE=OBJECT;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Execute("ControlSession")(@WAIT=TRUE;\
@ATOMID=SELF;@STRING="@SUGGEST";))
        )
)
(@METHOD=Suggest
        (@ATOMID=check_reference;@TYPE=OBJECT;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Execute("ControlSession")(@WAIT=TRUE;\
@ATOMID=SELF;@STRING="@SUGGEST";))
        )
)
(@METHOD=Suggest
        (@ATOMID=pump_breakdown;@TYPE=OBJECT;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Execute("ControlSession")(@WAIT=TRUE;\
@ATOMID=SELF;@STRING="@SUGGEST";))
        )
)
(@METHOD=Suggest
        (@ATOMID=device_rotation;@TYPE=OBJECT;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Execute("ControlSession")(@WAIT=TRUE;\
@ATOMID=SELF;@STRING="@SUGGEST";))
        )
)
(@METHOD=Suggest
        (@ATOMID=execute_rotation;@TYPE=OBJECT;)
        (@FLAGS=PUBLIC;)
        (@RHS=
                (Execute("ControlSession")(@WAIT=TRUE;\
@ATOMID=SELF;@STRING="@SUGGEST";))
        )
)
```

Strategy Listing

The final listing of the text format identifies the global inferencing and inheritance strategies specified for the primer knowledge base. In this case the strategies are the default ones provided with the system. It is possible to modify and save global strategies in the Rule Editor, Meta-Slot Editor, and Strategy dialog window.

From Primer.tkb

```
(@GLOBALS=
        @INHVALUP=FALSE;
        @INHVALDOWN=TRUE;
        @INHOBJUP=FALSE;
        @INHOBJDOWN=FALSE;
```

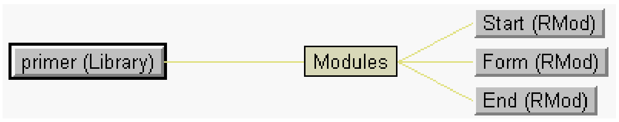
```
              @INHCLASSUP=FALSE;
              @INHCLASSDOWN=TRUE;
              @INHBREADTH=TRUE;
              @INHPARENT=FALSE;
              @PWTRUE=TRUE;
              @PWFALSE=FALSE;
              @PWNOTKNOWN=FALSE;
              @EXHBWRD=TRUE;
              @PTGATES=TRUE;
              @PFACTIONS=TRUE;
              @SOURCESON=TRUE;
              @CACTIONSON=TRUE;
              @VALIDUSER=FALSE;
              @VALIDENGINE=FALSE;
              @PFEACTIONS=FALSE;
              @PFMACTIONS=GLOBAL;
              @PFMEACTIONS=FALSE;
       )
```

Appendix C

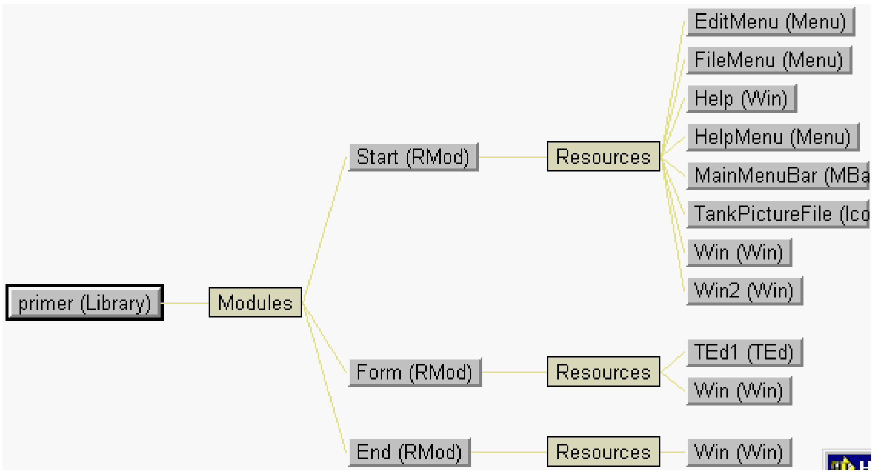
Primer.dat Scripts

This appendix supplements the last session described in Chapter Four that let you examine graphical user interface resources created with the Open Editor component. Each window of the `primer.dat` file that you loaded in the Resource Browser before conducting that primer session appears below with a description of various graphical elements used and the scripts attached through the Script Editor.

After loading the `primer.dat` file in the Resource Browser (as instructed in Session Three), the following diagram appears.



With righthand expansion of each module node in the resource browser network, we see the individual components that the modules contain:



Application Script

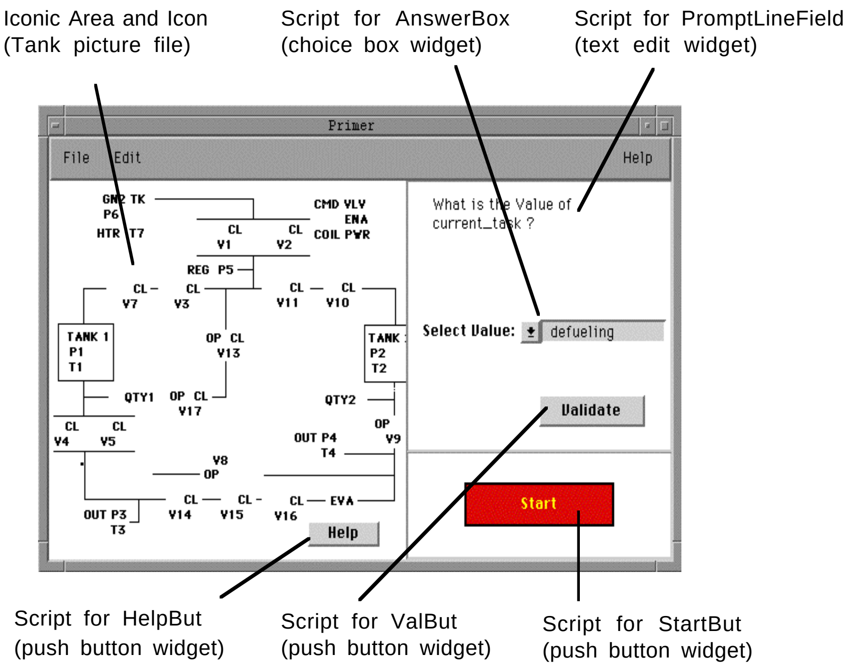
A scripted application is initiated with a startup script that appears in the main module of the application library. The following shows how the primer application is initialized when starting with the Application Script:

