Integration of a Rule Engine Component

With a Portal Platform

Master Thesis

Submitted by
Misumbi Sylvester Chanda
Matriculation Number: 12972

05.01.2004

A thesis submitted in partial fulfillment of the requirements of the degree of Master of Science in Information and Communication Systems

TECHNICAL UNIVERSITY HAMBURG-HARBURG
GERMANY

Supervised by:

Prof. Dr. Joachim W. Schmidt
Arbeitsbereich Softwaresysteme
Technische Universität Hamburg-Harburg

Prof. Dr. Florian Matthes
Arbeitsbereich Softwareengineering betrieblicher Informationssysteme,
Technische Universität München

Table Of Content

1. Introduction 1
  1.1 Objective and Goals 2
  1.2 Structure of the work 2

2. Concepts of Rule-Based Systems 3
  2.1 Rule Concepts 10
  2.2 Rule Languages 11
    2.2.1 IRL (ILOG Rule Language) 13
    2.2.2 Business Action Language (BAL) 14
    2.2.3 Custom Business Rule Languages 14
    2.2.4 Decision Tables 15
    2.2.5 Decision Trees 16
    2.2.6 The Technical Rule Language (TRL) 16
    2.2.7 BrokerRL: Business-Object-Oriented Rule Language 17
    2.2.8 XML-based Rule languages 18

3. Portal Platform Architecture and Rule Engine Component 21
  3.1 Portal platform architecture 21
  3.2 The Existing Rule Management System Model 23
  3.3 Integrated Rule Management System 24

4. Business Rule Engine Products and Standards 33
  4.1 Business Rule Engine Selection Criteria 33
  4.2 ILOG JRules 4.5 Product 34
  4.3 Java Expert System Shell – JESS 36
  4.4 Blaze Advisor from Brokat 37
  4.5 OPSJ from Charles Forgey 38
  4.6 QuickRules from Yasu Technologies 38
  4.7 CommonRules from IBM alphaworks 38
  4.8 ACQUIRE from acquired Intelligence 38
  4.9 CLIPS from Gary Riley 39
  4.10 InfoSapient 39
  4.11 Rule Engine Standards 39

5. Rule Engine Component Integration 41
  5.1 Design Options 43
    5.1.1 Option 1: Rules as business objects 45
    5.1.2 Option 2: Rules exchange in Ilog XML Format 46
    5.1.3 Option 3: Rule Processing Via RuleML and XSLT 47
    5.1.4 Option 4: Rule processing via ILog Rule set files 48
  5.2 Details of the Selected Integration Option 48
  5.3 Evaluation 49

6. Summary & Outlook 51
  6.1 Summary 51
  6.2 Outlook 51

Appendix A: SRML DTD 53

Appendix B: The RuleML DTD 55

Appendix C: ILOG Simple Rules DTD 57

Appendix D: Sample .ilr Rule File 61

Bibliography 62

1. Introduction

Nowadays the Internet has become ubiquitous. Portals provide a single point of access for staff, business partners and customers. They provide access to corporate knowledge and content. In this era management of corporate knowledge and content can make the difference between success and failure of a company.

Companies are making use of the Internet on an every day basis. In order to reach out to customers, companies use Enterprise Portals. Portals increase the competitive advantage of companies to a great extent.

Furthermore, portals support various critical corporate processes. These processes can be rule-driven, i.e. many processes are executed depending on certain conditions. For the smooth running of these processes, rule support is essential for enterprise portals.

In this sense, it is important to note that the increasing reliance on information technology (IT) as the basis for strategic business initiatives and decisions in the corporate world is driving an increase in both the complexity and the pace of change of portal applications. As a result, IT solution providers are being asked to implement data-driven applications containing business rules that are too complex, voluminous, and fast changing for traditional software architectures.

Many companies have realized the value and importance of managing business rules as assets separate from corporate data and application code. As a result, standard methods for deploying rules from repository to production need to be defined and supported by rule-based software tools.

Nowadays, performance and scalability decisions must take into account business rule execution, and application infrastructure must accommodate business rule engines efficiently. Rule-based portals make interactions between users such as policy managers**,** business analysts and application developers smoothly.

When a company wishes to introduce rule-based technoloy in its portal system, a lot of questions arise. These questions are of a varied nature. The company may ask its IT department the following questions among other things:

- Do we have many rule-based processes in the company to necessitate the use of a business rule-based system? Sometimes the use of a rule-based system might be an over-kill. This is mainly a question of balancing costs and benefits of a software system.

- Do we have enough in-house IT know-how to run the rule-based system? For instance, integrating a business rule engine, a component of a rule-based system, requires sound knowledge of rule-based technology.

- What rule-based system are we going to use? There are many rule-based systems on the market to choose from. Each rule-based system has its own strenghs and weaknesses. Choosing an approprate software product from many such products is a difficult task.

- How financially viable is the rule-based system vendor? The liquidity of a vendor gives a clue about its long-term availability. The vendor company has to exist for a long time in order to offer long-time consultancy, support and maintenance to its customers.

- Open source rule engines must be evaluated against commercial rule engines. How good is the support with commercial rule engines compared to community support given for open source rule engines? Will an open source rule engine be maintained for a longer period of time?

These issues have to be dealt with in order to decide whether to use a rule component in a corporate portal.

1.1 Objective and Goals

Enterprise Portals are nowadays realized using standard software, which can be called portal platform. One of these is the InfoAsset Broker portal platform. This portal platform has some rule support, but lacks general rule execution mechanism.

In this work, scenarios for general rule support for the InfoAsset Broker portal platform shall be analysed and an architecture for general rule support shall be defined, realized and evaluated. In this regard many options for general rule support shall be studied and evaluated with respect to their advantages and disadvantages. A final choice will be made among the various options for integration with the portal platform. The integration will then be evaluated.

1.2 Structure of the work

Chapter 2 will describe the general rule concepts that are relevant for rule-based systems. It will particularly concentrate on the concepts that are of importance to my work. It will then introduce various rule language classes that are supported by rule-based systems.

Chapter 3 will describe the general architecture of the portal platform on which my work is based. It will then describe the current rule-based management system model. Furthermore, it will describe the integration of the current rule management system in the portal platform.

Chapter 4 will describe important business rule engine systems and standards. It will describe the criteria, which should be taken into consideration when selecting a rule engine product. It will close up with a description of rule engine standards that are at an advanced stage of their development.

Chapter 5 will describe the integration of the selected external rule engine system into the portal system. It will start with a description of all the concepts that are common to all integration options. The various integration options for the external rule engine will also be described here. One integration option will be selected for evaluation. Furthermore, the selected option will be analysed in detail.

Chapter 6 will close up my work with a summary and outlook.

2. Concepts of Rule-Based Systems

The increasing reliance on information technology (IT) as the basis for strategic business initiatives in the corporate world is driving an increase in both the complexity and the pace of change of applications developed by IT departments. Business rules are often hardcoded in business logic which makes maintaince of code extremely difficult. The solution is to use rule engines of rule-based systems to externalize business rules from application code. A business rule engine can be understood as a standard component in business system architecture. The rule-based system software architecture improves the software development process by:

- Minimizing time required to modify software when business and market conditions change
- Making it easy to incorporate business people into the software development and maintenance process

The importance of rule-based systems is also given in [16, ILOG Jrules Technical White Paper]:

A rule-based system supports collaboration of business partners in an orderly and efficient manner. It provides a systematic way to model, implement, and deploy rules. It provides tools based on the skills and knowledge of individual roles, so that policy managers, business analysts and developers have the support they need to make the most of their business rule management. A rule-based system can be delivered quite easily in incremental pieces. If the first increment includes a solid data foundation (cast with the future in mind), incremental system releases become the delivery of upgraded or additional rule sets to an existing infrastructure.

Separating the business rules from application code and implementing them using a rule engine, a component of a rule-based system, can make applications adaptable and maintainable. Since business rules are externalized from the application code, they can be changed independently without recompiling the application. When represented in a natural business language that business people understand, business users and analysts can actually write and maintain rules. This will be made clear in the coming chapters.

In figure 1.1, the left hand-side diagram shows that in traditional software systems rules and application code are interspersed. The right hand side diagram shows that in rule-based systems rules and application code are clearly separated from each other.

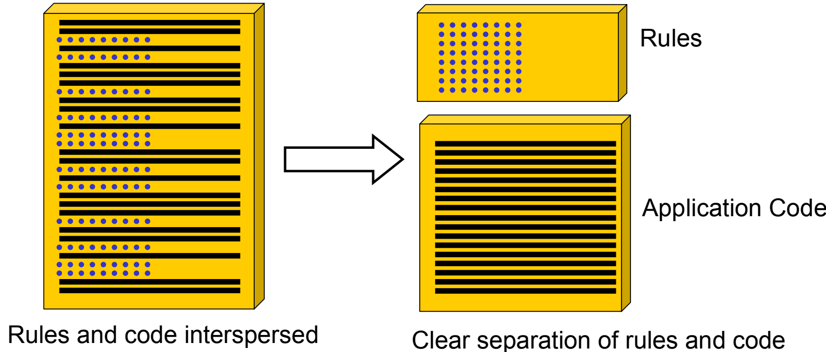


Figure 2.1: Separation of business rules from application code

A very important question now arises; when should rule-based programming be used? I will answer this question by explaining what rule-based programming is and what it is not. Much of the programming done in everyday life is procedural. Rule-based programming, however, is declarative.

In procedural programming, the programmer tells the computer what to do, how to do it, and in what order. Procedural programming is well suited for problems in which the inputs are well specified and for which a known set of steps can be carried out to solve the problem. In other words, procedural programming should be used to solve problems with clear algorithms, e.g. mathematical computations.

A purely declarative program, in contrast, describes what the computer should do, but omits much of the instructions on how to do it. Problems involving control, diagnosis, prediction, classification, pattern recognition, or situation awareness are best solved using declarative programming.

During the course of this work I will make use of the experience I gained by using the ILOG BRMS (Business Rule Management System) products to make the concepts of rule engines clear. Because of the dynamic nature of business rules, conventional application development fails to adequately support their unique demands. Business logic is encapsulated and spread throughout application objects, client software, database structures, and stored procedures. This is difficult and costly to maintain. When business policy changes, developers must read through application code to locate the logic, make the changes, recompile and test. In today's fast paced business and e-business environment, maintaining code in this way is no longer profitable. Competition has driven organizations to adapt to changing markets in ever-shorter cycles. Often, current applications cannot adapt fast enough to keep pace.

The figure below shows that in a rule-based system, the rules are separated (logically, perhaps physically) and shared across data stores, user interfaces and applications.

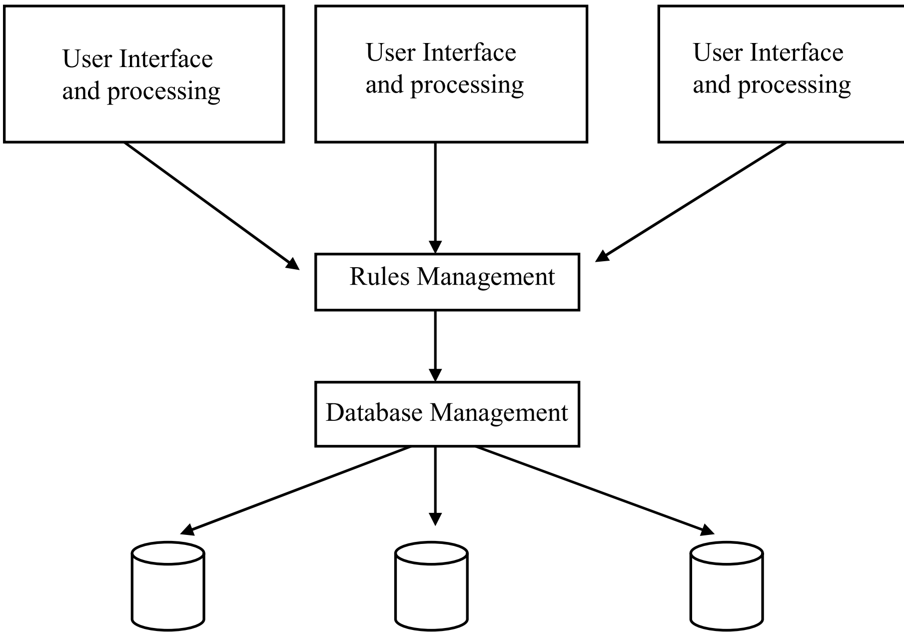


Figure 2.2 [4]: A High-Level Conceptual Architecture for a Rule-Based System

Adding a brief historical development of how these systems evolved from the field of Artificial Intelligence would be beneficial. However, my main emphasis will be on the architecture of these systems. One of the results of research in the area of artificial intelligence has been the development of techniques, which allow the modelling of information at higher levels of abstraction [1]. These techniques are embodied in languages or tools, which allow programs to be built which closely resemble human logic in their implementation and are therefore easy to develop and maintain. These programs, which emulate human expertise in well-defined problem domains, are called expert systems [2]. Expert systems whose knowledge is represented in rule form are called *rule-based systems*. I will explain the term *knowledge* later. The two terms, expert system and rule-based system, are usually used interchangeably. Business people usually prefer the term *business rule management system* to refer to this type of systems.

A rule-based system is a system that uses *rules* to derive conclusions from premises. In traditional computer programming, a program is made up of an algorithm and data structures [3]. An expert system is made up of an inference engine (rule engine) and knowledge. Rule-based programming is one of the most commonly used techniques for developing expert systems. In this programming paradigm, rules are used to represent heuristics, which specify a set of actions to be performed for a given situation. An expert system, see figure 1.3, consists of a *User Interface*, a *Knowledge Base* (rule base), an *Inference Engine* (Rule Engine) and the *Working Memory*. The Knowledge Base contains all the rules and data used by those rules. The Inference Engine controls overall execution of rules. The rule engine or the inference engine component of an expert system executes rules.

The main purpose of an expert system is to improve an existing operation by improving its profits, quality and/or security. This is done in three principal ways. First, an expert system can make the knowledge of a small number of experts available to many. Second, it can add uniformity to procedures and results where previously there may have been inconsistencies. Finally, expert systems can offer an active form of training - learning by doing.
The diagram below shows the architecture of a classical expert system with user interface, Knowledge base, agenda, inference engine and working memory components.

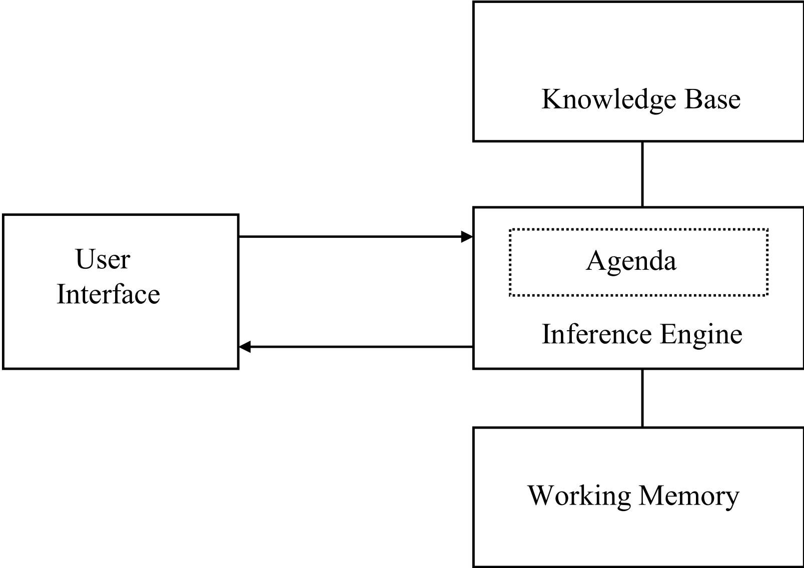


Figure 2.3: Components of Expert Systems

The Knowledge Base

The *knowledge base* is a component of expert systems containing both factual and heuristic knowledge.

Factual knowledge is that knowledge of the task domain that is widely shared, typically found in textbooks or journals, and commonly agreed upon by those knowledgeable in the particular field.

Heuristic knowledge is the less rigorous, more experiential, more judgmental knowledge of performance. In contrast to factual knowledge, heuristic knowledge is rarely discussed, and is largely individualistic. It is the knowledge of good practice, good judgment, and plausible reasoning in a particular field.

A knowledge base contains knowledge, which may include everything that makes up a database plus the relationships among the data and the ability to update itself through usage.

For example, a knowledge base about a family might contain the facts that John is David's son and Tom is John's son and the rule that the son of someone's son is their grandson. From this knowledge one could infer the new fact that Tom is David's grandson.

Algorithm versus Inference Mechanism

Understanding the difference between an algorithm and an inference mechanism is a key to understanding expert systems. An algorithm consists of a step-by-step resolution of a problem, as well as the logic, relating data. Once the algorithm for solving a problem is known, the programming is straightforward. What often makes life difficult is that the task of programming usually includes defining the algorithm.

