CA Aion® Rule Manager

Rule Engine: JSR-94 Implementation Guide

r11

CA Technologies Product References

This document references the following CA Technologies products:

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

Contact CA Technologies

Contact CA Support

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

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

Provide Feedback

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

If you would like to provide feedback about CA Technologies product documentation, complete our short customer survey, which is available on the CA Support website at http://ca.com/docs.

Contents

Chapter 1: The JSR-94 Specification

This section contains the following topics:

Introduction

The CA Rule Engine implements interfaces specified by Java Specification Request 94 (JSR-94). JSR-94 defines a Java runtime API for inference engines, which permits an inference engine to be called from a Java program. The JSR-94 standard is a result of the Java Community Process. Before this standard was established, each supplier of an inference engine was free to define its own way of interacting with its inference engine. This was the situation before the different suppliers of Java-based inference engines agreed upon a basic API for interacting with such inference engines. This API is embodied in the JSR-94 specification.

Java provides no inferencing services. The Java language provides no special commands for invoking inferencing. In addition, you cannot define rules within the Java language. The simple branching structures, such as the Java if statement, cannot be used as independent rules. Therefore, the inferencing capability must be provided by a source outside of the language itself. For inferencing to be performed from within the Java language, the Java language requires an API to an inference engine. How a Java program is to interact with the outside source of inferencing is defined by the JSR-94 standard.

The full JSR-94 specification is available on the Java Community Process website at: http://www.jcp.org (type **94** into the JSR text box and press Enter).

JSR-94 Implementations

CA provides CA Rule Engine, a fully compliant JSR-94 implementation, for invoking inferencing services directly from Java programs. A rulebase is more than a mere aggregation of rules. It also contains additional information to allow a rule engine to execute the rules. Since the JSR-94 standard only specifies acquisition and use of a rule engine, the format of the rulebase is specific to CA Rule Engine. For more information, see JSR-94 Implementations and Rule Languages.

An understanding of CA Rule Engine requires an understanding of the JSR-94 specification. JSR-94 provides a set of services, or interfaces, which a client uses to interact with an inference engine. These services are based on the assumption that client application can execute a basic multiple-step cycle that consists of:

1. parse rules
2. register the rules that are to be executed with the inferencing service
3. load the rules (pass them to the inference engine)
4. add objects to the engine for the rules' consideration
5. execute the rules (inference and get results back from the engine)

The JSR-94 specification permits several vendor-specific extensions. For more information on the use of CA Rule Engine, see CA Extensions to the JSR-94 Specification and Special Considerations.

JSR-94 Services Overview

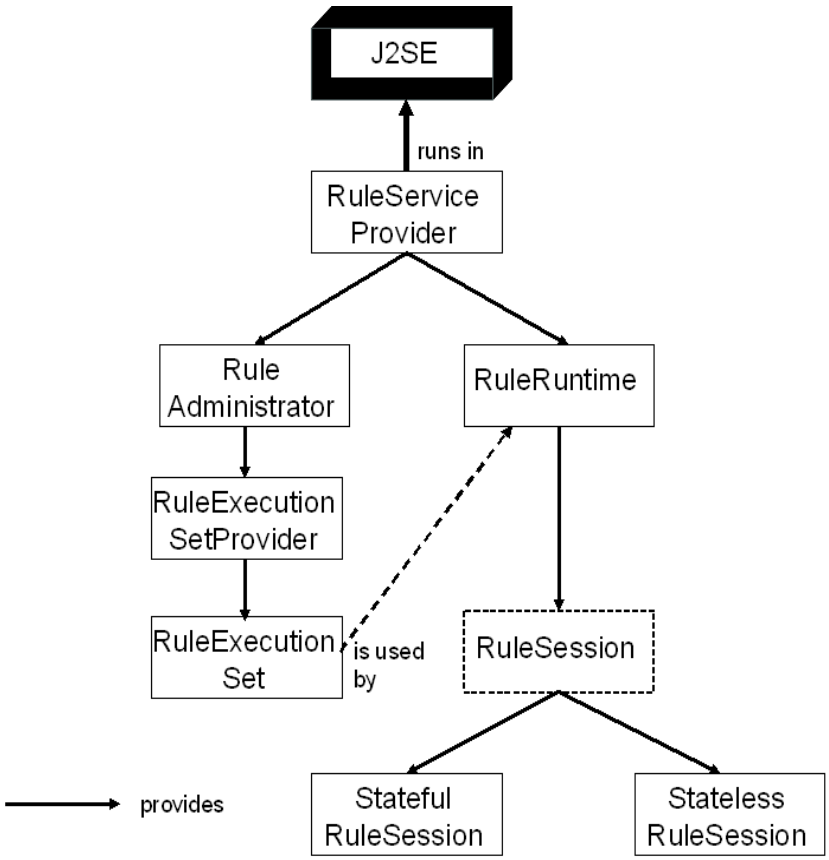
The JSR-94 specification provides for two types of rule-related services:

- Administrative services through the administrator API
- Runtime services through the runtime API

Administrative services create, register, and query what are called rule execution sets. A rule execution set is an executable collection of rules; that is, it is just a synonym of rulebase adopted by the JSR-94 standard. The RuleExecutionSet interface provides a Java program with a generic type that characterizes executable rule sets for any vendor-specific inference engine.

Runtime services address the actual inferencing operations. Runtime services provide the means of establishing an inferencing session with a registered rule execution set, introducing objects, i.e. data values, to an inferencing session, invoking inferencing, and retrieving the results of inferencing.

The following diagram shows the overall structure of JSR-94 implementations:



JSR-94 implementations run in J2SE environments. The Java client program begins by instantiating a RuleServiceProvider provided by the vendor specific JSR-94 implementation. For information on how to instantiate a RuleServiceProvider provided by the vendor-specific JSR-94 implementation, see Construction of Java Client Applications.

The RuleServiceProvider provides methods for obtaining the RuleAdministrator and the RuleRuntime, to instantiate the main service providers of the JSR-94 specification. The RuleAdministrator needs to be instantiated to obtain a RuleExecutionSet. To obtain a RuleExecutionSet, it is first necessary to instantiate a RuleExecutionSetProvider, whose role is to provide methods for taking a rule source, for example, a textual document or binary file, and returning a RuleExecutionSet based on that source. The vendor's implementation of the overloaded createRuleExecutionSet() methods does what is necessary to return an instance to the client application of the type RuleExecutionSet appropriate for the particular inference engine, depending upon the input to the method.

Note: There are two forms of RuleExecutionSet providers: RuleExecutionSetProvider and LocalRuleExecutionSetProvider. RuleExecutionSetProvider defines methods to create a RuleExecutionSet from serializable sources. LocalRuleExecutionSetProvider allows creation of RuleExecutionSet instances from non-serializable resources and is available only to inference engines that are running in the JVM of the caller.

In addition, the RuleAdministrator provides services to register and deregister a RuleExecutionSet with a URI. This must be done to invoke inferencing under the RuleRuntime service and to set and/or query user and vendor-specific properties of RuleExecutionSets and the rules they contain.

Once a RuleExecutionSet instance is obtained for the rules that will be executed, it is necessary to obtain an instance of the RuleRuntime from the RuleServiceProvider and to create a RuleSession. A RuleSession can be created by passing the URI of the registered RuleExecutionSet to the RuleRuntime. The JSR94 specification supports both stateful and stateless rule sessions. It is possible to submit the rules and objects to the inference engine and to receive the results in one synchronous call, i.e. a stateless rule session, or to conduct interaction with the inference engine, i.e. a stateful rule session.

Finally, JSR-94 provides facilities for filtering the results obtained from inferencing. An application can retrieve just the information in which it is interested.

Using a JSR-94 Implementation in Java Code

The following Java code provides a template of what is involved in using a JSR-94 implementation in Java code, only the RuleServiceProvider registration string and the name of CA's implementation class are specific to CA Rule Engine:

```
String RULE_SERVICE_PROVIDER = "com.ca.cleverpath.aion.jsr94";

// Load the rule service provider of the vendor implementation.
// For more information on loading the service provider, see
// Acquiring the RuleServiceProvider.
Class.forName("com.ca.cleverpath.aion.jsr94.RuleServiceProviderImpl");

// Get the rule service provider from the provider manager.
RuleServiceProvider svcProvider =
        RuleServiceProviderManager.getRuleServiceProvider(RULE_SERVICE_PROVIDER);

// For the next series of steps, see Rule Administration: Acquiring
// and Registering the RuleExecutionSet.
// Get the Rule Administrator
RuleAdministrator ruleAdmin = svcProvider.getRuleAdministrator();

// Get the RuleExecutionSet provider
RuleExecutionSetProvider ruleSetProvider =
        ruleAdmin.getRuleExecutionSetProvider(null);

// Get a sample rulebase file.
RuleExecutionSet ruleSet = ruleSetProvider.createRuleExecutionSet(
        "file:/D:/InstallSoftware/JSR94/lib/rulebase.bin",null);

String ruleSetUri = "rulebases://" + ruleSet.getName();
ruleAdmin.registerRuleExecutionSet(ruleSetUri, ruleSet, null);

// For the next series of steps, see Establish a Rule Session.
// Create a RuleRuntime
RuleRuntime runtime = svcProvider.getRuleRuntime();

// Create a Stateless Rule Session using the RuleRuntime
StatelessRuleSession session = (StatelessRuleSession)runtime.createRuleSession(
        ruleSetUri, null, RuleRuntime.STATELESS_SESSION_TYPE);

// Enter local code to create client instances on the Java side.
//  (Assume an instance, inst1, is created for input and an
//  inst2 is created for output.)
// Enter local code to set properties of created instances

// Perform inferencing over the added objects with the rules
// passed to this session at create time. See Stateless versus
// Stateful Rule Sessions.
```

```
List inputObjects = (List) new LinkedList();
inputObjects.add(inst1);
inputObjects.add(inst2);  //NOTE: Also pass the output instance to the rulebase
List objectsReadBack = session.executeRules(inputObjects);

// Enter local code to process results, for example, iterate over
// objectsReadBack.

// Close the session with the inference engine.
session.release();


ruleAdmin.deregisterRuleExecutionSet("rulebases://" + ruleSet.getName(), null);
```

For a detailed explanation of this code, see Construction of Java Client Applications. For a detailed example of an actual client application, see Client Application Class.

CA Technologies's JSR-94 Implementation

This section introduces the general features of the CA Rule Engine, CA Technologies's implementation of the JSR-94 specification. It summarizes features of the underlying inference engine and CA Technologies's extensions to the JSR-94 Specification.

Inference Engines

An inference engine is the foundation of every JSR-94 implementation. The inference engine that is made available to Java client applications by CA Rule Engine is implemented totally in Java. This is the same inference engine that is utilized by other Java components of CA Aion Rule Manager.

The CA Rule Engine inference engine has its own API. However, client applications of CA Rule Engine never use this API directly. Instead, CA Rule Engine provides an API that complies with the JSR-94 specification and performs any necessary transformations between the JSR-94 compliant API and the native API of the internal inference engine.

The rule language used by CA Rule Engine is known as *Rulebase Definition Language* (RDL). This manual describes this language and corresponding engine behaviors in the following places:

- Using CA Rule Engine provides a brief, high-level summary of the salient features of the language required to understand the CA Rule Engine extensions.
- Overview of RDL Rulebases provides a summary of the features of the language needed if writing a rulebase in native RDL.
- Overview of Inferencing provides a summary of the behaviors of the inference engine.

The separate RDL Specification provides the reference source and explains the features of the language in detail.

To facilitate direct authoring of rules, CA Rule Engine also provides an *infix* rule language that is much easier to understand by users. Infix rules can be used in place of actual RDL rules in an RDL rulebase to form an infix rulebase. CA Rule Engine provides an infix2rdl batch/script file tool to convert an infix rulebase to its RDL equivalent and an rdl2infix batch/script file tool to do the reverse. For more information, please refer to the separate guide for infix language.

CA Rule Engine provides the facilities to build, i.e. compile an RDL textual rulebase as an executable rulebase. CA Rule Engine also provides facilities to perform inferencing with such an executable rulebase. The structure of CA Rule Engine closely follows the Administrative and Runtime services specified in JSR94. There are subtle differences. For example, the input to JSR-94 createRuleExecutionSet() method can be a text file, written in RDL, or a pointer to an executable rulebase. Because the inference engine requires an executable rulebase as input, the method automatically compiles the text file. If the method is called with a pointer to an executable rulebase, it will not call the CA Rule Engine building service but just return the executable rulebase cast as a RuleExecutionSet. CA Rule Engine hides the mechanisms for determining how to pass back a rule execution set to the Java program from the user.

CA Technologies Extensions to the JSR-94 Specifications and Special Considerations

The JSR-94 specification allows inference engine vendors to extend the API specification in specific ways that are appropriate for their inference engine. Vendors may specify properties of RuleExecutionSets and Rules that are unique to their inference engine. CA Rule Engine introduces several properties that are special to the CA Rule Engine inference engine. For more information on these properties and how to access them, see Access CA Technologies's Extended RuleExecutionSet and Rule Properties.

There are special client-side considerations that deal with requirements for writing the Java application program that uses CA Rule Engine. For more information about these special considerations, see Constructing of Java Client Applications.

Many of these properties and special considerations depend on features of the underlying rule language used by the CA Rule Engine inference engine. For more information about these features, see RDL Rulebase Overview.

JSR-94 Implementations and Rule Languages

Although the JSR-94 specification specifies the API to any compliant inference engine, it does not mandate any specific rule language. One needs to formulate the rules in the vendor-specific rule language of the particular inference engine that is being invoked. For CA Rule Engine, this language is called the Rulebase Definition Language (RDL).

RDL is an XML-based programming language for representing objects and decision trees, the fundamental model on which rules are represented in RDL, in rulebases. For more information on RDL, see Rulebase Definition Language Fundamentals.

Rulebase Definition Language Fundamentals

To understand a CA Rule Engine rulebase, it is necessary to understand several concepts that are unique to RDL and the CA Rule Engine inference engine. For more information on these concepts, see RDL Rulebase Overview.

Rulebases expressed in RDL are hierarchically organized in three levels:

```
Domains
        Rulesets
                Rules
```

Domains, rulesets, and rules are *named* objects in an RDL rulebase; each may occur multiple times at their respective levels. *Domains* provide a means of grouping rulebase resources in functional units. Rulebase resources include classes, instances, and rulesets. A rulebase may contain multiple domains, each dealing with a separate aspect of the problem being solved. Rulesets are sets of rules. An RDL ruleset is not to be confused with a RuleExecutionSet in JSR-94 standard, which is really equivalent to a whole rulebase. Rules have a unique structure in RDL, which are based on a decision tree paradigm.

RDL supports classes. Classes are defined by their fields. Classes can be defined at the rulebase level, the domain level and the ruleset level. Classes defined at a higher level are accessible by objects defined in a lower level in the same hierarchy, but not conversely. The classes defined at the rulebase level may be interfaced with client applications. Instances of such classes pass information between the rulebase and client application. Instances can be either statically defined by a class or dynamically created at runtime, e.g. by rules. Additionally, if the rulebase designates a class as an *app-creatable* class, the client application can dynamically create instances for the class. All instances in RDL must be named.

Each field in a class is described by its collection type and its data type. The collection type can be either atomic or set. The data type can be one of Number, Boolean, String, DateTime, Duration and instance reference. When a RDL instance is instantiated, the state for a field is *unknown* and remains in that state until it is resolved to a value by a rulebase statement or client operation. At the time of resolution the field will become in the *known* state. The concept of unknown is fundamental in inferencing in the sense that the goal of inferencing is to use the available rules to resolve the unknown fields.

The purpose for an atomic instance reference field is to hold a reference to an instance of specified class defined in the rulebase. The instance reference type is different from other data types in the sense that it has three possible states: unknown, reference to another valid instance, and a special value, the *RDL NULL*, to indicate that it is known that that reference does not exist. For example, if a field of a Person class is spouse, the RDL NULL value for that field means that given person does not have a spouse, or is single.

In RDL, domains define critical aspects of how the rulebase will be exposed to the outside world. The domain contains the specification of how the Java client applications will invoke the rulebase. The following aspects of domains define this interaction:

- **The appshared domain**. A rulebase may contain multiple domains. Domains may be specified as either internal to the rulebase or shared with the client application. The latter are referred to as *appshared domains***,** after the attribute in the RDL that designates a domain to have this status. Appshared domains specify the interface for passing objects between the rulebase and client application.
- **Pre and postconditions**. The pre- and postconditions of an appshared domain define respectively the input values that the client application needs to provide the rulebase for inferencing, and the output values that the client application can retrieve from the rulebase. For client applications, preconditions specify values that are both read and write; postconditions specify read only values. Pre- and postconditions are designated fields of instances of classes defined at the rulebase level.

The Java client application must interact with one and only one appshared domain during any given invocation of the executeRules() method. The appshared domain determines which objects the Java application must provide to the inference engine and which objects can be retrieved from the inference engine based on the specified pre and postconditions.

To help the Java programmer understand the structure of the rulebase interface, CA Rule Engine provides an application interface document. This document is obtained as a vendor-specific extension of the rule execution set properties. For more information, see CA Extensions to the JSR-94 Specification and Special Considerations.

Install and Configure CA Rule Engine

If you obtained CA Rule Engine distribution as a ZIP archive, installation only requires extracting it to the desired location. A new folder containing the CA Rule Engine distribution will be created at that location. In this guide, the distribution folder is referred to as *aionjre_home*.

If you obtained CA Rule Engine as a component of another product, please refer to your product documentation to determine the location of *aionjre_home*.

CA Rule Engine is compatible with JDK 1.4.2 or later. JDK 1.4.2 or later must be installed to use CA Rule Engine to develop inference enabled applications. To compile and run the samples using supplied scripts, Apache Ant 1.6.1 or later also must be installed and found in PATH.

You must ensure that the following jar files are found on CLASSPATH when developing and running your inference enabled application:

- aionjre.jar
- jsr94.jar
- log4j.jar

The aionjre.jar file is located in the *aionjre_home* folder. The others are included in the lib folder in the CA Rule Engine distribution for your convenience.

Examples of CLASSPATH settings are shown in sample batch files (for Windows) and corresponding shell scripts (for UNIX or Linux) provided in the samples folder in the CA Rule Engine distribution. For more information on the sample applications provided with CA Rule Engine, see Documentation and Samples. The sample batch or script files also illustrate the use of log4j configuration for CA Rule Engine. For more information on configuring log4j, see Configure Logging and Execute for CA Rule Engine.

The CA Rule Engine specific files needed for running the JSR-94 Technology Compatibility Kit (TCK) are located in the tckconfiguration folder in the CA Rule Engine distribution.

More information

Verify JSR94 Compliance (see page 185)

Configure and Execute Logging for CA Rule Engine

CA Rule Engine logging uses the log4j package from Apache (http://logging.apache.org).

Note: All messages from the CA Rule Engine are written to the log4j log file. In the absence of log4j configuration, CA Rule Engine does not produce any log messages. We recommend that log4j configuration be performed for CA Rule Engine. We also recommend that UTF-8 encoding be configured for log4j appender to support non-ASCII characters that may be used by a localized rulebase.

The log4j logging framework must be configured initially by the user of CA Rule Engine. This configuration is normally specified using a property file. For a sample configuration property file, see Example: Configuration File for log4j. To learn more about log4j and its configuration using a property file, please see the log4j project documentation at http://logging.apache.org

Note: Other configuration techniques include an xml configuration file (which is analogous to the sample property file, but in XML) and the use of API calls in code.

Only the topmost logger for the com.ca.cleverpath.aion.jsr94 must be configured. Set the log level for CA Rule Engine to an appropriate level to log the desired CA Rule Engine messages. We recommend setting the logging level of the topmost logger to at least WARN. For more information on logging levels, see Guidelines for log4j Messages from CA Rule Engine. For a mapping of log4j logging levels and CA Rule Engine engine trace levels, see Log4j Logging Levels and CA Rule Engine Trace Levels.

To obtain a log file, execute the Java class with a -D parameter that sets the log4j.configuration property to the URI of the desired configuration file. The following example shows the case where the configuration file is in current folder:

```
java -Dlog4j.configuration=log4j.properties com.ca.cleverpath.aion.jsr94.samples.tlp.TLPSession
```

If the configuration file is not in current folder, it should be specified in the full URI format. For example, the –D parameter should be specified as:

```
-Dlog4j.configuration=file:../log4j.properties
```

if the file log4j.properties is located one level above the current folder.

Example: Configuration File for log4j

Example

In the following example, the logging level for the topmost logger is set to WARN and the path to the log file is defined to be C:\Temp\ and the name of the log file is aionjre.log:

```
# log4j configuration for CA Rule Engine
log4j.logger.com.ca.cleverpath.aion.jsr94=WARN, JSR94
#Define a file appender for output
log4j.appender.JSR94=org.apache.log4j.RollingFileAppender
log4j.appender.JSR94.File=C:\Temp\aionjre.log
# Keep one backup file
log4j.appender.JSR94.MaxBackupIndex=1
log4j.appender.JSR94.MaxFileSize=2000KB
log4j.appender.JSR94.encoding=UTF-8
log4j.appender.JSR94.layout=org.apache.log4j.PatternLayout
log4j.appender.JSR94.layout.ConversionPattern=%d{HH:mm:ss,SSS} %p %t %c - %m%n
```

See the log4j.properties file in the samples folder inside the CA Rule Engine distribution.

For more information on client applications using logging frameworks, see Client Applications with Logging Frameworks Notes.

Client Applications with Logging Frameworks Notes

If the logging framework used by application is log4j, CA Rule Engine reuses the application's log4j configuration and does not attempt to reconfigure log4j. Make entries for CA Rule Engine's top most logger as shown in the sample logging configuration file; for more information, see Example: Configuration File for log4j.

If the logging framework used by client application is not the log4j framework, note that CA Rule Engine's log4j configuration and logging are separate from the any client application's logging, configure log4j for CA Rule Engine.

Guidelines for log4j Messages from CA Rule Engine

The following guidelines show the correlation between log4j logging levels and types of CA Rule Engine messages (logging levels are shown in ascending order):

- DEBUG: Interesting information and significant stages for application can be very detailed if desired so (for example, whenever CA Rule Engine sets field values during a rule session).
- INFO: Similar to lifecycle or highly significant events (for example, when a session is created or released).
- WARN: Output may not be as expected but processing will continue. For example, CA Rule Engine warns users in the following situations:

  When unknown values cannot be reported back to the client.

  When Java instances cannot be created because there is no default constructor for the Java class or no verified Java class corresponds to the rulebase class.
- ERROR: For exceptions and other errors generated by CA Rule Engine.
- FATAL: For shutdown events when CA Rule Engine cannot continue, for example, when rule service provider class cannot be registered.

For more information on these logging levels, please visit the **javadoc** link for the log4j project documentation on http://logging.apache.org

Log4j Logging Levels and CA Rule Engine Trace Levels

The following mappings of log4j logging levels and CA Rule Engine engine trace levels define the corresponding messages for the log4j log file:

- ALL, DEBUG — 3 - All messages including load, infer, user, error, and warning messages will be allowed thru the message handler.
- INFO — 1 - Messages of type MSGTYPE_TRACE_L1 with user, error, and warning messages only allowed.
- WARN — 0 - Only messages of type error and warning are allowed.
- ERROR — 0 - Only messages of type error are allowed.
- FATAL — No engine messages
- OFF — 0 - Ignore all messages (rule engine posts some messages).

Note: There is no way to indicate CA Rule Engine's engine trace level 2. This keeps a consistent mapping between log4j logging levels and CA Rule Engine's engine trace level. Users can effectively trace level 1 or 3 by choosing logging level as DEBUG or INFO.

For a description of all messages generated by CA Rule Engine, go to the Overview page in the /doc/api folder inside the CA Rule Engine distribution. For more information on available documentation and example application, see Documentation and Samples.

Documentation and Samples

CA Rule Engine documentation, including API Javadocs, is provided in the /doc folder inside the CA Rule Engine distribution.

Sample Java source code for the following client applications is provided in the samples folder inside the CA Rule Engine distribution:

- The TLP sample which illustrates general usage.
- The shopping cart sample which illustrates the iterative decision feature.
- The expense reimbursement sample which illustrates incremental supply of preconditions.

To compile and run these samples, the following sample batch files for Windows and corresponding shell scripts for UNIX or Linux platforms are provided:

- compilesamples to compile sample client applications
- runXXXadmin to run the administrator part of the XXX sample.
- runXXXsession to run the whole rule session of the XXX sample.

For more information on the sample client applications, see Sample CA Rule Engine Applications.

Chapter 2: Using the CA Rule Engine

Using the CA Rule Engine involves two levels of knowledge: knowledge of the JSR-94 standard (see the chapter *The JSR-94 Specification*), and knowledge of specific features of CA Rule Engine. Knowledge of appropriate techniques for constructing Java applications is also necessary. These techniques involve close coordination between the Java client and the structure of the rulebase.

Building a JSR-94 application can occur under either of the following use cases or some combinations of the two:

- You begin with a pre-existing rulebase for which you want to write appropriate Java classes to call the rulebase from a Java application through the JSR-94 interface, but you may or may not have direct access to the rule author or RDL code to understand the rulebase interface. In this use case, you may need to know how to query the interface of the rulebase. Once you understand the requirements of the interface, you can construct the appropriate Java classes.
- You begin with pre-existing business objects defined by Java classes or by developing the rulebase concurrently with Java developers. This use case requires writing a rulebase to inference over these objects. Both sides, the Java classes of your client application and the RDL rulebase, must work together to make sure the Java classes are appropriate for the RDL rulebase as in the first case. In addition, the rulebase must properly implement the business logic behind the pre-existing business objects.

It is also possible to configure the use of CA Rule Engine in a Java application under different solution architectures. A single Java client application may be used to register a rule execution set and to execute that rule execution set or these functions can be done in different Java applications. For more information on rule execution sets, see JSR-94 Services Overview.

For more documentation of CA Rule Engine, see the contents in the doc folder in the CA Rule Engine distribution.

This section contains the following topics:

Construction of Java Client Applications

This chapter presents the general principles for constructing the Java client application. On the client side, CA Rule Engine makes the following extensions to the JSR-94 specification:

- Additional features, beyond standard bean features, are required for constructing Java classes that map instances between the client side and the inference engine.
- If the rulebase contains two or more appshared domains, the inference engine must be informed about which domain will be used by the client.

For more information, see CA Technologies Extensions to the JSR-94 Specification and Special Considerations.

The following major points must be understood when constructing the Java client application:

- There are CA Rule Engine specific rules for constructing Java classes so that CA Rule Engine can coordinate instances on the client side with instances on the rulebase/inference engine side.
- Once the application classes are constructed correctly, the Java client application must include JSR-94 specific coding to acquire a rule execution set and rule session.
- The application must add objects from the client application to the inference engine and retrieve the results of inferencing following JSR-94 specification. For more information, see Addition and Retrieval of Inference Engine Objects. The JSR-94 specification also provides a way for the client application to filter results of inferencing by supplying filter objects.

Note: For an existing rulebase, the supplied WrapperMaker tool can be used to automatically construct the Java classes that wrap the rulebase classes. The Java source code for the sample applications can be found in samples folder inside the CA Rule Engine distribution. The sample applications also provide client test applications.

More information

Rules for Constructing Java Classes (see page 27)
Specify a Shared Domain (see page 43)
Acquire the RuleExecutionSet and RuleSession (see page 35)
Using the WrapperMaker Tool (see page 33)

Rules for Constructing Java Classes

To be able to exchange information between the CA Rule Engine inference engine and its Java client application, it is necessary to have wrapper (stub) Java classes that wrap the corresponding rulebase level RDL classes that define pre- and post-condition (input and output) fields. A wrapper class only needs to declare property fields that correspond to RDL fields which are part of the rulebase application interface. At runtime, user instantiates objects of wrapper classes, initializes the pre-condition fields and adds the objects to rule session for inference and obtains inference results through the post-condition fields of those objects. The CA Rule Engine inference engine may also instantiate objects of wrapper classes due to rule actions.

The CA Rule Engine inference engine uses Java reflection technique to automatically match any user Java objects (instances) of wrapper classes added to the session with its internal instances of corresponding RDL classes. In addition to establishing correspondence between the wrapper class and the RDL class for just that added object, CA Rule Engine also establishes class correspondence for any instance reference fields defined in the class of that object. It will also implicitly and recursively add any objects as values of instance reference fields. When necessary, CA Rule Engine will also search for any wrapper classes in the package of the class of that added object.

In addition to matching objects of a wrapper class, CA Rule Engine also accepts objects of classes that are derived from that wrapper class and matches those objects to internal instances of the RDL class that corresponds to the wrapper class. In addition to normal Java class, the Java wrapper class for an RDL class may be a Java interface and CA Rule Engine accepts any objects of a class that implements that interface and matches those objects to internal instances of the RDL class that corresponds to the interface.