```
// Startup Module. It can be used either within the
// development environment or Runtime. It defines which are the
// files to be loaded at  startup and which windows to be opened.

// Global variables to keep ND servers, main window,
// question slot references
global object guisvr;
global object rulesvr;
global object coresvr;
global object engsvr;
global object questionSlot;
global object mainWin;

proc AppStartup
       //Get the Neuron Data servers
       coresvr := getserver(\"ND.Core\");
       rulesvr := getserver(\"ND.Nx\");
       guisvr := getserver(\"ND.Gui\");
       KBsvr := rulesvr.KBs;
       engsvr := rulesvr.Engine;

       // Set the Rules Handlers
       engsvr.SetLocalHandler(engsvr.PROC_QUESTION,
              \"Start::NXPQuestionProc\");
    engsvr.SetLocalExecuteHandler(\"FormInput\",
              \"Start::NXPFormInputProc\");

       // Load knowledge bases
       KBsvr.Load(\"primer.tkb\");
       KBsvr.Load(\"primer2.tkb\");

       // Display the main window
       mainWin := guisvr.Windows.Load (\"Start\",\"Win\");
       mainWin.Init();
       mainWin.ValidateBut.Enabled =0;
       mainWin.CBox.Enabled = 0;
       mainWin.Show();

end proc


// Question handler for the application
// integer proc NXPQuestionProc (object atom, string qstr)
// this question handler is only for the atom current_task.Value
       if (atom.Name != \"current_task.Value\")
              return 0;
       // Non modal question: the inference engine should be stopped
       engsvr.Stop();
       theCBox := mainWin.CBox;
       // Clear up the atom choices of previous questions
       num = theCBox.ItemCount;
```