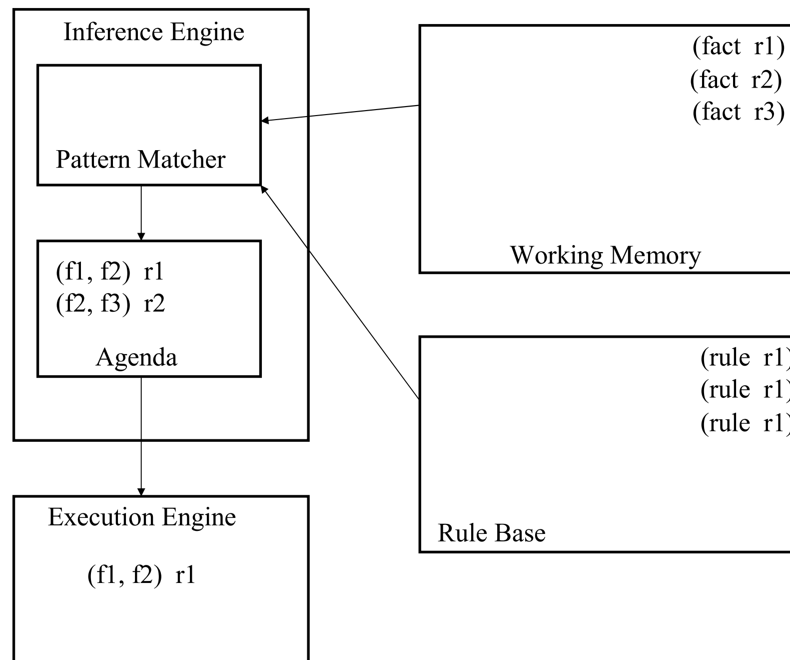
An inference mechanism consists of algorithms and the rules that relate the knowledge in a knowledge base. An inference mechanism is just a higher level programming language used to facilitate the development of an expert system in the same way that Lotus 1-2-3 and QuattroPro were higher level languages used to develop sophisticated spreadsheets. Today, users would never think of creating a spreadsheet without using a spreadsheet program. Similarly, a programmer would not contemplate building an expert system without using an inference mechanism.

User Interface

An expert system requires a suitable user interface and may require a more complex entry and retrieval of information. For example, in BERT, (Bank expERT), an expert system used by the US Comptroller of Currency to check on the fiscal health of US banks, financial information is downloaded from a database before the system is used by an investigator. An expert-system interface may also require the user to enter new relationships as it probes deeper into a problem.

Since in this work I will concentrate on the type of expert systems called rule-based systems, it is important at this stage to introduce the general architecture of modern rule-based systems.

The diagram below shows the main components of a rule-based system, user interface, Knowledge base, agenda, inference engine and working memory.



f: Fact
r: Rule

Figure 2.4: The architecture of a typical rule-based system

The Inference Engine

The inference engine (or rule engine) controls the overall execution of the rules. The primary purpose of a rule engine is to apply rules to data held in the working memory. It is the central part of a rule-base system. It controls the whole process of applying the rules to the working memory to obtain the outputs of the system. It searches through the knowledge base, attempting to pattern match facts or knowledge present in memory to the antecedents of rules. If a rule's antecedent is satisfied, the rule is ready to fire and is placed in the agenda. When a rule is ready to fire it means that since the antecedent is satisfied, the consequent can be executed. Usually the rule engine works in cycles that go like this:

- All the rule antecedents are compared to facts working memory (using the pattern matcher) to decide which ones should be activated during this cycle. This unordered list of activated rules, together with any other rules activated in previous cycles, is called the *conflict set*.
- The conflict set is ordered to form the agenda, the list of rules whose right hand-side will be executed or fired. The process of ordering the agenda is called *conflict resolution*. The conflict resolution strategy for a given rule engine will depend on many factors, only some of which will be under the programmer's control.

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

The Rule Base

The rules need to be stored somewhere. The rule base contains all the rules the system knows. They may simply be stored as strings of text, but most often a rule compiler processes them into some form that the inference engine can work with more efficiently. For an email filter, the rule compiler might produce tables of patterns to search for and folders to file messages in.

In addition, the rule compiler may add to or rearrange the premises or conclusions of a rule, either to make it more efficient or to clarify its meaning for automatic execution. Depending on the particular rule engine, these changes may be invisible to the programmer.. Some rule engines allow the programmer to store the rule base in an external relational database, and others have an integrated rule base. Storing rules in a relational database allows the programmer to select rules to be included in a system based on criteria like data, time, and user access rights.

The Working memory

The rule-base system stores all the data (objects) its rule engine is currently working on in the working memory. In a typical rule engine, the working memory, sometimes called the fact base, contains all the pieces of information the rule-base system is working with. The working memory can hold both the premises and conclusions (result objects) of the rules. Typically, the rule engine maintains one or more indexes, similar to those used in relational data-bases, to make searching a very fast operation.

Some implementations can hold only objects of a specific type, and others can include objects of any type, for example Java objects.

The Pattern Matcher

The inference engine has to figure out what rules to fire and when. The purpose of the pattern matcher is to decide which rules apply, given the current contents of the working memory. If the working memory contains thousands of facts, and each rule has two or three premises, the pattern matcher might need to search through millions of combinations of facts to find those combinations that satisfy the rules.
Often the pattern-matching techniques used by a particular rule engine will affect the kind of rules the programmer writes, either by limiting the possibilities or by encouraging him to write rules that would be particularly efficient.

The Agenda

Once the inference engine figures out which rules should be fired, it must still decide which rule to fire first. The list of rules that could potentially fire is stored on the agenda. The agenda is responsible for using the conflict strategy to decide which of the rules, out of all those that apply, have the highest priority and should be fired first. Each rule engine has its own approach to solve this problem.

The Execution Engine

Finally, once the rule engine has determined which rule to fire, it has to execute that rule's action part. The execution engine is a component of a rule engine that fires the rules. In classical rule-based systems, rules could do nothing but add, remove, and modify facts in the working memory, but now they can do more than this.

There are two methods for executing rules in rule-based systems, forward chaining and backward chaining.

Forward-chaining engines

Forward-chaining systems are data-driven, i.e. they compare objects/facts in the working memory against the conditions (IF parts) of the rules and determine which rules to fire. The facts in such systems are represented in a working memory that is continually updated. Furthermore, in these systems rules represent possible actions to take when specified conditions hold on items in the working memory - they are sometimes called *condition-action rules*. The conditions are usually patterns that must match items in the working memory, while the actions usually involve adding or deleting items from the working memory.

Backward-chaining engines

Backward-chaining systems are goal-driven. These systems look for the action in the THEN clause of the rules that matches the specified goal. In other words, they look for the rules that can produce this goal. If a rule is found and fired, they take each of that rule's conditions as goals and continue until either the available data satisfies all of the goals or there are no more rules that match.

In my work, I will only concentrate on forward-chaining systems.

Rule Engine Optimization

The typical rule-based system has a more or less fixed set of rules, whereas the working memory changes continiously. However, it is an empirical fact that in most rule-based systems, much of the working memory is also fairly fixed over time. Although new facts arrive and old ones are removed as the system runs, the percentage of facts that change is per unit time is fairly small.
There are many methods for optimizing rule engines to execute rules more efficiently. Most rule engines use the Rete (Latin for `net') Algorithm[15] for optimization. This algorithm

makes use of the above idea. The Rete Algorithm is intended to improve the speed of forward-chained rule-based engines by limiting the effort required to re-compute a conflict set after a rule is fired. In the Rete algorithm, executable rules are compiled into a network. Input data to the network consists of changes to the working memory. Objects are inserted, removed, and modified. The network processes these changes and produces a new set of rules to be fired. The network minimizes the number of evaluations by sharing tests between rules and propagating changes incrementally. Briefly, the rete algorithm eliminates the inefficiency in the simple pattern matcher by remembering past test results across iterations of the rule loop. Only new or deleted working memory elements are tested against the rules at each step. Furthermore, Rete orgonizes the pattern matcher so that these few facts are only tested against the subset of rules that may actually match.

The main drawback of this algorithm is its high memory space requirement.

2.1 Rule Concepts

A business rule is a precise statement that describes, constrains or controls some aspect of a business[9]. The strategic importance of business rules is to implement a company's objectives, that is, accomplish the company's vision and goals.

A *rule* (sometimes called a *heuristic*) is a kind of instruction or command that applies in certain situations. The following statements are some of the rules that we know in everyday life:

- No eating in STS [30] computer laboratory
- No walking with bare feet in snow
- No walking with pants on the street
- No chewing gums in class

Rules are like the *if-the*n statements of traditional programming languages. An STS laboratory order rule can look like this, in an English-like pseudocode:

```
IF
     A student is in the laboratory
AND
    He/She is hungry
THEN
    He/She should go to the canteen to eat
```

Rules have domains. The domain of a rule is the set of all information the rule could possibly work with. In the above example the domain of the laboratory rule is a set of facts about the location and oral fixations of one particular student.

The following paragraph is taken from the white paper [WP-JRules4[1].0]. Business rules describe and control the structure, operation and strategy of an organization. Often they are captured in policy and procedure manuals, customer contracts, supplier agreements, marketing strategies, and as the expertise embodied in employees (as is the case in expert systems). They are dynamic and likely to change with time, found in all manner of applications, from the click-stream of a website to the numerous business rules in B2B

(Business to Business) systems, in trading partner contracts , etc. Finance and insurance companies must enforce numerous internal policies and external regulatory policies. E-business, self-service and personilization on the web demand real time order tracking, sales histories, and customer preferences. Each of these business domains rely heavily on being able to convey the policies, regulations, and the strategies of their industry to IT departments for integration into software applications. All these scenarios are captured in business rules or simply rules.

A *rule* is composed of the condition part (or IF part) and the action part (or THEN part). The *condition part* of a rule is a series of patterns that specify the facts that must be matched for the rule to be applicable. The *action part* of a rule is also called the Left Hand Side (LHS) and likewise the *action* part is called the Right Hand Side (RHS).

Rules are grouped together to form a *rule set*. In other words, a rule set is a file containing all the rules in a particular domain. I will clarify the concepts of conditions, actions and firing of rules with an example. Given the following rule:

```
IF
      The Customer is older than 65
THEN
      Offer him a senior citizen discount
```

From the above it is clear that the condition part of our rule is <The Customer is older than 65>. The action part is likewise <Offer him a senior discount citizen discount>. In java syntax the condition part is `customer.getAge() > 56`. Likewise the action part is `offerSeniorDiscount(customer)`. The action part of this rule is executed if the condition part is true. Execution of a rule is also referred to as firing the rule. Firing the rule in java sense is calling the method `offerSeniorDiscount(customer)`.

2.2 Rule Languages

There are different rule languages for different users, from the system developer to end-users. There are basically two different types of rule languages (Figure 2.1):

- Rule authoring languages: These are rule languages that end users and programmers use to write rules. Rule authoring languages are provided to allow users to write business rules with different rule syntaxes. They usually have a syntax that resembles a natural language like English. Rule engines do not directly process rules written in these languages.

- Rule execution languages: These are rule languages used for writing rules that are directly executable by rule engines. Rule execution denotes the process of executing a ruleset in a language that a rule engine understands. All business rule languages must be translated into an executable rule language before the business rules can be executed

The diagram below shows the different types of rule language types supported by common rule engines. It further shows three examples of rule authoring languages. I will talk more

about the different types of rule languages in coming chapters. At this stage it is important to only note that there are different rule languages for different purposes.

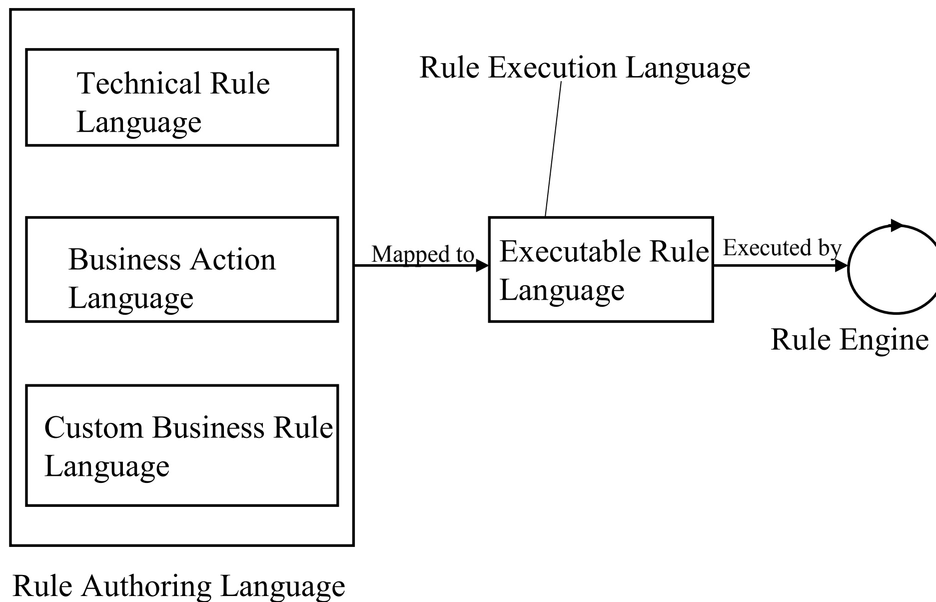


Figure 2.5: Rule Languages Supported By Rule Management Systems

There is no universal business rule language. Different applications and business domains require different business rule languages. The difference is not only in the classes and properties that are manipulated by the rules, but also in the rule structure.

Differentiation of rule languages is useful if all languages can be mapped to a more generic executable rule language. In many applications which embed a business rule engine, the rules will be written by business people who have mastered the application business domain, but do not necessarily have a programming background. The concept of a business rule engine will become clearer during the course of this paper. For the sake of understanding at this stage, a rule engine is simply an object in a business application for business rule processing. Business people might find it difficult to express rules directly in a technical rule language because of its developer-oriented design.

Before writing the business rules, a business rule language must be selected, or a custom language developed that supplies the natural language syntax required to create a custom language. Rule languages for business people are called Business Rule Languages (BRL). They are designed to be used by business analysts or policy managers and use a business-oriented rather than a technical vocabulary. They enable reasoning on an object model that reflects the structure of a given business domain, rather than the underlying implementation.

I will explain the differences between rule languages using the ILOG JRules business rule languages as examples[16].

A differentiation of rule languages is given in [16, ILOG Jrules Overview]:

One example of a business rule language is the Business Action Language (BAL) from ILOG. The BAL is a general purpose business rule language with a natural language syntax. The BAL is designed to cover most of the common needs when writing business rules. Rule

languages mainly used by developers or technical staff are called Technical Rule Languages (TRL). Technical Rule Languages are syntax driven. Rule languages which can be directly executed by rule engines are called executable Rule Langages. One example of such a language is ILOG Rule Language (IRL), which is the language that can be directly executed by the JRules rule engine. The IRL has a Java-like syntax and is mostly used by developers. Business rule languages, like the BAL or the TRL, can be used by a developer to write rules, too.

The relationship between the different rule languages in the Ilog Jrules Rule Managent System is made clear in the figure 2.2. The Ilog software vendor has yet introduced another rule language which is in the form of decision tables. In the diagram below it can be seen that the different types of the Ilog rule languages must first be converted into the Ilog rule language before being processed by the rule language.

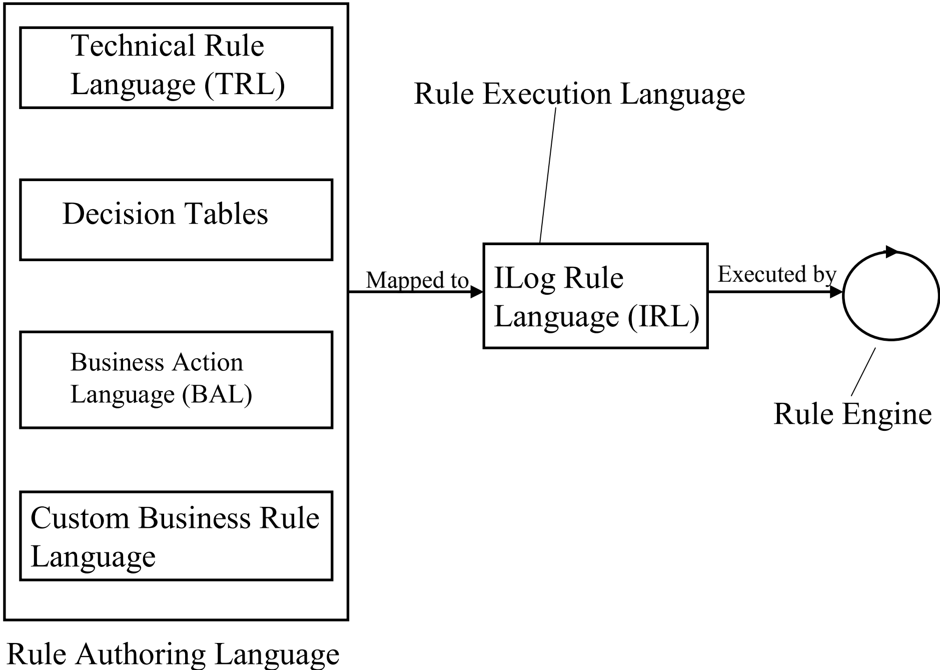


Figure 2.6: The ILog JRules Rule Language Architecture

In the following sections, the different ILOG rule languages will be explained in more detail.