For this matching to work correctly, the Java classes of objects added to the inference engine must comply with specific rules described in the following sections. The requirements can be met through inheritance or implemented interface in addition to direct implementation in that class.

Rules for Public No-Argument Constructor and Property Accessors follow Java rules for the construction of bean classes. The Java classes should also implement the Serializable interface if serialization is needed. The other rules are unique to using CA Rule Engine.

For more information on Java beans, please see the Java Beans tutorial at http://java.sun.com

Note: Java client classes that represent rulebase classes must implement the Serializable interface if the rule session needs to be serialized. The WrapperMaker tool automatically generates serializable classes.

Class Name Requirements

Correspondence between an RDL class and its Java wrapper class or interface is established by matching the unqualified name of the Java wrapper class with the RDL class name. Since RDL supports a much wider set of characters for its identifiers, if it is necessary to write an RDL class for an existing Java class, simply choose identical name should suffice. However, if it is necessary to write a Java wrapper class or interface that directly corresponds to an RDL class, the following method to map an RDL identifier to a Java identifier must be followed.

To establish the Java identifier that matches an existing RDL identifier, each character of the RDL identifier is checked in sequence to see if that character can be a Java identifier character at that position. If so, that character is directly copied. Otherwise, that character is replaced with the string Uxxxx, where xxxx represents the Unicode value of that character in the format of 4 hex digits. For example, the "?" character will be replaced by "U003f".

To follow Java class naming conventions, the first letter of the mapped identifier may be capitalized as needed and that results in the unqualified name of the Java wrapper class or interface. For a list of rulebase classes, see The App Interface document.

For example, the Java class, testclients.myrulebase.Person or testclients.myrulebase.person, may be the Java wrapper class for the RDL class person. For RDL class Trial#5, the unqualified Java class name should be TrialU00235.

Public No-Argument Constructor

The Java class should have a public no-argument constructor.

Example

```
public class RulebaseClass implements Serializable {
    //  Property declarations
    //  The constructor method
    public RulebaseClass()
    {
        // Any object initiation code (optional)
    }
    //  Other methods
}
```

Note: If no constructor is specified for a class, Java will provide a default constructor that fulfils the function of the public no-argument constructor. However, if any other constructor is specified for the class, it is necessary to explicitly state the no-argument constructor.

For more information

RDL Field Data Types and Java Property Types (see page 31)

Property Accessors

Correspondence between a property field of a Java wrapper class and a RDL field of corresponding class is also established through name matching. The name of a Java property field can be created from the corresponding RDL field name similar to that of class names. The only difference is that to follow Java field naming convention, the first letter of the mapped Java identifier may be converted to lowercase as needed. In addition, since the Java field name is obtained through bean introspection, supplemental BeanInfo class may be used as an alternative.

For name matching to succeed, classes must support bean introspection. Beans support introspection in the following ways:

- By adhering to Bean specific rules, known as *design patterns*, when naming Bean Property getter and setter method names. Every property must have public get / set methods named following these design patterns. Consider the following pair of methods:

  ```
  public PropertyType getPropertyName();
  public void setPropertyName(PropertyType v);
  ```

  If either of these is detected, the bean class exposes the property called propertyName of PropertyType, following the standard property naming convention. Both get and set methods must be defined for the property to be used with CA Rule Engine so that data can be transferred between the Java application side and the inference engine side.

  Note: Since RDL collection is set based, even though the type of the field is array, there is no need to provide separate elemental getter and setter methods required for an indexed property of a Java Bean class.

- By explicitly providing property, method, and event information with a related *Bean Information* class. A Bean information class implements the BeanInfo interface. A Bean Information class explicitly lists those Bean features that are to be exposed to introspection. For example, the following BeanInfo classes expose the properties *job* and *owns* of the *person* class to introspection:

```
public class personBeanInfo extends SimpleBeanInfo {
  public PropertyDescriptor[] getPropertyDescriptors() {
    try {
      PropertyDescriptor desc1 = new PropertyDescriptor("job", person.class);
      PropertyDescriptor desc2 =
        new PropertyDescriptor("owns", person.class);
      PropertyDescriptor[] modifiedProperties = {desc1,desc2};
        return modifiedProperties;
    }
    catch( IntrospectionException error ) {
      return new PropertyDescriptor[0];
    }
  }
}
```

Note: RDL permits the use of special symbols in field names that are not permissible in Java property names; for example, #sick_days_per_year is a valid field name in RDL. In this case, instead of naming the Java field u0023sick_days_per_year, a BeanInfo class may be used to map a Java property, sick_days_per_year, to the rulebase field. The following PropertyDescriptor provides the necessary mapping:

```
PropertyDescriptor descn = new PropertyDescriptor("#sick_days_per_year",
  person.class, "getSick_days_per_year", "setSick_days_per_year");
```

In addition to naming requirements described above, the Java property field should also meet the following requirements to allow data to be transferred properly between the client application and CA Rule Engine:

Read and Write methods

The property should have both read and write methods

Data type compatible

The Java class for the property should be compatible with the rulebase data type.

RDL Field Data Types and Java Property Types

The following list shows the RDL field data types with their Java property types. The first Java data type is preferred (when more than one type is acceptable):

number (atomic):

BigDecimal, BigInteger, Double, Float, Integer, Long; int, long, double, float (See note 1 below)

string (atomic):

String, StringBuffer

boolean (atomic):

Boolean, boolean (See note 1 below)

datetime (atomic):

String, StringBuffer

duration (atomic):

String, StringBuffer

inst_ref (atomic):

Java Class of same name (for example, Java class with unqualified name Customer matches an instance reference field of RDL class Customer). (See note 2 below)

set (collection):

Arrays of compatible data types, for example: BigDecimal [] for set of numbers; Person[] for set of instance reference fields. (See note 3 below)

Note:

1. Use of primitive types is not recommended in general. The use of boxed object types (for example, Integer or Float instead of integer and float) allows the user to specify Java *null* which indicates a field to be *unknown* to CA Rule Engine. If a primitive type is used, CA Rule Engine cannot accept or indicate any such fields to be unknown. Warnings are logged to the user in situations in which CA Rule Engine cannot update an object with unknown values. For more information, see Specify Unknown Rulebase Fields. In addition, it is not possible to use initialization callback on a field if a primitive type is used since the field will never be unknown. For more information, see Initialization Callback Methods.
2. For atomic instance reference fields, an object with null instance name stands for the special value of RDL NULL, i.e. asserting absence of a reference. For example, if a field of a class Person is spouse, and an instance of a Person is known to be single, a Person object with null instance name can be set as the value of the spouse field of that Person instance to indicate that that Person has no spouse. For more information, see Specify Instance Reference Fields.
3. Please note that rulebase sets contain unique elements - whereas Java arrays don't enforce uniqueness - so a resulting rulebase set may end up containing fewer elements than does the Java array.

For definitions of the data types provided by RDL, see the Rule Definition Language specification.

Instance Naming

Every Java object corresponding to a rulebase class instance is required to have the instanceName property to indicate an instance name for the object. The purpose of the instance name property is to establish correspondence between a Java object and its same named rulebase instance. The instance names must be unique for all Java objects corresponding to the instances of the same RDL class. This property should be compatible with a rulebase string field and have both get and set methods. For any object to be explicitly passed to the inference engine, the instance name cannot be null or empty. An object with null instance name can only be used as value for an instance reference field to indicate that instance reference field has the value of RDL NULL, i.e. asserting absence of a reference.

Note: In compliance with the Property Accessors requirement, accessor methods must be provided for the instanceName property or the property must be listed in a BeanInfo class.

For any static instance inside a rulebase to be accessible by Java, there must exist a Java instance of corresponding class with its instance name property exactly matching the name of that static rulebase instance. To obtain the names of rulebase static instances, see The App Interface document. CA Rule Engine automatically synchronizes any dynamically created instances between the Java side and the rulebase side.

Using the WrapperMaker Tool

The WrapperMaker tool facilitates the generation of Java classes that wrap the rule engine classes for an existing rulebase. The WrapperMaker tool takes either a rulebase file (RDL or binary) or a rulebase App Interface document file (see Rulebase Structures) and perform the following tasks:

- For any class defined at the rulebase level, the tool generates a corresponding Java class that wraps the rulebase class. The generated classes satisfy all requirements specified in Rules for Constructing Java Classes. In addition, each generated class also provides a second constructor that takes instance name as an argument, and methods equals(), hashcode(), and toString().
- The tool generates a template session Java class that declares all static instances defined at the rulebase level as private static members of the session class. The template session Java class also defines the getRuleServiceProvider() function (which wraps CA Rule Engine specific initializations for JSR-94) to facilitate the obtaining of the RuleServiceProvider.

The user can add rule session logic to the template session Java class to perform inference tasks.

For ease of invocation of the WrapperMaker tool, the wrappermaker batch or script files have been supplied in the /bin folder in the CA Rule Engine distribution. The syntax for invoking the tool is:

```
wrappermaker rulebase/APIfile [JavaPackageName [rootOutputFolder]]
```

where the user can also optionally specify the Java package name for the generated Java class files and the root folder to which the hierarchy of classes will be written. If a rulebase file is supplied as the first parameter, an App Interface document file will also be generated. When optional parameters are not supplied, the generated Java classes will be in the default package and files will be written to the same folder where the rulebase file or App Interface document file resides.

The WrapperMaker tool script also supports drag and drop operation if drag and drop is supported by the OS GUI. When a rulebase file or an App Interface document file is dropped onto the WrapperMaker tool script icon, it is equivalent to invoke the script with the name of that file as the first parameter. Since a Java IDE such as Eclipse can automatically adjust package names when a new Java source file is added to a project, drag and drop operation may be the simplest method for invoking the WrapperMaker tool.

For more information

Acquire the RuleExecutionSet and RuleSession (see page 35)

Adding Callback Methods

The CA Rule Engine inference engine supports callbacks. A callback is a user-defined method that is invoked by CA Rule Engine upon occurrence of certain events. A callback is specific for a particular RDL class and field and must be defined in the corresponding Java wrapper class or interface. Callback methods are optional. Currently, CA Rule Engine supports two types of callbacks, initialization callback and change callback. This section mentions only the syntax of the callback methods. For a more detailed description and effective use of callbacks, see Using Callbacks.

CA Rule Engine invokes the initialization callback method when it needs to access the value of a field in processing a rule. The initialization callback method is also called the *onInit* method. The exact name of the method depends on the name of the field. For example, for a field named xYZ, the signature of the "onInit" method should be defined as the following:

```
public void onInitXYZ()
```

CA Rule Engine invokes the change callback when it needs to modify the value of a field or to create or delete a dynamic instance. Depending on whether the field is a collection, or whether the operation is on the instance, the following five types of change callback methods may be defined. All change callback methods are optional.

To react to a change to the value of a field, client needs to define the "onChange" method. For example, for a field named xYZ, the signature of the "onChange" method should be defined as the following:

```
public boolean onChangeXYZ(fieldType fieldValue)
```

where fieldType is the actual data type of the field xYZ and fieldValue is the new value. The method should return true if it chooses to update the field to the specified value and false otherwise.

When the field is a collection field and the change is to add or remove elements of that collection rather than setting a whole new collection to the field, the following two callback methods will be invoked instead of the "onChange" method:

```
public boolean onAddElementXYZ(fieldType fieldValue, int index)
public boolean onDeleteElementXYZ(fieldType fieldValue, int index)
```

The index value gives the position of the changed element. Since these two methods tend to be more efficient in carrying out modification of the collection field, it is recommended to implement them for a collection field even if no other actions are needed. The WrapperMaker tool automatically generates them for collection fields.

To react to dynamic instance creation and deletion by CA Rule Engine in carrying out rule actions, the client needs to provide the following methods:

```
public static void onCreateInstance(instanceClass instanceObj)
public static void onDeleteInstance(instanceClass instanceObj)
```

where instanceClass is the name of the class enclosing the methods and instanceObj is the changed instance object.

Acquire the RuleExecutionSet and RuleSession

Once the Java classes that wrap the corresponding rulebase classes are properly defined (see Rules for Constructing Java Classes), the main application class needs to be constructed. This class provides methods for acquiring the rule execution sets and establishing a rule session that uses those classes that wrap the rulebase classes. The WrapperMaker tool generated template session class can be used as the starting point for the main application class.

An overview of the required elements in the construction of the main application class includes the following:

- The JSR-94 related packages that must be imported into the client application. For more information, see JSR-94 Packages.
- The first step in any client application is to obtain the RuleServiceProvider. The RuleServiceProvider is the entrance to all other services provided by a JSR-94 implementation. For more information, see Acquire the RuleServiceProvider.
- For all systems, it is necessary to obtain and register a RuleExecutionSet. However, some solution architectures may obtain and register the RuleExecutionSet outside of the application that performs inferencing. For more information, see Acquire and Register the RuleExecutionSet.
- Finally, the application must establish a rule session, which provides an executeRules() method for calling the inference engine with a RuleExecutionSet. For more information, see Establish a Rule Session.

JSR94 Packages

The classes and interfaces defined by the JSR-94 specification are contained in the following packages that correspond to the types of services specified in the JSR-94 specification:

- javax.rules provides the RuleServiceProvider, RuleRuntime, and RuleSession types
- javax.rules.admin provides RuleAdministrator and RuleExecutionSetProviders

These packages must be implemented by the vendor. It is necessary to import the javax.rules.* and javax.rules.admin.* namespaces to client applications. For more information, see JSR-94 Services Overview.

In addition, the Java programmer may want to include any specific vendor extensions that will be used in the client application. CA Rule Engine provides the following additional interfaces as extensions to the JSR-94 specification:

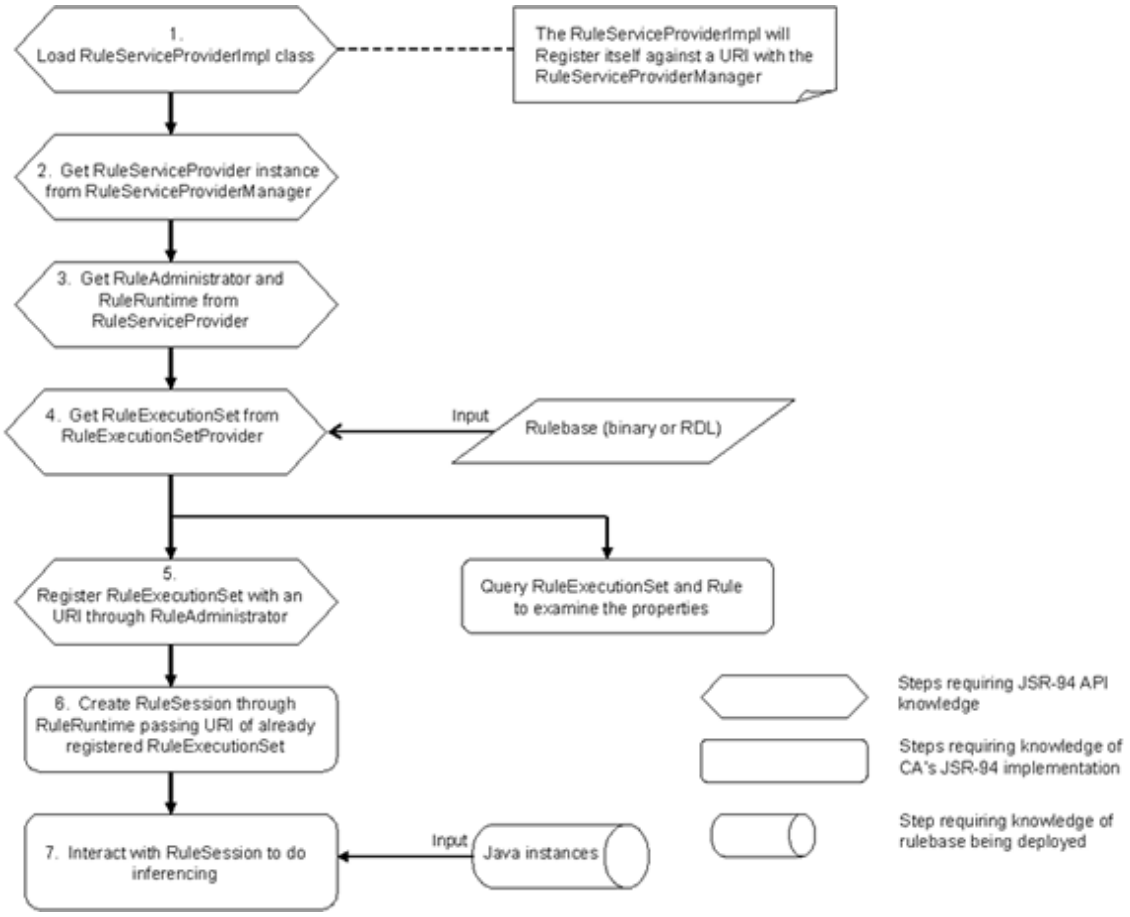
- com.ca.cleverpath.aion.jsr94.AionRulesEngineProperties provides constants for referring to CA's extended rule execution set and rule properties.
- com.ca.cleverpath.aion.jsr94.CARuleExecutionSetMetadata provides methods for accessing CA's extended rule execution set metadata from within a rule session

More information

Access CA Technologies's Extended RuleExecutionSet and Rule Properties (see page 56)

Basic Program Structure and Solution Architectures Flow Diagram

All JSR-94 client applications follow a similar basic flow. The following chart explains how a Java client application works with the JSR-94 interface.



How these steps are distributed in the client application is a matter of architectural design. In one common solution architecture, all the preceding steps may be implemented in a single Java application. In another solution architecture, the steps through registering the RuleExecutionSet using the RuleAdministrator may be performed in one client program, such as an admin application, while querying RuleExecutionSets for property information and inferencing with those RuleExecutionSets may be performed in a different client application, i.e. the business application.

For more information about the steps in the flow chart, see Acquire the RuleServiceProvider (step 1), Acquire and Register the RuleExecutionSet (steps 2 - 5), Establish a Rule Session (step 6), and Addition and Retrieval of Inference Objects (step 7). For more information about querying RuleExecutionSets and Rules, see Query RuleExecutionSets and Rules.

For a detailed example of an actual client application, see Client Application Class.

Note: Implementing some steps requires knowledge of the pure JSR-94 API while other steps require specific knowledge of CA Rule Engine. Knowledge of how to interface with the rulebase is also required. For more information on obtaining information about the interface of the rulebase, see Rulebase Structures.

Acquire the RuleServiceProvider

JSR-94 implementations provide a generally applicable RuleServiceProviderManager, which provides the means of acquiring the RuleServiceProvider. The RuleServiceProvider is the class that provides the functionality appropriate to the vendor's own JSR-94 implementation. For example, the RuleServiceProvider must provide vendor-specific implementations for the RuleAdministrator and RuleRuntime interfaces. This implementation must also register itself against a URI with the RuleServiceProviderManager.

Each vendor implementing the JSR-94 specification must provide a unique implementation of the RuleServiceProvider. CA Technologies's implementation of the RuleServiceProvider is named:

```
com.ca.cleverpath.aion.jsr94.RuleServiceProviderImpl
```

Acquiring CA Technologies's RuleServiceProvider is accomplished with the following steps:

- The application provides a private static String variable named RULE_SERVICE_PROVIDER, whose value is the URI of CA's rule service provider class. The URI of the CA Technologies's rule service is:

  ```
  com.ca.cleverpath.aion.jsr94
  ```

- To ensure uniqueness of the registration URI with the RuleServiceProviderManager, a unique name is used within the Java package namespace of the vendor's JSR-94 implementation. To load CA Technologies's RuleServiceProviderImpl and to register it with the RuleServiceProviderManager, use the Class.forName() method:

  ```
  Class.forName("com.ca.cleverpath.aion.jsr94.RuleServiceProviderImpl");
  ```

- The RuleServiceProviderManager provides a public method to get an instance of a RuleServiceProvider:

  ```
  RuleServiceProvider getRuleServiceProvider(String uri);
  ```

  Invoke this method using the RULE_SERVICE_PROVIDER variable:

  ```
  RuleServiceProvider svcProvider =
    RuleServiceProviderManager.getRuleServiceProvider(RULE_SERVICE_PROVIDER);
  ```

The template session class generated by the WrapperMaker tool, see Using the WrapperMaker Tool, supplies a getRuleServiceProvider() function that implements the above steps.

Acquire and Register the RuleExecutionSet

After a RuleServiceProvider instance is acquired (for more information, see Acquire the RuleServiceProvider), it is necessary to retrieve an instance of the RuleAdministrator:

```
RuleAdministrator ruleAdmin = svcProvider.getRuleAdministrator();
```

The RuleAdministrator provides the following public methods:

```
RuleExecutionSetProvider getRuleExecutionSetProvider(Map propertiesMap);
LocalRuleExecutionSetProvider getLocalRuleExecutionSetProvider(
  Map propertiesMap);
void registerRuleExecutionSet(String bindUri, RuleExecutionSet ruleSet,
  Map propertiesMap);
void deregisterRuleExecutionSet(String bindUri, Map propertiesMap);
```

The next step for the application is to get an instance of the RuleExecutionSetProvider or the LocalRuleExecutionSetProvider:

- The RuleExecutionSetProvider defines methods to create a RuleExecutionSet from a number of Serializable sources. Note that this Provider has methods to create RuleExecutionSets from sources referenced through a URI, a binary rulebase object, and an Element object that is the generated by an XML parser parsing through an RDL rulebase. The URI can either point to a RDL rulebase, or a binary rulebase file such as one obtained using the getBinaryRuleBase() method of CA's implementation class com.ca.cleverpath.aion.jsr94.admin.RuleExecutionSetImpl, which implements the RuleExecutionSet interface in JSR-94.
- The LocalRuleExecutionSetProvider defines methods to create a RuleExecutionSet from local resources, such as InputStreams, binary rulebase objects or character-based Readers. The Local Provider is easier to use when creating the RuleExecutionSet from an XML rulebase file in RDL format. The Local Provider requires that the rule engine be in the same JVM as the caller.

Example

For example, the following code fragment obtains an instance of the LocalRuleExecutionSetProvider:

```
LocalRuleExecutionSetProvider ruleSetProvider =
  ruleAdmin.getLocalRuleExecutionSetProvider(null);
```

The client application must instantiate the appropriate rule execution set provider and then use the appropriate createRuleExecutionSet() method of the rule execution set provider to obtain an instance of the RuleExecutionSet.

For more information on the createRuleExecutionSet() method and its allowable parameters, see documentation on the RuleExecutionSetProviderImpl and LocalRuleExecutionSetProviderImpl classes in the Javadoc. CA Rule Engine supports all input parameters of the createRuleExecutionSet() method. For example, the following code fragment creates a rule execution set from a rulebase file named "rulebase.xml" without any vendor-specific properties:

```
InputStream inStream = getResourceAsStream("rulebase.xml");
RuleExecutionSet ruleSet = ruleSetProvider.createRuleExecutionSet(
  inStream, null);
```

Note: In some cases, it may be necessary to create the appropriate input to the createRuleExecutionSet(). For the sample code above, it is necessary to create an InputStream from a rulebase file since LocalRuleExecutionSetProvider has been used.

To perform inferencing with the RuleExecutionSet instance, register the RuleExecutionSet through the RuleAdministrator. Specify the URI for the execution set and invoke the registerRuleExecutionSet() method:

```
ruleAdmin.registerRuleExecutionSet(ruleset_uri, ruleSet, null);
```

where *ruleset_uri* specifies the desired URI for the RuleExecutionSet, e.g. *rulebases://shoppingcart* for a rulebase named shoppingcart.

In a solution architecture where the administrative operations are performed in one program and inferencing is performed in another program, the administrative program must obtain an instance of the RuleRuntime from the RuleServiceProvider and serialize it to a binary file:

```
RuleRuntime runtime = svcProvider.getRuleRuntime();
```

Use the following method to serialize the RuleRuntime into a binary file:

```
ObjectOutputStream serRuleRuntime = new ObjectOutputStream(
  new FileOutputStream("runtime.out"));
serRuleRuntime.writeObject(runtime);
```

where *runtime* references the RuleRuntime instance obtained from the RuleServiceProvider.

For more information on obtaining the RuleRuntime, see Establish a Rule Session.

Establish a Rule Session

To conduct inferencing through the JSR-94 API, a rule session needs to be established, that is, an instance of either a stateless or stateful RuleSession needs to be obtained. To accomplish this, obtain an instance of the RuleRuntime first.

You can obtain a RuleRuntime instance in one of the following ways, depending upon the solution architecture of the client application:

- If the client application has created and registered its own RuleExecutionSet, then the RuleRuntime can be obtained directly from the RuleServiceProvider.

  ```
  RuleRuntime runtime = svcProvider.getRuleRuntime();
  ```

- If the client application needs to use a RuleExecutionSet that was created and registered in another program, obtain the RuleRuntime instance by deserializing the serialized binary file of the RuleRuntime that was created in that other program.

  ```
  ObjectInputStream in;
  in = new ObjectInputStream(new FileInputStream("runtime.out"));
  RuleRuntime runtime = (RuleRuntime) in.readObject();
  ```

For more information on serializing the RuleRuntime, see Acquire and Register the RuleExecutionSet.

RuleRuntime serves the following purposes:

- It provides the createRuleSession() method for obtaining a RuleSession instance.
- It provides the constants STATEFUL_SESSION_TYPE and STATELESS_SESSION_TYPE that allows the client application to specify the type of inferencing session it wishes to establish.
- It enables the programmer to retrieve the list of RuleExecutionSet URIs that are registered with the RuleAdministrator through the getRegistrations() method. This capability is important if the solution architecture involved in deserializing the RuleRuntime is on a different client application than the application that registered the RuleExecutionSet.

The createRuleSession() method requires the following parameters:

- The URI of the RuleExecutionSet that is to be used by the inference engine. This URI may be directly available in the client application from the registration of the RuleExecutionSet with the RuleAdministrator or obtained through the getRegistrations() method of the RuleRuntime instance.
- A property map. Under CA Rule Engine, the property map is used to specify the name of the RDL domain that is used with the client application when the rulebase contains more than one domain that is shared, and to specify the generation of the optional Inferencing Summary document. For more information, see Specify a Shared Domain and Obtain Rulebase Documents respectively.
- A specification of the rule session type. The method invocation should specify either STATEFUL_SESSION_TYPE or STATELESS_SESSION_TYPE as a parameter. This parameter determines how the client application interacts with the inference engine, and in particular, how application objects are passed to and retrieved from the inference engine. For more information, see Addition and Retrieval of Inference Engine Objects.

A typical creation of a (stateful) rule session can be:

```
StatefulRuleSession session = (StatefulRuleSession)runtime.createRuleSession(
  rulesetUri, null, RuleRuntime.STATEFUL_SESSION_TYPE);
```

Note: The need to cast the instance returned by the createRuleSession() method. The method returns a RuleSession, but the *session* instance needs to be of a specific kind of session to allow the client application access to the appropriate methods to interact with the inference engine.

Specify a Shared Domain

An RDL rulebase domain that is shared with a client application is called an *appshared domain*; an RDL rulebase may contain many appshared domains. For more information on appshared domains, see Rulebase Definition Language Fundamentals.

A Java client application should interact with only one appshared rulebase domain during an invocation of the executeRules() method. When a rulebase shares more than one domain with client applications, the Java client application must specify the domain for interaction.

The Java client application must identify the interaction domain to the inference engine. The JSR-94 specification provides a convenient means for client application to do this when the rule session is being instantiated. The second parameter of createRuleSession() method, the propertiesMap parameter, can be used to specify the name of the appshared domain to be used by the client application.

To pass the properties map specifying the desired appshared domain to the rule session, create a Java Hashtable and add the AionRuleEngineProperties.RULE_INFERENCING_DOMAIN_NAME property to the table mapped to the name of the target appshared domain name.

Example

```
Hashtable sessionProps = new Hashtable();
sessionProps.put(AionRulesEngineProperties.RULEBASE_INFERENCING_DOMAIN_NAME,
   "domain-name");
StatefulRuleSession session = (StatefulRuleSession)runtime.createRuleSession(
  rulesetUri, sessionProps, RuleRuntime.STATEFUL_SESSION_TYPE);
```

For a stateful rule session, it is possible to select a different domain for each invocation of the executeRules() method. This can be achieved through combined use of the property AionRulesEngineProperties.RULEBASE_AUTO_PUSH_POP_DOMAIN with the value Boolean.TRUE and the setInferencingDomainName() method of CARuleExecutionSetMetadata (see Reset Rules and Switch Domains).

For more information regarding other AionRulesEngineProperties and how they can be accessed, see Access CA Technologies's Extended RuleExecutionSet and Rule Properties.

