IBM Cúram Social Program Management

Cúram Rules Codification Guide

Version 6.0.4

Note

Before using this information and the product it supports, read the information in Notices at the back of this guide.

This edition applies to version 6.0.4 of IBM Cúram Social Program Management and all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Chapter 1

Introduction

1.1 Introduction

This guide describes the process of coding the rules (eligibility and entitlements for Social Welfare products) into a Cúram server application. This is an involved aspect of Cúram server development that uses tools and data structures provided within Cúram.

The following two companion guides are available:

- *Cúram Rules Definition Guide*
- *Cúram Rules Editor Guide*

The *Cúram Rules Definition Guide*, describes the procedures involved in defining the rules based on the relevant legislation.

The *Cúram Rules Editor Guide*, takes these rules as a starting point and describes the Cúram Rules Editor and the process of entering these rules into the Cúram server application.

1.2 Prerequisites

To successfully design and add rules to a Cúram server application, the reader will need the following:

- A basic understanding of social welfare systems and products;
- A basic understanding of Cúram and Cúram server application development;
- A basic understanding of data storage architecture and design;
- An adequate understanding of Unified Modeling Language (UML) for working with the Cúram application meta-model;

This guide assumes the above levels of competence as a starting point. Other documents in the Cúram Server Development Environment Documentation Suite are provided to introduce the developer to the different areas mentioned above.

There may be situations where a developer is charged with a specific subtask of the rules development and so requires expertise in an individual area only.

1.3 Audience

This document should be read by anybody who will be developing, adding or editing the rules of a Cúram server application model.

Chapter 2

Overview of Rules Codification

2.1 Introduction

This chapter provides a brief overview of the main elements involved in rules codification. It describes where and how the implementation of rules fits into application development. It also identifies the tools required for rules codification.

2.2 Rules

Rules are simple conditions that evaluate data and return a `true` or `false` result, indicating whether or not the rule was successful. Rules are extracted from relevant legislation and other criteria such as implicit conditions (i.e., the client is alive) or previous eligibility decisions for a product. Note that rules extraction is conducted by business analysts during the rules definition process (see the `Curam Rules Definition Guide` for more information).

2.3 Evidence (Data Items)

Evidence is the data that rules evaluate during rules execution. The evidence required for a particular rules execution is identified during the rules definition process (see the `Curam Rules Definition Guide` for more information).

A piece of evidence that is used by the rules during rules execution is referred to as a data item. During rules codification, the developer must model the relationship between the rules and the data items that the rules will evaluate.

2.4 Rules Data Objects (RDOs)

Rules Data Objects (RDOs) are UML stereotypes. Related data items are grouped together as attributes within RDOs.

The attributes for most RDOs are modeled according to the business logic of social welfare categories, i.e., case details, demographic details, contribution details, means details, dependencies details, and employment details. The attributes for these RDOs are populated and read by the rules engine during rules processing. Note that as part of rules codification, the developer needs to model RDOs to contain the necessary data items.

Rules engine provides an in-built RDO called Globals RDO. It's not required to model the Globals RDO. Data items of Globals RDO are accessible to all the eligibility rule sets and it usually acts as a key in the loaders to load the required evidence. These data items of this RDO are pre-initialized and passed into the Rules Engine interface methods rather than loading them using the loaders.

2.4.1 List RDOs

A List RDO is a special kind of RDO which can be seen as an array of RDOs. For example, representing the demographic data for a person and his or her children would require one RDO to represent the data for the person and one List RDO to represent the data for all his or her children.

2.4.2 Pre-initialized RDOs

RDOs can be pre-initialized by setting the values of their data items and passing them to the Rules Engine interface methods for rules execution. RDOs provide setter methods for each of their data items to set the value without the need of loading the value from the loader.

2.5 Loaders

Loaders are purpose-written Java classes used to populate RDOs with data (this data is usually read from the database or other RDOs). Whereas RDOs organize data according to the business logic of social welfare categories, loaders organize data according to data storage criteria. The design for a loader must therefore consider the underlying structure of the data which will be required for the implementation of the rules. Loaders will have a direct correspondence to this structure. However, because RDOs are structured according to business logic, there will not be a one-to-one correspondence between loaders and RDOs.