```
        for (i=0; i<num; i++) {
                theCBox.Item(0).Dispose();
        }
        // Set the new list of atom choices for the current question
        num = atom.ChoiceCount;
        for (i=0;i<num; i++) {
                theCBox.AddItem(i);
                theCBox.Item(i).Label = atom.Choice(i);
        }
        // Enable the controls of the question panel
        theCBox.Enabled = 1;
        mainWin.PromptLineField.String = qstr;
        mainWin.ValidateBut.Enabled = 1;
        questionSlot := atom;
        // Return the hand to the inference engine to
        // suspend the inference process
        return 1;
end proc


// Executes Handlers for the application
integer proc NXPFormInputProc (string execStr, integer nAtoms,
        object atom[])
        // Interrupt inference engine
        engsvr.Stop();
        win := guisvr.Windows.Load(execStr, \"Win\");
        win.Init();
        // The End Execute is a modal Window while
        // the FormInput one is non modal
        if (execStr == \"End\")
                theStr = win.ModalProcess();
        else
                win.Show();
        // Return the hand to the inference engine
        // to suspend the inference process
        return 1;
end proc
```

Start.Win

Lets begin by examining the `Start.Win` window itself. This window is used for several different tasks and serves as the application interface “main window”


Iconic Area and Icon (Tank picture file)
Script for AnswerBox (choice box widget)
Script for PromptLineField (text edit widget)
Primer
File
Edit
Help
What is the Value of current_task ?
Select Value:
defueling
Validate
Start
Help
TANK 1
TANK 2
Script for HelpBut (push button widget)
Script for ValBut (push button widget)
Script for StartBut (push button widget)

`Start.Win` has the following script definitions:

Script for Start.Window: Just before window appears restart session.

```
// When the window is opened the session
// is restarted to get ready for a new session.

on event WIN_OPENED
        mbar := SELF.MenuBar;
        mbar.Item(0).SubMenu.Item(0).Enabled = 0;
        mbar.Item(0).SubMenu.Item(2).Enabled = 0;
        engsvr.Restart();
end event
```

Script for File menu: Exits application when Quit menu item is selected.

```
// Quit menu option of File menu has been selected.

on event WIN_MITEMSELECTED itemid 105
//      NOIR_Exit();\
```

Script for File menu: Closes window when Close menu item is selected.

```
// Quit menu option of File menu has been selected.

on event WIN_MITEMSELECTED itemid 102
       SELF.Terminate();
end event
```

Script for HelpBut: Open window Start.Help (also in the “Start” module).

```
// Open a window (ModuleName.WindowName) on click.

on event TBUT_HIT
       theWin := guisvr.Windows.Load(\"Start\", \"Help\");
       theWin.Init();
       theWin.Show();
end event
```

Script for StartBut: Open window Start.Win2 to receive the suggested hypothesis.

```
// Start the application by offering the
// user with a set of potential faults.

on event TBUT_HIT
       rulesvr.Engine.Restart();
       theWin := guisvr.Windows.Load(\"Start\", \"Win2\");
       theWin.Init();
       theWin.Show();
end event
```

Script for ValidateBut: Process window as form and continue session when pushed.

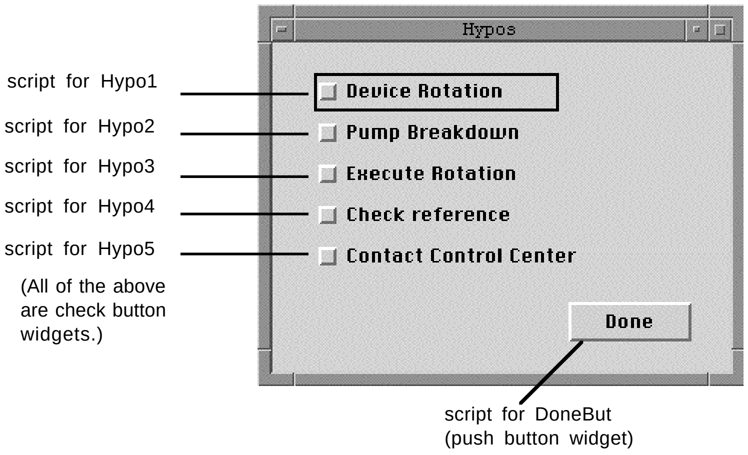
```
// Continue the session when the user
// has answered the question.