Note: Classes in the Java client application, which correspond to the rulebase level classes that are used by the interacting appshared domain, must properly wrap those rulebase level classes. For more information, see Rules for Constructing Java Classes. Noncompliance of those Java classes with these rules results in failure to establish the rule session or failure to add objects to the inference engine.

Add and Retrieve Inference Engine Objects

Because the inference engine, under JSR-94, is external to the Java client application, it is necessary to add instances from the client side to the inference engine side and to retrieve the results of inferencing back to the client application. How Java instances are added to and retrieved from the inferencing side depends upon whether the client application has established a stateful or a stateless RuleSession. For more information on establishing RuleSessions, see Establish a Rule Session.

Stateless Versus Stateful Rule Sessions

Both the StatelessRuleSession and StatefulRuleSession instances provide an executeRules() method for invoking the RuleExecutionSet with which those instances were created. The signatures of this method are radically different for each of these rule session types.

A stateless rule session means that the inference engine does not maintain state. That is, the whole inferencing process is completed from a single, synchronous invocation from the client application. The client application cannot go back to the same inferencing process after the single invocation is completed. When a stateless rule session is established, Java client instances are added to and results are retrieved from the inference engine in a single executeRules() call. The executeRules() method provided by the StatelessRuleSession object takes a list of object (references) as an input argument and returns a list of (references to) objects that provide the results of the inferencing session. Once an object reference is passed to the inference engine, it is impossible to modify that reference within the stateless rule session.

Note: The StatelessRuleSession also provides an overloaded version of its executeRules() method for also passing filter objects to the inference engine for filtering the returned results. For more information on filters, see Object Filters.

For information on the StatelessRuleSession.executeRules() methods, see the StatelessRuleSessionImpl class in the Javadoc.

A stateful inferencing session allows the client application to have multiple interactions with the inference engine during the rule session. The inference engine maintains state through these interactions. Objects can be added to and removed from the inference engine in separate methods, and these methods are independent of and can be interspersed with the StatefulRuleSession.executeRules() method. The StatefulRuleSession offers a number of methods to add objects, either singly or by means of a list, and to update, retrieve, and remove objects. The executeRules() method within a stateful rule session takes no arguments and returns void.

Important: The StatefulRuleSession is not intended to be used to execute certain rules repeatedly for different combinations of precondition values, such as applying the same rules for 100 individual cases. Such operations may not yield expected results due to side effects from previous executions and behaviors will be specific to each vendor implementation. The Java application should instead repeatedly use StatelessRuleSession with each combination of preconditions instead. The StatefulRuleSession is best used in situations where a variable number of preconditions may be needed to resolve a problem depending on the values of the supplied preconditions, and it is desirable to supply certain preconditions only when it is necessary. For a use case illustrating this kind of usage of the StatefulRuleSession, see Reimbursement Example.

If it is desired to have the stateful session to resolve to the same final results compared with those from a stateless session given the exact same preconditions just all at once, the kinds of interactions allowed during a stateful session should be limited to those operations implicitly performed on behalf of the stateless session. In general, it is always safe to add new objects or to update a previously unknown field with a value during the stateful rule session, while removing an object or changing the value of a known field without resetting the session should be avoided. For more information, see the guide on engine behavior.

It is possible to reset the state of the inference engine during a stateful rule session using the reset() method. For more information on the methods of the StatefulRuleSession, see the StatefulRuleSessionImpl class in the Javadoc.

Note: The object-add methods return Handles to the objects added to the inference engine. These handles are used to interact with those objects during the stateful rule session. For more information on Handles, see the HandleImpl class in the Javadoc.

A rule session should always be released after all interaction with the inference engine has been concluded:

```
session.release();
```

Adding and Creating Objects During Rule Sessions

The following principles affect the interaction of objects across the Java client and inference engine during a rule session when objects are either added to the inference engine or created by the inference engine:

- When an object is added that holds references to other objects as property values corresponding to rulebase *inst_ref* fields, the referenced objects are also added implicitly to the rule session. These implicitly added objects are verified against the corresponding rulebase classes. For more information, see Rules for Constructing Java Classes.
- An object added to a rule session is merely made available for use by the engine. Engine will only interact with an added object when any rule premise needs to get the value of a field of that object or any rule action needs to update the value of a field of that object. However, if an object is updated or removed, engine is notified of the change.
- CA Rule Engine may need to create new instances of Java objects corresponding to rulebase instances created as a result of inferencing. CA Rule Engine can create these objects only if the Java Class corresponding to the rulebase instance's class has been verified and known to it.
- Additionally, CA Rule Engine only creates or updates Java objects corresponding to rulebase instances when rule actions require such changes. In particular, if a post-condition field has been set previously, CA Rule Engine will not reset it to unknown. It is client's responsibility to make sure all post-condition fields are set to unknown (Java null value) prior to inference.

Difference between Objects for Static and Dynamic Instances

RDL makes a distinction between static and dynamic instances. Static instances are declared in the definition of RDL class. Dynamic instances are created by the engine based on rule actions or is created by the client if the RDL class is declared to be application creatable.

For the same RDL class, even though objects of the same Java wrapper class are used for both static and dynamic instances, client should note the following subtle differences between them so as to avoid potential problems:

- A static instance is always known to the engine. In principle, it is only necessary to add an object corresponding to a static instance if it contains pre and or post condition fields. For example, if the RDL class is intended for an enumeration type and static instances defined in that class represents values of the enumeration, it is only necessary to create objects for values which are actually referred by other objects, and to add them to the session prior to inference. CA Rule Engine will create additional objects for the remaining values when needed. However, it is always correct to add all objects for static instances to a rule session prior to inference.

- An object for a static instance should never be removed from a stateful rule session. This is again due to the fact that static instances are always known to the engine which makes it meaningless to remove such an object.
- The CA Rule Engine can create objects for dynamic instances of any defined RDL class. Client can only create objects for dynamic instances of RDL classes which have been declared application creatable.
- Any client created objects for dynamic instances must be added to the session for CA Rule Engine to be aware of their existence. However, such objects can be created or added prior to inference and during initialization callback.
- Client created objects for dynamic instances can be removed from a stateful rule session.

Object Filters

Object filters are Java objects (instances) that provide a method by which a JSR-94 implementation can return a subset of its objects to the client. These instances are passed to a JSR-94 implementation, where they are used to filter objects returned to the Java side. The JSR-94 specification stipulates several techniques for using object filters:

- The StatefulRuleSession.getObjects() and StatelessRuleSession.executeRules() methods are offered with overloaded forms that take an object filter.

  For example, if the Java client application wants to retrieve only Customer objects after executing rules, the filter instance that is passed to the getObjects() or the executeRules() method should check the class of an object to accept or reject it.
- The RuleExecutionSet class provides the setDefaultObjectFilter() method to set a default filter class name for a RuleExecutionSet instance. That default filter class is used whenever the rule execution set is used in a session, that is, if a rule execution set has a default object filter, then calling getObjects() without specifying object filter will cause filtering to occur with the default filter. For more information on the setDefaultObjectFilter(), see the RuleExecutionSetImpl class in Javadoc.

Filter classes must implement the ObjectFilter interface, which is provided by the javax.rules package. This interface stipulates the following public methods:

```
Object filter(Object obj);
void reset();
```

ObjectFilter class implementations require the following.

- A public constructor
- The main filter() method that performs the filtering

  The obj parameter is the object to be filtered. The return parameter is the result of the filtering or the null value.
- The reset() method to allow the filter to be reset to an initial state

  This method needs to be implemented for stateful filters.

Example

For example, the following ObjectFilter implementation (taken from the JSR-94 Specification document), filters objects based on Class.

```
public class ClassFilter implements ObjectFilter
{
    private Class filterClass;
    public ClassFilter( Class clazz )
    {
         filterClass = clazz;
    }

    public Object filter( Object obj )
    {
         if ( filterClass.isAssignableFrom( obj.getClass() ) )
         {
               return obj;
         }
         return null;
    }
    public void reset()
    {
    }
}
```

This example may not be suitable for an environment where multiple ClassLoaders are present.

For more examples of ObjectFilters, see Filtering the Returned Rulebase Objects.

Processing Considerations

This section explains various special processing considerations that are relevant to programming the Java client application.

Specify Unknown Rulebase Fields

A Java *null* value for a Java property to CA Rule Engine specifies that the corresponding rulebase field is *unknown* for all types of rulebase fields. For example, assuming a Java class *Person*, which contains the field *spouse*, has been constructed properly, the following specifies to CA Rule Engine that Audery's spouse is unknown.

```
Person p = new Person("Audery");
p.setSpouse(null);
```

However, Java null value cannot be assigned to a Java primitive type variable, such as an int type variable. Thus it is impossible to specify that a rulebase field corresponding to a Java property of a primitive type is unknown to CA Rule Engine on input, nor can CA Rule Engine on output indicate that the value for that field is still unknown. Therefore, use of primitive type should be avoided if possible.

Specify Instance Reference Fields

Instance reference fields in CA Rule Engine are used to refer to another rulebase objects of the same class or of different classes. Since a Java property of a class type is also a reference to an object of that class, that Java property can be used directly to indicate instance reference relationship to CA Rule Engine. For example, using the Person class mentioned above again, the spouse field apparently is a reference to another object of the Person class. Thus the Person class may be defined as the following:

```
public class Person implements Serializable {
    private String instanceName = null;
    private Person spouse = null;
    //  Other Property declarations
    //  The constructor method
    public Person()
    {
        // Optional object initiation code
    }
    public Person(String name)
    {
        this.instanceName = name;
        // Optional object initiation code
    }
    public String getInstanceName() {
        return this.instanceName;
    }
    public void setInstanceName(String instanceName) {
        this.instanceName = instanceName;
    }
    public Person getSpouse() {
        return this.spouse;
    }
    public void setSpouse(Person spouse) {
        this.spouse = spouse;
    }
    //  Other methods
}
```

And the following indicates to CA Rule Engine that Fred and Mary are spouses:

```
Person pFred = new Person("Fred");
Person pMary = new Person("Mary");
pFred.setSpouse(pMary);
pMary.setSpouse(pFred);
```

However, if it is known that Max is single, a special value must be reserved to specify to CA Rule Engine that Max does not have a spouse. That special value corresponds to the RDL NULL value. Now that the Java null value has been used to specify unknown already, the special instance with null instance name is used for this purpose. So the following should be used for Max:

```
Person pMax = new Person("Max");
Person pRDLNull = new Person(); // instanceName is null by default
pMax.setSpouse(pRDLNull);
```

Specify Array Property Values

Arrays are used to represent rulebase fields that are defined as sets. In this case:

- A null value of the array indicates unknown value. For example, Integer[] intArray = null defines an array that is unknown.
- Arrays of zero length indicate an empty set, which is a known value. For example, intArray = new Integer[] { } indicates that there are no current integers members in the array intArray.
- Arrays of length greater than zero are translated into corresponding rulebase sets with the corresponding instance members, but taking into account element uniqueness.

Any null value used as a member value in an array that corresponds to a rulebase set is ignored and a WARN message generated. Such usage is not meaningful to rulebase and thus should be avoided. For example, intArray[0] = null, where intArray is an array of Integers, yields an empty set (the null value is ignored) of numbers, if there is only one element in that array. If valid, unique and non-null members are also present in that array, that array defines a set of just those values to the inference engine. The reason to ignore any array element that is null is to satisfy the requirement that all objects in CA Rule Engine must be named and thus can never be unknown. Only fields of an object may be unknown. Therefore, it is impossible to have a rulebase set of objects that contain unknown objects.

Specify Datetime String Values

CA Rule Engine accepts and returns datetime string values in the World Wide Web Consortium (W3C) schema format. For more information on datetime handling and the W3C schema format, refer to the RDL specification.

For datetime values, the datetime string may specify:

- Only a date value (for example, "2000-01-23")
- Only a time value (for example, "15:34:23" or "10:43")

  Both a date and time value (for example, "2000-01-23T15:34:23")

Any form of the datetime string may optionally specify a “Z” suffix (for example, "2000-01-23T15:34:23Z") indicating a GMT value. The absence of this suffix indicates a local-time value.

Specify Duration String Values

CA Rule Engine accepts and returns duration string values in the World Wide Web Consortium (W3C) schema format. For duration, W3C schema format is extended to allow specification of an optional start datetime for precise duration handing. For more information on duration handling and the W3C schema format, refer to the RDL specification. The start date time extension for duration constants allows a datetime value, which must include a date portion, to be appended to a W3C duration value with ";" as delimiter between them to indicate that the duration value starts at the specified datetime.

For duration values, both positive and negative values are allowed, the duration string may specify:

- Only date constituent values (for example, "P3Y" and "-P1M2D")
- Only time constituent values (for example, "PT23M")
- Both date and time constituent values (for example, "P12Y9M15DT10H11M23S")
- Any of above with start date time (for example, "P3Y;2001-04-01", i.e. a duration of 3 years starting from April 1, 2001, and "-P3Y;2001-04-01", i.e. a duration of 3 years going backwards from April 1, 2001)

Reset Rules and Switch Domains

There are cases where it may be desirable for a stateful rule session to simply reset the rules while retaining the input data, such as in the case of carrying out experiments where most of the parameters are held constant and only one or few are modified at a time. Resetting the session also erases all input data, which means that all objects have to be added again even though most of them have not changed. It is more efficient to update the changed objects and reset the rules only. Resetting rules in the inference engine of CA Rule Engine is carried out through popping the domain and pushing it back in again. The JSR-94 API doesn't define a method that resets the rules, but user can use the property AionRulesEngineProperties.RULEBASE_AUTO_PUSH_POP_DOMAIN with the value Boolean.TRUE to configure the stateful rule session so that each invocation of the executeRules() method automatically performs the domain popping and pushing and thus resetting the rules without losing data.

The same feature, which carries out automatic domain popping and pushing by executeRules() method, can also be used to switch to a different appshared domain in combination with the setInferencingDomainName() method of CARuleExecutionSetMetadata. The following sample code illustrates domain switching during runtime:

```
Hashtable sessionProps = new Hashtable();
sessionProps.put(AionRulesEngineProperties.RULEBASE_AUTO_PUSH_POP_DOMAIN,
Boolean.TRUE);
sessionProps.put(AionRulesEngineProperties.RULEBASE_INFERENCING_DOMAIN_NAME,
"Domain1");
StatefulRuleSession session =
(StatefulRuleSession)runtime.createRuleSession(rulesetUri, sessionProps,
RuleRuntime.STATEFUL_SESSION_TYPE);
…
session.executeRules(); // Inference over Domain1
…
// Now switch to Domain2
CARuleExecutionSetMetadata mData =
(CARuleExecutionSetMetadata)session.getRuleExecutionSetMetadata();
mData.setInferencingDomainName("Domain2");
…
session.executeRules(); // Inference over Domain2
…
```

Note: The feature, which carries out automatic domain popping and pushing by executeRules() method, disables the incremental inference feature of the stateful rule session which requires rule states to be maintained across successive invocations of executeRules() method. However, these two features generally apply to different circumstances.

Exception Catching

Methods exposed by CA Rule Engine support exceptions. These exceptions are thrown by the inference engine. For more information on the methods that throw exceptions, see the Javadoc. For more information on exception messages, follow the "error, warning and trace messages" link that is in the Overview page of Javadoc.

It is recommended that the Java client application use standard Java try/catch programming around CA Rule Engine methods that throw exceptions.

Query RuleExecutionSets and Rules

The JSR-94 specification requires that vendors provide a method for querying RuleExecutionSets and Rules. This method is provided by the RuleAdministration functionality and the RuleSession API. The specification requires support of a common set of properties because the JSR-94 specification fully expects that individual vendors will want to include their RuleExecutionSet and Rule properties; the JSR-94 specification allows *extensions* to accommodate these vendor-specific properties. For more information on CA's property extensions, see Access CA's Extended RuleExecutionSet and Rule Properties.

Query Common RuleExecutionSet and Rule Properties

When a RuleExecutionSet instance is obtained from the appropriate RuleExecutionSetProvider (see Acquire and Register the RuleExecutionSet), that RuleExecutionSet may be queried. The RuleExecutionSet interface provides the following query operations:

```
String getName();

String getDescription();

String getDefaultObjectFilter();

List getRules();
```

For more information on these operations, see the RuleExecutionSetImpl class in the Javadoc.

The first two operations are straightforward. For more information on getDefaultObjectFilter(), see Object Filters. The operation getRules() returns a list of rules. For example:

```
List rules = ruleset.getRules();
```

Each rule provides the following operations to retrieve its name and description:

```
String getName();

String getDescription();
```

Rules may be accessed by iterating through the list of rules. For example:

```
Iterator iter = rules.iterator();
while (iter.hasNext() ) {
  Rule rule = (Rule)iter.next();
  System.out.println("\tRule name: " + rule.getName());
  System.out.pringln("\tRule description: " + rule.getDescription());
}
```

Both the RuleExecutionSet and Rule interface offer operations by which vendor specific properties can be set and retrieved. Each interface provides the following operations:

```
Object getProperty(Object key);
void setProperty(Object key, Object value);
```

For more information on these operations, see the RuleExecutionSetImpl and RuleImpl classes in the Javadoc.

For more information on properties used by CA Rule Engine, see Access CA's Extended RuleExecutionSet and Rule Properties.

Access CA Technologies's Extended RuleExecutionSet and Rule Properties

JSR-94 permits the inference engine vendor to extend the interface specification to allow a vendor to provide client applications with specific information about RuleExecutionSets and rules that are unique to the vendor's implementation of the standard. These methods include:

- User or vendor-defined properties can be accessed through the getProperty() and setProperty() methods of the RuleExecutionSet and Rule classes.
- RuleExecutionSet properties that pertain to the use of the RuleExecutionSet within a rule session can be accessed from a RuleSession instance using the RuleSession.getRuleExecutionSetMetadata() method, which returns an instance of RuleExecutionSetMetadata. The RuleExecutionSetMetadata API allows access to rule execution set properties such as the RuleExecutionSet URI, name, and description. CA Technologies has specifically extended this class. The accessor methods of CA's extended version of this interface are specifically designed to retrieve the CA Technologies-defined properties of RuleExecutionSets.

  Note: Rule information is not available through the CARuleExecutionSetMetadata interface.

  In CA Rule Engine, the sets of properties that can be obtained by RuleExecutionSet.getProperties() have corresponding specific CA Technologies accessors of CARuleExecutionSetMetadata. However, if a property is only applicable to a particular execution of the rule session (for example, the optionally generated Inferencing Summary document), then that property can only be obtained using CARuleExecutionSetMetadata. An inferencing summary document is an XML document that summarizes rule behavior during the course of a rule session. For more information on obtaining this optional document, see Obtain Rulebase Documents.

All object names (constants) for the CA extended RuleExecutionSet and Rule properties are provided in the AionRulesEngineProperties interface. This interface must be imported into any application program that will access RuleExecutionSet and Rule properties.

The following table summarizes extended properties provided by CA Rule Engine for rules:

Property	Property Constant
Unqualified name of the rule	RULE_NAME
Priority of the rule	RULE_PRIORITY
Name of the domain of the rule	RULE_RULESETDOMAIN
Name of the ruleset to which the rule belongs	RULE_RULESETNAME

Property	Property Constant
Priority of the ruleset to which the rule belongs	RULE_RULESETPRIORITY

Example

The following example retrieves the name of the ruleset to which a given rule belongs:

```
String ruleSetName = (String) rule.getProperty(
  AionRulesEngineProperties.RULE_RULESETNAME);
```

Important! These CA Technologies specific properties are exposed as Read-Only. Invocations of the RuleExecutionSet.setProperty() and Rule.setProperty() methods to set these properties will not be allowed. For more information, see the documentation on the setProperty() methods for the RuleExecutionSetImpl and RuleImpl classes in the Javadoc.

The following table summarizes the extended properties and the corresponding operations of the CARuleExecutionSetMetadata interface provided by CA Rule Engine for RuleExecutionSets that are not specific to a rule session:

Property	Property Constant	CARuleExecutionSetMetadata operation
The application interface document	RULEBASE_APP_INTERFACE_DOCUMENT	getAppInterfaceDocument()
Compilation date/time	RULEBASE_COMPILATION_DATETIME	getCompilationDateTime()
Binary format level of the rulebase	RULEBASE_FORMAT_LEVEL	getFormatLevel()
Currently supported binary format level for rulebases	RULEBASE_LATEST_FORMAT_LEVEL	getCurrentFormatLevel()
The load map XML document for the RuleExecutionSet (rulebase)	RULEBASE_LOAD_MAP_DOCUMENT	getLoadMapDocument()

Example

For example, to retrieve the rulebase format level of a RuleExecutionSet from outside a rule session the following code can be used:

```
String ruleSetFormatLevel = (String) ruleSet.getProperty(
    AionRulesEngineProperties.RULEBASE_FORMAT_LEVEL);
```

To retrieve this same information from within a rule session through the CARuleExecutionSetMetadata API, the following code can be used (where *session* references a previously created RuleSession):

```
CARuleExecutionSetMetadata ruleExecSetInfo = (CARuleExecutionSetMetadata)
    session.getRuleExecutionSetMetadata();
String ruleSetFormatLevel = ruleExecSetInfo.getFormatLevel();
```

Note that the instance returned by the getRuleExecutionSetMetadata() method must be explicitly cast as an instance of CARuleExecutionSetMetadata to make the CA Technologies-extended accessors available to the client application.

RuleExecutionSet metadata can be obtained from within a rule session regardless of whether the rule session is stateless or stateful.

For more information on the RULEBASE_APP_INTERFACE_DOCUMENT and RULEBASE_LOAD_MAP_DOCUMENT properties, see Rulebase Structures.

The following table summarizes the extended properties and the corresponding operations of the CARuleExecutionSetMetadata interface provided by CA Rule Engine for RuleExecutionSets that are rule session specific:

Property	Property Constant	CARuleExecutionSetMetadata operation
Name of the appshared domain to use	RULEBASE_INFERENCING_DOMAIN_NAME	getInferencingDomainName()/setInferencingDomainName()
Whether to generate inferencing summary	RULEBASE_GENERATE_INFERENCING_SUMMARY	isGeneratingInferencingSummary()
Whether to automatically push/pop domain	RULEBASE_AUTO_PUSH_POP_DOMAIN	isAutoPushPopDomain()

The values for the all the properties shown in the table above are user-defined. If used, they must be specified at the same time that the rule execution set is created. Since they are specific to a rule session, the value of these properties should be queried using the CARuleExecutionSetMetadata operations. The value for RULEBASE_INFERENCING_DOMAIN_NAME property should be the name of selected domain in String type. It is required only when more than one appshared domains are defined in the rulebase. For a rulebase with multiple appshared domains the setInferencingDomainName() method can be used to switch domains at runtime, see Reset Rules and Switch Domains. The values for properties RULEBASE_GENERATE_INFERENCING_SUMMARY and RULEBASE_AUTO_PUSH_POP_DOMAIN should be Boolean.TRUE to turn on the specified features for the rule session. Both properties are optional and need to be defined only when the respective feature is desired.

Since the optional inferencing summary document is a RuleExecutionSet property that pertains to the execution of a specific rule session, it is only available through the getInferencingSummary() method of the CARuleExecutionSetMetadata API. This optional document is generated if the RULEBASE_GENERATE_INFERENCING_SUMMARY property is set to Boolean.TRUE at the creation of the rule session. For more information, see Obtain Rulebase Documents.

Use Cases for Building JSR94 Applications

JSR-94 applications can be built according to the following use cases:

- You may begin with a pre-existing rulebase for which you want to write appropriate Java classes to send data to the rulebase, but you may or may not have direct access to the rule author or RDL code to understand the object interface. In this use case, you likely need to know how to query the interface of the rulebase. For more information, see Rulebase Structures.
- You may begin with pre-existing business objects defined by Java or by developing the rulebase concurrently with Java developers. In this use case, there is a need to prepare a rulebase (in RDL) to inference over these objects. For more information, see Construct an RDL Rulebase for Java Objects.
- Combination of the above two cases, such as that you may begin with a pre-existing rulebase, for which there is a need to add new rules to handle some new business objects.

Rulebase Structures

If the rulebase has been already been composed before defining the Java objects, for example, if the rulebase has been written by a person familiar with business logic, the Java programmer must have access to some details of the rulebase before constructing the Java application. Structural information about the rulebase is provided in the documents - the App Interface and Loadmap documents that can be obtained through CA Rule Engine. For more information, see Obtain Rulebase Documents.

Note: For the immediate purpose of constructing the Java client application, the App Interface document is the more important one of the two.

The App Interface and Loadmap documents are XML documents; knowledge of how to parse XML is necessary to read the documents. For more information on the content of these documents, see The App Interface Document and The Rulebase Loadmap Document. Details related to these documents are provided in the Javadoc in the com.ca.cleverpath.aion.rulesp.core.common package. See XMLDoc_AppInterface and XMLDoc_LoadMap within the rulesp.core.common package. In addition, the XML schemas for these documents are provided in the schema folder inside the CA Rule Engine distribution.

Obtain Rulebase Documents

Rulebase documents are obtained as values of CA extended properties of RuleExecutionSets. For more information on the CA extended properties for CA Rule Engine, see Access CA's Extended RuleExecutionSet and Rule Properties. Three kinds of documents are available: the App Interface document, the Loadmap document and the optionally generated Inferencing Summary document. See the respective sections for these documents below for more information.

The App Interface document and the Loadmap document are static for a rulebase and thus may be obtained for the RuleExecutionSet with the RuleExecustionSet.getProperty() method or directly through a CARuleExecutionSetMetadata accessor. The Inferencing Summary document is tied to a rule session and thus can only be obtained through the CARuleExecutionSetMetadata accessor method getInferencingSummary().

For example, to obtain the App Interface document through the getProperty() method, the following can be used (where *ruleset* references a RuleExecutionSet instance):

```
String xmldoc = (String) ruleSet.getProperty(
    AionRulesEngineProperties.RULEBASE_APP_INTERFACE_DOCUMENT);
```

Note: If a rulebase file is passed to the WrapperMaker tool to generate the Java classes, an App Interface document is also written out. It can be used to facilitate selection of the appropriate appshared domain if more than one such domains exist in the rulebase.

The corresponding AionRulesEngineProperties constant for obtaining the loadmap is RULEBASE_LOAD_MAP_DOCUMENT.

Equivalent code to obtain the App Interface document within a rule session using the CARuleExecutionSetMetadata API is (where *session* references a RuleSession instance):

```
CARuleExecutionSetMetadata ruleExecSetInfo =
    (CARuleExecutionSetMetadata) session.getRuleExecutionSetMetadata();
String xmldoc = ruleExecSetInfo.getAppInterfaceDocument();
```

The corresponding CARuleExecutionSetMetadata accessor to obtain the loadmap is getLoadMapDocument(). For more information on the CARuleExecutionSetMetadata accessors, see the CARuleExecutionSetMetadataImpl class in the Javadoc.

The Inferencing Summary document is generated only when the RULEBASE_GENERATE_INFERENCING_SUMMARY property is set to true at the creation of the rule session. The procedure to set that property is identical to that of specifying the desired appshared domain for a rule session. For example, the following code fragment creates a stateful rule session that will generate the Inferencing Summary document upon execution:

```
Hashtable sessionProps = new Hashtable();
sessionProps.put(AionRulesEngineProperties.RULEBASE_GENERATE_INFERENCING_SUMMARY,
    new Boolean(true));
StatefulRuleSession session = (StatefulRuleSession)runtime.createRuleSession(
    rulesetUri, sessionProps, RuleRuntime.STATEFUL_SESSION_TYPE);
```

After execution of the session, the Inferencing Summary document can be obtained by:

```
CARuleExecutionSetMetadata ruleExecSetInfo = (CARuleExecutionSetMetadata)
session.getRuleExecutionSetMetadata();
String xmldoc = ruleExecSetInfo.getInferencingSummary();
```

The App Interface Document

The App Interface document is an XML document that summarizes the rulebase objects that are shared with applications. It is the critical document for constructing the classes in a Java client application that will correspond to classes in the rulebase. For more information on the rules of constructing Java classes that will correspond to rulebase classes, see Rules for Constructing Java Classes.

The document covers the following features of the rulebase interface with the Java client application:

- **The appshared domains of the rulebase.** Although the rulebase may provide several domains that can be shared with Java client applications, a rule session can only interface with a single domain at a time. For more information, see Specify a Shared Domain. Information provided with the domains is their names (to be used as a RuleSession property when creating the rule session), the pre-/postconditions fields that correspond to the input and output values of the inferencing session, and if appropriate, the list of application creatable classes within the domain.
- **The classes that must be created in the Java client application.** This information determines how classes are to be named in the Java client application, see Class Name Requirements. The class definitions also provide the field names and data types and names of class instances.
- **The names and data types of the fields of the class.** The Java application client must define its classes as Beans with the proper properties and accessors. For more information, see Property Accessors and Property Matching. For more information on rules for matching rulebase field types to property types in the Java client application, see Relationship Between RDL Field Data Types and Java Property Types.
- **The names of rulebase class instances.** The names of instances in the rulebase are especially important because the Java client application classes must provide an instanceName property. Instances passed to the inference engine must hold the corresponding rulebase instance name as the value of this property. For more information, see Instance Naming.