Pre-initialized RDOs do not require loaders, as they are already populated.

2.6 Data flow within the Curam Server Application

One of the main strengths of Curam is its ability to offer clients information clearly while processing the necessary information in the background. Load-

ers, BPOs and rules achieve this by translating the client's business logic (RDOs) into a form that can be processed and stored in the database. This results in two different views of the data.

An advantage of maintaining two views of the data is that the abstraction created between the different architectural layers allows changes to be made to one layer without affecting another level. For example, new database structures can be implemented without RDOs having to be changed.

2.7 Development Steps

The following list describes each of development steps for rules codification:

1. Gather information for system design. This is performed by business analysts and is described in more detail in the `Curam Rules Definition Guide`.
2. Open the Curam Reference Model in in Rational Software Architect (RSA).
3. Add custom entities and processes to the Model. Business Process Objects (BPOs) are one of the fundamental building blocks of the Curam server application. This process is described in more detail in the `Curam Modeling Reference Guide` and the `Curam Server Developer's Guide`.
4. Extract rules from relevant legislation and other criteria. This is performed by business analysts and is described in more detail in the `Curam Rules Definition Guide`.
5. Design and add RDOs to the Curam application model within RSA. The process for designing loaders is outlined in Chapter 4, *Loaders*.
6. Write loaders in Java for all RDOs that require them. The process for writing loaders is outlined in Chapter 4, *Loaders*.
7. Create rule sets using the Curam Rules Editor. This process is described in more detail in the `Curam Rules Editor Guide`.
8. Generate server code by running the Curam Generator. For more information, see the `Curam Modeling Reference Guide`.
9. Write the java code for the modeled BPOs. For more information, see the `Curam Server Developer's Guide`.
10. Compile the server application. For more information, see the relevant server deployment guide, e.g., `Curam Deployment Guide for WebSphere Application Server`.
11. Invoke and test the new application, checking the implementation of the rules. Rules testing is described in more detail in Chapter 6, *Debugging the Rules*.

12. Adjust the model and/or code and rebuild as required.

2.8 Cúram Server Application Development Tools

The Curam Server Development Environment (SDEJ) consists of several tools that are used for different aspects and stages of application development:

Rational Software Architect (RSA)

Curam makes use of RSA primarily as a tool for UML modeling, analysis, and design.

The Rules Editor

This is a part of the Curam application that allows rules definitions to be entered. It is described in more detail in the Cúram Rules Editor Guide.

Chapter 3

Rules Data Objects (RDOs)

3.1 Introduction

This chapter describes the process for designing RDOs and adding them to a Cúram server application.

3.2 Designing RDOs

RDO design involves two main steps. The first of these is the identification of the data items (evidence) that are required during a rules execution. The data items identified must be modelled into the attributes for the RDOs. The second step is the grouping of attributes according to the business logic of social welfare categories.

3.2.1 Identifying Attributes

Identifying the data items that are required is a difficult process; the choice of attributes, their datatypes and what they are called draws from both social welfare business and Cúram development experience.

For example, an unemployment benefit product will have a requirement that the claimant is alive. This can be represented as either of the following:

- DeceaseDate [type date]
- AliveOrDead [type indicator (i.e., Boolean)]

The decision made has a bearing on:

- the underlying database tables
- the selection coding required in the BPO
- the definition of the struct used to pass the data between objects, and

- the information that the client both inputs and receives as output.

Worked Example

The example below outlines a subset of the list of criteria (rules) required for a typical unemployment benefit product:

- Must be alive;
- Must be capable of work;
- Must be available for work;
- ...
- The case must not be closed;
- ...
- Must have made at least 39 contributions;
- ...

From these criteria, we could identify the following attributes (evidence/data items) for the RDO:

- DeceaseDate [type date];
- CapableOfWorkIndicator [type Boolean];
- AvailableForWorkIndicator [type Boolean];
- AvailableForWorkEvidence [type text];
- ...
- CaseExistsIndicator [type Boolean];
- CaseCloseDate [type date];
- ...
- NoOfContributions [type integer];
- ...