on event TBUT_HIT
       // Clean-up the Question panel
       win := SELF.Win;
       theCBox := win.CBox;
       theCBox.Enabled = 0;
       theTed := win.PromptLineField;
       theTed.String = \"\";
       theTed.Enabled = 0;
       win.CBox.Enabled = 0;
       win.ValidateBut.Enabled = 0;
       SELF.Win.CBox.Unselect();
       // Resume the pending question
       engsvr.Continue();

end event
```

Script for AnswerBox: Use slot of question, and volunteer the slot with choice box selected choice.

```
// When IRE needs to ask a question the choice box is
// initialized with the current slot value options.
// When the user selects an option, the current slot is set.
// The choice box is disabled when the question ends since
// the question panel is part of the application main window.

on event CBOX_ITEMSELECTED
       questionSlot = SELF.ChosenItem.Label;
end event
```

Start.Win2

Now let's examine the `Start.Win2` window (appears in the "Start" module displayed in the Resource Browser) that lets you place the `pump_breakdown` hypothesis on the agenda:



`Start.Win2` has the following script definition:

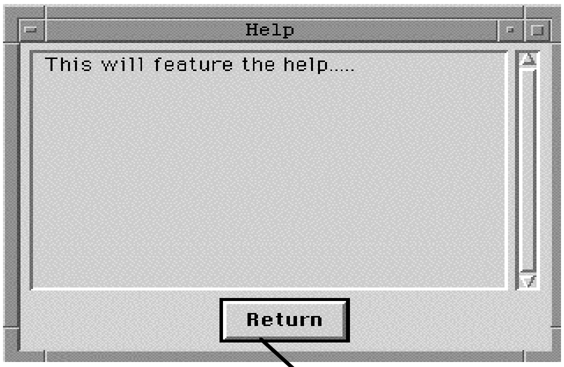
Script for DoneBut: process window as form, close the window and start session.

```
// When the user clicks on this button, the selected check
// buttons suggest the corresponding hypothesis.
// The window is closed and the KNOWCESS is then processed.
// WARNING: in this primer only one hypothesis is selectable

on event TBUT_HIT

       win := SELF.Win;
       objsvr := rulesvr.Objects;
       // Device rotation, Execute Rotation, Check Reference,
       // Contact Control Center hypotheses
       if (win.Hypo1.Selected == 1 || win.Hypo3.Selected == 1 ||
              win.Hypo4.Selected == 1 || win.Hypo5.Selected == 1 )
{
       guisvr.AlertDialogs.ShowInfo(\"Only Pump Breakdown is
supported\");
```

```
        }
        // pump_breakdown hypothesis
        if (win.Hypo2.Selected == 1) {
                objsvr.pump_breakdown.Value.Suggest();
                rulesvr.Engine.Start();
                win.Terminate();
        }

end event
```

Start.Help

The "Start" module also includes the Start.Help window attached to the "Help" button in `Start.Win` (see Script for `HelpBut` above). This help window is used to demonstrate a type of end user help:



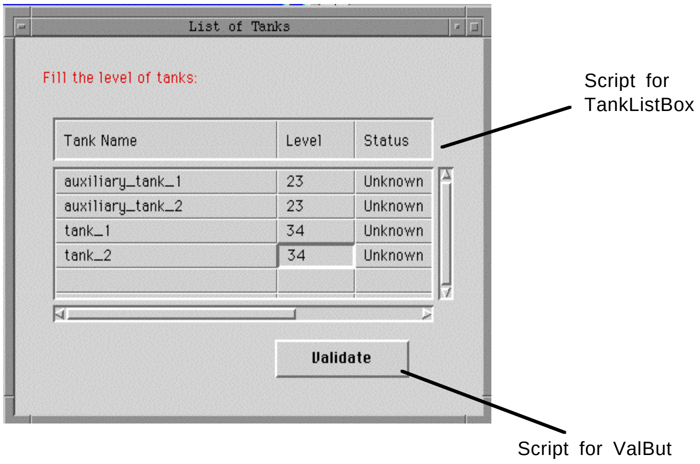
`Start.Help` has a single script definition:

Script for OkBut: Close window when done.