For a complete description and example of the App Interface document, see XMLDoc_AppInterface link in the com.ca.cleverpath.aion.rulesp.core.common package in the Javadoc. The schema of the App Interface document is provided in the schema/appinterface folder inside the CA Rule Engine distribution.

The Rulebase Loadmap Document

The Loadmap document is an XML document that summarizes the objects defined within a binary rulebase. This informative document is a concise and complete summary of a rulebase's content. However, Loadmap documents detail objects that may be inaccessible to client apps (for example, rulebase-private classes) and therefore are probably more useful to rulebase authors than to Java client programmers. The loadmap covers:

- **Rulebase initialization methods**. Initialization methods carry out operations, such as setting a field with certain value, prior to inferencing over the rules. For more information on initialization methods, see the guide on RDL.
- **Rulebase associations**. Associations are reciprocal relationships between class instances. For more information on associations, see the guide on RDL.
- **Rulebase classes**. Information on classes includes a class's parent and child classes, fields, and instances. For more information on classes, see the guide on RDL..
- **Domains**. The <domains> element of the Loadmap covers each domain in the rulebase and its contents. It covers the initialization, associations, classes, pre- and postconditions, creatable classes, and rulesets of each domain. For more information, see the guide on RDL.. More information on rulesets is also provided by this document, including the information on each rule in a ruleset. Rule information is limited to the effective dates of the rule and a list of terms (class/field/instance) used by the rule. The actual rule structure is not revealed.
- **Internal References**. References are cross-references between the rulebase elements as determined by the RDL parser.

For a complete description and example of the Loadmap document, see XMLDoc_LoadMap link in the com.ca.cleverpath.aion.rulesp.core.common package in the Javadoc. The schema of the Loadmap document is provided in the schema/loadmap folder inside the CA Rule Engine distribution.

The Inferencing Summary Document

The Inferencing Summary document is an optionally generated XML document that summarizes rule behavior during the course of an inferencing session. Though rule behavior can be gleaned from detailed log messages, infer summaries offer some important advantages:

- When summary totals, rather than event detail, are sought, the data format is more convenient.
- As compared to detailed tracing, summaries have less impact on Engine performance.
- As a summary, the data representation is compact.

This last advantage (compactness) is particularly important when analyzing behavior for rules with iterative decisions - rules with possibly thousands of rule threads.

The Inferencing Summary document is a valuable aid to tuning the performance of a rulebase and thus mostly useful to rule authors. It covers summaries at the following nested levels:

- **Rulebase level**. The document's root element (<infer_summary>) describes overall document properties and summarizes overall behavior of the inferencing session.
- **Domain level**. The root element's children are one or more Domain elements (<domain>). The Domain elements summarize domain-level inferencing activity.
- **Ruleset level**. A Domain element's children are zero or more Ruleset elements (<ruleset>). The Ruleset elements summarize ruleset-level inferencing activity.
- **Rule level**. A Ruleset element's children are one or more Rule elements (<rule>). The Rule elements summarize rule-level inferencing activity.
- **Decision level**. A Rule element's children are zero or more Decision elements (<decision>). The Decision elements summarize decision-level inferencing activity.
- **Alternative level**. A Decision element's children are zero or more Alternative elements (<alternative>). An alternative is one action branch off a decision. The Alternative elements summarize alternative-level inferencing activity.

For a complete description and example of the Inferencing Summary document, see XMLDoc_InferSummary link in the com.ca.cleverpath.aion.rulesp.core.common package in the Javadoc. The schema of the Inferencing Summary document is provided in the schema/infersummary folder inside the CA Rule Engine distribution.

Construct an RDL Rulebase for Java Objects

In some applications of CA Rule Engine, the business objects may have already been defined as Java classes. In this use case, the goal is to enhance the use of these business objects by providing a context in which the objects are processed by business rules.

The rulebase for this application needs to implement the business logic behind the set of business objects. When authoring a rulebase, it may be easier to use the more readable infix rule language to define rules. The Infix2RDL tool can be used to convert a rulebase with infix rules to a pure RDL rulebase. To facilitate understanding of existing RDL rulebase, the RDL2Infix tool can be used to convert an RDL rulebase back to one with infix rules. For more information on Infix2RDL tool, RDL2Infix tool and RDL features see Using the Infix2RDL Tool, Using the RDL2Infix Tool, and Overview of RDL rulebases respectively.

It may also be necessary to retrofit the Java class definitions to be used by the RDL rulebase that will be created. The retrofitting may require verifying that:

- The Java classes are fully defined according to the standard for Beans. For more information, see Public No-Argument Constructor and Property Accessors.

  Note: Any object name exposed to JSR-94 clients must be a valid Java-object name. BeanInfo may be used to map names not valid in Java.

- Class names and property names accord with the expected rules for interfacing with an RDL rulebase. For more information, see Class Name Requirements and Property Matching. Property types are compatible to the corresponding RDL data types. For more information, see Relationship Between RDL Field Data Types and Java Property Types.

- All classes must have an instanceName property with proper accessor methods. For more information, see Instance Naming.

- All Java client classes must implement the Serializable interface if serialization of session is needed.

One approach to retrofitting existing Java classes is to determine the rulebase structure first so that a prototype rulebase can be created with dummy rules if needed, then use the WrapperMaker tool (see Using the WrapperMaker Tool) to generate the required Java classes from the prototype rulebase and then merge the existing Java classes with the generated ones.

Using the Infix2RDL Tool

The Infix2RDL tool facilitates the process of authoring RDL rulebases by converting a rulebase with infix rules to a pure RDL rulebase, thus allowing the much more readable infix language to be used to define rules.

For ease of invocation of the Infix2RDL tool, the infix2rdl batch/script files have been supplied in the bin folder inside the CA Rule Engine distribution. The syntax for invoking the tool is:

```
infix2rdl rulebase_infix.xml [rulebase.xml]
```

where *rulebase_infix.xml* represents the file name of the rulebase with infix rules, and *rulebase.xml* represents the file name of the generated RDL rulebase. It is customary to name the infix rulebase with the *_infix.xml* suffix. If this convention is used, the file name of the generated RDL rulebase is optional, and the Infix2RDL tool script will by default generate the file name for the output RDL rulebase by removing the *_infix* portion from the input file name. If this convention is not used, one needs to explicitly specify the file name of the generated RDL rulebase.

The Infix2RDL tool script also supports drag and drop operation if drag and drop is supported by the OS GUI and the above naming convention is used. When an infix rulebase file is dropped onto the Infix2RDL tool script icon, it is equivalent to invoke the script with the name of that file as the first parameter.

Using the RDL2Infix Tool

The RDL2Infix tool facilitates the understanding of existing RDL rulebases by converting a pure RDL rulebase to one with infix rules.

For ease of invocating the RDL2Infix tool, the rdl2infix batch/script files have been supplied in the bin folder inside the CA Rule Engine distribution. The syntax for invoking the tool is:

```
rdl2infix rulebase.xml [rulebase_infix.xml]
```

where "rulebase.xml" represents the file name of the RDL rulebase, and "rulebase_infix.xml" represents the file name of the generated rulebase with infix rules. It is customary to name the infix rulebase with the "_infix.xml" suffix. If this convention is used, the file name of the generated infix rulebase is optional, and the RDL2Infix tool script will by default generate the file name for the output infix rulebase by appending the "_infix" portion to the input file name. If this convention is not used, you need to explicitly specify the file name of the generated infix rulebase.

The RDL2Infix tool script also supports drag and drop operation if drag and drop is supported by the OS GUI and if the above naming convention is used. When a RDL rulebase file is dropped onto the RDL2Infix tool script icon, it is equivalent to invoking the script with the name of that file as the first parameter.

Chapter 3: RDL Rulebase Overview

This chapter presents an overview of CA Rule Engine rulebases. The presentation is at a conceptual level; it does not get into the specifics of the RDL. For a complete description of RDL, refer to the RDL Specification.

CA Rule Engine rulebases are coded as XML documents in the Rulebase Definition Language (RDL). For clarity, this document often employs an *infix-style notation* for illustrating rule logic. For a human reader, this notation is easier to follow than the corresponding native RDL (XML) code.

This section contains the following topics:

Rulebase Fundamental Notions

Rule

A **rule** is an atom of knowledge expressing dependencies amongst field values. A rule consists of one or more **premises** (Boolean expressions typically testing field values) and, associated with each premise, an **action** (statements typically changing field values).

Rulebase

A **rulebase** is a collection of objects defining not only the rules but also organizational objects (e.g., rulesets, domains) and the objects referenced by the rules (e.g., classes, instances and fields). A rulebase may consist of hundreds, even thousands, of rules.

Rule author

A **rule author** or **rulebase author** is one who maintains rulebases - usually via a tool such as a rulebase editor. The rule author is a knowledge expert who encodes some or all of her/his knowledge in the form of rules. The author's expertise may involve any knowledge area - e.g., loan approval, medical analysis, chemical analysis, etc.

At inferencing time, a field may be in either a **resolved** (has a value) or **unresolved** (does not have a value) state.

Rules typically have dependencies on one another - i.e., one rule's premises may test field values resolved by another rule's action. Likewise, multiple rules may be alternatives to one another - i.e., perhaps resolving the same fields to different values depending on premise conditions.

The rule author may specify rules in any order - so there needs to be some sort of generic mechanism for determining how best to apply rules for resolving field values. We call this mechanism an **inference engine** (or **Engine**).

During inferencing, the Engine maintains a hierarchical **agenda** of active rulesets and rules.

When the Engine initializes a rule on the agenda, the Engine creates a **rule thread**. The thread is a runtime instantiation of the rule - and, due to decision iteration, the thread may, in turn, spawn additional child threads; and a child thread may do the same. As such, a rule extends the agenda hierarchy with its own hierarchy of threads.

When visiting a rule thread, the Engine evaluates the thread's premises and, if a premise is TRUE, the Engine performs the corresponding action for the thread. The Engine performs at most one action on behalf of a thread.

Rulebase Structure

An RDL rulebase is organized into a hierarchy of objects.

Rulebase Level

At the topmost level, a rulebase defines zero or more of the following:

- Rulebase-level Initialization Methods invoked when the Engine loads a rulebase

  These methods typically initialize "constant fields". For example, *MAX_AGE*.

- Rulebase-level Classes defining fields and static instances

- Rulebase-level Associations defining reciprocal relationships between fields of rulebase-level classes

  For example, an Ownership Association may establish a relationship between a Person.owns field (declared as a set of Ducks) and a Duck.owner field (declared as an instance-reference to a Person). The Engine maintains this relationship. For example, if a Duck changes ownership, the Engine automatically updates the Person.owns fields for both the previous and new owners.

- Domains for specifying categories of knowledge.

Domain Level

Each domain, in turn, may define zero or more of the following:

- Domain-level Initialization Methods (invoked when the Engine loads a domain)
- Domain-level Classes
- Domain-level Associations (applicable to domain-level classes)
- Rulesets for specifying collections of rules

Ruleset Level

Each ruleset, in turn, may define zero or more of the following:

- Ruleset-level Initialization Methods (invoked when the Engine loads a ruleset)
- Ruleset-level Classes
- Ruleset-level Associations (applicable to ruleset-level classes)
- Rules for specifying atomic knowledge

Scoping

Objects are scoped according to their position within the hierarchy. For example:

- A rule can access not only rulebase-level classes but also classes defined by its enclosing domain and ruleset.
- A domain-level initialization method can access not only rulebase-level classes but also classes defined by its enclosing domain.

A lower-level object will obscure same-named objects at higher levels. For example, both the rulebase-level and ruleset-level may define a class: *Class1* - but the ruleset's rules will only *see* the ruleset-level *Class1*. In order to reference the rulebase-level class, a rule would need to further qualify the class name.

RDL Characteristics

RDL

RDL is a specialized language for encoding rulebases. It supports only those features needed for the definition of rules and their associated objects.

RDL is *not* a general-purpose programming language. It does not include more general-use facilities such as I/O or data-formatting facilities. Likewise, it does not support IF/THEN/ELSE statements, because RDL centralizes decision making.

In other cases, RDL avoids language constructs that complicate or hinder inferencing. For example, RDL disallows expressions from having side-effects. As a result, an expression's evaluation cannot result in field-value changes.

The following terms describe RDL specific characteristics:

Object Naming

Object names consist of Unicode characters and are case insensitive.

Object names must be non-empty and may not include any of the following characters:

- Blank space ( )
- Double quote ("")
- Period (.)
- Comma (,)
- Colon (:)
- Parenthesis (open or close)
- Asterisk (*)

Important: Do not define object names that begin with an underscore (_). Such names are reserved for intrinsic identifiers defined in RDL. For example: *_global*, *_curr_datetime*.

Object References

When referencing a rulebase object, RDL code can reference the object directly or indirectly by references to other objects. As seen in examples, RDL code must qualify names only to the extent necessary to avoid ambiguity.

Direct References

If *field1* is declared by only one class (*Class1*) and that class defines only one static instance *(Instance1*), the RDL code can simply reference the field by its name:

```
field1
```

However, if the class defines multiple instances, the RDL code must further qualify the field name:

```
Instance1.field1
```

If multiple classes declare fields and instances of the same name, the RDL code may need to further qualify the field name:

```
Class1.Instance1.field1
```

RDL code may also need to qualify names when different levels of the rulebase hierarchy declare objects of the same name. In this case, the lowest-level declaration will obscure the higher-level declarations.

So, if the rulebase-level, domain-level and ruleset-level all declare instances of Class1, RDL code may need to qualify names in the following manner:

```
_global.Class1.Instance1.field1
domain1.Class1.Instance1.field1
ruleset1.Class1.Instance1.field1
```

In cases where the rulebase employs the same object name for classes, domains and rulesets, the RDL may need to further qualify a name by anchoring it at the rulebase-level:

```
_global.domain1.ruleset1.Class1.Instance1.field1
```

Indirect References

RDL code can also reference objects indirectly using a series of instance references:

```
Duck1.owner.spouse.manager.spouse.myDog.age
```

In contrast to direct references (which can be resolved at compilation time), indirect references require resolution at inferencing time.

When resolving such references, all kinds of things can go "wrong". For example, *Duck1* may not have an owner. This situation is discussed in a subsequent section of this chapter.

Data Types

RDL supports data types for both atomic (individual) values and collection values.

Atomic Data Types with Magnitude

Rule logic can compare values of these data types not only for equality/inequality but also for magnitude (for example, A > B):

Number

A numeric data type of specified precision (digits to the right of the decimal point) that does not distinguish between integers and floating-point values. The values are of arbitrary size and precision. The Engine automatically rounds values appropriately.

String

A sequence of Unicode characters.

DateTime

A data type for a datetime value (e.g., 2004-01-20 14:46:22), a date-only value (e.g., 2004-01-20) or a time-only value (for example, 14:46:22). This data type makes provision for both GMT and client-local time.

Note: In Engine server environments, one may need to take into account the possibility that the client and server may not share the same local time; and that different concurrent clients of the server may have different local times. The Engine resolves these discrepancies by converting all DateTime values to client-local time - so a given Engine instance will only see DateTime values in terms of its own client's local time.

The Engine's geographic location is therefore transparent to client applications. Those clients don't care whether the Engine's server is in Bombay or Hong Kong - or if the server changes location.

Duration

A data type for a span of time expressed in years, months, days, hours, minutes or seconds - or of any combination of these. Duration values may be signed (positive/negative).

Duration values may optionally specify a start DateTime. The Engine applies the start DateTime for more-precise interpretation of year and month counts. When evaluating expressions involving a Duration value lacking a start DateTime, the Engine attempts to infer a start DateTime from other expression operands - thereby improving the interpretation of the Duration value. If all else fails, the Engine applies a default start DateTime (2001-01-01T00:00:00).

Atomic Data Types without Magnitude

Rule logic can compare values of these data types only for equality/inequality:

Boolean

A data type with values: TRUE and FALSE.

Instance Reference

A reference to a class instance. For example, a Duck owner-field may be an instance reference to a Person - indicating the Person owning a Duck.

This data type also defines a constant: NULL that can be used to indicate the absence of a reference. For example, if a Duck had no owner, the Duck's owner-field would reflect a NULL value.

Note: that a NULL value is a value - so, if a Duck has no owner, the owner field is resolved. The owner field would be unresolved if we didn't know if the Duck had an owner.

Sets

The RDL also supports Sets: ordered collections of unique elements of the same data type. For example, a Set of instance references may indicate the Duck instances owned by a given Person. A Set of Numbers may indicate relevant chapters of a Book.

Note: that Set elements must be unique, so a Number Set for Student test scores reflects only distinctive (and not duplicate) test scores. In order to retain duplicate test scores, you can use a Set of Grade instances.

The RDL does not support Sets of Sets.

Operators

In the following sub-sections, operators are classified according to their return types. For example, an operator calculating a substring position within a String value returns a Number value, so this operator is classified as a Number (not String) operator.

Operators that Return a Numeric Value

- Calculating unary plus/minus of a Number expression
- Adding, subtracting, multiplying and dividing of a Number expression with another Number expression
- Calculating the maximum/minimum of two Number expressions
- Calculating the length of a String
- Calculating the number of elements in a Set
- Calculating the number of instances in a Class
- Calculating the position of a substring within a String value. For example:

  ```
  strIndex of description for "Mr."
  ```

- Calculating trigonometric relationships (sin, cos, tan, acos, asin, atan, and atan2) and converting between degrees and radians
- Calculating mathematical values (absolute value, ceiling, exponentiation, floor, log, modulus, power and square root)
- Decoding DateTime components (year, month, day-of-month, day-of-week, hour, minute, second)
- Decoding Duration components (years, months, days, hours, minutes, seconds)
- Filtering and/or transforming Class instances or Set elements to obtain a Number value. For example:

  ```
  select one p in Person
    such that p.age >=21
    return p.weight
    default -1
  ```

- Calculating the sum, average and standard deviation of Number values associated with Class instances or Set elements. For example:

  ```
  summation of p in Person using p.age
  ```

Operators that Return a Boolean Value

- Calculating NOT of a Boolean expression
- Calculating AND and OR of a variable number of Boolean expressions
- Comparing two atomic expressions of the same data type for equality, inequality or magnitude (less-than, less-than-or-equal-to, greater-than, greater-than-or-equal-to)
- Testing for value inclusion within a range of values. For example:

```
>=1 .. <100 or >dtm:2001-01-01 .. <=dtm:2005-03-17
```

- Comparing two Set expressions of the same data type for equality or inequality
- Comparing two Set expressions of the same data type for subset or superset relationships
- Testing for element inclusion within a Set
- Testing for intersection of two Sets
- Determining whether a field is currently resolved or not
- Determining whether a String value matches a regular-expression pattern. For example:

```
ipAddr matches "[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+"
```

- Filtering and/or transforming Class instances or Set elements to obtain a Boolean value. For example:

```
select one p in Person
  such that p.age >=21
  return p.isAvailable
  default false
```

- Determining the existence of Class instances or Set elements satisfying selection criteria. For example:

```
exists p in Person such that p.age >=21
```

Operators that Return a String Value

- Concatenating the textual values of a variable number of expressions (of any data type)
- Calculating the maximum or minimum of two String expressions
- Calculating the substring of a String value. For example:

```
subString of description from n1 to n2
```

- Eliminating and/or consolidating of white-space within a String value

- Replacing substrings via a regular-expression pattern. For example:

```
strReplace description with "xyz" for "a*b"
```

- Filtering and/or transforming Class instances or Set elements to obtain a String value. For example:

```
select one p in Person
  such that p.age >=21
  return p.name
  default "<none>"
```

Operators that Return an Instance Reference Value

Filtering or transforming Class instances or Set elements to obtain an Instance Reference value. For example:

```
select one p in Person
  such that p.age >=21
  return p.spouse
  default null
```

Operators that Return a DateTime Value

- Adding a Duration expression to a DateTime expression
- Subtracting a Duration expression from a DateTime expression
- Calculating the maximum or minimum of two DateTime expressions
- Encoding a DateTime value. For example:

```
make datetime with dtmYear 1947 dtmMonth 5 dtmDayOfMonth 17 dtmGMT false
```

- Filtering or transforming Class instances or Set elements to obtain a DateTime value. For example:

```
select one p in Person
  such that p.age >=21
  return p.birthDate
  default dtm:1900-01-01
```

- Calculating the average of DateTime values associated with Class instances or Set elements. For example:

```
average of p in Person using p.birthDate
```

In addition to these operators, the RDL also defines intrinsic identifiers for fetching the current time-of-day (*_curr_datetime*) and the Engine-instance's creation datetime (*_base_datetime*).

Operators that Return a Duration Value

- Adding two Duration expressions
- Subtracting two Duration expressions
- Subtracting two DateTime expressions
- Multiplying a Duration expression by a Number expression
- Dividing a Duration expression by a Number expression
- Calculating the maximum or minimum of two Duration expressions
- Calculating the absolute value of a Duration value
- Rephrasing of a Duration value into an equivalent, more-regular form at a specified level. For example:

  ```
  durNormalize dur:2y145m90dt49h123m73s to durYears
  ```

  The result is: dur:14y4m3dt3h4m13s
- Filtering or transforming Class instances or Set elements to obtain a Duration value. For example:

  ```
  select one p in Person
    such that p.age >=21
    return p.vacationTaken
    default dur:0y
  ```
- Calculating the sum, average and standard deviation of Duration values associated with Class instances or Set elements. For example:

  ```
  summation of p in Person using p.vacationTaken
  ```
- Calculating the standard deviation of DateTime values associated with Class instances or Set elements. For example:

  ```
  stddev of p in Person using p.birthDate
  ```

Operators that Return a Set Value

- Filtering or transforming Class instances or Set elements to obtain a new Set value. For example, for a Set of spouses for adult Persons:

  ```
  select all p in Person such that p.age >=21 return p.spouse
  ```
- Sorting Class instances or Set elements according to one or more sort levels. For example, to sort vouchers overall into ascending sequence by expiration date, and then to sort same-date vouchers into descending sequence by value:

  ```
  sort v in Vouchers
    using v.expirationDate ascending
    using v.value descending
  end
  ```

- Selecting Set elements by their position. For example, for selecting the first three elements of a Set (similar to the substring operator for String values):

```
subCollection of personSet from 0 to 3
```

Statements

The following types of statements are provided:

- Statement for local-variable declaration
- Assignment statement
- Statements for generation of error, warning and trace messages
- Instance creation statement
- Instance deletion statement
- Sub-inferencing and stop-inferencing statements
- Statements for Set union, intersection and difference
- Statements for element addition to or removal from a Set
- Statements for seeding a random-number generator and fetching values from it

There is no need for IF-type statements because the RDL locates all decision making in rule premises. In this way, decisions are more centralized and accessible; and rule actions tend to be cleaner and more atomic.

Class-Inheritance Hierarchies

The RDL supports a simplified form of class-inheritance hierarchies.

A hierarchy must be fully contained within a given level of the rulebase hierarchy. For example, a domain-level class must be derived from another domain-level class of the same domain. The class cannot be derived from a rulebase-level class or a class from another domain.

All fields are instance-level (not class-level) fields.

All fields are at a public (not private or protected) access level.

Only leaf (not parent) classes can be instantiated.

Extensibility

A rulebase author can decorate RDL rulebases with custom XML elements and attributes.

The author distinguishes these elements or attributes from RDL elements or attributes by prefixing them with namespace prefixes.

Example

```
<rulebase name="rulebase1" myEditor:dtCreate="2005-01-20">
    ...
    <myEditor:data>
        <desc>Some descriptive text</desc>
        ...
    </myEditor:data>
    ...
</rulebase>
```

The CA Rule Engine Compiler ignores all such elements and attributes. The Compiler also ignores any sub-elements or textual content of those ignored elements.

Rulesets and Rules

A rule author can organize rules into rulesets. The following topics describe their characteristics.

Priorities

Both rulesets and rules may specify priority values.

An object's priority value affects its position within the inferencing agenda. Objects of higher priority precede objects of lower priority. The ordering of objects of equal priority is purposely undefined.

The default priority is 0 (zero).

Rule priorities only have meaning within the containing ruleset; ruleset priorities only have meaning within the containing domain. So a rule with priority 1000 in a ruleset with priority 10 will precede lower-priority rules within the same ruleset - but will follow all rules within a ruleset with priority 11.

Generally speaking, the use of priorities is discouraged because it can defeat the whole point of inferencing. There are a few situations that require the use of priorities. One might legitimately employ priorities when defining *validation rules* (rules for validating input-field values) or *default rules* (rules that fire when higher-priority rules fail to resolve field values). However, even in those cases, there may be cleaner solutions that can be found without resorting to priorities.

Effectiveness Criteria

Both rulesets and rules may specify effectiveness criteria.

An object's effectiveness criteria are defined in terms of datetime relationships and determine whether the Engine will include the object on the inferencing agenda.

By default, the Engine always includes rulesets and rules on the agenda.

Effectiveness criteria can range from simple to complex.

Example

Simple criteria:

```
Between 2000-08-31 15:34 and 2003-03-30 11:54:12
```

Complex criteria:

```
On 3rd Saturday/Sunday
  between 09:00-15:00
  on 1st 15 days of April and September
  of any year
```

Multiple criteria can be specified for a given object. For example, the criteria might specify different time-ranges for different days of the week for different months of different years.

When testing criteria, the Engine applies its own instance-creation timestamp (*_base_datetime*) - so all criteria are consistently tested against the same DateTime value.

Effectiveness criteria may be specified in either GMT or client-local time. In the former case, the Engine will enforce the criteria uniformly for all clients (regardless of geographic location). In the latter case, the Engine will enforce the criteria differently for a San Francisco client than for a Paris client.

Decision-Tree Rules

The RDL applies a "minimalist" approach when it comes to rule types. The RDL supports only one rule type: a decision-tree rule.

The idea here is that this rule type is sufficiently powerful and flexible so that end-user rulebase editors should be able to map all of their higher-level rule types to this one RDL rule type.

Overall Structure

A decision-tree rule consists of one or more named decisions:

```
rule "Rule1"
decision "main"
   ...
decision "next"
   ...
decision "next2"
   ...
...
```

The decisions can have any name (not just the names shown) - but the names must be unique as decision names within the enclosing rule.

Each decision, in turn, specifies a base expression followed by zero or more test cases:

```
decision "main"
    eval <baseExpr>
    then
        case <selExpr1>:
            <action1>
        case <selExpr2>:
            <action2>
        otherwise:
            <action3>
        unknown:
            <action4>
    end
```

The above decision is a non-iterative decision. The RDL also supports iterative decisions - decisions specifying an iteration variable and iteration expression:

```
decision "main"
    for <iterVar> in <iterExpr>
    eval <baseExpr>
    then
        ...
    end
```

Some test cases specify selection expressions and others (OTHERWISE, UNKNOWN) don't specify such expressions. All forms of test cases are optional.

Although a decision may specify multiple test cases with selection expressions, a decision may only specify at most one OTHERWISE and/or UNKNOWN test case.

Test cases involving selection expressions can share a test-case action:

```
decision "main"
    eval <baseExpr>
    then
        case <selExpr1>:
            <action1>
        case <selExpr2>:
        case <selExpr3>:
            <action2>
        otherwise:
            <action3>
        unknown:
            <action4>
    end
```

The base expression and any selection expressions can be of any data type or complexity - but all must be of the same data type.

We collectively refer to the base, selection, OTHERWISE and UNKNOWN expressions as the decision's premise expressions. A rule involving multiple decisions will have multiple sets of premise expressions (one set per decision).

Overall Semantics

Other sections will go into more detail - but here are some gross details concerning decision semantics:

The Engine evaluates premises and performs actions in the context of a rule thread. Every rule is associated with at least one thread.

- For a non-iterative decision, the decision is associated with exactly one thread - and the Engine processes the decision in the context of that thread.
- For an iterative decision, the decision may be associated with multiple threads. The Engine evaluates the decision's iteration expression in the context of the decision's initial thread - but then spawns additional threads according to the elements generated by the iteration expression. All subsequent decision processing takes place in the contexts of the additional threads.

The Engine will evaluate the base expression (within an appropriate thread) and compares its value against those of the various selection expressions (if any) in the order specified.

- If a match is found, the Engine will perform that test case's action (using the same thread).
- If a match is not found, the Engine exercises the action for the OTHERWISE test case (if defined) using the same thread.
- If the Engine is unable to evaluate the base expression due to its reference to an unresolved field, the Engine exercises the action for the UNKNOWN test case (if defined) using the same thread.