2.2.1 IRL (ILOG Rule Language)

The ILOG Rule Lanaguage (IRL) is the core language of the ILOG rule engine. The IRL is an executable rule language and all business rules of different rule languages are translated into this language before parsing by the rule engine.

IRL is a more concrete language than the business rule languages and can reference any application object, that is, a Java object or an object derived from XML data. It puts full support of Java operators in expressions, tests, and Java arrays at the disposal of rule-based system developers.

There is support for relationships between objects so that a rule can reference not only objects in the working memory (a part of the rule engine that stores objects against which a ruleset is

executed), but objects that are linked to those in the working memory by data members or method invocations. Temporal reasoning is available for applications that need to operate in realtime, and is supported by extended features of the language.

2.2.2 Business Action Language (BAL)

While the Ilog rule language is ideal for developers, the business action language (BAL) is intended for business analysts and end users. With the business action language, business rules can easily be created by non-technical users using business terms, instead of Java constructs, and automatically translated for execution by the rule engine to IRL. BAL can be localized for different languages, enabling business rules to be expressed in English, for example, and viewed in French, and vice versa. It can be used as is or customized to the language of an analyst's business.
It enables business people to write rules using a natural and readable syntax.

The diagram below shows an example of a business action language rule as defined in the rule editor:

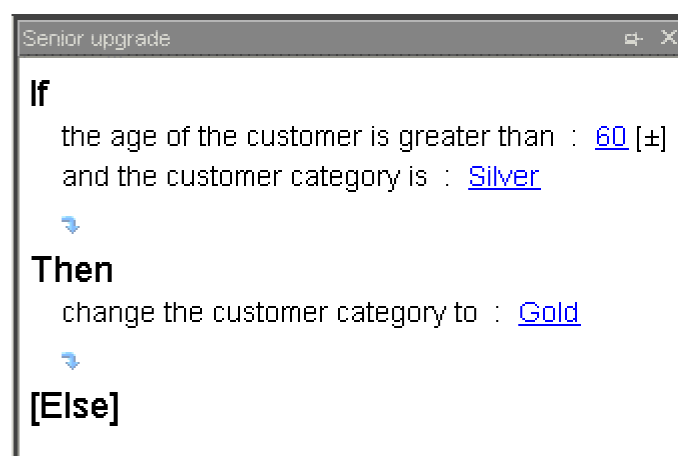


Figure 2.7: Business Action Language rule example from ILOG []

The above rule in normal english language (Oxbride English) reads as follows:

If the customer is older than 60 years and his category is silver, his category should be changed to gold.

2.2.3 Custom Business Rule Languages

Developers can implement their own business rule language using the business rule language support provided by ILOG JRules. The custom business rule language can be made as simple or as complex as necessary, providing business users with the freedom to write complex rules or constraining them to work within a predefined scope of the business domain.

A given business rule language can be customized to the terms and vocabulary of a particular business domain, allowing business users to edit business rules in a language relevant to their business.

2.2.4 Decision Tables

Business analysts, who are used to creating their business rules in table format, can do so using the decision tables. Many companies implement decisions through the use of decision tables. This is especially true where the business rules are based on a small number of conditions, with each rule corresponding to a particular combination of data values or value ranges. Decision tables are look-up tables where rows and columns represent different conditions and the resulting action or returned data is defined by their intersection.

Common examples of such tables include rate tables, pricing charts, discount schedules, etc. The postage chart on the wall of a post office is an example of a decision table. This chart usually shows the shipping method, the weight, and the destination of a particular item. Rules in form of decision tables are easy to visualize and work with.

Decision tables can be displayed as desired in a variety of formats, e.g. single-axis or double-axis, multiple condition rows or columns, by single value or value ranges, and with color and font formatting. Cell contents can be simple return values or can be any complex action. Tables can efficiently handle many thousands of rule conditions and actions. Decision tables provide a concise view of a set of business rules that has flexible browsing and exploration capabilities.

The diagram below shows an example of a decision table.

Decision Table Rule *

Age	Revenues	Category	Free Shipping	Gift	Discount
Young	Low	Silver	.	.	.
		Gold	X	Puppet	10
	Medium	Silver	.	.	.
		Gold	X	Lighter	5
	High	Silver	.	.	.
		Gold	X	Champagne	5
MiddleAge	Low	Silver	.	.	.
		Gold	X	Puppet	10
	Medium	Silver	.	.	.
		Gold	X	Lighter	5
	High	Silver	.	.	.
		Gold	X	Champagne	5
Senior	Low	Silver	.	.	.
		Gold	X	Puppet	10
	Medium	Silver	.	.	.
		Gold	X	Lighter	5
	High	Silver	.	.	.
		Gold	X	Champagne	5

Senior High Gold Discount

Figure 2.8: Decision Table Example

In the above example the columns represent the varios parts of a rule. The *age*, *revenues* and *category* Columns represent the condition part of a rule. Likewise, *free shipping*, *gift* and *discount* represent the action of a rule. The rows represent a particular rule.

Here is an example of a rule read from the above table:

IF
The Age of the customer is young, his revenue is low and his category is gold
THEN
The custmer should be given free shipping, a puppet gift and a discount of 10%

2.2.5 Decision Trees

Another convenient way to represent sets of interrelated decisions is with decision trees. Decision trees allow users to trace a chain of conditions to a single appropriate action. Looking at branches coming from a decision point (node) in the tree lets the user quickly confirm that all applicable possibilities have been accounted for.

As with Decision Tables, action nodes can contain a single return value or can specify a complex rule language construct.

The diagram below shows a decision tree example. The gray boxes represent actions of a rule. The left-hand side up to the gray boxes represent the condition part of a rule. An example of a rule in this decision tree would read as follows:

IF
The yearly income of a customer is greater than $100,000
THEN
Offer the customer a loan limit of $20,000

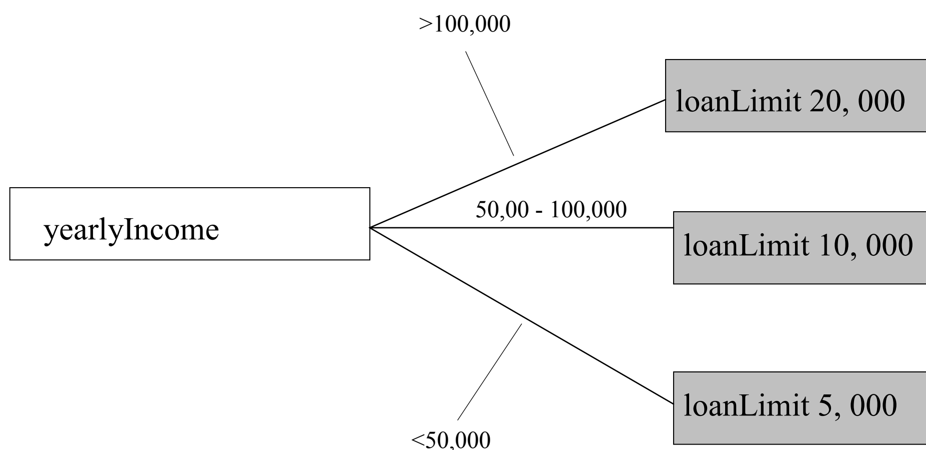


Figure 2.9: Decision Tree Example

2.2.6 The Technical Rule Language (TRL)

The technical rule language is a version of Ilog rule language with emphasis on rule syntax. Rules written in this language can be edited by the rule editor using point and click techniques to access lists of classes and methods.
It is mostly oriented towards developers since it resembles a programming language and it has programming language oriented constructs.

The diagram below shows an example of a technical rule language rule. The *WHEN* represents the condition part of a rule and the *THEN* is the usual action part of rule. This rule reads in normal english syntax as follows:

```
IF
      The customer has a shopping cart with free shopping status

THEN
       The customer should be offered free shipping
```

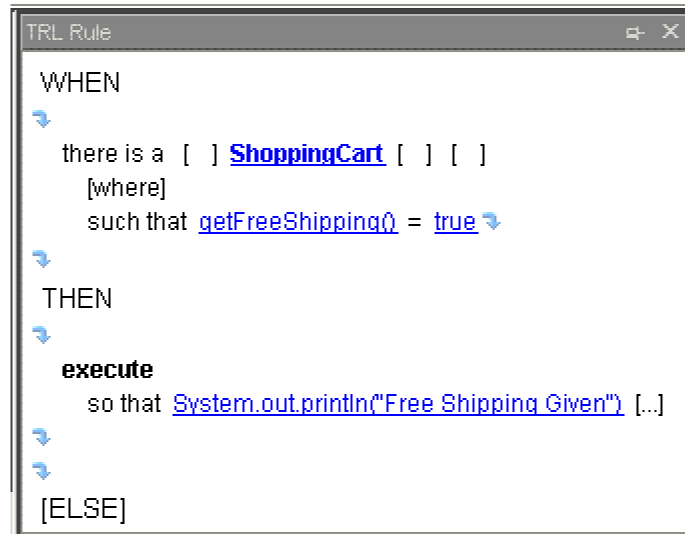


Figure 2.10: Technical Rule Language Rule Example

2.2.7 BrokerRL: Business-Object-Oriented Rule Language

The broker rule language (BrokerRL) [12][37] is a business object-oriented rule language defined specifically for the WIPS research project. The WIPS IT 1.1 portal is a maritime industry portal system based on the InfoAsset Broker. In the WIPS IT 1.1 research project, a rule management system has been designed and implemented to manage ship survey-specific rules [12][37]. The broker rule language is suitable for constructing event-condition-action rules (ECA).

An example of a BrokerRL rule is as follows:

```
IF
      (Ship.Classification = "MC") AND (Ship.Flag = German)
THEN
      AddSurveyItem 413 to SurveyItemList
```

2.2.8 XML-based Rule languages

A number of projects exist with the aim of defining a standard rule language based on XML, the Extensible Mark-up Language [43]. XML is a self is a self-describing, text-based structured data file format. The power of XML lies not so much with XML itself, but with the wealth of available tools for working with it. Web browsers can visualise it. Commercial XML editors are available. XSLT can be used to write simple scripts that transform an XML document into a new XML document in a different format, or into a non XML document. Many parsers and APIs are available to for working with XML from Java programs.

Representing rules as XML makes sense for many reasons, among them interoperability, editability, searchability. XSLT scripting languages and XML parsers transform rules from XML into the language used by any rule engine. However, transforming in the other direction is usually very hard and one needs a parser for the rule language. Therefore it is recommended to define rules as XML, and only transforming them into the native format of a rule language for execution.

The drawback of XML is that it is not very readable compared to vendor specific rule languages.

There have been a number of initiatives working on XML-based rule languages. The initiatives that are currently at an advanced stage are:

- Simple Rule Markup Language (SRML)
- Rule Markup Language (RuleML)

2.2.8.1 Rule Mark-up Language (RuleML)

The RuleML [38] project is a defining a standard representation for rules. It covers both forward-chaining and backward-chaining rules as well as transformation and mapping of rules.

RuleML largely grew out of the Business Rules Markup Language (BRML) [13] which was developed at IBM Research and which is implemented in IBM CommonRules [10], a rule engine library. The goal of the RuleML Initiative is the eventual adoption as a Web standard, e.g., via the World Wide Web Consortium (W3C). CommonRules enables exchange of executable business rules over the Internet between enterprises using heterogeneous rule systems, and enables incremental specification of executable business rules by non-programmer business domain experts.

Here is an example of a RuleML rule[39]:

```
<rule>
    <prem>
         <p>You want to review rule principles</p>
    </prem>
   <conc>
         <p>You may look at
         <a href="http://www.cs.brandeis.edu/...">Rule-Based Systems</a>
         </p>
```

```
    </conc>
</rule>
```

In the above rule the condition part of the rule starts with *<prem>* and ends with *</prem>* tags. Likewise the action part of the rule starts with *<conc>* and ends with *</conc>* tags. The rest of the syntax in the rule is normal XHTML [44] syntax.

2.2.8.2 Simple Rule Mark-up Language (SRML)

The Simple Rule Mark-up Language [11] is an XML-based rule language format initiated by the ILog software vendor company. SRML describes a generic rule language consisting of a subset of language constructs common to forward-chaining rule engines. As it does not use constructs specific to a proprietary vendor language, rules specified using it can easily be translated and executed on any conforming rule engine, making it useful as an interlingua for rule exchange between Java rule engines.

Here is an example of a SRML rule [40]:

```
<rule name="Discount">
    <conditionPart>
        <simpleCondition className="ShoppingCart" objectVariable="s">
            <binaryExp operator="gt">
                <field name="purchaseAmount"/>
                <constant type="float" value="100"/>
            </binaryExp>
        </simpleCondition>
    </conditionPart>
    <actionPart>
        <modify>
            <variable name="s"/>
                <assignment>
                        <field name="discount"/>
                        <constant type="float" value="0.1"/>
                </assignment>
        </modify>
    </actionPart>
</rule>
```

In the above rule the condition part of the rule starts with *<conditionPart>* and ends with *</conditionPart>* tags. Likewise the action part of the rule starts with < *actionPart* > and ends with </ *actionPart* > tags. The above rule can be written in traditional terms as follows:

```
IF
    the total purchase amount of a shopping cart is > 100$
THEN
    Set the discount for the shopping cart to 0.1%
```

2.2.8.3 In-house representations

One of the limitations of the existing standard rule languages is their minmal support for describing actions on the right-hand (RHS) side of a rule. The RHSs can be very complex and working with the confines of a standard representation can be very limiting. Currently most vendors using XML and rules use in-house XML representations. In-house representations are tailored for the vendor's specific application. In-house XML representations can easily be transformed into standard formats via XSLT.
An example of an in-house XML format is Ilog XRL (eXecutable Rule Mark-up Language). XRL is a business rule interchange format, that conforms to a Document Type Definition (DTD) provided by Ilog JRules. It allows users to export business rules in an XML format so that they can load the rules into an external application, or so that they can be processed with any XML tool.

3. Portal Platform Architecture and Rule Engine Component

The Info Asset Broker provides a system platform for the construction of enterprise portal in Internet as well as corporate knowledge portals in intranet. It is described in more details in [42].

In the following chapters I will describe the InfoAsset Broker, WIPS IT 1.1 portal and the WIPS rule management architectures. Furthermore, I will explain how these systems fit together.

3.1 Portal platform architecture

The InfoAsset Broker architecture

The info Asset broker is platform-independent portal software written entirely in java. It delivers information regardless of format or medium and allows file systems, content management systems and databases to serve as information repositories. A combination of these repositories can be used as required.

The diagram 3.1 shows the architecture of the InfoAsset Broker.

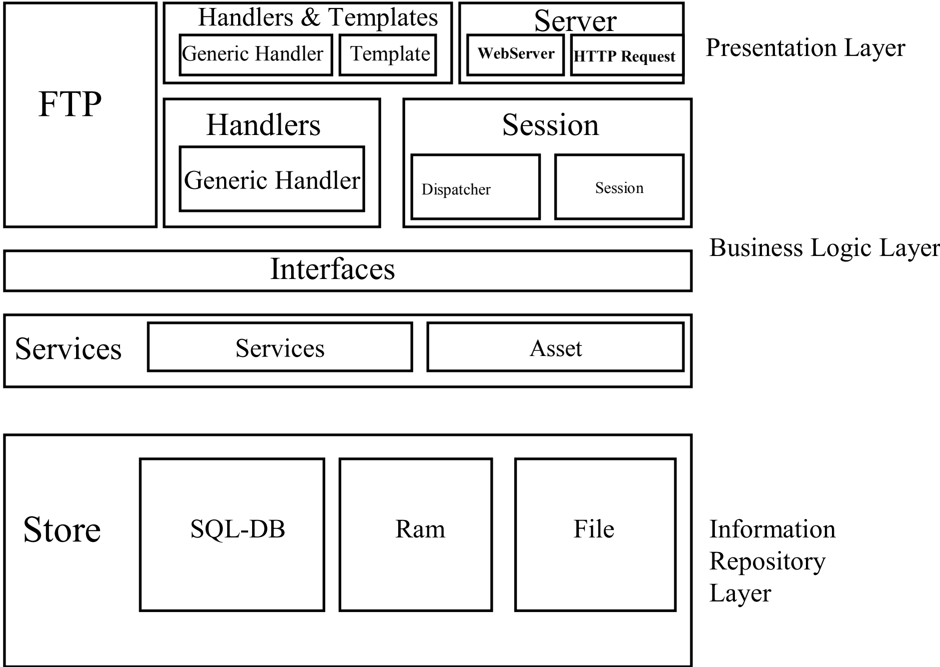


Figure 3.1: The InfoAsset Broker Architecture

The InfoAsset Broker has a 3-tier client server architecture, i.e. it consists of the presentation, business logic and information repository layers. The presentation layer is made up of templates and handlers. They are used for authentication purposes and transport parameters in handler invocation chains.