```
// Close the window when the user clicks on the button

on event TBUT_HIT
        SELF.Win.Terminate();
end event
```

FormInput.Win

The “FormInput” module contains a single window labeled FormInput.Win.



Fill the level of tanks:

Tank Name	Level	Status
auxiliary_tank_1	23	Unknown
auxiliary_tank_2	23	Unknown
tank_1	34	Unknown
tank_2	34	Unknown

`FormInput.Win` has the following script definitions:

When opened:

```
on event WIN_OPENED
       mbar := SELF.MenuBar;
       mbar.Item(0).SubMenu.Item(0).Enabled = 0;
       mbar.Item(0).SubMenu.Item(2).Enabled = 0;
end event
```

Script for TankListBox: Initialize with `tank_XX.level` value and volunteer the value back to the Rules Element.

```
// This is an example of an input table.
// You should first link it to a class
// then initialize the columns.
use Start; // to use the Start module global variables

on event WGTSINITIALIZED
       tanks := rulesvr.Classes.tanks;
       rProps := rulesvr.Properties;
       // Use a table data source to link the listbox
       // to the class Tanks
       ds := rulesvr.NxTableDataSources.Create();
       ds.RowColumnCount(0,3);// set the size of the data source
       ds.Atom = tanks;
       ds.RegisterView( SELF);
       //set the column mapping  with field and column labels
       ds.ColumnProperty(0) = rProps.Name;
       ds.Columns(0).Title = \"Tanks\";
       ds.ColumnProperty(1) = rProps.level;
```

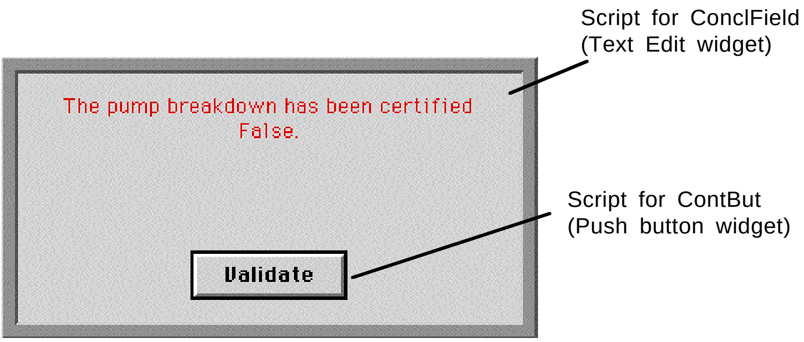
```
        ds.Columns(1).Title = \"Level\";
        ds.ColumnProperty(2)= rProps.problem;
        ds.Columns(2).Title = \"Has Problem\";
        // set the view non editable for columns 0 and 2
        ds.ViewOption(SELF, \"noeditcols\") = \"[0...0][2...2]\";
end event
```

Script for ValBut: Process window and Continue session when pushed.

```
// When the user clicks on the button
// the information entered in the table
// are processed
use Start;
// to access Start module global variables

on event TBUT_HIT
        engsvr.Continue();
end event
```

End.Win

Now let's examine the `End.Win` window contained in the "End" module. This window is used to conclude the session and display the user the results:



`End.Win` has the following script definitions:

Script for ConclusionField: Use specified value: "The pump_breakdown has been certified: @V(pump_breakdown.value)."

```
// Initialize the text output field with
// the results of the session, i.e. pump breakdown
use Start;

// to access the Start module global variables

on event WGTSINITIALIZED
        theSlot= rulesvr.Objects.pump_breakdown.Value;
        // setting the conclusion into the ConclusionField
        if (theSlot == 0)
                theString = \"False\";
```

```
       else
              theString = \"True\";
       SELF.String = \"The pump breakdown has been certified \" +
theString;
end event
```

Script for ContinueBut: Close window and Continue session when pushed.