When processing a decision, the Engine performs at most one action per thread. In some cases, the Engine may not perform any action for a thread.

Test Case Actions

There are two varieties of test-case actions:

- A *statement action* specifies a block of zero or more RDL statements. The Engine performs this action by executing all of the RDL statements (if any) in the order specified.
- A *decision action* references another of the rule's decisions. The Engine performs this action by pursuing the referenced decision as a sub-decision of the current one.

Example Rules

This section illustrates both single-decision and multiple-decision rules, as well as rules with and without iterative decisions.

Simple Single-Decision Rule

The following example is a single-decision rule similar to IF/THEN rules found in traditional rule languages:

```
rule "Example1"
decision "number"
    eval (num1 + num2) < 0
    then
        case = true:
            do X=1 Y=10 end
    end
```

The rule's decision is non-iterative, so the Engine associates only one thread with the rule and processes the decision in the context of that thread.

Examples

```
For: num=5, num2=-7
Results: X=1, Y=10

For: num=5, num2=-4
Results: X unresolved, Y unresolved
```

Multi-Decision Rule - All Data Types

The following example is a multi-decision rule illustrating base or selection expressions of all data types.

All of the rule's decisions are non-iterative - so the Engine associates only one thread with the rule and processes all decisions in the context of that thread.

Example

Each selection expression involves a literal constant - so each decision is similar to a SWITCH statement within a conventional programming language:

```
rule "Example2"
decision "number"
    eval (num1 + num2)
    then
        case < 0:
            do X=1 Y=10 end
        case > 10:
            decision boolean
        case = 5:
            decision string
        default:
            decision boolean
        unknown:
            decision instref
    end
decision "boolean"
    eval ((num1 > num2) and bool1)
    then
        case = true:
            decision string
        case = false:
            do X=2 Y=12 end
        unknown:
            decision instref
    end
```

```
decision "string"
    eval (str1 & str2)
    then
        case = "abcdef":
            decision datetime
        default:
            decision instref
    end
decision "instref"
    eval spouse
    then
        case = person1:
            decision datetime
        case = null:
            decision duration
    end
decision "datetime"
    eval (dtm1 + dur:5d)
    then
        case >= dtm:2001-06-01:
            decision set
        default:
            decision duration
    end
decision "duration"
    eval (dur1 - dur2)
    then
        case < dur:10d:
            do X=3 end
        default:
            decision set
    end
decision "set"
    eval scores
    then
        case includes 100:
            do X=4 end
        case intersects set(10, 20, 30, 40, 50):
            do X=6 Y=16 end
        case = set():
            do X=5 end
        case >= set(70, 80, 90):
            do Y=14 end
    end
```

Examples

```
For: num1=5, num2=-7
Results: X=1, Y=10

For: num1=5, num2=0, str1="x", str2="y", spouse=null,
    dur1=dur:100d, dur2=dur:91d
Results: X=3, Y unresolved

For: num1=5, num2 unresolved, spouse=person1, dtm1=dtm:2001-05-28,
    scores=set(60, 70, 80, 90)
Results: X unresolved, Y=14

For: num1=5, num2 unresolved, spouse=person2
Results: X unresolved, Y unresolved
```

Note: The Engine might not find any action to perform.

Single-Decision Rule - Complex Selection Expressions

This example is a single-decision rule where the base expression is a literal constant and the selection expressions are non-constant expressions:

```
rule "Example3"
decision "main"
    eval true
    then
        case = (A < B and B < C):
            do X=1 end
        case = (B < A):
            do X=2 end
        case = (B > 10 or B > C):
            do X=3 end
    end
```

As with all earlier examples, the Engine associates only one thread with the rule.

Examples

```
For: A=1, B=2, C=3
Results: X=1

For: A=1, B=0
Results: X=2

For: A=5, B=5, C=1
Results: X=3

For: A=5, B=5, C=10
Results: X unresolved
```

Single-Decision Rule - with Iteration

The following example is a single-decision rule with an iterative decision:

```
// Filter out vouchers that have expired.
// Retained vouchers are added to filteredVouchers.
rule "Example5"
decision main
    for v in ShoppingCart.theCart.theCustomer.vouchers
    eval v.expirationDate >= _base_datetime
    then
        case true:
            do
                combine v into filteredVouchers
            end
    end
```

When processing this rule, the Engine spawns an additional thread for each instance of a customer voucher.

Each thread checks the voucher's expiration date and, if the voucher has not yet expired, the thread adds that voucher to a Set (filteredVouchers).

The result is a Set (possibly empty) specifying all unexpired vouchers.

Multi-Decision Rule - with Iteration

Example

The following example is a multi-decision rule illustrating a mix of iterative and non-iterative decisions:

```
// Apply filtered vouchers to the item groups.
// Any vouchers applied are marked as "isUsed".
rule "Example6"
decision main
    for v in filteredVouchers
    eval true
    then
        case true: decision findGroup
    end
decision findGroup
    for g in ItemGroup
    eval g.category = v.itemCategory
    then
        case true: decision checkAmount
    end
decision checkAmount
    eval v.value <= g.totalPurchases
    then
        case true:
            do
                g.totalPurchases = g.totalPurchases - v.value
                v.isUsed = true
            end
    end
```

When processing this rule, the Engine first spawns an additional thread for each instance of a filtered voucher. For each of these, the second decision finds an item group associated with a filtered voucher - by spawning additional threads for each instance of class ItemGroup. For each pairing of a filtered voucher with an item group, the third decision conditionally reduces the group's total purchases and marks the voucher as "used".

As a result, the group total-purchases will reflect applied vouchers; and the vouchers will indicate whether they have been "used".

Note: Iteration expressions may be defined either by Sets (filteredVouchers) or classes (ItemGroup).

Additional Notes

- Except for exactly one decision (the root decision), each decision must be referenced as a sub-decision by some other decision within the rule.
- A given decision can reference multiple different decisions as sub-decisions (in different test-case actions). The decision may also reference the same decision multiple times as a sub-decision (in different test-case actions).
- A given decision may be referenced as a sub-decision by multiple other decisions.
- The rule can specify the decisions in any order without regard to how decisions reference one another as sub-decisions. The CA Rule Engine Compiler determines the root decision.
- The Compiler detects and rejects cyclical references amongst decisions.
- A sub-decision can inherit iteration variables from other decisions.

Binary Rulebases

A binary rulebase is a more-efficient runtime representation of an RDL rulebase.

The CA Rule Engine Engine accepts only binary (not RDL) rulebases - so one must first employ CA Rule Engine administrative services to compile an RDL rulebase to a binary rulebase; and then employ CA Rule Engine runtime services to inferencing over the binary rulebase.

Portability

The binary rulebase format is portable across hardware, software and language environments.

A binary rulebase does not include any platform-specific objects (for example, Java-specific or OS-specific objects). All integral values employ a standardized big-endian format. All textual values employ a standardized Unicode UTF-8 encoding.

Security

Due to its unpublished format, a binary rulebase is reasonably secure against reverse engineering. Therefore, the intellectual property contained therein is reasonably protected. Such protection may be particularly important to third-party rulebase vendors.

Durability

A binary rulebase is durable across future CA Rule Engine releases because CA Rule Engine compilation services make it easy to upgrade older-format rulebases to the latest format level.

If the CA Rule Engine encounters an older-format rulebase, it automatically upgrades that rulebase (with a warning message). Note that the Engine does not retain the results. Therefore, in order to avoid repeated automatic-upgrades, one should re-compile the rulebase manually.

Chapter 4: Inferencing Overview

This chapter presents an overview of CA Rule Engine inferencing. The presentation is at a conceptual level. For a fuller description of inferencing, refer to the Engine Behavior Specification. For clarity, this document often employs an *infix-style* notation for illustrating rule logic. For a human reader, this notation is easier to follow than is the corresponding *raw* RDL (XML) code. Note that this infix-style notation is for illustrative purposes only - and that the CA Rule Engine Compiler supports only RDL as a textual language for defining rulebases.

The end-user must first apply CA Rule Engine administrative services to compile RDL-coded rulebases into binary rulebases; and then applies CA Rule Engine runtime services to invoke the inference engine (Engine) for rule processing.

This section contains the following topics:

Fundamental Notions

Rule

A **rule** is an atom of knowledge expressing dependencies amongst field values. A rule consists of one or more **premises** (Boolean expressions typically testing field values) and, associated with each premise, an **action** (statements typically changing field values).

Rulebase

A **rulebase** is a collection of objects defining not only the rules but also organizational objects (e.g., rulesets, domains) and the objects referenced by the rules (e.g., classes, instances and fields). A rulebase may consist of hundreds, even thousands, of rules.

Rule author

A **rule author** or **rulebase author** is one who maintains rulebases - usually via a tool such as a rulebase editor. The rule author is a knowledge expert who encodes some or all of her/his knowledge in the form of rules. The author's expertise may involve any knowledge area - e.g., loan approval, medical analysis, chemical analysis, etc.

At inferencing time, a field may be in either a **resolved** (has a value) or **unresolved** (does not have a value) state.

Rules typically have dependencies on one another - i.e., one rule's premises may test field values resolved by another rule's action. Likewise, multiple rules may be alternatives to one another - i.e., perhaps resolving the same fields to different values depending on premise conditions.

The rule author may specify rules in any order - so there needs to be some sort of generic mechanism for determining how best to apply rules for resolving field values. We call this mechanism an **inference engine** (or **Engine**).

During inferencing, the Engine maintains a hierarchical **agenda** of active rulesets and rules.

When the Engine initializes a rule on the agenda, the Engine creates a **rule thread**. The thread is a runtime instantiation of the rule - and, due to decision iteration, the thread may, in turn, spawn additional child threads; and a child thread may do the same. As such, a rule extends the agenda hierarchy with its own hierarchy of threads.

When visiting a rule thread, the Engine evaluates the thread's premises and, if a premise is TRUE, the Engine performs the corresponding action for the thread. The Engine performs at most one action on behalf of a thread.

If the Engine is unable to evaluate a premise or to complete an action due to an unresolved-field value, the Engine will **pend** the thread - i.e., defer further processing of the thread until relevant fields are resolved (by the actions of other threads).

Agenda Management

When the Engine loads a domain, the Engine establishes a hierarchical inferencing agenda for that domain. The Engine populates the agenda with rulesets for that domain (assuming any effectiveness criteria are satisfied). The rulesets, in turn, then populate the agenda with rules for that ruleset (assuming any effectiveness criteria are satisfied).

The Engine initializes an agenda rule with an initial thread specific to that rule. The initial thread may, in turn, subsequently spawn additional child threads - and each of those threads may spawn yet other threads - so the rule may eventually extend the agenda hierarchy with a thread hierarchy.

During the course of inferencing, the Engine may retire rules from the agenda. If all of a ruleset's rules have retired, the Engine also retires the ruleset from the agenda.

At the end of inferencing, the agenda may or may not be empty.

The Engine retains the agenda so long as the domain remains loaded.

Rule Reactivity

The Engine currently supports *discretely-reactive* rules.

For this type of rule, a rule's threads react to field-value modifications only while threads are pended and, even then, only in a restricted manner.

In the future, the Engine may also support *continually-reactive* rules - rules that are continuously and more-fully reactive to field-value modifications.

Discretely Reactive Rules

For this type of rule, a rule's threads react to field-value modifications only while threads are pended and, even then, only in a restricted manner.

Discretely reactive rules are inherently more efficient, more stable and safer than continually reactive rules. For example, it should be impossible for a rule author to write discretely-reactive rules that loop endlessly. Such looping would be possible for continually reactive rules.

Thread States

The Engine processes rule threads independently of one another.

All threads are initialized in a READY state.

As the Engine visits rule threads, thread states change:

- If the Engine can find a non-empty statement-action to perform and can complete execution of all the action statements, the Engine sets the thread's state to FIRED and retires the thread from the agenda.

- If the Engine cannot find a statement-action to perform - or only finds an empty statement-action - the Engine sets the thread's state to FAILED and retires the thread from the agenda.
- If the Engine is unable to evaluate premise expressions or to complete statement-actions due to references to unresolved fields, the Engine sets the thread's state to PENDED but retains the thread on the agenda. Other threads (of the same or different rules) may subsequently modify field values - thereby unpending the pended thread and return it to a READY state.

The Engine visits only READY rule threads. When the Engine visits a thread, the thread's state always changes to a non-READY state.

A thread may remain indefinitely in a PENDED state because relevant fields were never resolved.

Examples with Non-Iterative Decisions

Example

An example rule with a non-iterative decision:

```
rule "rule1"
decision "main"
    eval (A + B)
    then
        case = C:
            do X=3 Y=Z end
        case >= 10:
            do end
    end
```

Assume that the rule thread is initially in a READY state. Since the rule consists of a single, non-iterative decision, the Engine will employ this thread as the context for all of this rule's processing.

Examples

```
For: A=1, B=2, C=3, Z=0
Results: FIRED

For: A=1, B=9, C=4
Results: FAILED (empty statement action)

For: A=1, B=2, C=4
Results: FAILED (no statement-action found)

For: A=1, B unresolved
Results: PENDED

For: A=1, B=9, C unresolved
Results: PENDED

For: A=1, B=2, C=3, Z unresolved
Results: PENDED
```

Note: how the Engine insists on processing test cases in the order specified. For example, for A=1 and B=9, the Engine will never advance past the first test case while C is unresolved - even though the second test case matches the base-expression value.

Here is another example rule with an UNKNOWN test case:

```
rule "rule2"
decision "main"
    eval (A + B)
    then
        case = C:
            do X=3 Y=Z end
        case >= 10:
            do end
        unknown:
            do X=4 end
    end
```

Examples

```
For: A=1, B=2, C=3, Z=0
Results: FIRED

For: A=1, B=9, C=4
Results: FAILED (empty statement action)

For: A=1, B=2, C=4
Results: FAILED (no statement-action found)

For: A=1, B unresolved
Results: FIRED

For: A=1, B=9, C unresolved
Results: PENDED

For: A=1, B=2, C=3, Z unresolved
Results: PENDED
```

Note: The UNKNOWN test case only applies when the Engine is unable to evaluate the base-expression. It does not apply when the Engine is unable to evaluate either selection or action expressions.

Example with Iterative Decisions

Example

An example rule with an iterative decision:

```
rule "ruleA"
decision "main"
    for p in Person1.spouse.relatives
    eval p.age
    then
        case = Person1.age:
            do X=3 Y=p.weight end
        case = p.spouse.age:
            do end
    end
```

Since the rule consists of an iterative decision, the Engine will begin rule processing in an initial thread - but then spawn additional threads for the remainder of the processing.

Example

If Person1's spouse is unknown or the spouse's relatives are unknown, the Engine pends the rule's initial thread on the iteration expression.

Assuming that relatives are known, the Engine spawns a new thread for each of the relatives.

The new threads will fire, fail or pend independently of one another.

Assume that Person1's age is 50 and that there are six relatives (PersonA, PersonB, PersonC, PersonD, PersonE and PersonF) and that:

- PersonA's age is unknown
- PersonB's age is 50 but weight is unknown
- PersonC's age is 50 and weight is 100
- PersonD's age is 35 but spouse is unknown
- PersonE's age is 37 and her/his spouse's age is 37
- PersonF's age is 42 and her/his spouse's age is 39

In this case, the Engine will spawn six new threads (one per relative) and:

- PersonA's thread will pend on its base expression
- PersonB's thread will pend on a test-case action
- PersonC's thread will fire and retire from inferencing
- PersonD's thread will pend on a test-case expression
- PersonE's thread will fail (due to an empty action) and retire from inferencing
- PersonF's thread will fail (no test-case found) and retire from inferencing

At this point, the rule will be associated with three threads.

Unpending Rule Threads

The Engine unpends threads in one of two situations:

- When relevant unresolved fields are resolved
- When relevant resolved fields change value

The Engine returns an unpended thread to a READY (ready to visit) state. Handling varies according to whether the thread was pended on a premise reference or an action reference.

When Pended on a Premise Reference

The Engine unpends the rule according to whether the rule specifies iterative decisions.

For Rules Without Iterative Decisions

Consider the following rule:

```
rule "rule1"
decision "main"
    eval A
    then
        case = B:
            do X=C Y=D end
        case >= 10:
            decision next
    end
decision "next"
    eval E
    then
        case = F:
            do X=G Y=H end
    end
```

In cases where the rule thread is pended on a premise reference, other premise fields (including those in other decisions) may be relevant for unpending:

If the thread pended on a reference to A, the Engine unpends the thread only on resolution of A. The thread's active decision remains *main*.

If the thread pended on a reference to B, the Engine unpends the thread on either resolution of B or a value change for A. The thread's active decision remains *main*.

If the thread pended on a reference to E, the Engine unpends the thread on either resolution of E or a value change for A or B. If E is resolved, the thread's active decision remains *next*. If A or B change value, the active decision becomes *main*.

If the thread pended on a reference to F, the Engine will unpend the thread on either resolution of F or a value change for A, B or E. If F is resolved or E changes value, the thread's active decision remains *next*. If A or B change value, the active decision becomes *main*.

Note: As a result of unpending a thread, the Engine may also redefine the *active decision* for the thread.

For Rules With Iterative Decisions

Unpending for rules with iterative decisions is very similar to rules without iterative decisions, except that only fields referenced *since the latest iteration* are relevant for unpending.

Consider the following rule:

```
rule "rule1"
decision "main"
    eval A
    then
        case = B:
            do X=C Y=D end
        case >= 10:
            decision next
    end
decision "next"
    for p in Person
    eval p.E
    then
        case = p.F:
            do X=p.G Y=p.H end
    end
```

If the thread is premise-pended on p.E or p.F, the Engine unpends the thread only as those fields are resolved or change value. In that case, the thread's active decision remains *next.*

Note: The thread does not unpend as any of the premise references for earlier decisions change value. Likewise, the Engine ignores any changes in the membership of class Person.

When Pended on an Action Reference

In cases where a thread is pended on an action reference, only fields referenced by the problem statement are relevant for unpending. This handling applies to both iterative and non-iterative decisions.

Exmaple

Returning to the earlier example:

```
rule "rule1"
decision "main"
    eval A
    then
        case = B:
            do X=C Y=D end
        case >= 10:
            decision next
    end
decision "next"
    eval E
    then
        case = F:
            do X=G Y=H end
    end
```

If the thread pended on a reference to C, the Engine unpends the thread only on resolution of C. The thread's active decision remains *main*.

If the thread pended on a reference to D, the Engine unpends the thread only on resolution of D or a value change for C. The thread's active decision remains *main*.

If the thread pended on a reference to G, the Engine unpends the thread only on resolution of G. The thread's active decision remains *next*.

If the thread pended on a reference to H, the Engine will unpend the thread only on resolution of H or a value change for G. The thread's active decision remains *next*.

Revisiting Unpended Rule Threads

During the course of inferencing, the Engine may visit, pend, unpend, or revisit a given thread multiple times:

- When revisiting a thread previously pended on a premise reference, the Engine returns to the thread's active decision and completely re-evaluates its premise expressions (starting with the base expression).
- When revisiting a thread previously pended on an action reference, the Engine returns to the thread's active decision and re-executes the problem action statement (starting at the beginning) - without re-executing any previous action statements.

Rule Retirement

The Engine retires the rule from the inferencing agenda when all of the rule's threads have retired (by either firing or failing). The rule remains on the agenda as long as at least one thread is in a PENDED or READY state.

Instance Deletion

The Engine supports both statically-defined and dynamically-created class instances.

During inferencing, RDL code (or Engine client) may delete dynamically-created instances. In response, the Engine automatically cleans up references to a deleted instance:

- For any atomic field references, Engine will reset the field to NULL. The Engine will similarly handle atomic local-variable references.
- For any collection field references, the Engine will remove the reference as an element from the collection. The Engine will similarly handle collection local-variable references.
- For any rule-thread references, the Engine will, depending on the circumstances, either unpend the thread or terminate threads associated with the deleted instance.

For further information, refer to the Engine Reference document.

Special Handling

The Engine provides special handling for some expressions in order to improve efficiency or simplify usage.

For ANDing and ORing

The Engine processes some expressions in such a way so as to avoid needlessly pending threads.

For example, consider the following ORed expression:

```
isHealthy or isWealthy or isWise
```

If a Person is wise but we don't know if s/he is healthy or wealthy (i.e., those fields are unresolved), the Engine avoids pending and evaluates the overall expression as TRUE.

Similar handling applies to the ANDed expression:

```
isHealthy and isWealthy and isWise
```

If a Person is not wise but we don't know if she is healthy or wealthy, the Engine again avoids pending the thread by evaluating the overall expression as FALSE.

Note: The ANDing and ORing may be implicit. For example, sub-range expressions:

```
A >=B .. <C
```

are implicit AND operations.

Likewise, "stacked" rule-decision test cases. For example:

```
case < B:
case > C:
    do X=1 end
```

are implicit OR operations.

For NULL Values

Fields may be indirectly referenced using paths of instance-references. For example:

```
Duck1.owner.spouse.age < 35
```

In order to evaluate the above expression, the Engine must follow these links to find the appropriate field value (for example, the age of Duck1's owner's spouse).

If while following the links, the Engine detects an intermediate field with a NULL value, the Engine evaluates the immediate comparison as FALSE. So, if Duck1 has no owner or the owner has no spouse, the Engine evaluates the comparison result will be FALSE.

This handling simplifies logic so that it does not have to be awkwardly coded as:

```
Duck1.owner <> null
    and Duck1.owner.spouse <> null
    and Duck1.owner.spouse.age < 35
```

Note: As a consequence, both of the following exhaustive comparisons evaluate as FALSE when Duck1 has no owner or the owner has no spouse:

```
Duck1.owner.spouse.age < 35
Duck1.owner.spouse.age >= 35
```

The Engine applies the same handling for indirect references to Boolean fields, for example:

```
Duck1.owner.spouse.isWise
```

If Duck1 has no owner or the owner has no spouse, the Engine evaluates the field value as FALSE.

This special handling only applies in the above contexts. If NULL values are detected outside these contexts, the Engine generates an *invalid NULL reference* error. Consider the statement:

```
age = Duck1.owner.spouse.age
```

In this case, if Duck1 has no owner or the owner has no spouse, the Engine generates an error.

Forward Chaining

The Engine employs a forward-chaining strategy for resolving field values.

This means that:

- The Engine visits all rules at least once during inferencing - unless a rule thread prematurely terminates inferencing via a stop-inferencing statement.
- The Engine seeks to maximize the number of FIRED or FAILED threads.
- When selecting a thread to visit, the Engine always selects the first READY thread within the inferencing agenda.

Forward-chaining is useful for determining all the consequences of the pre-condition values.

Consider an agenda with the following discretely-reactive rules:

```
rule "Rule1" (READY)
decision "main"
    eval A > 10
    then
        case =TRUE: do D=1 end
    end
rule "Rule2" (READY)
decision "main"
    eval B > 0 or C < 10
    then
        case =TRUE: do X=2 end
    end
rule "Rule3" (READY)
decision "main"
    eval B < 10
    then
        case =TRUE: do Y=D end
    end
rule "Rule4" (READY)
decision "main"
    eval A < 10
    then
        case =TRUE: do C=12 end
    end
rule "Rule5" (READY)
decision "main"
    eval A < 10
    then
        case =TRUE: do B=3 end
    end
```

Note: All decisions are non-iterative - so the Engine processes each rule in the context of a single rule thread.

Assume that field: A is currently resolved to the value: 5 - but that all other fields are unresolved.

The Engine first visits Rule1's thread. The Engine fails the thread and retires it from the agenda.

The Engine next visits Rule2's thread and pends it on premise references to fields B and C.

The resulting agenda is:

```
rule "Rule2" (thread PENDED on B & C)
decision "main"
    eval B > 0 or C < 10
    then
        case =TRUE: do X=2 end
    end
rule "Rule3" (thread READY)
decision "main"
    eval B < 10
    then
        case =TRUE: do Y=D end
    end
rule "Rule4" (thread READY)
decision "main"
    eval A < 10
    then
        case =TRUE: do C=12 end
    end
rule "Rule5" (thread READY)
decision "main"
    eval A < 10
    then
        case =TRUE: do B=3 end
    end
```

The Engine next visits Rule3's thread and pends it on a premise reference to field: B.

The resulting agenda is as follows:

```
rule "Rule2" (thread PENDED on B & C)
decision "main"
    eval B > 0 or C < 10
    then
        case =TRUE: do X=2 end
    end
rule "Rule3" (thread PENDED on B)
decision "main"
    eval B < 10
    then
        case =TRUE: do Y=D end
    end
rule "Rule4" (thread READY)
decision "main"
    eval A < 10
    then
        case =TRUE: do C=12 end
    end
rule "Rule5" (thread READY)
decision "main"
    eval A < 10
    then
        case =TRUE: do B=3 end
    end
```

The Engine next visits Rule4's thread and fires it. The action unpends Rule2's thread. The Engine removes Rule4 from the agenda.

The resulting agenda is as follows:

```
rule "Rule2" (thread READY)
decision "main"
    eval B > 0 or C < 10
    then
        case =TRUE: do X=2 end
    end
rule "Rule3" (thread PENDED on B)
decision "main"
    eval B < 10
    then
        case =TRUE: do Y=D end
    end
rule "Rule5" (thread READY)
decision "main"
    eval A < 10
    then
        case =TRUE: do B=3 end
    end
```

The Engine next visits Rule2's thread but pends it on a premise reference to field: B.

The resulting agenda is as follows:

```
rule "Rule2" (thread PENDED on B)
decision "main"
    eval B > 0 or C < 10
    then
        case =TRUE: do X=2 end
    end
rule "Rule3" (thread PENDED on B)
decision "main"
    eval B < 10
    then
        case =TRUE: do Y=D end
    end
rule "Rule5" (thread READY)
decision "main"
    eval A < 10
    then
        case =TRUE: do B=3 end
    end
```

The Engine next visits Rule5's thread and fires it. The action unpends threads for both Rule2 and Rule3. The Engine removes Rule5 from the agenda.

The resulting agenda is as follows:

```
rule "Rule2" (thread READY)
decision "main"
    eval B > 0 or C < 10
    then
        case =TRUE: do X=2 end
    end
rule "Rule3" (thread READY)
decision "main"
    eval B < 10
    then
        case =TRUE: do Y=D end
    end
```

The Engine next visits the Rule2 thread, fires it and removes it from the agenda.

The resulting agenda is as follows:

```
rule "Rule3" (thread READY)
decision "main"
    eval B < 10
    then
        case =TRUE: do Y=D end
    end
```

The Engine next visits Rule3's thread - but pends it on its action reference to field: D.

Since there are no more rule threads to visit, the Engine terminates inferencing with the final agenda:

```
rule "Rule3" (thread PENDED on D)
decision "main"
    eval B < 10
    then
        case =TRUE: do Y=D end
    end
```

Rule3's thread may remain pended indefinitely - unless D is a pre-condition field and the client application resolves it.

The final field status is:

```
A=5, B=3, C=12, D unresolved, X=2, Y unresolved
```

Sub-Inferencing

During the course of inferencing, a rule-thread action may sub-inference into another rulebase domain - in order to pursue an inferencing sub-problem.

The Engine imposes no limits on sub-inferencing - i.e., a thread in a sub-inference domain may, in turn, sub-inference into yet another domain - so Domain1 may sub-inference to Domain2 and Domain2 may, in turn, sub-inference to Domain3, and so on.

The Engine will disallow a given domain from being loaded more than once during sub-inferencing. For example, Domain1 may sub-inference to Domain2 but Domain2 may not then sub-inference (directly or indirectly) back to Domain1.

When a thread action invokes sub-inferencing, the Engine suspends inferencing within the current domain, de-activates the current domain, activates the new domain and then begins inferencing within that domain. The Engine assigns a separate inferencing agenda to the new domain - so rules from different domains are not mixed together within the same agenda.

As the Engine fires threads within the new domain, the Engine may unpend threads in ascendant domains - but the Engine will not revisit those threads until their owning domain again becomes active.

At the completion of inferencing within the new domain, the Engine de-activates it, unloads it, re-activates the prior domain and then resumes execution of the interrupted rule-thread action.

Chapter 5: Using Callback

The CA Rule Engine inference engine provides a level of interactivity beyond the standard stateful rule session using callback methods.

A callback is a client-defined method invoked by the CA Rule Engine inference engine upon the occurrence of certain events. A triggering event is specific for a particular RDL class and field and the corresponding callback method must be defined or declared in the corresponding Java wrapper class or interface and not just in any derived or implementation classes.

Callback methods are event-specific and permit the client to supplement or to react to engine processing in a just-in-time-fashion.

Currently, two types of callbacks are supported by CA Rule Engine that can be used by client application: initialization callback and change callback. Using of callbacks by client is always optional. The client determines whether to define any callback methods for both types of callbacks.