In the business logic layer, the infoAsset Broker uses handlers to define business process fragments. Business processes are not supported as first-class objects themselves, which is a limitation of the portal platform. The business layer is mainly made up of business objects called *assets*. Sessions objects are used to track consecutive requests made by a user.

The information repository layer mainly provides storage facilities that abstract from concrete persistence mechanisms for the portal system, e.g. SQL-DB, File-System and Content Management Systems.

I will now explain the main technical concepts of the InfoAssetBroker architecture:

- Template
- Handler
- Asset

Templates

A template is a single user interface page. It can be easily created and changed using current design tools such as Macromedia Dreamweaver. It contains static content as well as placeholders to allow creation of dynamic content. The associated handler provides the dynamic content.

Handlers

A handler is a process fragment component. Incoming user requests are dispatched to different handlers. Each handler can handle a specific request type, A handler enacts a fragment of a business process, as it might create and delete business objects, change the state of business objects, and it can send a response page to the user. In order to create dynamic pages, the handler makes use of the portal's template engine, which allows creating dynamic pages.

Assets

To represent entities, i.e. persistent stored information objects in a portal, the platform provides business objects defined as *Assets* that can be, for example, groups, persons, documents Developers can define new Asset types, following the application and domain-specific requirements.

3.2 The Existing Rule Management System Model

I will now look in more detail at the conceptual architecture of the Rule Management System for the InfoAsset portal platform.

The following architectural diagram describes the concepts identified in a rule management system. It further shows that a rule management system consists of the execution and integration layer, a generic rule editor, and a customized management GUI. All the components are interconnected by clearly defined interfaces.

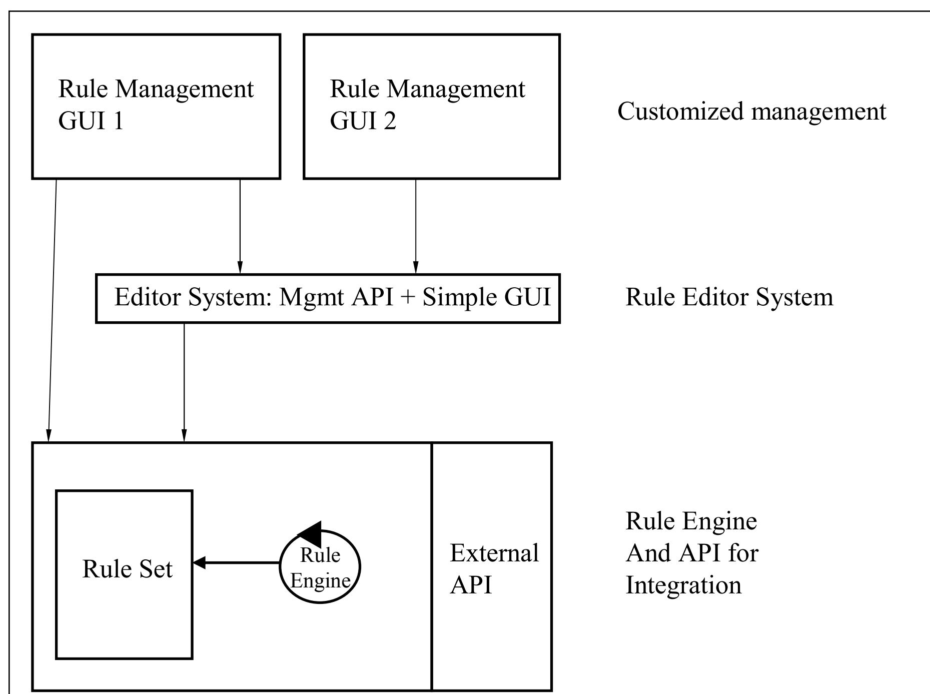


Figure 3.2: The Architecture of a general Rule Management System [12][37]

External API

Rule engine and the external API form the bottom layer of a rule management system.
The external API provides an interface to external systems. The API provides the bridge between external system and rules management system. The interface allows management of rules. The API provides the integration point for the rules management system into the application architectures.

Rule Editor Layer

A good rules management system allows the business logic of a system to be specified external to the system itself. Rules can be changed directly by rule maintainers and editors. Many rules management system provide natural language or wizard-style GUIs for designing rules, allowing rule editors e.g. product managers or business analysts to actually specify the logic. However most rule editor systems are not specific for a domain, so rule maintainers must learn the syntax and structure of the rule language.

Rules Management GUI Layer

A Rules Management Graphical User Interface (GUI) allows rules to be safely constructed and modified by the rule editor with the right data entry controls. The GUI hides the structured rule language and the programming involved. It also provides user-specific and application-specific customization.

For details about rule engines, see chapter 2.

3.3 Integrated Rule Management System

I will now describe the system designed and realised following the architecture defined in chapter 3.2. This system has been realized in the context of the WIPS project. The WIPS IT 1.1 rule management system consists of the following components as described in chapter 3.2:

- The rule editor: Realized as handlers and templates
- The rule set: the rule space
- The rule engine: This is the heart of the rule management system for rule execution
- Context: A context defines the dynamic binding of terms and expression to variables
- Business objects: These are application objects needed for rule execution

The diagram below shows the main components of the WIPS 1.1 Rule Management System.

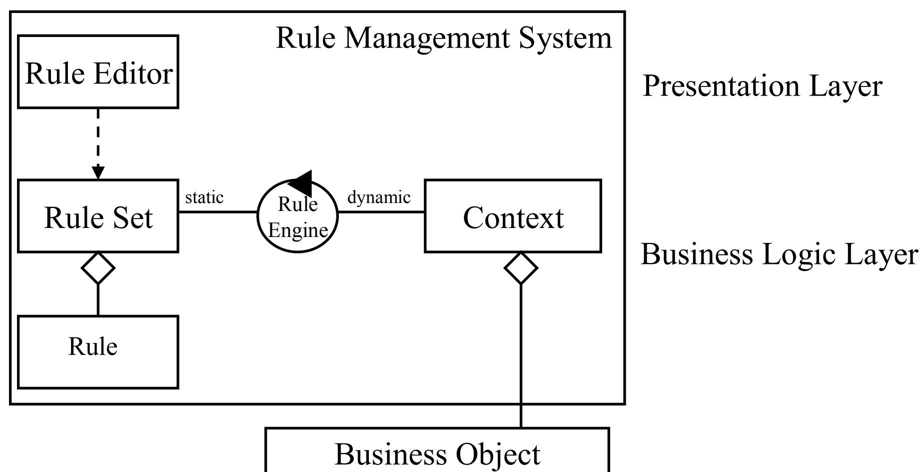


Figure 3.4: Integrated Rule System Model

The diagram below shows how the WIPS Rule Management System fits in the overall InfoAssetBroker architecture.

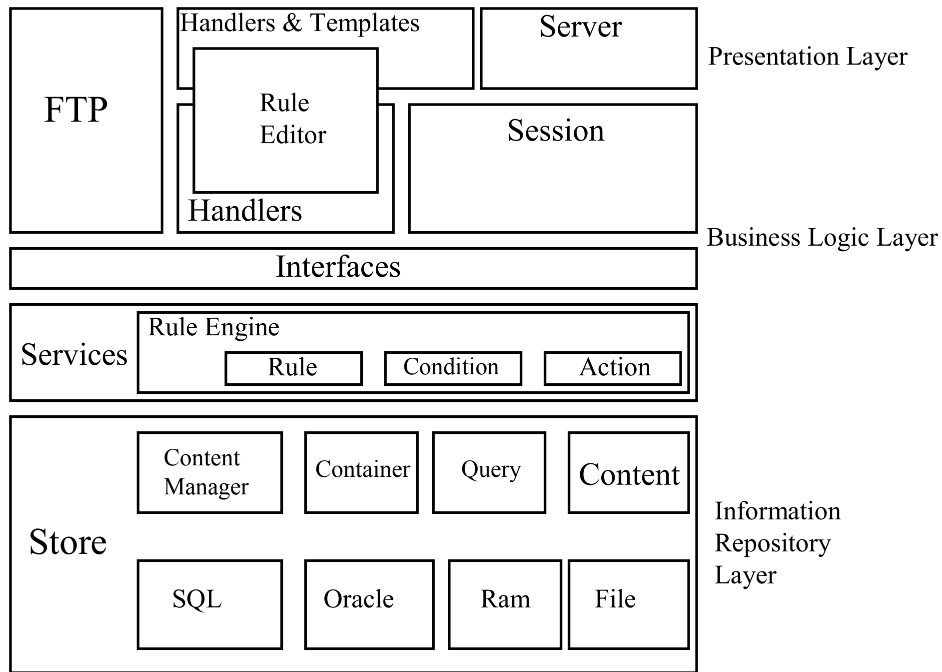


Figure 3.4: The InfoAssetBroker and the WIPS Rule Management System Architecture

WIPS Rule Structure

The diagram below shows that a *rule* in the BrokerRL is composed of a *condition part* and an *action part*. Literals and variables are specializations of *terms* .In turn terms are combined together by *comparison operators* to form a simple condition. Logical operators are used to combine conditions together to form a complex condition. An *action* is composed of terms *methods invocation on terms, usually on variables or* assignments (not shown in the diagram).

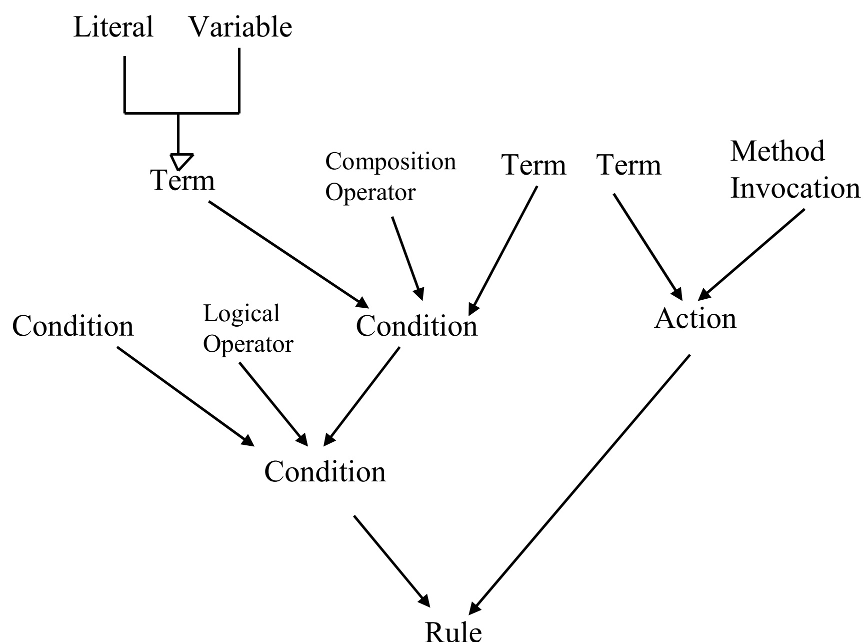


Figure 3.6: BrokerRL Rule Structure

I will now describe the elements of a BrokerRL rule with the help of UML diagrams. I will add a few notes to every diagram to make it clearer.

BrokerRL Rule

The diagram below shows that a BrokerRL rule is of the event-condition-action type (ECA). A rule class is composed of the condition and action class. The diagram also shows that a rule is additionally composed of an enabling condition class.

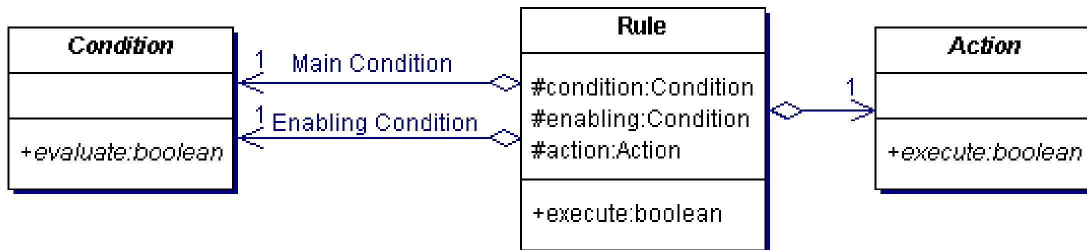


Figure 3.7: BrokerRL rule structure

Conditions

As a loose definition, we might say that a Pattern is a coded expression that is capable of designating one or more objects of that kind. Each condition implements the method evaluate that returns a boolean value. The Condition class is an abstract class, and each its concrete subclass defines a specific condition type. Here are typical conditions:

Simple Condition

The Simple condition class is a concrete subclass of condition. Its evaluate() method returns the evaluation of the operator with the terms.
The syntax of a simple condition is *LeftTerm Operator RightTerm.* An operator to form a simple condition combines two terms. Let me make this clear with an example:

```
Ship.Classification == MC
```

In the above example the terms *Classification* and *MC* are combined by the operator == to form a simple condition.

Complex Condition

The ComplexCondition class is an abstract subclass of the condition class. It takes two conditions as inner conditions. The two inner conditions are logically connected by the evaluate() method. As the example and syntax below shows two conditions *Ship.Classification == MC* and *Flag == German* are combined together by the logical operator AND to for a complex condition. In other word, two simple conditions are combined by a logical operator to forms a complex condition.

Syntax: Condition LogicalOperator Condition
Example: (Ship.Classification == MC) AND (Flag == German)

NOT Condition

The NOT condition wraps an inner condition, thus the evaluate() method returns the inverted inner condition evaluation. As the example and syntax below shows, the NOT Condition is the negation of the Complex Condition above. Two simple conditions are combined together and negated by the negation logical operator *NOT*.

NOT Condition Syntax: NOT Condition
Example NOT ((Classification == MC) AND (Flag == German))

The class diagram below shows that the SimpleCondition, ORCondition, ANDCondition and NOTCondition classes are subclasses of the condition class

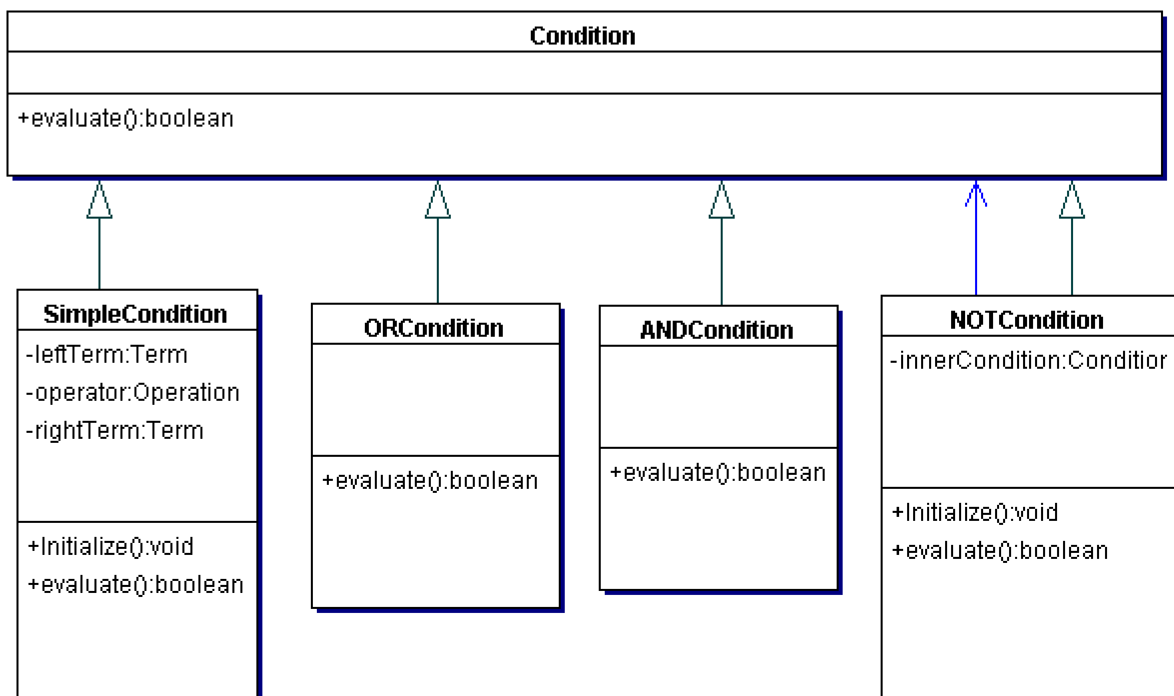


Figure 3.8: The UML Model Of The Condition Class And Its Subclasses

Terms

A term can be a Literal, Variable, an Enumeration, or a Range. Each term is of a particular data type. e.g. Integer, Double, Date, and String. “Literals” are immutable values e.g. “100A5” where else “Variables” are mutable values, that can represent object properties (attributes), like e.g. ship length, ship age, etc. “Enumeration” refers to a set of values while a “Range“ term refers to a term whose value must be within a certain range. The lower and upper bound of a rage term must be of the same data type. The compareTo() method returns –1 if the value supplied is less than term value, 0 if they are equal, and 1 if the value supplied is greater than term value. Terms can now also be of type *Object*, e.g. variable “ship” is of type object.