```
// The IRE engine is suspended at that moment
// and requires a CONTINUE to end the session.
use Start;

on event TBUT_HIT
  engsvr.Continue();
  SELF.Win.ModalReturn(\"Done\");
end event
```

Index

Symbols

- 1-66
- @V 1-70
- | | 1-73

A

- actions
  - forward action-effects 2-21
  - globally disabling and enabling 2-55
  - If Change 1-26
  - interpretations 1-68
  - locally disabling and enabling 2-57
  - methods 1-18, 1-20
  - Order of Sources 1-23
  - rules 1-32
- adaptability 2-1
- addressee 2-10
- agenda
  - backward chaining priority 2-14
  - contexts 2-25
  - gates 2-19
  - hypothesis forward priority 2-17
  - purpose 2-1
  - suggest priority 2-16
  - volunteering data 2-23
- agenda search mechanisms
  - disabling 2-10
- application programming interface 2-65
- arguments
  - methods 1-22
- Assign operator 2-22, 2-58
- automatic goal generation (see gates)

B

- backward chaining 2-14
  - example A-4
  - interpretation 2-28
  - locally disabling 2-56
  - pattern matching 2-33
- best first search 1-56
- boolean 1-6
- breath-first 1-54

C

- children objects 1-4
- class hierarchy 1-4
- class selectivity 2-40
- classes
  - definition 1-3
  - interpreting 1-66
- class-first 1-54
- closed world assumption 2-4
- conditions
  - evaluation of A-2
  - forwarding data A-7
  - interpretations 1-66, 1-69
  - methods 1-21
  - pattern matching 1-71
  - rules 1-29
- conflict resolution 2-42
  - example 2-45
  - inheritance down 1-54
  - inheritance priority 1-14
  - inheritance up 1-56
  - nonmonotonic reasoning 2-74
  - rules 2-9
- conflict resolution cycle 2-45
- context link A-7
- context links
  - agenda priority 2-25
  - globally disabling and enabling 2-55
  - interpretations 2-32
  - locally disabling and enabling 2-57
- control
  - inference mechanisms 2-54
  - nonmonotonic reasoning 2-74

D

- data 1-29
  - affect on inferencing A-12
- data structures 1-2
- data types 1-6
- data validation attribute
  - inheritance 1-34, 1-79
  - pattern matching 1-79

usage 1-12
date 1-6
Delete level 2-82
depth-first 1-54
DisableStrong level 2-81, 2-82
DisableWeak level 2-81, 2-82
disabling
agenda search mechanisms 2-10
rules 2-10
dynamic links 2-75
dynamic objects 2-75
dynamic structures
inheritance of 1-61–1-65
links between objects 1-59
objects 1-58
usage 1-58–1-65

E

Enable level 2-81
evaluation
exhaustive 2-8
methods 2-10–2-13
rules 2-3–2-9
slot 1-45
example
inference conflict resolution 2-45
Execute operator 2-22, 2-58
exhaustive evaluation 2-8
existential qualifier 1-71
external routines
pattern matching 1-80

F

file name
interpretation 1-68
float 1-6
focus of attention 2-1
forward action-effects
agenda priority 2-21
globally disabling and enabling 2-55
interpretations 2-31
locally disabling and enabling 2-57
pattern matching 2-37
forward chaining
context 2-25
forward action-effects 2-21
gates 2-19
hypothesis forward 2-17
volunteering data 2-23

G

gates
agenda priority 2-19
globally disabling and enabling 2-55
interpretations 2-30
locally disabling and enabling 2-57
pattern matching 2-35
generalization 1-3, 1-56
goal generation
explicit vs implicit 2-16

H

heuristics 1-28, 2-2
hypotheses
context links 2-25
evaluation 2-4
explicit vs implicit investigation 2-16
hypo forward mechanism 2-17
purpose 1-31
subgoal 2-15
terminal 2-15
hypothesis