Callbacks are transparent to the rulebase author. The author can design rulebases without knowing or caring whether clients will specify handlers. Likewise, for rulebases shared by multiple clients, some clients may specify handlers and others may not.

While the client may carry out any action unrelated to rule inference in a callback handler method, only limited actions related to rule inference may be carried out in a particular type of callback method.

This section contains the following topics:

Initialization Callback Methods

The CA Rule Engine inference engine invokes an Initialization Callback Method for resolving unresolved pre-condition field values.

Some situations where the callback method may be particularly useful are:

- When the Engine may potentially need to draw upon a large volume of input data - but for a given inferencing scenario, the Engine needs to access only a small subset of the data (as determined by the rules at runtime)

  For example, in a large corporation, the rules may need to reference the management hierarchy for a given employee. There may be thousands of employees but for any given session, the rules only need to reference the managers associated with the given employee. In this scenario, it would be impractical for a client to initialize pre-condition values for all possible employees - on the chance that the rules might need to reference those employees.

- When the client needs to access external resources at runtime - resources inaccessible to RDL code - e.g., databases, remote servers, code libraries.

Initialization callback is tied to a particular precondition field. To use the initialization callback feature, client needs to define the "onInit" method. The exact name of the method depends on the name of the field. For example, for a field named xYZ, the signature of the "onInit" method should be defined as the following:

```
public void onInitXYZ()
```

The CA Rule Engine inference engine invokes an initialization callback method only as necessary - as it actually needs values in order to make a decision. As such, the initialization callback is supportive of "first-chance" processing.

Any initialization callback handling logic implemented in the "onInit" method must not carry out any actions that modify the status of any of the rules. In a stateful rule session, the client method may optionally get any other objects from the rule session for inspection only. However, the client wrapper class may need to separately retain a reference to the stateful rule session object for that purpose in order to use the "getObject" calls of the stateful rule session.

Note: Adding new objects to the session does not modify the status of any of the rules. It is fine for the initialization callback method to add one or more new objects which serve as a value to an instance reference field that is being initialized. In fact, adding new objects that correspond to dynamic instances in initialization callback is more preferable to adding all needed dynamic instances upfront especially when the number of potentially needed dynamic instances is large.

The "onInit" method normally sets a value to the field for which it is defined. If the handling logic decides not to initialize the field, it can simply leave the field untouched.

The engine will invoke an initialization callback method at most once for each object and will cache the returned value for subsequent use. Client can avoid the invocation of a defined initialization callback method on a particular object by setting a non-null value to the concerned field of that object.

Even though in principle callbacks and rulebase design are not dependent on each other, there is a subtle effect on the use of initialization callback and application created dynamic instances related to rulebase design. This is due to the fact that initialization callback is only available at the field level.

If an application created dynamic instance is referred in rules through an instance reference field, that dynamic instance may be created and added to the session in the initialization callback method of that instance reference field. If such a dynamic instance is referred in rules only through class iteration, the dynamic instance must be added to the session prior to inference for the rule engine to be aware of its existence. Therefore, writing rules that refer to dynamic instances through instance reference variables may allow better flexibility on using initialization callback to create objects for dynamic instances.

For the case of class iteration, one can instead use an instance reference field in a container class that refers to a collection of that particular class. This instance reference field iterates over the collection so as to allow objects for client created dynamic instances to be instantiated in the initialization callback of that instance reference field.

Change Callback Methods

The CA Rule Engine inference engine invokes a Change Callback Method upon modifying a pre or post-condition field value, and when dynamically creating or deleting class instances visible to the client. However, a change callback method will not be invoked if the change is requested by the client, such as invoking the "updateObject" call of a stateful rule session, or as the result of client provided initialization callback.

Some situations where the Change Callbacks may be particularly useful:

- When the Engine may potentially generate a broad range of results - but, for a given inferencing scenario, the Engine may need to generate only a small subset of those results (as determined by the rules at runtime).

  For example, in a large corporation, the rules may need to modify the data for managers in the hierarchy of a given employee. For any given session, the rules only need to update the managers associated with the given employee. In this scenario, it would be impractical for a client to fetch post-condition values for all possible managers on the chance that the rules might have updated for those managers.

- When the client needs to access external resources at runtime, resources that are inaccessible to RDL code - e.g., databases, remote servers, code libraries.

- When it is necessary to take certain external actions as soon as the value of a concerned field is changed rather than wait till the end of inference.

A Change Callback Method is the *inverse* of an Initialization Callback Method and may be used in conjunction with it.

Unlike an initialization callback method which is invoked at most only once, a change callback method may be invoked multiple times if the engine needs to update the same field multiple times. For value change callback, the new value provided may be the Java null value if the field is to be set to unknown by the engine and the client provided callback is expected to handle that special case.

Another difference between an initialization callback method and a change callback method is that new objects cannot be created and added to the session in a change callback method.

Depending on whether the field is a collection, or whether the operation is on an instance itself, the following three types of change callback methods may be defined.

Field Value Change Callback

The CA Rule Engineinference engine invokes the field value change callback method when the value of a field is changed by the engine. One exception is that if the field is a collection field and the change is to add or remove an element of that collection, instead of invoking this callback method, CA Rule Engine invokes the more specific Element Add/Remove change callback for improved processing efficiency.

To react to a change to the value of a field, client needs to define the *onChange* method. For example, for a field named xYZ, the signature of the *onChange* method should be defined as the following:

```
public boolean onChangeXYZ(fieldType fieldValue)
```

where fieldType is the actual data type of the field xYZ and fieldValue is the new value.

The existing field value of the Java wrapper class is not modified when a client supplied change callback method for a field change event is invoked. This may provide the *old* value to the method. The callback method has the option to update the field value of the Java wrapper object to the provided new value if desired. The method should return true if it chooses to update the field to the specified value and false otherwise. The CA Rule Engine inference engine updates the field value of the Java wrapper class immediately after the user supplied callback method returns false, to keep it synchronized.

A client field value change callback method may not carry out any actions that modify state of the current rule session including updating a changed field to a value other than the provided new value. In a stateful rule session, the client handler may optionally get any other objects from the rule session for inspection only. However, the client wrapper class may need to separately retain a reference to the stateful rule session object for that purpose in order to use the *getObject* calls of the stateful rule session.

As an alternative to the above approach, if all the actions to be taken are also appropriate for the Java Bean setter method of the field, the actions can be directly implemented in the setter method without defining the *onChangeXYZ()* method since default behavior without a client supplied change callback for CA Rule Engine is to invoke the setter method to update the value of the field.

Collection-Element Addition/Deletion Callback

When the field is a collection field and the change is to add or remove elements of that collection rather than setting a whole new collection to the field, the following two callback methods will be invoked instead of the *onChange* method:

```
public boolean onAddElementXYZ(fieldType fieldValue, int index)

public boolean onDeleteElementXYZ(fieldType fieldValue, int index)
```

where fieldType is the actual elemental data type of the field xYZ and fieldValue is the changed element. The index value gives the position of the changed element. The index value is zero based.

The reason to have these two separate callbacks for collection element addition and deletion is to allow possibly more efficient handling of the collection than that can be done through direct array modifications. The assumption is that a Java collection object such as an ArrayList is actually used in implementation in the Java wrapper class for a collection field but is exposed as an array to CA Rule Engine. CA Rule Engine requires array interface for better type verification purposes. Due to this reason, it is recommended that client always implements them for a collection field for carrying out the actual change even if no other actions are needed. The WrapperMaker tool automatically generates such callback implementations for collection fields.

The existing field value of the Java wrapper class is not modified when a client supplied change callback method for an element addition or deletion change event is invoked. This may provide the *old* value to the method. It is preferred for efficiency purposes for the element addition or deletion change callback to update the field value by actually adding or deleting the provided element. The method should return true if it chooses to do so and false otherwise. The CA Rule Engine inference engine updates the field value of the Java wrapper class immediately after the user supplied callback method returns false, to keep it synchronized.

A client element addition or deletion change callback method may not carry out any actions that modify the state of the current rule session including updating the changed field to a value other than the intended new value. In a stateful rule session, the client handler may optionally get any other objects from the rule session for inspection only. However, the client wrapper class may need to separately retain a reference to the stateful rule session object for that purpose in order to use the *getObject* calls of the stateful rule session.

Instance Creation/Deletion Callback

The CA Rule Engine inference engine invokes the instance creation or deletion change callback method when rule actions creates or deletes a dynamic instance.

To react to dynamic instance creation and deletion by CA Rule Engine, the client needs to provide the following methods:

```
public static void onCreateInstance(instanceClass instanceObj)

public static void onDeleteInstance(instanceClass instanceObj)
```

where instanceClass is the name of the class enclosing the methods and instanceObj is the changed instance object.

Upon entry to the onCreateInstance method, the Java wrapper object corresponding to the newly created engine instance would have been automatically added to the session. However, user should not expect this instance to have any fields initialized except the instanceName field.

In the onCreateInstance method, it is not forbidden to set initial values to any pre-condition fields of the new instance using the setter methods of that instance provided no new objects need to be added to the session. However, unless such field initialization is only appropriate for engine created instances, client should instead use initialization callback for that purpose. An initialization callback will be invoked for all objects of that class regardless if the object is static, created by user or by engine. This callback method will only be invoked for engine created instances.

The Java wrapper object corresponding to the deleted engine instance will be automatically deleted from the session upon the return of the onDeleteInstance method.

A client instance creation or deletion change callback method may not carry out any actions that modify the state of the current rule session. For example, while the use of setter methods to initialize pre-condition fields is allowed, use of the "updateObject" calls of a stateful rule session to push the changes to the engine is not allowed.

Chapter 6: Tutorial

This chapter explains the procedures of using CA Rule Engine in detail, both from the client side and from the rulebase side. Starting with a sample business scenario, this tutorial explains the authoring of RDL rulebase with the infix rule language and the Infix2RDL tool, the use of the WrapperMaker tool to generate the Java classes that wrap the rulebase classes, the addition of session details to the generated template session class and the invocation of inferencing services.

A properly installed CA Rule Engine distribution is required to follow the steps in this tutorial. See Install and Configure CA Rule Engine for more information. In addition, if drag and drop invocation of tool scripts is not used, it is also necessary to add the bin folder of the CA Rule Engine distribution on the PATH. In order to run the resulting application, it is also necessary to make sure the current folder, or ".", have been put on the CLASSPATH.

All tutorial related files are located in the tutorial folder of the CA Rule Engine distribution kit.

This section contains the following topics:

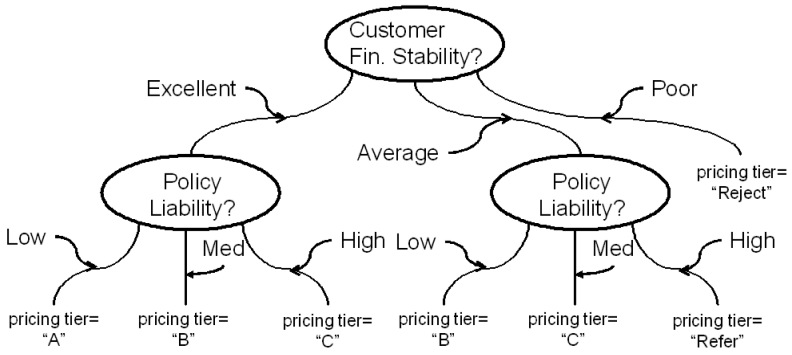
Tutorial Scenario

The business scenario for the system that is developed in this tutorial is to determine the pricing tier of a customer's insurance policy for a fictitious company, Forward Insurance. Both rulebase authoring and development of Java application are covered. In order for this tutorial to be applicable for both use cases mentioned in Using CA Rule Engine, extra requirements that simulate one or the other situations will be added in the course of this tutorial.

At Forward Insurance, the pricing tier is determined on the basis of the customer's financial stability as well as information on the level of liability involved in the policy. Additional rules are used to determine the customer's financial stability. This example represents these rules as decision trees that can be mapped directly to rule structures in RDL.

Pricing Tier Decision Tree

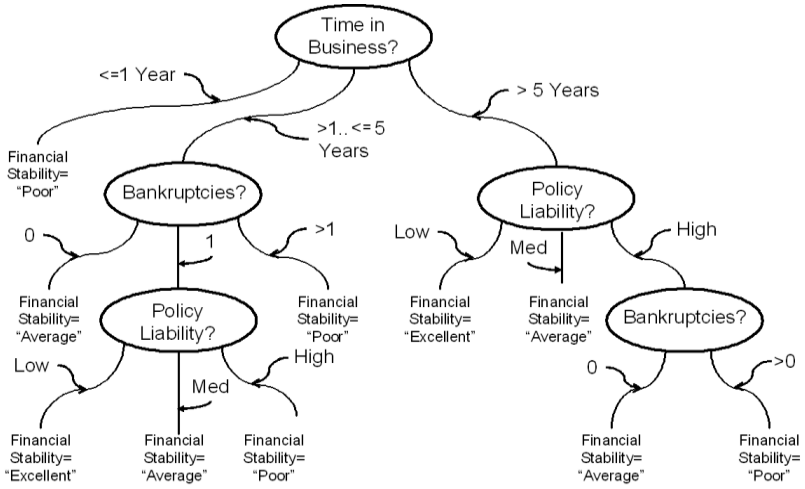
The pricing tier rule looks at customer financial stability in terms of whether it is Excellent, Average, or Poor. Examining of policy liability is relevant only if the financial stability is Excellent or Average. If the financial stability is Poor, the customer's application will be rejected. Policy liability can be High, Med, or Low.



The pricing tier decision tree can also be expressed as a decision table.

Customer Financial Stability Decision Tree

Customer financial stability is based on the customer's time in business, the number of bankruptcies experienced by the customer, and the policy liability. If the customer has been in business for 1 year or less, Forward Insurance considers the financial stability to be Poor. If the customer has been in business for between 1 and 5 years and has one bankruptcy in that period, then the policy liability must be considered to determine financial stability. If the customer has been in business for more than 5 years, Policy liability is the principal determinant of financial stability. Bankruptcies need to be considered only if the policy liability is High. The determination of customer financial stability is shown in the following decision tree:



The customer financial stability rule is a true decision tree and cannot be expressed as a single decision table. Notice that the order in which bankruptcies and policy liability are considered depends on which branch of time in business is pursued, and the difference in branch conditions for the bankruptcies.

Rulebase Interface Requirements

It is important to consider how these rules interface with their environment. The system is expected to accept the following as inputs related to the customer and the coverage.

The following information relates to the customer:

- The time in business is to be specified to the rulebase as a duration value.
- Bankruptcies experienced by the customer. The information will be passed to the rulebase as a set of dates on which the bankruptcies occurred.

The coverage must specify the liability involved in the policy. The policy liability can be High, Med, or Low. Moreover, it should be possible to pass policy liability as unknown to the rulebase, in which case the rulebase assumes that the policy liability is High.

The system returns the pricing tier.

To simulate the use case where rulebase authoring is subject to existing business objects, the policy liability must be passed to the rulebase as an object of type LiabilityType. The pricing tier will also be returned through an object of type InsuranceAgent. The InsuranceAgent object is returned to the client application from the rulebase, providing the pricing tier.

From the above requirements, four classes will be needed to form the interface between the rulebase and the client application: Customer, Coverage, LiabilityType and InsuranceAgent. For further information on the classes used in the system, see Rulebase Class Definitions. The structure of the rulebase interface with client applications is specified in Domain Interface Definition.

The Rulebase

For this tutorial, it is assumed that a suitable XML editor is available for creating an RDL rulebase. In fact, a simple text editor suffices. You create a rulebase named Forward_Insurance_Pricing_Tier. The infix rule language will be used in authoring the rules in this tutorial, so *forward_infix.xml* may be used as the filename of the rulebase. The rulebase code is opened with the following skeleton XML code:

```
<?xml version="1.0" encoding="UTF-8"?>
<rulebase name="Forward_Insurance_Pricing_Tier">
</rulebase>
```

Rulebase Class Definitions

A required step in constructing an RDL rulebase is to define the classes that are used by the rules. In general, it is necessary to settle the scope of the classes to be defined. Classes can exist at three levels: the rulebase level (with full visibility to all elements of the rulebase), the domain level, or the ruleset level. However, any rulebase class that is part of the application interface of the rulebase, i.e. have corresponding Java classes that wrap them, must be defined at the rulebase level. On the other hand, if a class is only internally used during rule inferencing, it may be better to settle the class at domain or ruleset level as appropriate.

This is a design issue that must be considered in relationship to the domains and rulesets that are projected to be part of the rulebase. In the case of Forward Insurance scenario, however, the four interface classes will be all that are needed. That means the scope of the classes in this tutorial will be at the rulebase level.

The Customer class requires two fields to support rulebase inputs: timeInBusiness of type duration, and bankruptcies of type set of datetimes. In addition, it provides the financialStability field of type string, which will be resolved internally by rules. To store values for these fields, an instance must be specified. Since the system only needs to process one customer at a time, a static instance suffices. The instance is named *theCustomer*. The RDL code for this class is given below:

```
<class name="Customer">
    <field name="financialStability">
        <datatype_string coll_type="none"/>
    </field>
    <field name="timeInBusiness">
        <datatype_duration coll_type="none"/>
    </field>
    <field name="bankruptcies">
        <datatype_datetime coll_type="set"/>
    </field>
    <instance name="theCustomer"/>
</class>
```

Since this class is at the rulebase level, the above <class> element should be a direct child of the <rulebase> element. In fact, since all classes in this tutorial are at the rulebase level, each should be a direct child of the <rulebase> element.

The Coverage class requires one field, the policy liability. This field, however, must reference an instance of another class, the LiabilityType class. The field is of type instref and points to an instance of LiabilityType. The possible liabilities of a policy are represented by three instances of LiabilityType: Low, Med, or High. The liability field of Coverage points to one of these instances. Again a static instance will suffice for the Coverage class and it is named *thePolicy.*

```
<class name="Coverage">
    <field name="liability">
        <datatype_instref coll_type="none">
            <identifier name="LiabilityType"/>
        </datatype_instref>
    </field>
    <instance name="thePolicy"/>
</class>
```

The LiabilityType class represents an enumeration. In RDL an enumeration can simply be represented as a list of static instances of the given class.

```
<class name="LiabilityType">
    <instance name="Low"/>
    <instance name="Med"/>
    <instance name="High"/>
</class>
```

Finally, define the InsuranceAgent class and its instance, *agent*. Define this class with a single field, pricingTier, of type string.

```
<class name="InsuranceAgent">
    <field name="pricingTier">
        <datatype_string coll_type="none"/>
    </field>
    <instance name="agent"/>
</class>
```

This completes the necessary class definitions to support the System Specification.

Domain Interface Definition

The domain, named *PricingTierDomain*, must be specified as appshared in order to interface with the Java client application. The <domain> element must also be a direct child of the <rulebase> element. The following skeleton RDL code declares this <domain> element:

```
<domain appshared="true" name="PricingTierDomain">
</domain>
```

Because this domain is appshared, you must specify the interface fields- both preconditions and postcondition. The specifications consist of the full identifier_path specification to the instance field that is part of the interface. The levels of identifiers involved in these path specifications are class, instance, and field. The customer's timeInBusiness and bankruptcy information, as well as the policy's liability type are the precondition fields. Only the pricingTier field of the agent needs to be specified as a postcondition.

```
<interface_fields>
    <precondition_list>
        <identifier_path>
            <identifier name="Customer"/>
            <identifier name="theCustomer"/>
            <identifier name="timeInBusiness"/>
        </identifier_path>
        <identifier_path>
            <identifier name="Coverage"/>
            <identifier name="thePolicy"/>
            <identifier name="liability"/>
        </identifier_path>
        <identifier_path>
            <identifier name="Customer"/>
            <identifier name="theCustomer"/>
            <identifier name="bankruptcies"/>
        </identifier_path>
    </precondition_list>
    <postcondition_list>
        <identifier_path>
            <identifier name="InsuranceAgent"/>
            <identifier name="agent"/>
            <identifier name="pricingTier"/>
        </identifier_path>
    </postcondition_list>
</interface_fields>
```

There are no application creatable classes in this example. In general, if a variable number of objects of a class are needed to be passed to CA Rule Engine by the client, the class needs to be declared as application creatable. For example, suppose a customer can buy multiple policies from Forward Insurance and the pricing tier for each policy will be related to other ones, which means the policies must all be considered at once. The Coverage class will then need to be declared as application creatable. In this tutorial, the rulebase specifies an empty <appcreatable_classes> element:

```
<appcreatable_classes/>
```

Both <interface_fields> and <appcreatable_classes> elements are optional, and if used must be direct children of the <domain> element.

Rulesets and Rules

Rulesets provide an arbitrary means of grouping rules. In this case all the rules could be placed into a single ruleset; however, two rulesets are used for illustration purposes:

- The pricing tier ruleset, which contains the single pricing tier decision tree rule.
- The customer financial stability ruleset, which comprises the customer financial stability decision tree rule and the default policy liability decision tree rule.

Both the pricing tier decision tree and the customer financial stability decision tree use multiple levels of nodes. These are expressed as multiple decisions within each decision tree.

The <ruleset> elements should be the direct children of the <domain> element to which domain the rulesets belong. Similarly the <rule> elements should be direct children of the <ruleset> element to which ruleset the rules belong.

Pricing Tier Ruleset and Rule

The following skeleton RDL code declares the pricing tier ruleset:

```
<ruleset name="PricingTierRuleSet">
    <effective_ranges/>
</ruleset>
```

If needed, an optional priority attribute can also be added to the ruleset declaration, e.g. <ruleset name="PricingTierRuleSet" priority="10">. The ruleset is defined with an empty <effective_ranges> element. This element is actually optional since effective ranges are not used in the Forward Insurance case. It is included just to illustrate where the element would be if it is used.

In the pricing tier ruleset, there is only the pricing tier rule. The following skeleton RDL code declares the pricing tier rule. This <rule> element should be a direct child of the pricing tier <ruleset> element. As in the <ruleset> element, an optional priority attribute can be added and an empty <effective_ranges> element is included just to illustrate where the element would be if it is used.

```
<rule name="PricingTierRule">
    <effective_ranges/>
</rule>
```

Up to now, standard RDL elements have been used in the construction of the Forward Insurance pricing tier rulebase. While RDL elements can also be used directly in authoring rules, the infix rule language, which is documented separately, is much more concise and readable in authoring rules. This tutorial will illustrate the use of the infix language and the conversion from an infix rulebase to pure RDL rulebase using the Infix2RDL tool.

To use the infix language in authoring the body of a rule, a CDATA section is needed to enclose the rule body. This CDATA section should be direct child of the <rule> element. The code below shows the infix version of the pricing tier rule:

```
<![CDATA[
decision PricingTier
eval theCustomer.financialStability
then
        case "Excellent": decision ExcellentCustomer
        case "Average": decision AverageCustomer
        case "Poor": do agent.pricingTier = "Reject" end
end

decision AverageCustomer
eval thePolicy.liability
then
        case Low: do agent.pricingTier = "B" end
        case Med: do agent.pricingTier = "C" end
        case High: do agent.pricingTier = "Refer" end
end

decision ExcellentCustomer
eval thePolicy.liability
then
        case Low: do agent.pricingTier = "A" end
        case Med: do agent.pricingTier = "B" end
        case High: do agent.pricingTier = "C" end
end
]]>
```

Referring back to The Pricing Tier Decision Tree, it can be seen that each decision section in the above code corresponds to a decision node in the decision tree. The item in the *eval* portion of the decision corresponds to that to be evaluated in the decision node. The different *case* clauses of the decision correspond to the branches off the decision tree node.

From another point of view, a decision block is also similar to the switch…case statement in programming languages such as Java. However, the case conditions allowed in the infix language can be much more complicated.

Just to illustrate how much more concise and readable the infix rule language is compared with the raw RDL code in authoring rules, the following is the RDL equivalent of the pricing tier infix rule:

```
<dec_tree_body>
    <decision name="PricingTier">
        <identifier_path>
            <identifier name="theCustomer"/>
            <identifier name="financialStability"/>
        </identifier_path>
        <dec_test_group>
            <part_eq_op>
                <string_constant value="Excellent"/>
            </part_eq_op>
            <dec_ref name="ExcellentCustomer"/>
        </dec_test_group>
        <dec_test_group>
            <part_eq_op>
                <string_constant value="Average"/>
            </part_eq_op>
            <dec_ref name="AverageCustomer"/>
        </dec_test_group>
        <dec_test_group>
            <part_eq_op>
                <string_constant value="Poor"/>
            </part_eq_op>
            <dec_statements>
                <assign_stmt>
                    <identifier_path>
                        <identifier name="agent"/>
                        <identifier name="pricingTier"/>
                    </identifier_path>
                    <string_constant value="Reject"/>
                </assign_stmt>
            </dec_statements>
        </dec_test_group>
    </decision>
```

```
<decision name="AverageCustomer">
    <identifier_path>
        <identifier name="thePolicy"/>
        <identifier name="liability"/>
    </identifier_path>
    <dec_test_group>
        <part_eq_op>
            <identifier name="Low"/>
        </part_eq_op>
        <dec_statements>
            <assign_stmt>
                <identifier_path>
                    <identifier name="agent"/>
                    <identifier name="pricingTier"/>
                </identifier_path>
                <string_constant value="B"/>
            </assign_stmt>
        </dec_statements>
    </dec_test_group>
    <dec_test_group>
        <part_eq_op>
            <identifier name="Med"/>
        </part_eq_op>
        <dec_statements>
            <assign_stmt>
                <identifier_path>
                    <identifier name="agent"/>
                    <identifier name="pricingTier"/>
                </identifier_path>
                <string_constant value="C"/>
            </assign_stmt>
        </dec_statements>
    </dec_test_group>
```

```
    <dec_test_group>
        <part_eq_op>
            <identifier name="High"/>
        </part_eq_op>
        <dec_statements>
            <assign_stmt>
                <identifier_path>
                    <identifier name="agent"/>
                    <identifier name="pricingTier"/>
                </identifier_path>
                <string_constant value="Refer"/>
            </assign_stmt>
        </dec_statements>
    </dec_test_group>
</decision>
<decision name="ExcellentCustomer">
    <identifier_path>
        <identifier name="thePolicy"/>
        <identifier name="liability"/>
    </identifier_path>
    <dec_test_group>
        <part_eq_op>
            <identifier name="Low"/>
        </part_eq_op>
        <dec_statements>
            <assign_stmt>
                <identifier_path>
                    <identifier name="agent"/>
                    <identifier name="pricingTier"/>
                </identifier_path>
                <string_constant value="A"/>
            </assign_stmt>
        </dec_statements>
    </dec_test_group>
```

```
            <dec_test_group>
                <part_eq_op>
                    <identifier name="Med"/>
                </part_eq_op>
                <dec_statements>
                    <assign_stmt>
                        <identifier_path>
                            <identifier name="agent"/>
                            <identifier name="pricingTier"/>
                        </identifier_path>
                        <string_constant value="B"/>
                    </assign_stmt>
                </dec_statements>
            </dec_test_group>
            <dec_test_group>
                <part_eq_op>
                    <identifier name="High"/>
                </part_eq_op>
                <dec_statements>
                    <assign_stmt>
                        <identifier_path>
                            <identifier name="agent"/>
                            <identifier name="pricingTier"/>
                        </identifier_path>
                        <string_constant value="C"/>
                    </assign_stmt>
                </dec_statements>
            </dec_test_group>
        </decision>
    </dec_tree_body>
```

Customer Financial Stability Ruleset and Rules

The customer financial stability ruleset consists of two decision trees: the customer financial stability rule, and the default policy liability rule assigning a liability of *High* to the policy if the liability is unknown.