The class diagram below shows that the literal, variable, enumeration and range classes are subclasses of the term class.

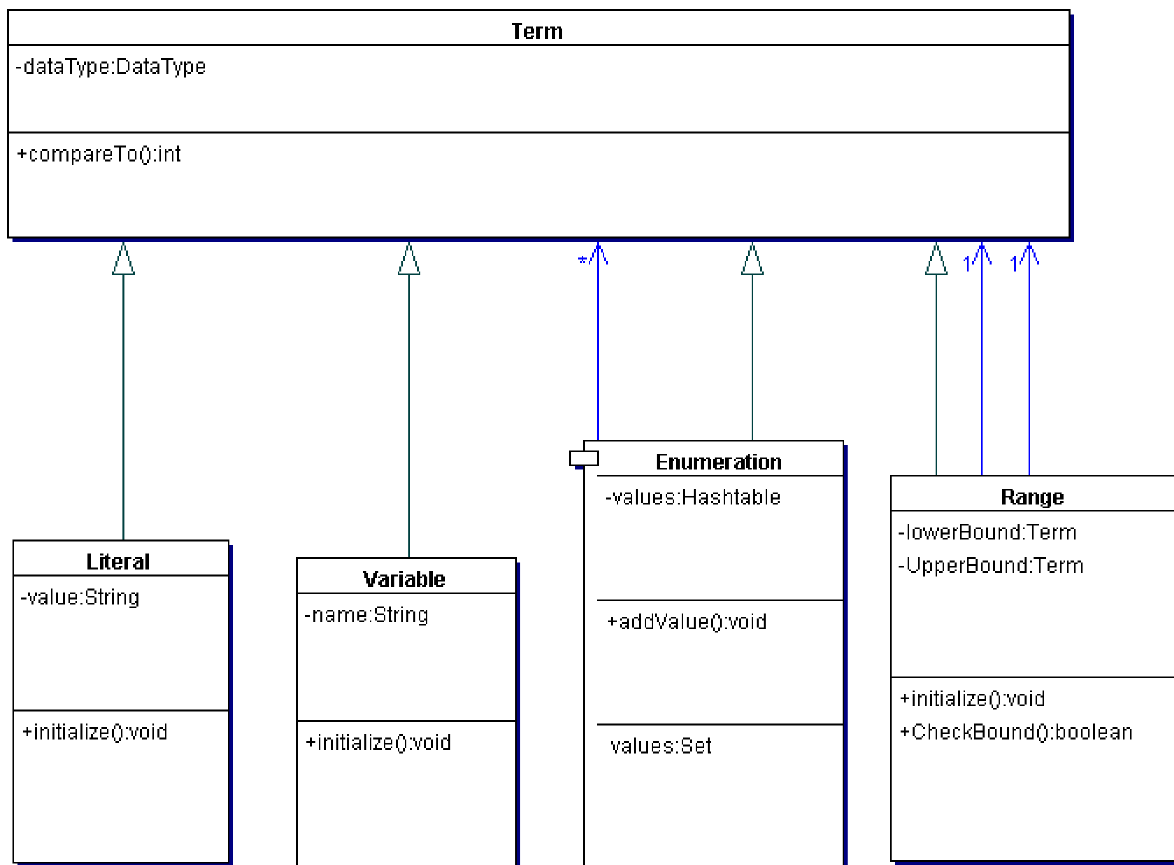


Figure 3.9: The UML Class Diagram Of The Term Class And Its Subclasses

Data Types

Data Types are predefined actual data type. They refer to Integer, Double, Date, and String thus each data type has a name. Each data type implements a *getValue* method that returns a string representation of its value. They also store a list of operators that can be performed on values of that datatype as left-hand terms. To maintain integrity the method checkValidity() return true if the value is valid else false. The method getValue() return string representation of a value while the method getName() returns the name of the data type. These can be represented in an object model as shown in Figure 5.4.

The class diagram below shows that the IntergerType, TextDataType, BooleanDataType, DateDataType, DoubleDataType and the ObjectType are subclasses of the DataType class.

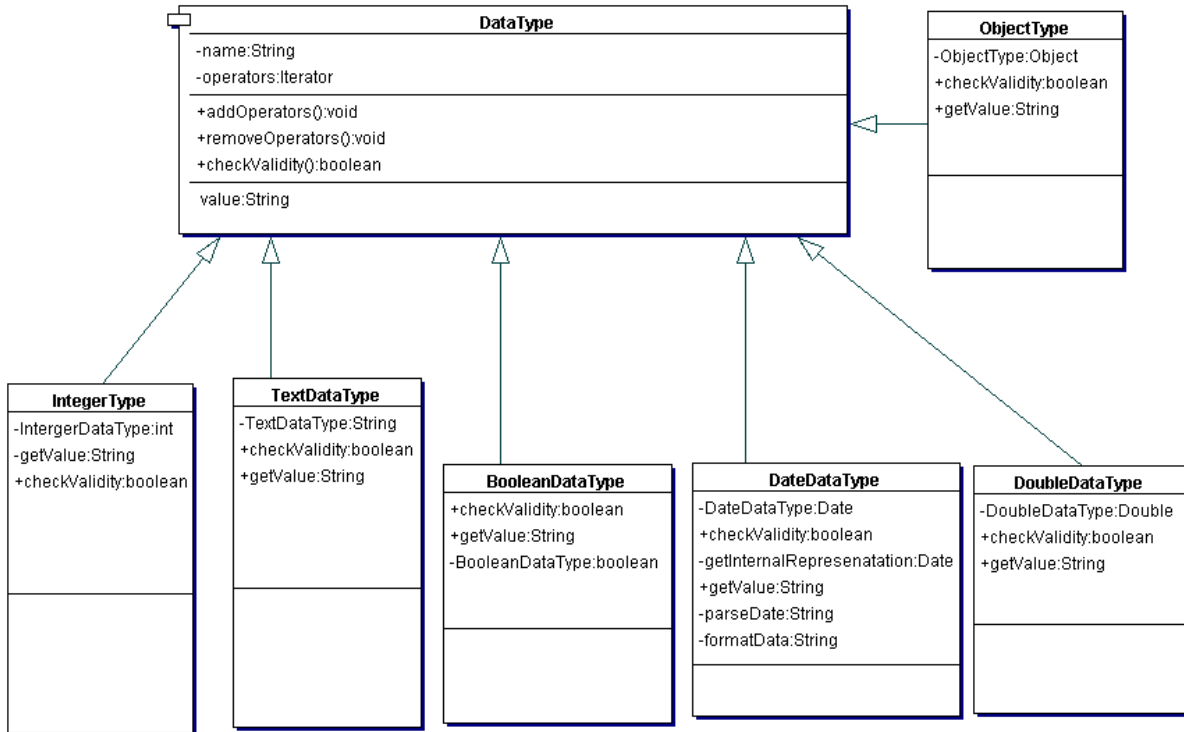


Figure 3.10: The UML Class diagram of The Data Type Class And Its Subclasses

Operators

Operators are needed for comparison purposes. Each Operator has a name e.g. "==", "<", "<=", ">", ">=", and "IN"
The method evaluate() takes two terms, evaluates them with respect to the chosen operator and returns a Boolean value.

The class diagram below shows that LessOperator, UnEqual, LessEgualOperator, RangeOperator, GreaterOperator, GreaterEqualOperator and EqualOperator are subclasses of the *AbstractOperator* class.

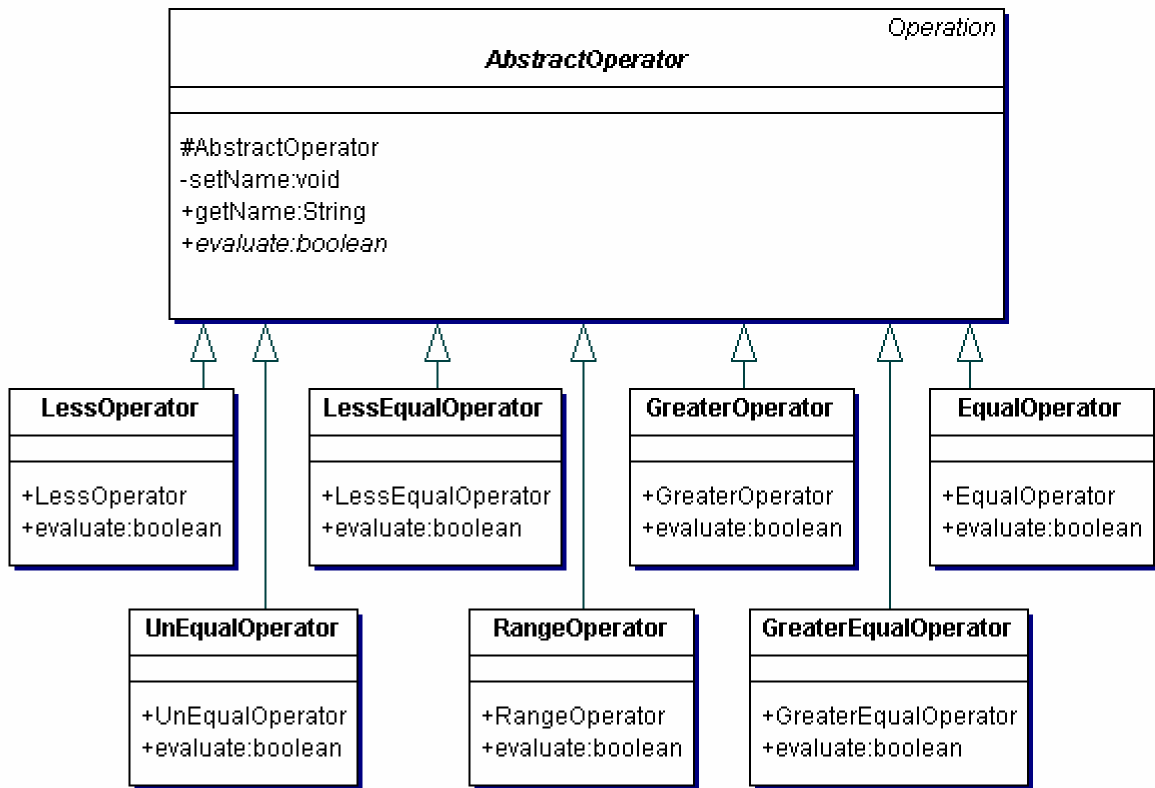


Figure 3.11: The UML class diagram for Operator and related concepts

Actions

Actions are description of operations to be performed whenever all patterns in a condition are successfully matched. An Action is a statement to be invoked / executed when the condition part of a rule is successfully matched.

Changes made to the latest version of the WIPS Rule Management System

Like any other software product, the WIPS Business Rule Management System has undergone a lot of changes. In the latest version of the BrokerRL an action can be:

- Method invocation on a term of type object, e.g. resutlist.addSurveyItem().
- Assignment: Assigment of a term to the result of a method invocation on a term of type object, e.g. shipSize = ship.getSize().

Furthermore, actions have been reworked to use JXPath expressions for method call and assignment. Variables are now mapped to JXPath expressions on evaluation context.

Variables are now mapped to JXPath expressions on evaluation context

Actions have been reworked to use JXPath expressions (for method call and assignment). JXPath expressions are now used to address objects and their properties. I will explain the new phenomena with the following rule:

```
IF
      ShipAge > 8 AND ShipAge < 600
THEN
      Add SurveyItem313 to SurveyItemList
```

Here “ShipAge” is a term name and the corresponding Java expression is “ship.getAge()”. This expression takes the form “/Ship.age” in JXPath. Similarly, the term “Ship” is expressed as “/Ship” in XPath. SurveyItemList is mapped on to /SurveyItemList.

4. Business Rule Engine Products and Standards

In this chapter, a number of rule engines will be examined and evaluated. There are many business rule engines on the market, both open source and commercial. Here is a list of the most popular commercial business rule engines:

- JRules from ILOG[16]
- Advisor from Brokat[19]
- OPSJ from Charles Forgey[20]
- QuickRules from Yasu Technologies[27]
- CommonRules from IBM alphaworks[26]
- exteNd Director from Novell[28]
- ACQUIRE from acquired Intelligence[29]

The list of the most popular open source rule engines is as follows:

- JESS (Java Expert System Shell) from Sandia National Labs[15]
- Mandarax[18]
- CLIPS from Gary Riley[17]
- InfoSapient[26]

4.1 Business Rule Engine Selection Criteria

There are many criteria to be considered when selecting a business rule engine to be used in a software project. But the main driving force in this decision should be the application requirements. The other aspect to consider is the vendor of the business rule engine. The elements to consider in this regard are:

- Consultation availability
- The financial stability of the vendor
- Size of the vendor company
- Consultation availability

The rule engine features that I considered and found important in my project are:

- Support for standard XML Rule Format
- Licensing (R&D)
- Long-Term use at the research department STS [30] and for other research projects

Taking into consideration of the above criteria I decided to investigate ILog JRules (commercial product) and JESS (open source product) further. I will use ILog JRules to implement my prototype business rule engine integration. Therefore, I will only introduce the other rule engine products.

4.2 ILOG JRules 4.5 Product

ILOG JRules [16] is a business rule-based management system (BRMS) software package. Its rule development environment supports three rule formats: BAL (Business Analyst Language) – an IF-THEN-ELSE rule format, TRL (Technical Rule Language) - a powerful language intended for use by technical staff, and Decision Table rules, where rules input and outcomes are specified in a "decision table" format.

ILOG JRules helps to develop new or adapt existing Java applications that utilize rules. It is a high performance rule engine for the Java platform. This product enables software developers to create applications that can be maintained with minimal effort. It allows developers to combine rule-based and object-oriented programming to add rules to new and existing applications.

The ILOG JRules product includes a Java rule engine and a rule kit. The rule kit consists of:

- Rule Builder: It is a fully integrated graphical environment for developing, debugging, deploying and managing business rule applications.
- Business Rule Repository: A central location for storing business rule metadata. It is extensible and based on industry standards. It is the foundation for extensive business rule management functionality and can be persisted to files or databases. It is also the foundation for extensive business rule management functionality.
- Business Rule Management: This is made up of tools for performing business rule queries, viewing business rule history, assigning and tracking business rule versions, controlling access to business rules, and managing business rule status and effectively enable the management of business rules across the enterprise.
- Business Rule Languages: Is a predefined set of business rule languages that can be used or customized to facilitate end-user access to business rules.
- Business Rule Editors: These are editors that enable non-technical user to create business rules using a point and click interface with cut and paste, pop-up windows and drop-down list boxes.

It is packaged as a set of Java class libraries. The rule engine is thread safe and optimized for Java, and provides an API (Application Programming Interface) that gives developers control over the engine. Rules can be compiled directly to Java code or optimized automatically by the rule engine. Rules are automatically optimized to maximize performance and make better use of application resources, see the Rete Algorithm for optimization in chapter 2.

ILOG JRules rule management system has the following features that are important for my work:

1. *The Programming interface:* JRULES has a set of graphical tools such as the rule editor for rule editing and the debugger for rule debugging.

2. *The Programming language:* JRULES does not have a dedicated programming language but directly uses JAVA. The inference process can directly manipulate JAVA and XML objects.

3. The *Rule engine:* The JRULES rule engine uses the RETE algorithm for rule execution optimization. It is a forward chaining rule engine, applying instantiated rules on working memory objects. Rules are selected considering priority and particular criteria are used to resolve conflicts when several rule instances are candidates for firing at the same time. For more information about the Rete algorithm and forward chaining see chapter 2.