evaluation status A-1
terminal A-4
hypothesis forward
description 2-17
interpretations 2-30
locally disabling 2-56
pattern matching 2-35

I

IC (see If Change methods)
If Change methods
default 1-27
inheritance 1-50
pattern matching 1-80
purpose 1-26
inference
example 2-45
globally disabling and enabling 2-54
locally disabling and enabling 2-56
inference number 1-13
inference priority 1-13, 1-31

inference slot 1-13
inferencing
  behavior ??–A-9
inferencing mechanisms
  action-effects 2-21
  backward chaining 2-14
  context 2-25
  control 2-54
  gates 2-19
  hypo forward 2-17
  multiple knowledge bases 2-79
  suggest 2-16
  volunteer 2-23
information hiding 2-13
inheritability setting 1-15
inheritance
  conflict resolution
    from children 1-56
    from parents 1-54
  data validation attributes 1-12
  default behavior
    method 1-50–1-53
    properties 1-37–1-42
    values 1-45–1-48
  disabling
    methods 1-53
    properties down 1-43
    values down 1-48
  dynamic
    methods 1-64
    properties 1-62
    values 1-63
  enabling
    properties up 1-43
    values up 1-48
  initial value attribute 1-11
  of properties A-11
  overview 1-34–1-36
  prompt line attribute 1-12
  usage 1-34–1-57
inheritance number 1-14
inheritance priority 1-14
inheritance slot 1-14
inheritance strategy 1-17
initial value attribute 1-11
instantiation 1-3
integer 1-6
interpretations
  actions 1-68
  backward chaining 2-28
  conditions 1-66, 1-69
  context link 2-32
  database operations 1-69
  file name 1-68
  forward action-effects 2-31
  gates 2-30
  hypothesis forward 2-30
  nonmonotonic reasoning 2-72
  pattern matching usage 1-78
  prompt line 1-69
  properties 2-28
  root 1-67
  slots 1-66, 2-28
  strings 1-68
  suggest 2-30
  usage 1-66–1-70
interpreted slot 2-28
investigating hypotheses 2-16

K

knowledge base
  current 2-76
  description 1-82
  inferencing with more than one 2-79
  loading 2-80
  merging 2-79
  modular architecture 2-77
  multiple 2-75
  text format ??–B-12
  unloading 2-82
knowledge island 1-81, 2-27
knowledge processing
  conducting a session 3-3

L

links
  context 2-25
  dynamic 1-59
  static 1-4
  strong 2-25
  weak 2-25
list usage 1-74
loading knowledge bases 2-80
LoadKB operator 2-54

M

merging knowledge bases 2-79

message passing 2-10
meta-slot attributes
- data validation 1-12
- inference priority 1-13
- inheritability setting 1-15
- inheritance priority 1-14
- inheritance strategy 1-17
- initial value 1-11
- prompt line 1-12

meta-slots A-12
methods
- actions 1-18, 1-20
- agenda control 2-58
- attaching 1-19
- condition 1-21
- default inheritance 1-50–1-53
- dynamic inheritance 1-64
- evaluation 2-10–2-13
- forward action-effects 2-21
- globally disabling and enabling 2-60
- If Change 1-26
- local arguments 1-22
- locally disabling and enabling 2-60
- message passing 2-10
- naming 1-19
- non-inheritable 1-53
- operators 1-20
- Order of Sources 1-23
- pattern matching 1-80, 2-37
- SELF variable 1-51
- specialization 1-52
- system methods 1-18
- usage 1-18–1-27
- user-defined inheritance 1-50
- user-defined methods 1-18

multi-value 1-6

N

nonmonotonic reasoning
- conflict resolution 2-74
- description 2-69
- interpretations 2-72
- pattern matching 2-73

NOTKNOWN 1-9
notknown status A-2
NXP_AINFO_BREADTHFIRST 2-68