As in the pricing tier ruleset, the following skeleton RDL code declares the customer financial stability ruleset. But this time the optional <effective_ranges> element is dropped to show that this element is indeed optional.

```
<ruleset name="CustomerFinancialStabilityRuleset">
</ruleset>
```

Customer Financial Stability Rule

The following skeleton RDL code declares the customer financial stability rule. This <rule> element should be a direct child of the customer financial stability <ruleset> element. As in the <ruleset> element, the optional <effective_ranges> element is dropped just to illustrate that the element is indeed optional.

```
<rule name="CustomerFinancialStabilityRule">
</rule>
```

The code below shows the infix version of the customer financial stability rule. It should be a direct child of the above <rule> element:

```
<![CDATA[
decision customerFinancialStability
eval theCustomer.timeInBusiness
then
        case <= dur:1Y: do theCustomer.financialStability = "Poor"
end
        case > dur:1Y .. <= dur:5Y: decision
oneToFiveYearsInBusiness
        otherwise: decision overFiveYearsInBusiness
end

decision oneToFiveYearsInBusiness
eval sizeof theCustomer.bankruptcies
then
        case 0: do theCustomer.financialStability = "Average" end
        case 1: decision underFiveYearsInBusinessOneBankruptcy
        otherwise: do theCustomer.financialStability = "Poor" end
end
```

```
decision overFiveYearsInBusiness
eval thePolicy.liability
then
        case Low: do theCustomer.financialStability = "Excellent"
end
        case Med: do theCustomer.financialStability = "Average" end
        otherwise: decision overfiveYearsInBusinessHighLiability
end

decision underFiveYearsInBusinessOneBankruptcy
eval thePolicy.liability
then
        case Low: do theCustomer.financialStability = "Excellent"
end
        case Med: do theCustomer.financialStability = "Average" end
        otherwise: do theCustomer.financialStability = "Poor" end
end

decision overfiveYearsInBusinessHighLiability
eval sizeof theCustomer.bankruptcies
then
        case 0: do theCustomer.financialStability = "Average" end
        otherwise: do theCustomer.financialStability = "Poor" end
end
]]>
```

Referring back to the decision tree figure in The Customer Financial Stability Decision Tree, it can be seen that as in the case of the pricing tier rule, the same correspondence exists between the decision blocks in above infix rule body and decision nodes of the decision tree, between the item in the *eval* part of the decision and that inside the decision tree node, and between the *case* clauses of each decision and the branches off the corresponding decision node.

One thing to note is that instead of listing all the conditions of each *eval* test as the in the case of the pricing tier rule, the customer financial stability rule uses the *otherwise* clause for the last branch of the decisions. In comparison to the switch…case statement in programming languages such as Java, the *otherwise* clauses in rule language corresponds to the *default* clause in Java statement.

Default Liability Rule

The following rule insures that liability points to the High LiabilityType if it is passed to the rulebase as unknown. This time the CDATA block for the infix rule is already put inside the rule body:

```
            <rule name="DefaultLiabilityRule">
<![CDATA[
decision main
eval liability
then
        unknown: do liability = High end
end
]]>
            </rule>
```

The *unknown* clause in the above decision block is a distinct feature of rule language. It specifies the action to take if the item in the *eval* part of the decision is unknown. The unknown clause has no correspondence in the switch…case statement in programming languages such as Java.

Note: The CDATA block is not indented. However, indention is not significant. Since CDATA block is shown verbatim from a base indention determined by an XML viewer, such as in the case of the Internet Explorer, any indention introduced here will become excessive when the XML viewer renders the rulebase document. The CDATA blocks of previous rules are not indented either. But that might not be obvious since those blocks are not shown inside the <rule> tags.

Save the completed rulebase. It is necessary to convert this rulebase with infix rules, or the infix rulebase, to one that uses pure RDL so that it can be used by CA Rule Engine.

Convert Infix Rulebase to RDL

The infix rulebase cannot be used with CA Rule Engine directly. It must be converted to pure RDL first. This can be achieved by using the Infix2RDL tool script that is provided in the bin folder of the CA Rule Engine distribution.

Assuming the requirements outlined at the beginning of this chapter have been met, if the OS GUI supports drag and drop, like Windows Explorer for example, one can drag the icon of the infix rulebase file and drop it onto the icon of the Infix2RDL tool script in the bin folder of the CA Rule Engine distribution to generate the pure RDL rulebase as the *forward.xml* file that will be located in the same folder as the infix rulebase file.

As an alternative to drag and drop as described above, one can also start a command shell, change to the directory of the infix rulebase file, and use the following command to convert the infix rulebase to one that uses pure RDL:

```
infix2rdl forward_infix.xml
```

Since the file name of the infix rulebase follows the naming convention of having the *_infix.xml* suffix, there is no need to specify the optional name for the generated RDL rulebase file. That also made it possible to use drag and drop. The script will automatically save the RDL rulebase in the *forward.xml* file.

Java Client Classes

The Java client application consists of five classes, four corresponding to those of the rulebase (Customer, LiabilityType, Coverage, and InsuranceAgent), and one class to construct the objects and to invoke the rulebase. The classes that correspond to the domain of the rulebase must comply with the Rules for Constructing Java Classes. In particular, unqualified class names must agree with the rulebase class names. In addition, property names must also agree with their counterparts in the rulebase. These classes must:

- Provide an instanceName attribute and proper accessor methods.
- Comply with the standards of Java Beans.

In addition, these classes should implement the Serializable interface. Otherwise the client is prevented from using any serialization features.

Care must be taken to insure appropriate mapping of Java Property types to corresponding rulebase data types. For example, where the rulebase specifies that timeInBusiness is of duration type, the corresponding property must be treated as a string in the Java class. For more information on type mapping, see Relationship between RDL Field Data Types and Java Property Types.

The WrapperMaker tool can be used to generate the Java classes that correspond to the rulebase classes and a template class for invoking the rulebase. The generated classes satisfy the above requirements. In addition to being used directly, the generated classes can also be used as guides in retrofitting existing classes.

Generate Java Client Classes from Rulebase

To invoke the rulebase for inferencing in Java using JSR-94 API of CA Rule Engine, it is necessary to create the Java classes that wrap the corresponding rulebase classes and an additional application class. This can be achieved by using the WrapperMaker tool script that is provided in the bin folder of the CA Rule Engine distribution.

Assuming the requirements outlined at the beginning of this chapter have been met, if the OS GUI supports drag and drop, like Windows Explorer for example, one can drag the icon of the rulebase file and drop it onto the icon of the WrapperMaker tool script in the bin folder of the CA Rule Engine distribution to generate the Java class files that will be located in the same folder as the rulebase file. By doing that, the following five Java files are generated:

- Customer.java
- Coverage.java
- InsuranceAgent.java
- LiabilityType.java
- Forward_Insurance_Pricing_TierSession.java

The first four Java classes correspond to the rulebase classes of the same name. The last one is a template application class for invoking the rulebase. In addition, the WrapperMaker tool also generates the *forward_api.xml* file that contains the App Interface document for the rulebase, see The App Interface Document.

As an alternative to drag and drop as described above, one can also start a command shell, change to the directory of the rulebase file, and use the following command to generate the same set of files:

```
wrappermaker forward.xml
```

Using drag and drop or the above command, the generated Java classes will be put in the default package and the files will be put in the same folder as the rulebase file. If command line approach is used, one can also optionally specify the package name for the Java classes and the root location of the Java files. For example, if one is using a Windows computer, would like to put the generated classes in the *com.forwardinsurance* package and would like to have the root of the generated class hierarchy in the "C:\PricingTier" folder, the following command can be used for that purpose:

```
wrappermaker forward.xml com.forwardinsurance C:\PricingTier
```

Since changing package name and copying Java files to desired locations can be easily carried out by a Java IDE, drag and drop invocation of the WrapperMaker tool likely suffices. For the purpose of this tutorial, the Java classes will be kept in the default package.

Retrofit Existing Classes

If one is starting with an existing rulebase and writing a Java application from scratch, the next step can be simply to add code to invoke the rulebase to the template rulebase application class. To simulate the process of retrofitting existing classes for use with CA Rule Engine, it is assumed that there are existing classes for the insurance agent class and the policy liability class. However, since customer's time in business and bankruptcy information, and the policy liability will be input to the rulebase directly, there is freedom in the definition of the Customer and Coverage classes that merely serve as containers for above information. The WrapperMaker generated version of those two classes can be used without modification.

Note: In the Customer class, the WrapperMaker tool automatically implemented the collection field *bankruptcies* with an ArrayList and generated the following element change callback methods:

```
        public boolean onAddElementBankruptcies(String
bankruptciesElem, int index) {
                // TODO: Add additional handling logic here
                if (this.bankruptcies == null)
                        this.bankruptcies = new ArrayList();
                this.bankruptcies.add(index, bankruptciesElem);
                return true;
        }

        public boolean onDeleteElementBankruptcies(String
bankruptciesElem, int index) {
                // TODO: Add additional handling logic here
                this.bankruptcies.remove(index);
                return true;
        }
```

Since the data type for bean property *bankruptcies* is an array of Strings for the RDL collection of date time values, the above callbacks can carry out any element additions and deletions much more efficiently compared with constructing a whole new array and setting it to using the setter, which would otherwise be done by CA Rule Engine.

Retrofit the InsuranceAgent Class

Suppose the existing class for an insurance agent is defined like the following:

```
public class InsuranceAgent {
        private String claimStatus = null;
        private String name = null;
        private String pricingTier = null;

        /**
         * Constructs from the agent name
         * @param name the agent name
         */
        public InsuranceAgent(String name) {
                super();
                this.name = name;
        }

        /**
         * Returns the value of name
         * @return String
         */
        public String getName() {
                return this.name;
        }

        /**
         * Returns the value of pricingTier
         * @return String
         */
        public String getPricingTier() {
                return this.pricingTier;
        }

        /**
         * Sets the value of pricingTier
         * @param pricingTier value to be set
         */
        public void setPricingTier(String pricingTier) {
                this.pricingTier = pricingTier;
        }
```

```
        /**
         * Processes a claim and return status
         * @param claimNum the claim number
         * @return String
         */
        public String processClaim(int claimNum) {
                // Processing details omitted.
                return this.claimStatus;
        }
}
```

It is apparent that the existing class has problems on all requirements mentioned earlier. However, it is rather straightforward to merge the WrapperMaker tool generated version of this class with the existing class:

- The easiest one to fix is to implement the Serializable interface. It is just a matter of importing the interface and adding the declaration to the opening of the class. Copying those from the WrapperMaker version suffices.
- The instanceName attribute overlaps with the existing name field. Since name is a private field, it can simply be renamed to instanceName and the existing getName() method and constructor can be modified to use *this.instanceName* instead without affecting current clients of this class. The accessor methods for the instanceName attribute can be simply copied over from the WrapperMaker version.
- The missing public no-argument constructor can also be copied over from the WrapperMaker version.

Note: The WrapperMaker version of the InsuranceAgent class also contains methods equals(), hashcode() and toString(). These methods are not required to invoke rulebase inference but help user of a generated class to perform actions such as comparison and report/logging. Since the original InsuranceAgent class does not contain those functions but does contain additional field that necessities changes to those functions and this tutorial will not make use of them, these functions are not merged into the resulting InsuranceAgent class.

The following listing shows the merged results of the InsuranceAgent class (also save the code as InsuranceAgent.java to replace the WrapperMaker generated copy if manual changes described above have not been made):

```
import java.io.Serializable;

public class InsuranceAgent implements Serializable {
        private String claimStatus = null;
        private String instanceName = null;
        private String pricingTier = null;

        /**
         * The default empty constructor
         */
        public InsuranceAgent() {
                super();
                this.instanceName = null;
                this.pricingTier = null;
        }

        /**
         * Constructs from the instance name
         * @param instanceName the instance name
         */
        public InsuranceAgent(String instanceName) {
                super();
                this.instanceName = instanceName;
        }

        /**
         * Returns the value of instanceName
         * @return String
         */
        public String getInstanceName() {
                return this.instanceName;
        }

        /**
         * Sets the value of instanceName
         * @param instanceName value to be set
         */
        public void setInstanceName(String instanceName) {
```

```
        this.instanceName = instanceName;
    }

    /**
     * Returns the value of name
     * @return String
     */
    public String getName() {
        return this.instanceName;
    }

    /**
     * Returns the value of pricingTier
     * @return String
     */
    public String getPricingTier() {
        return this.pricingTier;
    }

    /**
     * Sets the value of pricingTier
     * @param pricingTier value to be set
     */
    public void setPricingTier(String pricingTier) {
        this.pricingTier = pricingTier;
    }

    /**
     * Processes a claim and return status
     * @param claimNum the claim number
     * @return String
     */
    public String processClaim(int claimNum) {
        // Processing details omitted.
        return this.claimStatus;
    }
}
```

Adapt the Liability Class

Suppose the existing class for policy liability is defined like the following (also save the code as Liability.java in the same folder as other source files for later use):

```
public class Liability {
    private static int enumCount = 0;
    private int enumVal;
    private String name;

    private Liability(String type)
    {
      name = type;
      enumVal = enumCount;
      enumCount++;
    }

    public String toString() { return name; }
    public int toInt() { return enumVal; }

    public static final Liability LOW = new Liability("Low");
    public static final Liability MED = new Liability("Med");
    public static final Liability HIGH = new Liability("High");
}
```

Like the original InsuranceAgent class, the Liability class also fails to meet any of the requirements given earlier. However, the situation is much direr for retrofitting this class to work with CA Rule Engine. It is apparent that this class follows the type-safe enum idiom, and for the idiom to work, no public constructors are allowed. That precludes the possibility of retrofitting the Liability class to work with CA Rule Engine since a public no-argument constructor is required in that case, let alone to mention that implementing serialization for type-safe enum is quite complicated.

The workaround is to introduce a new class that adapts the Liability class. In fact, the naming of the policy liability class in the rulebase as LiabilityType rather than simply Liability fits exactly this purpose, and the WrapperMaker generated LiabilityType class can be modified slightly to suit the purpose. In principle, the LiabilityType class should support conversion between itself and the Liability class. In the context of this tutorial, since policy liability is just an input to the rulebase, conversion from the Liability class to the LiabilityType class suffices. That can be done by simply introducing a new constructor to the WrapperMaker generated LiabilityType class that takes a Liability typed object as parameter. This new constructor can be defined as follows:

```
/**
 * Constructs from existing Liability object
 * @param objLiability the existing Liability object
 */
public LiabilityType(Liability objLiability) {
        this(objLiability.toString());
}
```

That is the only needed modification to the generated LiabilityType class.

Client Application Class

Finally, there must be a class for invoking the rulebase, and the WrapperMaker generated Forward_Insurance_Pricing_TierSession class is a template for such a class.

Note: Unlike the Java classes that have rulebase correspondents, the name of the session template class can be changed to any desired one. The WrapperMaker tool appends *Session* to the name of the rulebase to form the default name for that class.

The WrapperMaker generated session template class mainly provides two convenience features for a user planning to use the JSR-94 API to invoke an CA Rule Engine rulebase:

- The first is that the template class defines all static instances of rulebase classes that are part of the application interface as private static fields of the session class. In many cases, the static rulebase instances may be all that are needed to perform the require inference task and with their corresponding Java instances to be fields of the session class, it will be easy to set and retrieve properties before and after the inference execution since there is no need to keep track of handles to them. These fields are declared as static in anticipation that function that invokes the inference may be declared static as well for ease of invocation by users of this session class. However, depending on specific situation, one may choose a different approach and can simply remove such field definitions in that case.
- The other convenience feature is that the template class defines the getRuleServiceProvider() function that encapsulates the CA Rule Engine specific initialization for using the JSR-94 API to carry out inferencing.

The remaining task is to implement a method that accepts inputs to the rulebase and returns desired output. In this tutorial, the static method infer() takes this role. To be more realistic, this session class will also provide methods to register and deregister the pricing tier rule execution set. In order to make the result runable, a main() method and a performPricingTierInference() method are also added to invoke the infer() method on a couple of test cases.

The following is the complete listing of the resulting Forward Insurance pricing tier session class:

```
import javax.rules.*;
import javax.rules.admin.*;

import java.io.InputStream;
import java.util.List;
import java.util.LinkedList;

/**
 * Invoke inference using JSR-94 API to get pricing tier
 */
public class Forward_Insurance_Pricing_TierSession {

    // Static instances defined in rulebase
    private static Customer theCustomer = new
Customer("theCustomer");
    private static Coverage thePolicy = new Coverage("thePolicy");

    // The rule service provider ID as defined by CA implementation.
    private static final String RULE_SERVICE_PROVIDER =
"com.ca.cleverpath.aion.jsr94";

    // The URI for the rule execution set
    private static String ruleSetUri = "rulebases://forward";

    /**
     * Get the CA implementation of rule service provider
     * @return the rule service provider object
     * @throws Exception
     */
    private static RuleServiceProvider getRuleServiceProvider()
throws Exception {
        // Load the CA implementation of rule service provider.
        // Loading this class will automatically register this
        // provider with the provider manager.
        Class.forName(RULE_SERVICE_PROVIDER +
".RuleServiceProviderImpl");
        // Get the rule service provider from the provider manager
and return it.
```

```
        return
RuleServiceProviderManager.getRuleServiceProvider(RULE_SERVICE_PR
OVIDER);
    }

    /**
     * Register the pricing tier rule execution set
     */
    public static void registerPTRuleExecutionSet() {
        try {
            // Get the rule service provider
            RuleServiceProvider svcProvider =
getRuleServiceProvider();
            // Get the Rule Administrator
            RuleAdministrator ruleAdmin =
svcProvider.getRuleAdministrator();
            // Get the Local Rule Set provider
            LocalRuleExecutionSetProvider ruleSetProvider =
                ruleAdmin.getLocalRuleExecutionSetProvider(null);
            // Get an input stream to the Rulebase XML file.
            InputStream inStream =

Forward_Insurance_Pricing_TierSession.class.getResourceAsStream("
forward.xml");
            // Create the RuleExecutionSet for the rulebase file
            RuleExecutionSet ruleSet =
ruleSetProvider.createRuleExecutionSet(inStream, null);
            // Close the reader
            inStream.close();
            // Register the rule execution set
            ruleAdmin.registerRuleExecutionSet(ruleSetUri,
ruleSet, null);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    /**
     * Deregister the pricing tier rule execution set
     */
    public static void deregisterPTRuleExecutionSet() {
        try {
            // Get the rule service provider
            RuleServiceProvider svcProvider =
getRuleServiceProvider();
            // Get the Rule Administrator
            RuleAdministrator ruleAdmin =
svcProvider.getRuleAdministrator();
            // Deregister the rule execution set
            ruleAdmin.deregisterRuleExecutionSet(ruleSetUri,
null);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    /**
     * Executes pricing tier rulebase with the given inputs
     * @param customerTimeInBusiness, W3C style duration string
     * @param customerBankruptcies, array of W3C style datetime
strings
     * @param policyLiability, one of the three enum instances or
null for unknown
     * @return the InsuranceAgent that has the pricing tier info
     */
    public static InsuranceAgent infer(String
customerTimeInBusiness,
            String[] customerBankruptcies, Liability
policyLiability) {
        // Create an agent object to hold results
        InsuranceAgent agent = new InsuranceAgent("agent");
        try {
            // Get the rule service provider
            RuleServiceProvider svcProvider =
getRuleServiceProvider();
            // Create a runtime
            RuleRuntime runtime = svcProvider.getRuleRuntime();
            // Create a Stateless Rule Session using the Runtime
instance
            // If the pricing tier rule execution set has not been
registered,
            // register it and note the fact.
            boolean previouslyRegistered = true;
            StatelessRuleSession session = null;
            try {
                session = (StatelessRuleSession)
runtime.createRuleSession(
                        ruleSetUri,
                        null,
                        RuleRuntime.STATELESS_SESSION_TYPE);
            } catch (RuleExecutionSetNotFoundException e) {
                registerPTRuleExecutionSet();
                session = (StatelessRuleSession)
runtime.createRuleSession(
                        ruleSetUri,
                        null,
                        RuleRuntime.STATELESS_SESSION_TYPE);
```

```
                previouslyRegistered = false;
            }

            // Set the input fields

theCustomer.setTimeInBusiness(customerTimeInBusiness);
            theCustomer.setBankruptcies(customerBankruptcies);
            // Need to support unknown liability so test if input is
null
            thePolicy.setLiability((policyLiability == null) ? null
: new
                LiabilityType(policyLiability));
            // Add the objects to the list to be passed to the rule
engine.
            List inputObjects = new LinkedList();
            inputObjects.add(theCustomer);
            inputObjects.add(thePolicy);
            inputObjects.add(agent);
            // EXECUTE THE RULES!
            // LiabilityType object will be passed implicitly
            // because it is referenced by policy.
            session.executeRules(inputObjects);
            // Release session
            session.release();
            // Deregister rule execution set if it is registered by
this call
            if (!previouslyRegistered)
                deregisterPTRuleExecutionSet();
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
        return agent;
    }
```

```
    /**
     * Carry out and report pricing tier inference using the given
inputs to rulebase
     * @param customerTimeInBusiness, W3C style duration string
     * @param customerBankruptcies, array of W3C style datetime
strings
     * @param policyLiability, one of the three enum instances or
null for unknown
     */
    private static void doPricingTierInference(String
customerTimeInBusiness,
            String[] customerBankruptcies, Liability
policyLiability) {
        System.out.println("Customer time in business: " +
            ((customerTimeInBusiness == null) ? "unknown" :
customerTimeInBusiness));
        System.out.println("Customer bankrupted " +
            ((customerBankruptcies == null) ? "unknown" :
                Integer.toString(customerBankruptcies.length)) + "
time(s)");
        System.out.println("Policy liability is " +
            ((policyLiability == null) ? "unknown" :
policyLiability.toString()));
        InsuranceAgent agent = infer(customerTimeInBusiness,
               customerBankruptcies, policyLiability);
        String tier = agent.getPricingTier();
        System.out.println("Resulting pricing tier is: " +
            ((tier == null) ? "unknown" : tier));
        System.out.println();
    }

    public static void main(String[] args) {
        doPricingTierInference("P3Y", new String[] {"2003-06-24"},
null);
        doPricingTierInference("P7Y", new String[] {},
Liability.MED);
        doPricingTierInference("P2Y", null, Liability.MED);
    }
}
```

A few things about the implementation of the session class to note are:

- Additional imports are necessary to support (1) invoking the JSR-94 admin functions, (2) converting the forward.xml file into an InputStream to feed into CA Rule Engine, and (3) passing a list of objects to the rulebase.
- The WrapperMaker generated field definitions for the three liability types and the insurance agent object have been removed. For the case of liability type objects, they are not explicitly used. As for the insurance agent object, returning a dynamically constructed one will reduce the chance that the caller will refer to an object that has been changed by a subsequence inference session. However, if it is known that the caller will not hold the returned reference to the insurance agent object for later use, the WrapperMaker generated static insurance agent object can be used to reduce memory usage.
- The return value of the executeRules() call is not used. In principle, that return value contains the references to objects returned after inferencing and should be examined to retrieve the results. In this case, since the insurance agent object is in the scope of the infer() function, there is no need to retrieve another reference to it from the return value of the executeRules() call.
- The infer() method registers the pricing tier rule execution set on the fly and deregister afterwards if the caller has not registered it before. This structure allows easy usage by a causal user. However, the caller should separately register and deregister the rule execution set for batch operation to improve efficiency.

Execute the Application

In principle, a separate caller program will invoke the Forward_Insurance_Pricing_TierSession.infer() method to perform the pricing tier analysis. For simplicity, the session class has been made directly runnable. To execute it, just start a command shell and change directory to the folder that contains the code files for this tutorial. Assuming default has been taken when running the Infix2RDL and the WrapperMaker tools, this folder should contain the following 3 XML files and 6 Java source files:

- forward_infix.xml
- forward_api.xml
- forward.xml
- Customer.java
- Coverage.java
- InsuranceAgent.java (modified as previously described)
- Liability.java (saved as previously described)
- LiabilityType.java (modified as previously described)
- Forward_Insurance_Pricing_TierSession.java (modified as previously described)

First the Java source files need to be compiled. Assuming PATH and CLASSPATH have been set as described at the beginning of this chapter, the following command compiles the Java source files:

```
javac *.java
```

The following command can be used to run the application:

```
java Forward_Insurance_Pricing_TierSession
```

The following outputs should be the results of the execution:

```
Customer time in business: P3Y
Customer bankrupted 1 time(s)
Policy liability is unknown
Resulting pricing tier is: Reject

Customer time in business: P7Y
Customer bankrupted 0 time(s)
Policy liability is Med
Resulting pricing tier is: C

Customer time in business: P2Y
Customer bankrupted unknown time(s)
Policy liability is Med
Resulting pricing tier is: unknown
```

For the first two cases, the pricing tier rulebase successfully resolved the pricing tier and the results can be easily verified by referring back to the decision trees given in The Pricing Tier Decision Tree and The Customer Financial Stability Decision Tree, and taking the default liability type rule into account. For the last case, with the customer's bankruptcy information as unknown, the existing rules cannot resolve the case and the resulting pricing tier is still unknown.

Obtain Log of Execution

To obtain a log of execution, it is necessary to configure the logging for CA Rule Engine, see Configure and Execute Logging for CA Rule Engine. For this tutorial, the sample configuration file "log4j.properties" available in the samples folder in the CA Rule Engine distribution can be used. That file configures CA Rule Engine for logging at the WARN level, in append mode. Just copy that file to the folder containing the tutorial files and run the application with the following command:

```
java -Dlog4j.configuration=log4j.properties
Forward_Insurance_Pricing_TierSession
```

The file *JSR94-wrapper.log* can be found after the execution. Since *log4j.properties* is loaded by log4j's default configuration algorithm, the option *-Dlog4j.configuration=log4j.properties* is actually optional. The content of the file will be similar to the following (note that the lines are wrapped):

```
14:24:47,461 ERROR com.ca.cleverpath.aion.jsr94.RuleRuntimeImpl
main - JSR94-6003: Specified uri: rulebases://forward cannot be
resolved to a registered RuleExecutionSet
14:24:48,853 ERROR com.ca.cleverpath.aion.jsr94.RuleRuntimeImpl
main - JSR94-6003: Specified uri: rulebases://forward cannot be
resolved to a registered RuleExecutionSet
14:24:48,993 ERROR com.ca.cleverpath.aion.jsr94.RuleRuntimeImpl
main - JSR94-6003: Specified uri: rulebases://forward cannot be
resolved to a registered RuleExecutionSet
```

The preceding three lines are the results of executing the first createRuleSession() call in the infer() method. Since the main() method neglected to register the pricing tier rulebase beforehand, the call results in a RuleExecutionSetNotFoundException being thrown, which the infer() method handled by dynamically registering the rule execution set and created the rule session by executing the second createRuleSession() call.

Obtain Inferencing Summary Documents

Changes to the Forward_Insurance_Pricing_TierSession class will be needed to obtain the optionally generated Inferencing Summary documents for rule behaviors during an inference session. For general information for obtaining Inferencing Summary documents, see Obtain Rulebase Documents. Specific to this tutorial, the following changes to *Forward_Insurance_Pricing_TierSession .java* file will be needed:

1. Add the following import statements at the beginning of the file for defining a property map and for obtaining and writing out the Inferencing Summary documents:

```
import
com.ca.cleverpath.aion.jsr94.AionRulesEngineProperties;
import java.util.Hashtable;
import
com.ca.cleverpath.aion.jsr94.CARuleExecutionSetMetadata;
import java.io.FileWriter;
```

2. Just before the inner try block of the infer() method, add the following definition of the property map:

```
Hashtable sessionProps = new Hashtable();
sessionProps.put(

AionRulesEngineProperties.RULEBASE_GENERATE_INFERENCING_SUMMA
RY,
   new Boolean(true)
);
```

Change the third parameter of the two createRuleSession() calls in the infer() method from null to sessionProps.

Add the following in between the session.executeRules() call and the session.release() call:

```
// Obtain and write out Inferencing Summary document
   CARuleExecutionSetMetadata ruleExecSetInfo =
(CARuleExecutionSetMetadata)
       session.getRuleExecutionSetMetadata();
   String xmldoc = ruleExecSetInfo.getInferencingSummary();
   FileWriter xmlFile = new FileWriter("inferSummary-" +
       customerTimeInBusiness + ".xml");
   xmlFile.write(xmldoc);
   xmlFile.close();
```

Save and recompile the *Forward_Insurance_Pricing_TierSession .java* file and run the application again. Three Inferencing Summary document files corresponding to the three sample cases are generated in the same folder. Comparing the rule behavior for the PricingTierRule between the first case (the *P3Y* case):

```
<rule abort_count="0" fail_count="0" fire_count="1"
name="PricingTierRule"
        objid="" pend_count="1">
    <decision base_pend_count="1" iter_pend_count="0"
name="PricingTier" objid="">
        <alternative abort_count="0" action_pend_count="0"
altpos="3" apply_count="1"
            premise_pend_count="0"/>
    </decision>
</rule>
and the third case (the P2Y case):
<rule abort_count="0" fail_count="0" fire_count="0"
name="PricingTierRule"
        objid="" pend_count="1">
    <decision base_pend_count="1" iter_pend_count="0"
name="PricingTier" objid=""/>
</rule>
```

It can be seen that the rule fired in the first case and thus resolved the pricing tier for that case but not in the third case so the pricing tier was still unknown for it.

Filter the Returned Rulebase Objects

In the preceding implementation of the session class, the executeRules() method returned a list of objects that included the rulebase precondition objects as well as the postcondition objects. That can become more evident if the executeRules() call in the infer() method is replaced following:

```
List objectsReadBack = session.executeRules(inputObjects);
System.out.println("Number of objects returned is " +
objectsReadBack.size());
```

Recompile and run the application will cause the following line of output being added to all three cases:

```
Number of objects returned is 4.
```

With the four objects being the *theCustomer*, *thePolicy* and *agent* objects that was explicitly passed to CA Rule Engine engine and the implicitly passed policy liability object of the class LiabilityType.