4. *The Rule set syntax (ILOG Rule Language rule set):* I will explain the syntax of the of an ILOG rule with an example. As can be seen in the example below a rule set is composed of:

```
fishAquariumAdvanced.ilr

import demo.fishAquariumAdvanced.*;
import ilog.rules.engine.*;
ruleset fishAquariumAdvanced
setup {
      assert fish(blue, clown);
      assert fish(red, angel);
      System.out.println("Initial Actions:");
      System.out.println("Add a blue clown fish.\n");
      System.out.println("Add a red angel fish.\n\n");
};
rule Clown1 {
   priority = maximum;
   when
   {
      ?fish1: Fish(?f1: type; ?f1 == clown);
      ?fish2: Fish(?f2: type; ?f2 == angel);
   }
   then
   {
      retract ?fish1;
      retract ?fish2;
      assert Fish(green, shark);
      System.out.println("Fire Rule:");
      System.out.println("Remove a "+?fish1.getColorName()+"
                  "+?fish1.getFishName()+" fish.");
      System.out.println("Remove a "+?fish2.getColorName()+"
                  "+?fish2.getFishName()+" fish.");
      System.out.println("Add a green shark fish.\n\n");
};
rule Clown2 {
   priority = high;
   when
   {
      ?fish: Fish(?f1: type; ?f1 == clown);
   }
   then
   {
      assert Fish(yellow, trigger);
      System.out.println("Fire Rule:");
      System.out.println("Add a yellow trigger fish.\n\n");
   }
}; }
```

As can be seen in the example above a rule set is composed of a rule set name, import statements, and a collection of rules. In this example the rule set name is fishAquariumAdvanced. The import statement is used for importing necessary Java classes. For more details on this topic see the ILOG Rule Language Reference [16].

4.3 Java Expert System Shell – JESS

JESS (The Java Expert System Shell) [15] is an expert system shell written entirely in Java. JESS was originally a copy of the kernel of CLIPS[17], but now it has been standardized on a Java-based system and at present it has many features that differentiate it from its parent. JESS is used to increase the decision-making ability of Java applets and applications. JESS can be operated with commands written in the Java-language without compiling any Java code, also rules can access Java objects and vice versa. JESS uses the Rete-algorithm to process rules.

JESS is a shell for rule-based systems. Jess's purpose is to continuously apply a set of *if…then* statements (rules) to a set of data (working memory).

Rules in JESS are defined using the *defrule* construct. JESS rules are executed whenever their *if* parts are satisfied. Programming variables in JESS are atoms that begin with a question mark.

A JESS rule looks like this:

```
(defrule library-rule-1
      (book (name ?x)  (status late) (borrower ?y))
      (borrower (name ?y)  (address ?z))
      =>
      (send-late-notice ?x ?y ?z)
)
```

The above rule might be translated into pseudo-English as follows:

```
Library rule #1:
IF
    A late book x exists, borrowed by a person y
AND That borrower's address is known to be z
THEN
      Send a late notice to y at z about the book x
END
```

In this JESS rule example, the name of the rule is library-rule-1. The first part contains the conditions to be tested and the second the actions to take if all the conditions specified in the first part are verified. The patterns in the conditional part are used for selecting objects in the working memory that match the patterns.

A JESS rule has two parts, separated by the "=>" symbol, which can be read as *then*. The first part consists of the *left-handside* patterns *(choice false)* and *(msg type Request).* The second part consists of the *right-hand-side* actions. The *left-hand-side* of the rule consists of patterns that are used to match facts, while the *right-hand-side* contains function calls.

The JESS rule engine can be integrated in any JAVA application.

JESS characteristics that are important for my work are as follows:

1. *Programming interface:* JESS programming interface is represented by a simple console. There are no tools for object management or rule management and the language uses a complex LISP like syntax. The development of applications is difficult and not suitable for non-skilled users. The language also has a Java API, which make it possible to manipulate its engine via Java code.

2. *Language syntax:* As in the LISP language, the JESS language main concept is that of atom. The head of a list represents a specific function and the elements in the tail represent the function arguments. A variable is an atom in which has ? as first character, and may contain a single atom, a number or a string. Rules evaluation modifies facts in the working memory.

3. *Rule engine: The* JESS rule engine uses the RETE algorithm. This algorithm grants better performance than standard pattern matching exhaustive techniques. Rules may be applied in forward and backward chaining. In forward chaining two different strategies may be applied for conflict resolution: depth or breadth. In the depth strategy the last activated rule is applied before the other ones. In the breadth strategy rules are applied in the activation order.

4. Inference on objects: JESS inference engine can not directly interact with java objects so it is necessary to build an interface between JESS objects and JAVA objects. This characteristic makes JESS application development quite difficult.

- JRules from ILOG[16]
- Advisor from Brokat[19]
- OPSJ from Charles Forgey[20]
- Eclipse from Haley Enterprise[21]
- PegaRules from PegaVision[22]
- QuickRules from Yasu Technologies[27]
- CommonRules from IBM alphaworks[26]
- ACQUIRE from acquired Intelligence[29]

The list of the most popular open source rule engines is as follows:

- JESS (Java Expert System Shell) from Sandia National Labs[15]
- CLIPS from Gary Riley[17]
- JTP from Stanford University[23]
- drools from Werken Company[24]
- InfoSapient[26]

4.4 Blaze Advisor from Brokat

Blaze Advisor is a 100% Java business rule management system . It has graphical user interface environment for developing rule-based systems. It is designed for more technically oriented developers, although it can also be use by business people.

Rules are developed using the Blaze Advisor Structured Rule Language (SRL), a proprietary language that allows end users to define business rules using common business terms. With SRL, rules can be developed and modified using one of three syntax options. Two of these options are for business users, enabling them to use common business terminology. For developers, rules can be coded in an object-oriented manner similar to Java.

4.5 OPSJ from Charles Forgey

OPSJ is a tool for adding rules to Java software systems. The OPSJ rule language is compatible with all standard Java programs. Its rule engine is written entirely in Java. A rule-based system developed using OPSJ can be run on any machine that supports Java version 1.3 or later.

4.6 QuickRules from Yasu Technologies

QuickRules a Java rule engine that can be used to design, develop, and manage business rules. It enables architects, business users, and business managers to design, develop, deploy, and manage business. It also offers support for using XML in business rules.

4.7 CommonRules from IBM alphaworks

CommonRules is a rule-based framework for developing rule-based applications with emphasis on separation of business logic and data, conflict handling, and interoperability of rules. It is a 100% java product. CommonRules can be integrated with existing applications at a specific point of interest, or it can be used to create applications composed only of rules. CommonRules uses a rule language called CLP (Courteous Logic Program).

4.8 ACQUIRE from acquired Intelligence

Acquire is both an authoring tool for rule-based systems and an expert system shell. It provides a step-by-step method for acquiring and structuring knowledge. It can also be embeded in other software applications.

4.9 CLIPS from Gary Riley

CLIPS is an expert system tool for constructing rule-based and object-oriented expert systems. It is written entirely in C. The standard version of CLIPS provides an interactive, text oriented development environment, including debugging aids, on-line help and an integrated editor.

4.10 InfoSapient

InfoSapient is a rule engine that allows business analysts to enter and maintain business rules using natural languages. The resulting automated decision-making is similar to the way human experts make decisions in a particular domain. Many variables may be considered simultaneously and used to weigh risks and opportunities in order to arrive at the best course of action. Given ambiguous situations that are typically found in business processes, InfoSapient will arrive at the most appropriate course of action. .

4.11 Rule Engine Standards

Various commercial off-the-shelf products (other than application) servers can be designed to work together with rule engines. Each rule engine has its own programmer's interface. The Java Rule Engine API [34] , defined by the javax.rules package, is a standard enterprise API for accessing rule engines, currently being developed by a consortium of rule engine vendors under the Java Community Process. The javax.rules package will allow the host program to interact generally with multiple rule engines just like the Java Database Connectivity (JDBC) API [35] makes it possible to write vendor neutral database programs.

The API will include mechanisms for creating and managing sets of rules; for adding, removing, and modifying objects in working memory; and for initializing, resetting, and running the engine. However, the javax.rules API will not specify a standard rule language.

5. Rule Engine Component Integration

In this chapter I will investigate the various possible ways of using the ILOG rule engine to harness the full power of the BrokerRL rule language expressiveness. I will investigate the possibility of replacing the WIPS rule engine with the ILOG rule engine and the possibility of making the ILOG rule engine work together with the WIPS rule engine. One of these possibilities will be chosen, realized and evaluated.

Observations

The BrokerRL rule language has an expressiveness that is higher than what the WIPS Rule Engine can handle (Figure 5.1). In short rules can be defined in the language that cannot be handled by the WIPS rule engine. This is because the rule language is suitable for normal interdependent Event-Condition-Action (ECA) rules, while the rule engine can only handle IECA (Independent-Event-Condition-Action) rules [12]. Interdependent rules are rules that operate on the same objects in the working memory. Execution of an interdependent rule causes changes, e.g. state change, to objects referenced by the condition parts of other rules. Standard rule engines like ILOG JRules can handle both rule types.

The diagram below shows the language expressiveness of the standard rule languages compared to the BrokerRL rule language. It also shows that the WIPS Rule Engine cannot handle the full expressiveness of the BrokerRL rule language.

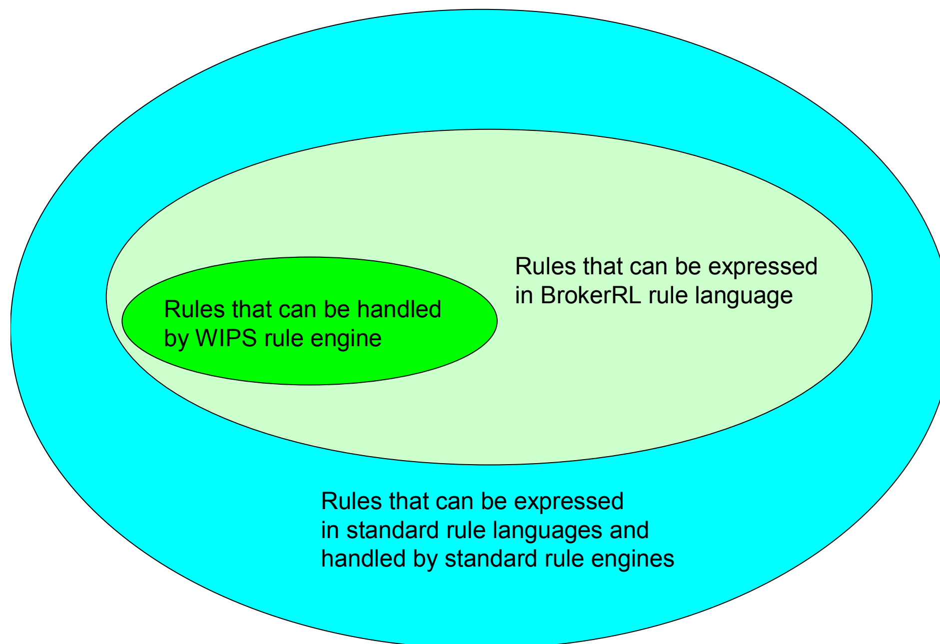


Figure 5.1: Rule Language Expressiveness Classes

We would like to make use of the full power of the BrokerRL rule language. In order to achieve this objective, a standard rule engine will be used in conjunction with the WIPS IT 1.1 Rule Management System.

Two approaches could be thought of to solve this:

1. The WIPS rule engine can be replaced by a standard rule engine either commercial or open source, see figure 5.2.
The diagram below shows that the WIPS Rule Engine is replaced by a standard rule engine.

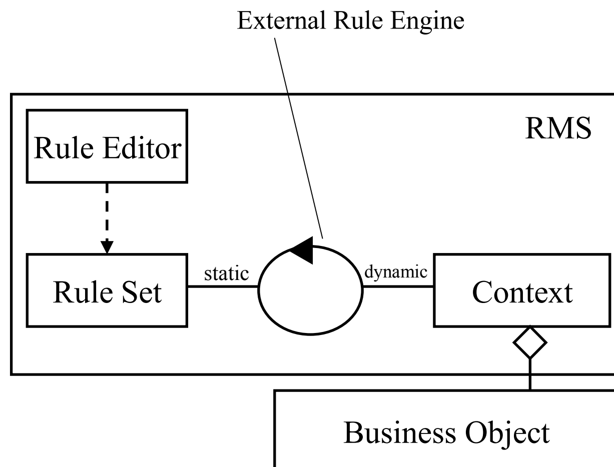


Figure 5.2: Replacing the WIPS Rule Engine With an External Rule Engine

2. If it can be determined that for a given rule set, the rules are either interdependent or independent, evaluation of interdependent rules can be delegated to an external standard rule engine. This implies that the rule set and its evaluation context must be sent to a standard rule engine for evaluation, see figure 5.3.

The diagram below shows the WIPS Rule Engine working in conjunction with an external rule engine on the same rule set and evaluation context.

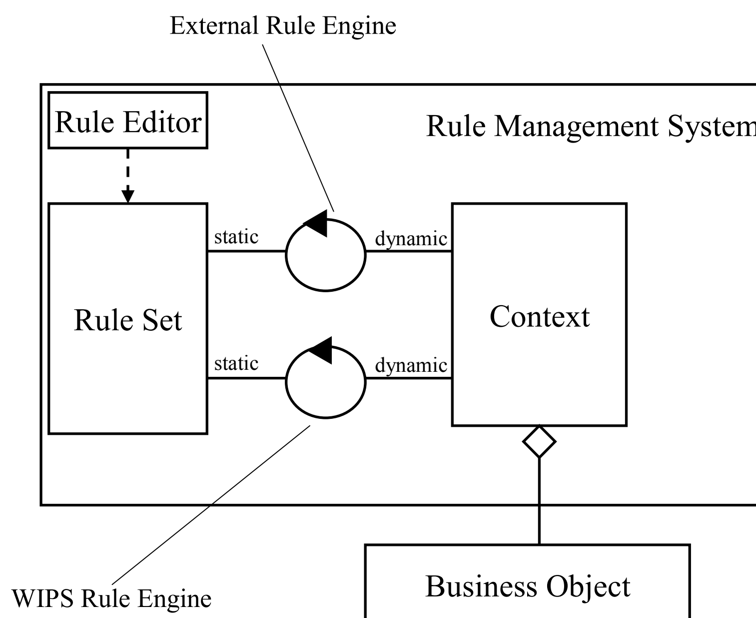


Figure 5.3: The WIPS Rule Engine Working With A Standard Rule Engine

As the WIPS rule engine is tightly integrated with its portal platform, InfoAsset Broker, solution 2 above is the chosen solution.

The existing rule engine shall still be used for appropriate tasks while the additional one shall be used for tasks where BrokerRL rules are interdependent.

The external rule engine is more difficult to integrate because of the following reasons:

- The rule sets must be mapped to an appropriate format for the external rule engine
- The context (variable bindings) must be provided to the external rule engine in a form that it can handle
- Variables must be published to the external rule engine's working memory.

Assuming that these problems are solved, the external rule engine can work on the business objects contained in the rules defined in the rule set and thus execute the business rules.

5.1 Design Options

There are four options to address the above problem. I will analyse each potential option with respect to its advantages and disadvantages. I will then choose one solution for integration and evaluation.

General Process Steps

The following are the general process steps to be taken in all the options. In all the options both the WIPS rule engine and the Ilog rule engine are working on the same business, i.e. they share the same working memory. At this stage it is important to note that the external rule engine will only execute if a rule set is found to contain interdependent rules. Rule sets of this kind will be loaded into the extenal rule engine in an appropriate format for execution. It is not necessary to import the results from the external rule engine because both rule engines work on the same business objects (assets). The process of rule format transformation will involve a number of steps for each process. Some processes will operate on rules represented as business objects and others on rules represented as documents. In the portal, rules are represented as business objects. A rule object is made up of a condition object and an action object, see figure below. These objects are instances of IMPRule, IMPCondtion and IMPAction, see figure 5.4.

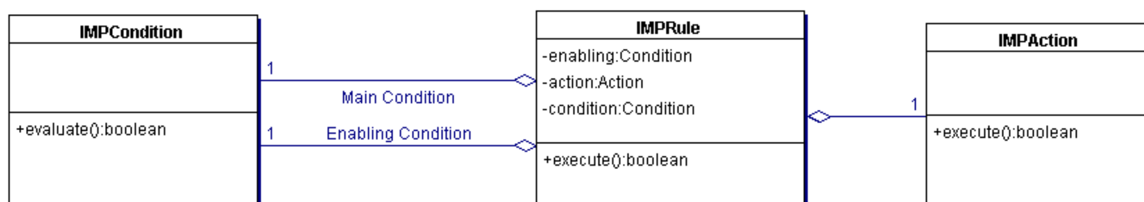


Figure 5.4: The portal rule object composition

In my work, I am going to use the visitor pattern [33] to traverse a rule set structure, extract the rules and their conditions and actions. The purpose of the visitor pattern is to encapsulate a traversal operation to be performed on elements of a data structure. In this way, the operation being performed on a structure can be changed without the need of changing the classes of the elements that are being operated on. Using a Visitor pattern makes it possible to decouple the classes for the data structure and the algorithms used upon them. Each node in

the data structure "accepts" a Visitor, which sends a message to the Visitor that includes the node's class. The visitor will then execute its algorithm for that element. In my case this is equivalent to extracting information from a rule set structure.