NXP_AINFO_CACTIONSUNKNOWN 2-68
NXP_AINFO_EXHBWRD 2-67
NXP_AINFO_INHCLASSUP 2-68
NXP_AINFO_INHOBJDOWN 2-68
NXP_AINFO_PARENTFIRST 2-68
NXP_AINFO_PWFALSE 2-67
NXP_AINFO_PWNOTKNOWN 2-67
NXP_AINFO_PWTRUE 2-67
NXP_AINFO_SOURCESCONTINUE 2-68
NXP_SPRIO_CNTX 2-65
NXP_SPRIO_SUG 2-65
NXP_SPRIO_UNSUG 2-65
NXP_VSTRAT_CURFWRD 2-66
NXP_VSTRAT_QFWRD 2-66
NXP_VSTRAT_VOLFWRD 2-66

O

object network 1-10
object-first 1-54
object-oriented systems
- information hiding 2-13
- message passing 2-10

objects
- definition 1-3
- dynamic 1-58
- interpreting 1-66
- links between 1-4
- subobjects 1-5

operators
- methods 1-20
- rules 1-33

opportunistic reasoning (see gates)
or graph 2-5
Order of Sources methods
- inheritance 1-50
- pattern matching 1-80
- purpose 1-23

OS (see Order of Sources methods)

P

parameterized query 1-68
parent class 1-4
part of 1-5
pattern matching
- backward chaining 2-33
- class 1-73
- data validation 1-79
- existential 1-72

external routines 1-80
forward action-effects 2-37
gates 2-35
hypothesis forward 2-35
interpretations usage 1-78
list 1-74
methods 1-80
multiple 1-76
nonmonotonic reasoning 2-73
object 1-74
universal 1-71
usage 1-71–1-80
volunteer 2-41
primer knowledge base B-1
priorities 2-10
private methods 1-53
procedural knowledge 1-28
prompt line attribute
interpretations 1-69
purpose 1-12
properties
default inheritance 1-37–1-42
definition 1-6
dynamic inheritance 1-62
inheritability control 1-43
interpreting 1-66
public methods 1-53

R

reasoning path A-5
reasoning, nonmonotonic 2-69
relations 1-28
Reset operator 2-70
resolved slot 2-28
restart
programatically 2-67
Retrieve operator 2-22, 2-58
revisions 2-69
rule evaluation
globally disabling and enabling 2-54
locally disabling and enabling 2-56
rule priorities 2-9, 2-10
rules
basic structure 1-28
condition 1-29
conflict resolution 2-9
context links 2-25
determining its value 1-29
disabling 2-10
evaluation 2-3–2-9
forward action-effects 2-21
gates mechanism 2-19
hypothesis 1-31
hypothesis forward 2-17
investigating hypotheses 2-16
multiple rules evaluation 2-5
operators 1-33
right-hand side actions 1-32
symmetry 1-29
usage 1-28–1-33
volunteering data 2-23

S

SELF variable
data validation usage 1-79
method usage 1-51
prompt line usage 1-70
semantic gates (see gates)
SendMessage operator 2-10
slots
behavior
meta-slot attributes 1-11
methods 1-18
definition 1-7
evaluation 1-45
interpretations 1-66
private versus public 1-11
specialization 1-4, 1-53
strategy
programatic control 2-67
strings
data type 1-6
interpretation of 1-68
strong link A-9
strong links 2-25
subclass 1-4
subgoals 1-29, 2-15
subobjects 1-5
suggest
interpretations 2-30
programatically 2-65
purpose 2-3
suggesting hypotheses 2-16
symmetry 1-29
system methods 2-11
definition 1-18

inheritance 1-50
pattern matching 1-80

T

temporary.kb 2-75
terminal hypotheses 2-15
time 1-6

U

undefined.kb 2-76
universal qualifier 1-71
UNKNOWN 1-9
unloading knowledge bases 2-82
UnloadKB operator 2-54
untitled.kb 2-75
user-defined methods 2-11
definition 1-18
inheritance 1-50
pattern matching 1-80

V

Value property 1-7
values
default inheritance 1-45–1-48
dynamic inheritance 1-63
inheritability control 1-48
variablized query 1-68
volunteer
pattern matching 2-41
programatically 2-66
volunteering data 2-23

W

weak link 2-25
Wipeout level 2-82