Suppose that the list of objects were prepared by a different function, and thus the *agent* object is not in the scope of the function that executes the inference, and it must be retrieved from the returned list of object references. It will be desirable then to return just the InsuranceAgent object. The executeRules() method would still return a list of objects, but that list would contain only one object, namely the postcondition object, *agent*.

It is possible to achieve this level of control over what is returned by the rulebase through the use of ObjectFilters. For an explanation of ObjectFilters, see Object Filters. This section shows how to write an ObjectFilter that would return just the InsuranceAgent instance and how to incorporate this object into the session implementation code.

Write ObjectFilters

An ObjectFilter is a special type of object that can be passed to either a stateful or stateless rule session. CA Rule Engine knows how to use this special object to decide which objects it will return to the client application. It is essential that the object passed to the session implements the Object filter(Object object) method of the ObjectFilter class. If an ObjectFilter has been provided, CA Rule Engine calls this method and passes as the input argument the object that it wants to return to the client application. The implementation code has the option of returning that object (if it is to be returned to the client application) or returning null if the object is not to be returned.

A simple ObjectFilter for filtering only the InsuranceAgent object is as follows (also save the code as AgentObjectFilter.java in the source file folder):

```
// java imports
import javax.rules.ObjectFilter;

/**
* Utility filter class for the tutorial application.
 * <p>
 * This class implements an ObjectFilter that will be used to select
the
 * InsuranceAgent instance to return to the Java client application.
*/
public class AgentObjectFilter implements ObjectFilter
{
        /** AgentObjectFilter constructor. */
    public AgentObjectFilter() { }

        /** @see javax.rules.ObjectFilter#filter */
        public Object filter(Object object)
        {
        if (object instanceof InsuranceAgent)
                        return object;

                return null;
        }

        /** @see javax.rules.ObjectFilter#reset */
        public void reset() { }
}
```

Returning Only the InsuranceAgent Object

The code in this section further modifies the Forward_Insurance_Pricing_TierSession class code, see Client Application Class, to create the filter object and then passes the filter object to the session object by using the overloaded form of the executeRules() method. Replace the executeRules() call in the infer() with the following:

```
//create an ObjectFilter that returns only InsuranceAgent objects.
AgentObjectFilter insuranceAgentFilter = new AgentObjectFilter();
List objectsReadBack = session.executeRules(inputObjects, insuranceAgentFilter);
```

Compile the AgentObjectFilter.java file, recompile the Forward_Insurance_Pricing_TierSession.java file and run the application again, the number of returned objects changed to one as indicated by the following line of output in all three cases:

```
Number of objects returned is 1.
```

Appendix A: Samples

This appendix describes the sample applications included in the CA Rule Engine distribution. Each sample application includes a rulebase and Java source files for invoking inference using that rulebase through the JSR-94 API provided by CA Rule Engine. The description given for each sample explains the scenario behind it and provides notes on specific features of the rulebase and its associated Java implementation.

All sample related files are located in the samples folder in the CA Rule Engine distribution. The Java source files of each sample are put inside the *com.ca.cleverpath.aion.jsr94.samples.sampleName* package, where *sampleName* is the name of the sample in lowercase letters without space. The samples folder in the CA Rule Engine distribution is the root folder of the packages. For each sample rulebase, both infix version and pure RDL versions are provided respectively in the infix and xml folders which are at the same level as the *sampleName* folders that contain Java source files. Script files for building and running the provided samples can be found directly under the samples folder.

Running a Sample Application

In order to build and run the provided sample applications, the CA Rule Engine distribution must have been properly installed. See Install and Configure CA Rule Engine for more information. The batch/script files provide in the samples folder do take care of CLASSPATH settings for the sample applications.

The following sample batch files for Windows and corresponding shell scripts for UNIX/Linux platform are provided to compile and run the samples:

- compilesamples to compile all sample client applications.
- run*sampleName*admin to run the administrator part of the *sampleName* sample. For example, the runtlpadmin script runs the administrator part of the TLP sample.
- run*sampleName*session to run the whole rule session of the *sampleName* sample. For example, the runtlpsession script runs the rule session part of the TLP sample.

These sample batch files also illustrate the CLASSPATH settings needed and the use of log4j logging framework configuration for CA Rule Engine. In the sample batch files, the default way to configure log4j framework is using the System property log4j.configuration. Applications can choose to configure log4j in one of the many ways that the framework supports.

Since the purposes for the sample applications are more for illustrating ideas and they evolved from test cases for the CA Rule Engine, the implementation of the Java session classes for the sample applications is not polished. User is encouraged to modify the sample applications, especially the session classes to their own needs, e.g. to test different precondition values. The provided scripts help in build and run the customized samples.

Also see the Readme in the samples folder in the CA Rule Engine distribution for any updated instructions on running the sample applications.

TLP Sample

The TLP sample is a version of the Forward Insurance case used in the Tutorial chapter. To avoid conflicting names, the name of the company in this sample is changed to TLP. This sample demonstrates the use of multiple domains to organize rules for different purposes.

TLP System

The TLP insurance company wants to use inferencing for the calculation of pricing tier to be used by its insurance agents in preparing quotes. In the process, additional rules are also used for the calculation of intermediate results such as the property-insurance risk based on building materials, customer information, and so forth.

The following are the inputs to this sample application:

- Customer Information:

  Time in business

  Occurrences of bankruptcies (date and times for them)

  Years at current address

- Coverage or policy information:

  Type of coverage (“Building,” “Contents” or “Malpractice”)

  Type of risk (“Medical,” “Construction” or “Retail”)

- Property information:

  Type of construction (“Brick,” “Stick” or “Straw”)

  Property value

  Construction date

  Information on whether excessive prior losses have been paid

The following are the outputs from this sample application:

- Intermediate results calculated:

  Customer financial stability ("Poor," "Average" or "Excellent")

  Coverage liability ("Low," "Med" or "High")

  Property Risk ("Low," "Med" or "High")

- Final result calculated:

  Pricing Tier for use by Insurance Agent ("A," "B," "C" or "Reject")

Additionally, in the sample application, the pricing tier is adjusted based on the region of the property.

The following are inputs for adjusting pricing tier:

- Property information:

  Region of the property ("Region1," "Region2" or "Region3")

- Insurance agent information:

  The calculated pricing tier ("A," "B," "C" or "Reject")

The following are the outputs from this adjustment:

- Insurance agent information:

  The adjusted pricing tier ("A," "B," "C" or "Reject")

TLP Sample Rulebase

The TLP sample rulebase defines two appshared domains, *PricingTierDomain* and *RegionAdjustmentDomain*, for calculating the pricing tier and the adjusted pricing tier respectively. In both appshared domains, the corresponding inputs (see Sample JSR-94 Applications) appear as pre-conditions and the resulting outputs including the intermediate terms appear as post-conditions.

In the *PricingTierDomain,* the following are the rules for calculating outputs from inputs:

- The coverage liability rule calculates coverage liability based on type of coverage and type of risk.
- The property age rule calculates the age of the property based on its construction date.
- The property risk rule calculates property risk based on type of construction, age of the property, information on whether excessive prior losses have been paid and property value.

  Note: The age of the property is calculated based on its construction date.
- The customer financial stability rule calculates customer financial stability based on years in business, number of bankruptcies and years at current address.
- The pricing tier rule calculates the final result-pricing tier based on previously calculated intermediate results (coverage liability, property risk and customer financial stability).
- Additionally, there are some rules for assigning default values to the pre-conditions when their values are unknown. For example, if the construction type of the property is unknown, it is assigned a value of "Straw" by default.

In the *RegionAdjustmentDomain,* outputs are calculated from inputs. The region adjustment rule calculates the adjusted pricing tier based on the region of the property and the calculated pricing tier.

The sample rulebase defines the following rulebase classes and static instances to capture the business objects:

- *Customer*, static instance: *theCustomer*

  Fields:

  timeInBusiness, rulebase data type: duration

  bankruptcies, rulebase data type: set of datetime

  yearsAtCurrentAddress, rulebase data type: number with precision 0

  financialStability, rulebase data type: string

- *Coverage*, static instance: *thePolicy*

  Fields:

  typeOfCoverage, rulebase data type: string

  typeOfRisk, rulebase data type: string

  liability, rulebase data type: string

  Note: The fields typeOfCoverage and typeOfRisk can also be defined as inst-ref type that refers to CoverageType and RiskType classes that have static instances enumerate the types of coverage and types of risk. The Property class below adopts such an approach for its typeOfConstruction and region fields. The advantage of that approach is better error handling at compile time.

- *Property*, static instance: *theProperty*

  Fields:

  value, rulebase data type: number with precision 0

  constructionDate, rulebase data type: datetime

  hasExcessiveLossesPaid, rulebase data type: boolean

  typeOfConstruction, rulebase data type: inst_ref to ConstructionType class

  region, rulebase data type: inst_ref to PropertyRegion class

  risk, rulebase data type: string

- *InsuranceAgent*, static instance: *agent*

  Fields:

  pricingTier, rulebase data type: string

  adjustedTier, rulebase data type: string

- *ConstructionType*, static instances: *Brick, Stick, Straw*
- *PropertyRegion*, static instances: *Region1, Region2, Region3*

Java Applications for the TLP Sample

The package used for the TLP sample application is:

```
com.ca.cleverpath.aion.jsr94.samples.tlp
```

The following bean classes are defined to match the rulebase classes defined in the sample rulebase:

- *Customer*

  Properties*:*

  instanceName, Java data type: String

  yearsAtCurrentAddress, Java data type: Integer

  timeInBusiness, Java data type: String

  bankruptcies, Java data type: String[]

  financialStability, Java data type: String

  Note: For a number field like the *yearsAtCurrentAddress* field, the preferred Java data type is BigDecimal, which is used by the WrapperMaker tool for the Java data type corresponding to the number type in RDL regardless of precision specified by RDL. However, depending on actual values for that field and other constraints, alternative data types such as Integer may be used instead.

- *Coverage*

  Properties*:*

  instanceName, Java data type: String

  typeOfCoverage, Java data type: String

  typeOfRisk, Java data type: String

  liability, Java data type: String

- *Property*

  Properties*:*

  instanceName, Java data type: String

  value, Java data type: Integer

  constructionDate , Java data type: String

  hasExcessiveLossesPaid, Java data type: Boolean

  typeOfConstruction, Java data type: ConstructionType class

  region, Java data type: PropertyRegion class

  risk, Java data type: String

- *InsuranceAgent*

  Properties*:*

  instanceName, Java data type: String

  pricingTier, Java data type: String

  adjustedTier, Java data type: String

- *ConstructionType*

  Property: *instanceName*, Java data type: String

- *PropertyRegion*

  Property: *instanceName*, Java data type: String

These classes define the business objects. Note that the class names have been carefully chosen so that the unqualified Java class names correspond to matching rulebase class names. The bean class property names match the rulebase class field names and the Java data types for the bean class properties are compatible with the rulebase field types. For CA Rule Engine component to verify the Java classes and objects, these rules need to be followed.

For the TLP sample, the following sample application programs are provided to illustrate the use of CA Rule Engine, CA's implementation of the JSR-94:

- *TLPAdmin.java* illustrates creation of a *RuleExecutionSet* from the sample rulebase TLP.xml file and shows all properties of the rule set using the administrator API of JSR-94.
- *TLPSession.java* illustrates a full cycle of operations for inferencing using the sample rulebase .xml file. This includes not only some administrative functionality, such as registering the rule set, but also runtime functionality, such as creating a stateful rule session, adding objects to it and executing rules on it. In particular, it illustrates the selection of a domain in creating a rule session.

Another point to note is that the WrapperMaker tool was not used in the creation of the Java classes for the TLP sample since this sample was developed before the tool. For those classes that wrap the corresponding rulebase classes, the difference is limited to the selection of a different numerical Java type for the rulebase number type, e.g. use of Integer rather than BigDecimal. For the session class, it doesn't declare the private static fields corresponding to the static instances inside the rulebase as would the WrapperMaker tool. It rather defines those inside the function that carries out inference. However, the difference is more on the side of programming style.

Shopping Cart Sample

The Shopping Cart Sample illustrates the use of iterative expressions and decisions in rulebase design. The design of the rulebase also illustrates the use of client creatable classes. It is also illustrates the use of private classes (for example, classes that are not used in either preconditions or postconditions) and the private classes may be defined at a lower level than the rulebase level, depending how such a class is used.

Shopping Cart System

A fictitious online merchant would like to use rules to automatically apply vouchers a customer has against eligible purchases in his shopping cart, computes overall discounts and issuing new vouchers based on various properties of the customer and the items purchased.

The following are the inputs to this sample application:

- Customer Information:

  ID number

  One of three regions ("Region1," "Region2" or "Region3") where customer is located

  Vouchers the customer has

- Shopping cart Information:

  Customer using this cart

  Items the customer purchased

- Item:

  Price

  Description

- Voucher:

  Value

  Expiration date

  Category

  Whether used

The following are the outputs from this sample application:

- For the shopping cart:

  Total amount of purchase, i.e. sum of item price

  Net amount of purchase, i.e. after applying qualified vouchers

  Average item price

  Standard deviation of item price

- For the customer:

  Discount rate for the customer to be applied to the overall transaction.

  Updated vouchers with new ones if applicable.

Shopping Cart Sample Rulebase

The Shopping cart sample rulebase defines the *DiscountsAndVouchers* appshared domain for calculating discounts and processing vouchers. There are four rulesets defined in that domain:

- The *Validation* ruleset groups rules to validate the preconditions that the user input to the system.
- The *PrelimCalcs* ruleset groups rules to calculate the purchase amount and statistics for the items in the shopping cart.
- The *ApplyDiscountRate* ruleset groups rules to calculate the discount rate for the customer to be applied to the overall transaction.
- The *ApplyVouchers* ruleset contains the rule to issue new vouchers to customer for eligible transaction.

The four rulesets are prioritized according to the above listing order so that they are invoked in that sequence. Within the *PrelimCalcs* and the *ApplyDiscountRate* rulesets, rules are also prioritized so that they can be applied sequentially.

The design of the class structure in the Shopping Cart sample rulebase largely follows those dictated by the input and outputs of the system with some less intuitive design changes just to illustrate certain functionality of the RDL. The following classes are defined at the rulebase level:

- *Customer*, creatable by application

  Fields*:*

  id, rulebase data type: number with precision 0

  discountRate, rulebase data type: number with precision 1

  region, rulebase data type: inst-ref to CustomerRegion class

  vouchers, rulebase data type: inst-ref to set of Vocuher class objects

  iiCategory, rulebase data type: inst-ref to IICustomerCategory class

- *CustomerRegion*, static instances: *REGION1, REGION2, REGION3*
- *Item*, creatable by application

  Fields*:*

  price, rulebase data type: number with precision 2

  description, rulebase data type: string

  iiCategory, rulebase data type: string

- *ShoppingCart*, static instance *theCart*

  Fields*:*

  theCustomer, rulebase data type: inst-ref to Customer class

  netPurchases, rulebase data type: number with precision 2

  totalPurchases, rulebase data type: number with precision 2

  avgPrice, rulebase data type: number with precision 2

  stdDevPrice, rulebase data type: number with precision 6

- *Voucher*, creatable by application

  Fields*:*

  value, rulebase data type: number with precision 2

  expirationDate, rulebase data type: datetime

  itemCategory, rulebase data type: string

  isUsed, rulebase data type: boolean

- *IICustomerCategory*, static instances: *GOLD, SILVER, BRONZE*

- *IIItemGroup*

  Fields*:*

  category, rulebase data type: string

  totalPurchases, rulebase data type: number with precision 2

  minItemPrice, rulebase data type: number with precision 2

  items, rulebase data type: inst-ref to set of Item class objects

Note: Any field whose name begins with *ii* is used internally by the rulebase. This is also true for any class whose name begins with *II*.

In addition to classes defined at the rulebase level, this sample rulebase also defines a private class at the ruleset level for the *PrelimCalcs* ruleset.

Even though the items conceptually should belong to the shopping cart, this sample chooses to use rules to load the items from all client-created Item instances so that the shopping cart class does not need to contain references to them.

Other design details of the rules of interests include:

- Rule: *Validation.ValidateItems* illustrates use of the SUBSTRING and STRINDEX operators.
- Rule: *PrelimCalcs.SortFilteredVouchers* illustrates use of SORT operator for sorting vouchers at multiple levels.
- Rule: *PrelimCalcs.CalcitemGroups* illustrates use of the iterative EXISTS and SELECT ONE operators. It also creates instances of a class (IIItemGroup).
- Rule: *PrelimCalcs.SummarizeGroupItems* illustrates use of the iterative SUMMATION, SELECT ONE and SORT operators - and includes an example of *nested expressions*.
- Rule: *PrelimCalcs.ApplySortedVouchers* illustrates use of "nested iteration".
- Rule: *PrelimCalcs.SummarizeCartContents* illustrates use of the iterative SUMMATION, AVERAGE and STANDARD DEVIATION operators.
- Rule: *ApplyDiscountRate.IncreaseDiscountRate* illustrates use of the sizeOf operator for calculating the number of Item instances.
- Rule: *ApplyVouchers.AwardNewVouchers* illustrates use of the SUBCOLLECTION, SORT and AVERAGE operators to calculate voucher values; and the use of DateTime and Duration values to calculate a 90-day expiration date for a new voucher. It also creates instances of a class (Voucher).

Sample Java Applications for the Shopping Cart Sample

The package used for the Shopping Cart sample application is:

```
com.ca.cleverpath.aion.jsr94.samples.shoppingcart
```

The bean classes that match the rulebase classes are created using the WrapperMaker tool so the details of those are omitted. Please refer to Using the WrapperMaker Tool for more information on construction of them.

Like the TLP sample, two application programs are provided. The ShoppingCartAdmin.java illustrates the use of some Administrator API of JSR-94 using the shoppingcart.xml rulebase.

The ShoppingCartSession.java illustrates the inference process:

1. The session sample application sets up the rulebase for use through the JSR-94 API.
2. The session sample application creates a Customer object, initializes some of its fields, and assigns it to the static ShoppingCart object. The application also creates four Voucher instances, initializes them, and sets them to the Customer object.
3. The application defines the contents of the ShoppingCart by creating eight Item instances and initializing them. Note that each Item description includes a prefix describing the Item's category (for example, "Dec" for "Decoration", "Rec" for "Recreation", and so on). These categories correspond to the previously-described Voucher itemCategories.

At this point, the application adds the object to CA Rule Engine and invokes inferencing. Note that some object are added implicitly via reference.

During inferencing, the rules:

- Validate pre-condition values
- Selectively apply vouchers against purchased items
- Calculate various ShoppingCart statistics and a discount rate to be applied to the overall transaction
- Conditionally generate additional vouchers

Upon returning from inferencing, the application fetches the results and verifies them.

If the ShoppingCart contains no Items, the avgPrice (average) and stdDevPrice (standard deviation) fields are unresolved.

The returned vouchers indicate not only older vouchers applied to this transaction but also any new vouchers generated by the rules.

Expense Approval Sample

The Expense Approval Sample is a simple application that illustrates the use of stateful session to provide preconditions incrementally only as they are needed. The design of the rulebase also illustrates the use of associations that automatically match client created receipt and approval objects to the expense object. The rulebase also illustrates the use of initialization method to set value to certain rulebase fields.

Expense Approval System

A fictitious company wants to use rules to automatically process expense approval. It wants to cut expenses by asking the user to provide only enough information on the expense case for a decision can be made.

For any expense approval item, an amount and a description are required. The user enters those two items to start the process. Based on the expense amount and description, the system also may require the user to provide a receipt and manager approval before a decision can be made.

The following are the inputs to this sample application:

- Amount of expense
- Description of expense
- Receipt
- Manager approval

Receipt and Manager Approval are entered only if necessary.

The outputs from this sample application are:

- Approval status
- Message indicating what is needed next (if decision can't be made).

The Expense Approval Sample Rulebase

The rules for this expense approval sample are:

- If amount is less than or equal to limit for tips and the description contains the word *tip*, the expense is approved. Otherwise, receipt is required.
- If provided receipt does not match the expense item or of different amount, the expense is denied.
- If amount is less than or equal to limit for small expense and a receipt matching this expense and for the exact amount is provided, the expense is approved.
- Manager review and approval is needed for amount exceeding limit for small expense.

The Expense Approval sample rulebase defines the *ExpensePolicy* appshared domain to process items to be reimbursed. There is a single ruleset, the *ExpenseRuleSet*, defined inside that domain. Inside this ruleset, the *ExpenseRule* rule, implements the rules for expense approval as its decisions. In order to provide feedback to the user on what needs to be supplied next, two reporting rules, *requestReceipt* and *requestApproval*, are also defined in the *ExpenseRuleSet*. Different priorities are assigned to the three rules so that the main *ExpenseRule* is processed first and reporting of receipt request occurs before that of approval.

The rulebase structure is designed to illustrate the use of association in RDL and to use stateful session to request additional preconditions only when that is necessary. The following classes are defined at the rulebase level:

- *ExpenseItem*, static instance *theItem*

  Fields*:*

  amount, rulebase data type: number with precision 2

  description, rulebase data type: string

  receiptObj, rulebase data type: inst-ref to Receipt object

  approvalObj, rulebase data type: inst-ref to Approval object

  approvalStatus, rulebase data type: Boolean

  tipLimit, rulebase data type: number with precision 2

  smallLimit, rulebase data type: number with precision 2

- *Receipt*

  Fields*:*

  amount, rulebase data type: number with precision 2

  item, rulebase data type: inst-ref to ExpenseItem object

- *Approval*

  Fields*:*

  item, rulebase data type: inst-ref to ExpenseItem object

- *Message*, static instance *theMessage*

  Fields*:*

  msgText, rulebase data type: string

The first four fields of the *theItem* instance, all fields of Receipt objects and Approval objects are preconditions. The *approvalStatus* field of the *theItem* instance and the *msgText* field of the *theMessage* instance are postconditions. The *tipLimit* and *smallLimit* fields of the *theItem* instance are internal rulebase fields that are initialized with an initialization method of the rulebase.

This rulebase also defines two associations, *ReceiptAndItem* and *ApprovalAndItem*. They associate receipt with its expense item and approval with its expense item respectively. CA Rule Engine makes sure associated fields are synchronized, e.g. if a Receipt object is set to refer to an ExpenseItem object, CA Rule Engine makes sure the ExpenseItem object is also set to refer to that Receipt object.

The two reporting rules and the Message class are purely for purposes of reporting to user what is needed to complete the expense approval process. If such feedback is not needed, those two rules and the Message class can simply be omitted.

Sample Java Applications for the Expense Approval Sample

The package used for the Expense Approval sample application is:

```
com.ca.cleverpath.aion.jsr94.samples.expense
```

The bean classes that match the rulebase classes are created using the WrapperMaker tool so the details of those are omitted. Please refer to Using the WrapperMaker Tool for more information on construction of them.

Like previous samples, two application programs are provided. The ExpenseAdmin.java illustrates the use of some Administrator API of JSR-94 using the expense.xml rulebase.

The ExpenseSession.java illustrates the whole inference process:

1. The session sample application sets up the rulebase for use through the JSR-94 API.
2. The session sample application sets amount and description and starts inferencing. What distinguishes this sample from the previous two is that the session contains multiple calls to execute the rules.
3. After the initial inference terminates, this sample application checks and sees that expense approval process is not yet resolved and receipt is needed to continue. Then it creates a Receipt object for this expense and added that to the session and starts inference again. After the inference terminates again without resolving the approval process for lack of manager's approval, this sample application creates an Approval object for the expense and continues with inference. This time the expense process is resolved with the expense approved.

This sample illustrates the use of stateful rule session to carry out *incremental inferencing*. Incremental inferencing means that just a minimal set of preconditions are provided to start inferencing and additional preconditions are provided only when it is apparent that they are needed under the circumstance.

When the Java Receipt object and the Java Approval object were created, it was the Java ExpenseItem object that was set to the item fields of Receipt and Approval objects. These two new objects were not set to the inst-ref fields declared in the rulebase ExpenseItem object. Those two fields of the rulebase ExpenseItem object are not even declared as preconditions so it is impossible to set them on the Java side anyway. However, inside the rulebase, those two objects are actually addressed from the ExpenseItem object. What happened was that the two associations *ReceiptAndItem* and *ApprovalAndItem* defined in the rulebase automatically updated the two internal inst-ref fields of the ExpenseItem object to refer to the Receipt and Approval objects.

Appendix B: Verify JSR94 Compliance

The TCK for a Java standard specification allows a vendor implementing the standard to determine if the implementation is compliant with the specification. CA has confirmed that CA Rule Engine, CA's JSR-94 implementation, passes the TCK. For vendor-specific configuration files and rule execution set files to be used with the test compatibility kit, see the tckconfiguration folder inside the CA Rule Engine distribution.

Note: A TCK tests only for compliance and does not perform any functional tests on the vendor's implementation.

The JSR-94 Java Rule Engine API specification is bundled with its own TCK. More information about this TCK can be found with the document on TCK that comes with the specification download.

Verify CA Rule Engine for Compliance

CA Technologies has verified that CA Rule Engine passes the TCK. The installation folder for CA Rule Engine is referred to as *aionjre_home* in the steps mentioned below.

To verify CA Rule Engine for JSR-94 standard compliance, follow these steps:

1. Download the JSR-94 v1.0 specification and install it. The remaining steps in this procedure refer to the installation directory as *jsr94-1.0_home*.

   The Technology Compatibility Kit (TCK) for JSR-94 (the Java Rule Engine API) is officially available as part of the full specification download from the Java Community Process website at: **http://www.jcp.org** (type **94** in the JSR text box and press Enter). TCK 1.0 requires Java 1.4.2.

2. Back up the original TCK-related rule set .xml files:

   - *jsr94-1.0_home*/lib/tck_res_1.xml
   - *jsr94-1.0_home*/lib/tck_res_2.xml

   Copy the following CA implementation-specific .jar files to the *jsr94-1.0_home*/lib folder:

   - *aionjre_home*/aionjre.jar
   - *aionjre_home*/tckconfiguration/aion-jsr94-tck.jar

3. Copy the *aionjre_home*/lib/log4j.jar file to the *jsr94-1.0_home*/lib folder.
4. Copy the following CA implementation-specific rule set .xml files, to the *jsr94-1.0_home*/lib folder:
   - *aionjre_home*/tckconfiguration/tck_res_1.xml
   - *aionjre_home*/tckconfiguration/tck_res_2.xml
5. Back up the original TCK configuration file, *jsr94-1.0_home*/lib/tck.conf.

   Copy the *aionjre_home*/tckconfiguration/tck.conf file to *jsr94-1.0_home*/lib folder and go to the next step.

   Alternatively, one can edit it to have a tck.conf file with the configuration for CA Technologies's implementation:

```
<tck-configuration>
    <rule-service-provider>

com.ca.cleverpath.aion.jsr94.RuleServiceProviderImpl</rule-service-provider>

<rule-service-provider-jar-url>file:lib/aionjre.jar</rule-service-provider-jar-ur
l>
    <rule-execution-set-location>./lib</rule-execution-set-location>

<test-factory>com.ca.cleverpath.aion.jsr94.tck.util.TestFactory</test-factory>
    <rule-execution-set-uri>file:lib/tck_res_1.xml</rule-execution-set-uri>
</tck-configuration>
```

6. Back up the original TCK Ant script file, *jsr94-1.0_home*/run_tck.xml.

   Copy the *aionjre_home*/tckconfiguration/run_tck.xml file to *jsr94-1.0_home* folder and go to the next step.

   Alternatively, one can edit it to have a run_tck.xml file with the classpath updated for the tck.run.tests target, reflecting all the jar files needed for CA's implementation.

   Add the following jar files from *jsr94-1.0_home*/lib folder to the classpath:

   - aionjre.jar
   - aion-jsr94-tck.jar
   - log4j.jar

   An example of the entries to be added to the classpath for the two junit tasks in the tck.run.tests target is as follows:

```
<pathelement location="${lib.dir}/aionjre.jar"/>
<pathelement location="${lib.dir}/aion-jsr94-tck.jar"/>
<pathelement location="${lib.dir}/log4j.jar"/>
```

7. Run the Ant script file run_tck.xml from the *jsr94-1.0_home* folder. A sample command on Windows resembles the following:

   ```
   ant\bin\ant -f run_tck.xml
   ```

You can find a report of all the tests run by the TCK in the folder, *jsr94-1.0_home*/reports.

In addition to TCK 1.0, CA Technologies has confirmed that CA Rule Engine passes TCK 1.0.1. The steps above can also be used to verify compliance to TCK 1.0.1.

Index