Likewise, I will use the builder pattern [32] to convert portal rule objects to ILog rule objects and formats. A Builder is a converter class. The Builder pattern separates the algorithm for interpreting an original format from how a converted format gets created and represented. A software design pattern, the builder pattern is used to enable the creation of a variety of complex objects from one source object. The source object may consist of a variety of parts that contribute individually to the creation of each complex object through a set of common interface calls of the *Abstract Builder* class.

At this stage we have an ILog-conform representation of rules. When the rules are ready for processing, they are put together to form a ruleset, an instance of the *IlrRuleset* class. In Ilog JRules, the IlrRuleset class represents a rule set. The ruleset is finally loaded into the Ilog rule engine, an instance of the *IlrContext* class. IlrContext is the engine class in the Ilog rule management system. The IlrContext, IlrRule and IlrRuleset are the most important classes in the JRules rule engine system.

I will now look the IlrContext, IlrRule and IlrRuleset classes [Figure 5.5] in more detail. The ilrContext class is the rule engine class in ILog JRules business rule engine system. It has methods to control the rule engine and to trigger rules on demand. An object of type ilrContext is created with an associated rule set and working memory. Similarly the ilrRuleset and the ilrRule classes are the rule set and rule classes respectively. An instance of ilrRuleset class is composed of ilrRule objects. The rule set class maintains a collection of rules. The figure 5.5 below shows the ilrContext class with some of its important methods.

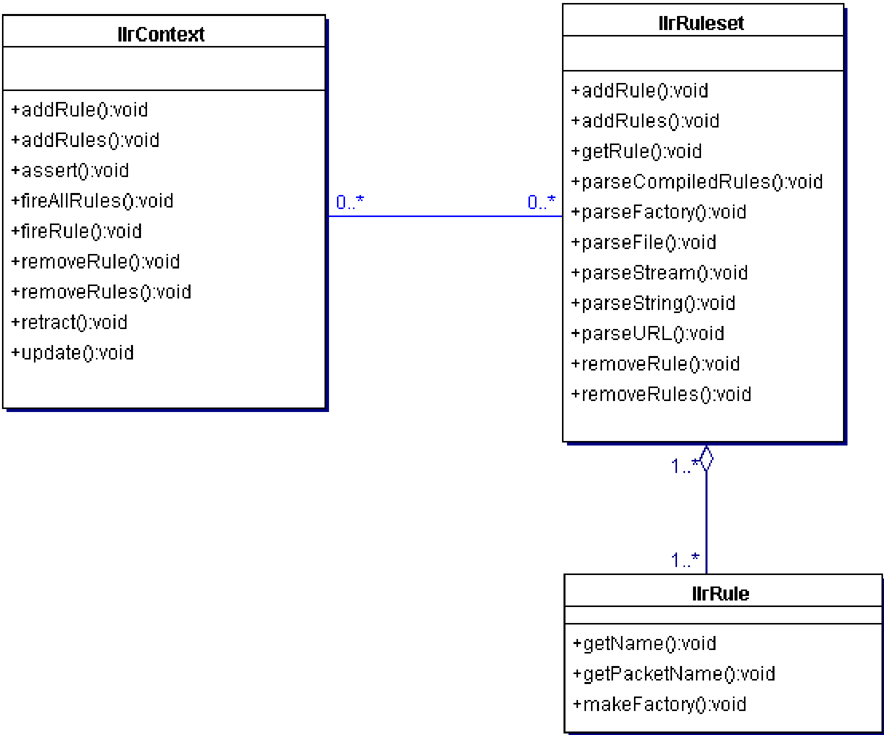


Figure 5.5: The Ilog JRules Rule Engine Class and its Associated Rule Set and Rule Classes

The figure 5.6 gives a general picture of the various options and process steps involved in transforming portal rules to Ilog rules.

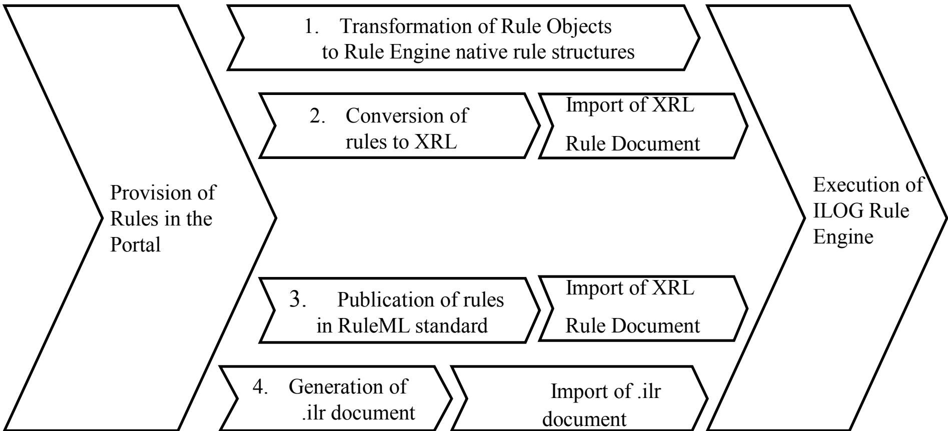


Figure 5.6: Different processes for integrating an external rule engine

5.1.1 Option 1: Rules as business objects

Option 1 process works on rules in the form of business objects. The builder pattern is responsible for generating ILog-specific objects by making use of the ILog's *Factory* class to instantiate the objects.

The rule objects, interdependent rule objects, are mapped on to the corresponding Ilog rule objects, see figure 5.7.

The diagram below shows the conversion of portal rule objects to Ilog rule objects.

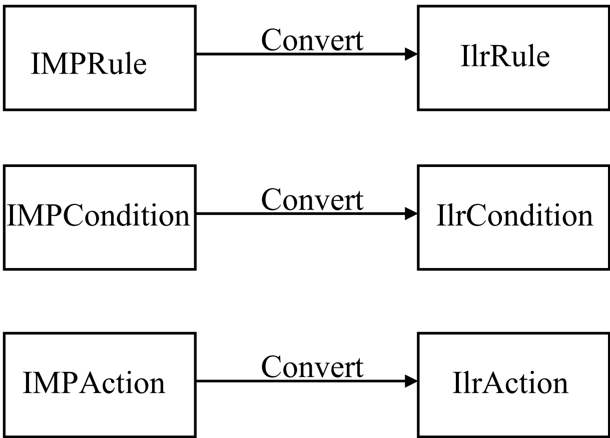


Figure 5.7: conversion of portal rule objects

The *IlrCondition* is combined with the *IlrAction* to form a rule. The *IlrRule* objects are then put together to form a rule set, of type *IlrRuleset*. An instance of the IlrRuleset is finally loaded into the Ilog rule engine.

The figure below shows the whole process of option 1. Portal rules are transformed into ILog rules and finally loaded into the rule engine.

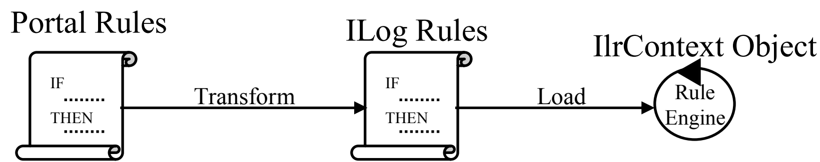


Figure 5.8: Conversion Process steps For Portal Rule objects

Advantages of option 1

- Shortest process due to one to one conversion of rule objects
- Most appropriate solution because portal rules are simply business objects
- ILog rule engine integrated as normal business object in the portal application.

Disadvantages of option 1

- Very difficult to implement due to poor documentation in the ILog API.

5.1.2 Option 2: Rules exchange in Ilog XML Format

In this option rules are loaded into the Ilog rule engine as XML documents. Portal rules are converted into an ILog XML format called XRL (eXecutable Rule Language). The eXecutable Rule Language enables rules to be expressed using XML and directly parsed by the ILog rule engine. The ilrXMLReader is the ILog class capable of parsing XML rules and loading into the Ilog rule engine. Finally the rule engine processes the rules.

The diagram below shows the whole process of rule processing in option 2. Starting from the portal rules, the rules are transformed into an XRL document, parsed by the reader and finally loaded into the ILog rule engine for execution.

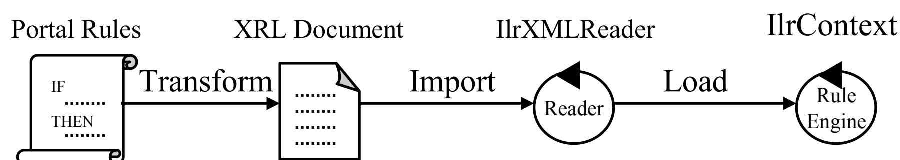


Figure 5.9: Rule processing via XRL XML documents and the irlXMLReader class

Advantages of option 2

- The XRL rule format is well documented in the ILog API
- XRL rule objects are processed just like normal java objects by ILog rule engine.

Disadvantages of option 2

- Many intermediate process steps
- Excessive memory consumption
- XRL is a proprietary ILog XML format
- The ILog rule engine is integrated as an external component into the portal application.

5.1.3 Option 3: Rule Processing Via RuleML and XSLT

Option 3 is similar to option 2 with the exception of a) the creation of standard RuleML representations and b) the intermediate conversion step between RuleML (Rule Mark-up Language) documents and XRL documents via XSLT (Extensible Stylesheet Language Transformations) [31]. XSLT provides an implementation of a tree-oriented transformation language for transmuting instances of XML using one vocabulary into XML instances using any other vocabulary imaginable. The XSLT language is used to specify how an implementation of an XSLT processor is to create a desired output from a given marked-up input. Using XSLT, the generated RuleML documents are converted into XRL documents and the process continues as in option 2.

The diagram below shows rule processing via XSLT and ruleML. Starting from the portal rules, the rules are transformed into a RuleML document, XRL document via XSLT, parsed by the reader and finally loaded into the ILog rule engine for execution.

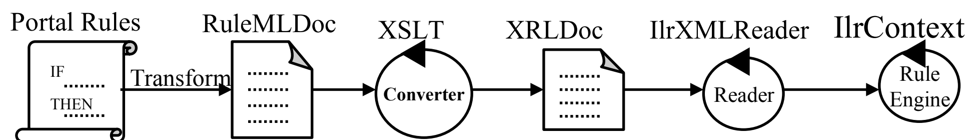


Figure 5.10: Rule Processing Via XSLT and RuleML

Advantages of option 3

- RuleML is a standard rule format supported by many rule engine vendors, which allows rules from other sources to be loaded.

Disadvantages of option 3

- Too many intermediate process steps
- Process consumes a lot of memory
- RuleML not yet fully supported by ILog rule engine
- The ILog rule engine is integrated as an external component in the portal application.

5.1.4 Option 4: Rule processing via ILog Rule set files

In option 4, the portal rules are directly converted into an ILog rule set file using the Builder pattern. The rule file is finally loaded into the Ilog rule engine for execution.

The diagram below shows the whole process of rule processing in option 4. Starting from the portal rules, the rules are transformed into an Ilog rule set file and finally loaded into the ILog rule engine for execution.

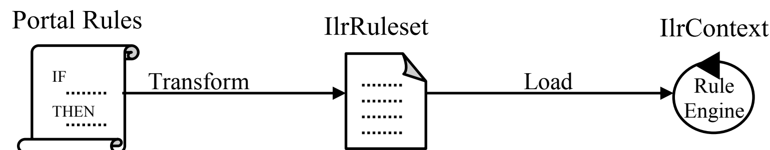


Figure 5.11: Rule processing via Ilog rule set files

Advantages of option 4

- This process is the one recommended by the ILog rule engine vendor
- Full documentation of the process in the Ilog API
- Vast experience in this process by ILog rule engine vendor.

Disadvantages of option 4

- The ILog rule engine is used as an external component in the portal application, thereby making it cumbersome and slow to import and export rule sets between the two rule engines.

5.2 Details of the Selected Integration Option

Upon analysing the four integration options, I decided to choose option 4 for the external rule engine integration. The main driving force for this decision was full support by ILog for this integration option.

I will now describe this option in more detail. For general process steps, the readers are advised to see chapter 5.1 above.

From the portal side of the process, it must be analysed whether rules are dependent or independent thereby determining whether to use the WIPS rule engine or the ILOG rule engine. For this, all business objects referenced from the action parts of rules are collected. Then, all references to business objects in condition parts are checked whether they reference business objects used in action statements. If this holds, the rules are said to be interdependent, otherwise they are said to be independent. However, transitive reachability of objects and the outcome of operations performed on objects cannot be traced there is one limitation in this process. I will clarify this limitation with an example: A method invocation

(an action) on object o1 could have side effects, e.g., state changes, of another object o2, which is used in a rule condition. This dependency is not detected if o2 does not explicitly appear in the rule action.

Portal rules that are dependent are put together to form a rule set and the conditions and actions of their rules are analysed and converted into the ILog rule format (.ilr) structure. For this, the process follows the Design Pattern *Builder* (node-wise generation of target format) as described in the previous chapter.

The objects that are referenced by the condition parts of the rules are put into the working memory of the Ilog rule engine. These objects are removed from the working memory after execution to avoid unexpected alterations on them by subsequent rule engine executions. This requires that the business objects on which the rules work can be identified in their condition and action parts. This is no problem, as the rule model defines variables (Variables of type object) as a specialization of terms where business objects can be identified. Literals (fixed values, e.g. Integer, Boolean, Date and String fixed values) need not be put into the working memory. They are instead inserted into the .ilr file (rule set). Range inclusions such as today in `[1.1.2003 to 31.12.2003]`, are mapped to two comparisons, `1.1.2003 < today AND today < 31.12.2003`. Enumerations, which are further specializations of terms are currently not supported.

5.3 Evaluation

The selected integration option is a good solution since it is fully supported by the Ilog rule engine vendor. However, it has a number of drawbacks. It is cumbersome to import and export rule sets between rule engines. It would be much easier and more effective to handle rule sets at an object level, see integration option 1 above for details on this idea.

It would be advisable to use option 1 in the next integration phase of an external rule engine into the current portal system. This can best be achieved by using the *QuickRules rule engine* [27] instead of ILOG JRules rule engine. The rule engine API is very well documented compared to the ILog one. Besides, QuickRules has numerous examples for external rule engine integration.

6. Summary & Outlook

In this chapter, a summary of the work and an outlook of future work and research will be given.

6.1 Summary

In this work the integration of rule-based component into a portal platform was elaborated.

Rule-based system concepts important for my work have been described. Different types of rule languages supported by rule-based systems have been described. Two types of rule languages were identified, rule authoring and rule execution languages. The most important rule languages in each of the two rule language types were identified.

The InfoAsset Broker portal platform and the WIPS rule-based portal on which my work is based were described. In this regard, the InfoAsset Broker portal platform architecture, the rule management system model for the InfoAsset Broker and the integrated rule management system were described

An overview of the important rule engine products on the market and standards has been given. Two categories for the rule engine products were identified, i.e. open source and commercial products. A number of criteria for choosing a rule engine product were identified. ILOG JRules rule engine was chosen as the commercial product for integration. Likewise, JESS was chosen as the open source product for integration.

Scenarios for general rule support for the InfoAsset Broker portal platform have been analysed and an architecture for general rule support has been defined, realized and evaluated. Four options for general rule support were studied and evaluated with respect to their advantages and disadvantages. The general process steps for the four integration options were identified and described. Each integration option was analysed in detail. A final choice among the various integration options was made for realization and evaluation.

6.2 Outlook

I would recommend that for the next InfoAsset Broker portal rule support customisation a fuzzy logic rule engine be integrated. One such rule engine is the InfoSapeint [26] rule engine. The WIPS case could benefit from support for linguistic variables, e.g. old, new, large and small, as classifiers for business objects. The variables *old, new, large* and *small* are linguistic variables that can best be manipulated using fuzzy logic rule engines. For the given WIPS scenario, fuzzy classifications like old ship could help in the maintenance and administration of rules.

It would be advisable to integrate an open source rule engine such as JESS as an alternative to the commercial ILOG JRules rule engine. This is recommended because open source products tend to be less expensive compared to their commercial counterparts. Communication with the Ilog rule engine vendor proved to be very cumbersome. This would not be the case with an open source product. There is always some expert in the open source community willing to help in case a problem arises.

As described in the evaluation in chapter 5.3, when using a commercial rule engine, I would recommend the *QuickRules rule engine* [27] which provides better support for a business object-level integration of the rule engine component, as it was described in option 1 in chapter 5.1.

Appendix A: SRML DTD

```
<!-- SRML (Simple Rule Markup Language) DTD -->
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT ruleset (rule*)>
<!ATTLIST ruleset
  name NMTOKEN #IMPLIED>
<!ELEMENT rule ( priority?, conditionPart, actionPart ) >
<!ATTLIST rule
  name NMTOKEN #REQUIRED>
<!ELEMENT priority (%expression;)>
<!ELEMENT conditionPart (%condition;)+>
<!ELEMENT actionPart (%action;)* >
<!ENTITY % condition "(simpleCondition | notCondition)">
<!ENTITY % action "(assignment | bind | assert | assertobj
              | modify | retract)">
<!ELEMENT simpleCondition (%expression;)*>
<!ATTLIST simpleCondition
  className CDATA #REQUIRED
  objectVariable NMTOKEN #IMPLIED>
<!ELEMENT notCondition (%expression;)*>
<!ATTLIST notCondition
  className CDATA #REQUIRED>
<!ELEMENT assert (assignment | bind)* >
<!ATTLIST assert
  className CDATA #REQUIRED>
<!ELEMENT assertobj (%expression;)>
<!ELEMENT retract (variable)>
<!ELEMENT modify (variable, (assignment | bind)+)>
<!ENTITY % expression
 "(%assignable; | constant | unaryExp | binaryExp | naryExp)">
<!ELEMENT unaryExp (%expression;)>
<!ATTLIST unaryExp operator (plus | minus | not) #REQUIRED>
<!ELEMENT binaryExp (%expression;,%expression;)>
<!ATTLIST binaryExp
  operator (eq | neq | lt | lte | gt | gte) #REQUIRED>
<!ELEMENT naryExp (%expression;)+>
<!ATTLIST naryExp
  operator (add | subtract | multiply | divide |
          remainder | and | or) #REQUIRED>
<!ELEMENT assignment (%assignable;,%expression;)>
<!ELEMENT bind (%expression;) >
<!ATTLIST bind
  name NMTOKEN #REQUIRED>
<!ELEMENT constant EMPTY>
<!ATTLIST constant
  type (string | boolean | byte | short | char | long
       | int | float | double | null) #REQUIRED
  value CDATA #REQUIRED>
<!ENTITY % assignable "(variable | field)">
<!ELEMENT variable EMPTY>
<!ATTLIST variable
  name NMTOKEN #REQUIRED>
<!ELEMENT field (%expression;)?>
<!ATTLIST field
  name NMTOKEN #REQUIRED>
```

Appendix B: The RuleML DTD

```
<!-- An XML DTD for a Datalog RuleML Sublanguage: Stand-Alone Version -->
<!-- Last Modification: 2001-01-25 -->

<!-- ENTITY Declarations -->

<!-- a conclusion and premise will be usable within 'if' implications -->
<!-- conc element uses an atomic formula -->
<!-- prem element uses an atomic formula or an 'and' -->

<!ENTITY % conc "atom">
<!ENTITY % prem "(atom | and)">

<!-- ELEMENT and ATTLIST Declarations -->

<!-- 'rulebase' root element uses 'if' implications as top-level rules -->

<!-- label attribute allows naming of an entire individual rulebase; -->
<!-- e.g., this can help enable forward inferencing of selected rulebase(s)  -->

<!-- direction attribute indicates the intended direction of rule inferencing;  -->
<!-- it is a preliminary design choice and has a 'neutral' default value -->

<!ELEMENT rulebase (if*)>
<!ATTLIST rulebase label CDATA #IMPLIED>
<!ATTLIST rulebase direction (forward | backward | bidirectional) "bidirectional">


<!-- 'if' implications are usable as top-level rules -->
<!-- 'if' element uses a conclusion followed by a premise -->
<!-- "<if>conc prem</if>" stands for "conc if prem", i.e., "conc is true if prem is true" --
>

<!-- label attribute is a handle for the rule: for various uses, including editing -->

<!ELEMENT if (%conc;, %prem;)>
<!ATTLIST if label CDATA #IMPLIED>


<!-- an 'and' is usable within premises -->
<!-- 'and' uses zero or more atomic formulas -->
<!-- "<and>atom</and>" is equivalent to "atom"-->
<!-- "<and></and>" is equivalent to "true"-->

<!ELEMENT and (atom*)>


<!-- atomic formulas are usable within conc's, prem's, and 'and's -->
<!-- atom element uses rel(ation) symbol followed by a sequence of -->
<!-- zero or more arguments, which may be ind(ividual)s or var(iable)s -->

<!ELEMENT atom (rel, (ind | var)*)>


<!-- there is one kind of fixed argument -->
```

```
<!-- individual constant, as in predicate logic -->

<!ELEMENT ind  (#PCDATA)>

<!-- there is one kind of variable argument -->

<!-- logical variable, as in logic programming -->

<!ELEMENT var  (#PCDATA)>

<!-- there are only fixed (first-order) relations -->

<!-- relation or predicate symbol -->

<!ELEMENT rel  (#PCDATA)>
```

Appendix C: ILOG Simple Rules DTD

```
<!-- ILOG Simple Rules DTD -->
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
  A constant expression has a required type (enumerated) and
  a value (cdata). The DTD provides the common types that
  exist in the programming languages. The notations allowed
  for specifying the literal values (for instance hexadecimal
  numbers or special characters) are the same as for the Java
  programming language.
-->
<!ELEMENT constant EMPTY>
<!ATTLIST constant
  type (string | boolean | byte | short | char | long
       | int | float | double | null) #REQUIRED
  value CDATA #REQUIRED>
<!--
  Defines an assignable entity. An assignable entity is a variable
  or a field. An assignable entity has a value and can change values
  using assignments. As an assignable entity is also a value, it
  can appear in an expressison.
-->
<!ENTITY % assignable "(variable | field)">

<!--
  A variable has simply a name. A variable is created using "bind"
  or as the object variable of a simple condition. A variable is
  an "assignable" and can change values using "assignment".
-->
<!ELEMENT variable EMPTY>
<!ATTLIST variable name NMTOKEN #REQUIRED>
<!--
  A field value. A field value operates on an object (an expression)
  and a field name. If the object is not specified, it is set to
  be the current default object. In a simple condition or a not
  condition, the default object is the one currently matched by the
  condition. For "assert" and "modify", the default object is the
  target of the assertion or the modification.
-->
<!ELEMENT field (%expression;)?>
<!ATTLIST field
  name NMTOKEN #REQUIRED>
<!--
  Defines the expressions of the language. An expression can be
  an assignable (variable or field), a constant (a literal), an
  arithmetic expression or a boolean expression (a test).
-->
<!ENTITY % expression
 "(%assignable; | constant | unaryExp | binaryExp | naryExp)">
<!--
  Defines the unary expressions. A unary expression acts on a
  single argument. The operator is specified using an enumerated
  type (the "operator" attribute). Note: the traditional
  signs like +, -, ! can not be part of enumerations. This
```

```
  justifies that we use symbolic names for operators.
-->
<!ELEMENT unaryExp (%expression;)>
<!ATTLIST unaryExp operator (plus | minus | not) #REQUIRED>
<!--
  Defines the binary expressions. A binary expression acts on two
  arguments. As for unary operators, binary operators are
  specified using enumerated symbolic names which correspond
  respectively to the following signs: ==, !=, <, <=, >, >=
-->
<!ELEMENT binaryExp (%expression;,%expression;)>
<!ATTLIST binaryExp
  operator (eq | neq | lt | lte | gt | gte) #REQUIRED>
<!--
  A N-ary expression acts on N expressions. The number of expressions
  must be at least two. The operators are specified using symbolic
  names. Those names correspond to these signs:
    +, -, *, /, %, &&, ||
-->
<!ELEMENT naryExp (%expression;)+>
<!ATTLIST naryExp
  operator (add | subtract | multiply | divide |
          remainder | and | or) #REQUIRED>
<!--
  An assignment. This uses two expressions: a first expression
  (an "assignable") which will be assigned the value of a second
  expression.
-->
<!ELEMENT assignment (%assignable;,%expression;)>
<!--
  A variable binding declares a variable and sets it to some
  initial value. A variable can change values by "assignment".
-->
<!ELEMENT bind (%expression;) >
<!ATTLIST bind name NMTOKEN #REQUIRED>
<!--
  The ruleset is composed of a list of rules. A ruleset has a name
  (optional). The ruleset is the root element of an XML document.
-->
<!ELEMENT ruleset (rule*)>
<!ATTLIST ruleset
  name NMTOKEN #IMPLIED>
<!--
  A rule has a name (required), a priority (optional), a condition
  part and an action part. The priority is an expression.
  There must be at least a condition and the action part can be empty.
-->
<!ELEMENT rule ( priority?, conditionPart, actionPart ) >
<!ATTLIST rule
  name NMTOKEN #REQUIRED>
<!ELEMENT priority (%expression;)>
<!ELEMENT conditionPart (%condition;)+>
<!ELEMENT actionPart (%action;)* >
<!--
  A condition is either a simple condition or a not condition.
  These conditions differ by the fact that a simple condition match
  an object which can be bound to a variable, while the not condition
  does not match any object.
```

```
-->
<!ENTITY % condition "(simpleCondition | notCondition)">
<!--
  Defines the possible actions allowed in the rule action part.
  This includes variable declarations and assignments, as well as
  the traditional assert, retract and modify statements of rule
  languages.
-->
<!ENTITY % action "(assignment | bind | assert | assertobj
              | modify | retract)">
<!--
  A simple condition has a class name (required), a variable
  name (optional) to which the object is bound. The body of the
  condition is composed of test expressions. Under the scope
  of a simple condition, the default object is the one tested
  by this condition.
-->
<!ELEMENT simpleCondition (%expression;)*>
<!ATTLIST simpleCondition
  className CDATA #REQUIRED
  objectVariable NMTOKEN #IMPLIED>
<!--
  A not condition has a class name (required). The body of the
  condition may contain the same test expressions as for
  the simple condition. A not condition can not be bound to
  a variable. Under the scope of a not condition, the default
  object is the one tested by this condition.
-->
<!ELEMENT notCondition (%expression;)*>
<!ATTLIST notCondition
  className CDATA #REQUIRED>
<!--
  The assert action. This action requires a class name and may
  specify field assignment statements. An instance of the class
  is created, the statements (assignments) are executed, and the
  object is then added to the working memory. Under the scope of
  an assert, the default object is the one currently asserted.
-->
<!ELEMENT assert (assignment | bind)* >
<!ATTLIST assert
  className CDATA #REQUIRED>
<!--
  Another assert action. This action asserts an object computed from
  an expression. It differs from the previous "assert" by the fact
  that the object may be already created and returned as the value
  of the expression.
-->
<!ELEMENT assertobj (%expression;)>
<!--
  The retract action. This action removes an object from the working
  memory. The variable represents an object previously bound from
  the condition part.
-->
<!ELEMENT retract (variable)>
<!--
  The modify action. The action modifies an object of the working
  memory. The object is identified by a variable. The block of
  statements the the same as for the assert action. Under the scope
```

```
   of a modify, the default object is the one currently modified.
-->

<!ELEMENT modify (variable, (assignment | bind)+)>
```

Appendix D: Sample .ilr Rule File

```
import java.lang.*;
import de.tuhh.chanda.wipstoilogrulesdemo.*;
import ilog.rules.engine.*;
import java.util.*;
ruleset r
{
    Ship theShip = new Ship(9,60);
    ArrayList resultlist = new ArrayList();
};
setup
{
    insert(theShip);
    System.out.println("Check:"+theShip.shipAge);
    System.out.println("Initial actions:\n");
};
rule shipAgeRule
{
  when
    {
      Ship(theShip .shipAge > 8 & < 600);
    }
  then
    {
      System.out.println("shipAgeRule: Add SurveyItem_1512 to the Inspection
      List .");
      resultlist.add("SurveyItem_1512");
      System.out.println(resultlist);
    }
}
rule shipLengthRule
{
  when
    {
      Ship(theShip.shipLength > 50 & < 120);
    }
  then
    {
       System.out.println("shipLengthRule: Add SurveyItem_1667 to the
       Inspection List .");
       resultlist.add("SurveyItem_1667");
       System.out.println(resultlist);
    }
```

Bibliography

[1] JXPERT: A JAVA/XML-INTELLIGENT INFORMATION SYSTEM by XIN ZHANG, Master Thesis, 1998

[2] von Luck, K., Nebel, B., Schneider, H.-J., Aspects of Knowledge Base Management Systems, Wissensrepräsentation in Expertensystemen, ed, Rahmstorf, G., Springer-Verlag, Berlin, pp 146-157, 1988.

[3] Expert Systems Tutorial, December 2003, #http://carlisle-www.army.mil/usacsl/divisions/std/branches/keg/expert/intro.htm

[4] DM REVIEW, 2003, www.dmreview.com

[5] Baader, F., Sattler, U, Knowledge Representation in Process Engineering, Proceedings of the International Workshop on Description Logics - DL-96, Cambridge, MA, pp 74-78, 1996

[6] Reasoning in rule-based systems, December 2003, http://www.expertise2go.com/webesie/tutorials/ESIntro/ESIntro16.htm

[7] Databases and Artificial Intelligence 3, December 2003, http://www.cee.hw.ac.uk/~alison/ai3notes/subsection2_4_4_2.html#SECTION0044200000000000000

[8] Rule-Based Systems and Identification Trees, December 2003, http://ai-depot.com/Tutorial/RuleBased.html

[9] Questions Frequently Asked of the Business Rules Group, December 2003, www.businessrulesgroup.org

[10] Overview of IBM CommonRules 1.0 Alpha, December 2003, Releasehttp://www.research.ibm.com/rules/commonrules-overview.html

[11] Simple Rule Markup Language (SRML), December 2003, http://xml.coverpages.org/srml.html

[12] Rule-based Process Support for Enterprise Information Portal, Henry Kamau Gichahi, master thesis, February 2003

[13] Business Rules Markup Language (BRML), December 2003, http://xml.coverpages.org/brml.html

[14] infoAsset Broker
Standardsoftware für das Wissensmanagement, December 2003. http://www.infoasset.de

[15] Jess Information, December 2003, http://herzberg.ca.sandia.gov/jess/

[16] ILog JRules, ILog Inc., 2003, http://www.ilog.com/

[17] CLIPS, December 2003, http://www.ghg.net/clips/CLIPS.html

[18] The Mandarax Project, December 2003, http://mandarax.org/

[19] BLAZE ADVISOR, December 2003,
http://www.blazesoft.com/product/advisor/index.html

[20] OPS/J, December 2003, http://www.pst.com/

[21] Haley, December 2003, http://www.haley.com/1761418434055184/

[22] Pega, December 2003, http://www.pega.com/Default.asp

[23] JTP: An Object-Oriented Modular Reasoning System, December 2003,
http://www.ksl.stanford.edu/software/JTP/#obtain

[24] drools, December 2003, http://drools.org/

[25] InfoSapient, December 2003, http://info-sapient.sourceforge.net/

[26] CommonRules, December 2003, http://www.alphaworks.ibm.com/tech/commonrules

[27] QuickRules Rules Engine, December 2003, http://www.sys-con.com/java/wbg/CurrentSearch_Detail.cfm?ID=1268

[28] Novell exteNd Director, December 2003,
http://www.novell.com/products/extend/director/

[29] Acquired Intelligence Inc. , December 2003, http://www.aiinc.ca/

[30] http://www.sts.tu-harburg.de/

[31] http://www.w3.org/TR/xslt

[32] http://en.wikipedia.org/wiki/Builder_pattern

[33] http://exciton.cs.oberlin.edu/javaresources/DesignPatterns/VisitorPattern.htm

[34] http://www.jcp.org/en/jsr/detail?id=94

[35] http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/

[36] http://www.daml.org

[37] Analysis and Design of a Rule-based System for Process Support in a Maritime Information Portal, Henry Kamau Gichahi Student Project, May 2002

[38] The Rule Markup Initiative, December 2003, http://www.dfki.uni-kl.de/ruleml/

[39] RuleML Example Document, December 2003, http://www.relfun.org/ruleml/discount-roundtrip.ruleml

[40] Simple Rule Markup Language (SRML), January 2004, http://xml.coverpages.org/SRML-examples20010517.txt

[41] CommonRules, January 2004, http://www.alphaworks.ibm.com/tech/commonrules

[42] The InfoAsset Broker, April 2001, http://www.infoasset.de

[43] Extensible Markup Language (XML), December 2003, http://www.w3.org/XML/

[44] XHTML™ 1.0, December 2003, http://www.w3.org/TR/xhtml1/