3.2.2 Grouping Attributes

The list of attributes identified must be broken down into groups according to the business logic of social welfare categories. Typically, these groupings include case details, demographic details, contribution details, means details, dependencies details, and employment details. These social welfare categories form the basis of the RDOs upon which rules will operate. The way the data is organized at this level ultimately determines the data that the client will enter and receive as output from the Cúram application.

Note that it is possible to maintain constants in an RDO at this level. These will be used during rules execution.

3.3 Adding RDOs to a Cúram Application Model

RDOs are added to a Cúram application model using RSA. This topic is covered in the *Working with the Cúram Model in Rational Software Architect*.

3.3.1 Adding Data Items to a RDO

Data items are added to RDOs in the same way as adding attributes to a struct or entity. Use the context menu on the RDO element in the project explorer of RSA to add the required data items.

If a loader is required for this dataitem please specify the name of the loader in the dataitem property panel (see Chapter 4, *Loaders*).

3.4 Attributes (data items) Documentation

Each attribute (data item) within an RDO has a documentation field, which represents a naturalistic language phrase describing what the attribute represents. This can be added to an attribute (data item) by following the steps below

1. Select a RDO or List RDO in the model and expand it to view the data items.
2. Select the data item to which you want to add the documentation.
3. In the properties (Properties can be viewed either by right clicking the data item and selecting Properties or in the Properties view) of the data item select the "Documentation" tab to see the text area where the documentation can be added.
4. Save the model after adding the documentation.

3.5 Adding List RDOs to a Cúram Application Model

To add a List RDO to a Cúram application model, you have to follow the same procedure as the one described for adding RDOs in Section 3.3, *Adding RDOs to a Cúram Application Model*.

Additionally, you must define one Description Item for a List RDO. Description Item is a attribute of a List RDO. Its stereotype is set to “description”.

3.5.1 Defining Description Items

There must be exactly one Description Item on a List RDO. Rules Engine and Loaders uses it to uniquely identify a record in the List RDO.

In order to define the Description Item you have to follow the procedure described below.

1. Select a List RDO in the model and expand it to view the data items.
2. Select the data item you want to define as description. window.
3. In the properties (Properties can be viewed either by right clicking the data item and selecting Properties or in the Properties view) of the data item select the "Stereotypes" tab to see the property "Description". Set this to true.
4. Save the model.

3.6 Immutable RDOs

An immutable RDO is an RDO whose values cannot be changed after they have been set.

An RDO can optionally be made immutable by adding the "final" attribute to the RDO declaration in the rule set. e.g.

```
<DataAccess name="rdoName" final="true">
```

If the value of an RDO attribute is changed after it has been set then the rules engine will throw an exception.

3.7 Qualified RDOs

An RDO declared in a sub-rule set can optionally take a qualifier to restrict it's scope within the entire rule set. An RDO can be qualified by adding the "qualifier" attribute to the RDO declaration in the sub-rule set. e.g.

```
<DataAccess name="rdoName" qualifier="qualifier">
```

The following qualifiers can be used:

- local
- input
- output

3.7.1 Local RDO

The scope of a local RDO is the sub-rule set; the values of a local RDO are not visible outside of the sub-rule set in which the RDO is declared.

3.7.2 Input RDO

An input RDO is passed to the sub-rule set from the immediate outer rule set/sub-rule set. The scope of the input RDO is the rule set/sub-rule set in which it is declared plus any sub-rule set in which it is passed as an input parameter.

3.7.3 Output RDO

An output RDO is passed from the sub-rule set to the immediate outer rule set/sub-rule set. The scope of an output RDO is the sub-rule set where it is declared plus the outer rule set/sub-rule set.

Chapter 4

Loaders

4.1 Introduction

This chapter describes the process for designing loaders and adding them to a Curam server application.

4.2 Designing Loaders

As described in Section 2.6, *Data flow within the Curam Server Application*, the Curam server application takes two different views of the data. Whereas RDOs organize data according to the business logic of social welfare categories, loaders organize data according to data storage criteria. Good loader design therefore requires an understanding of how both the RDOs and the database are structured. With this understanding, the developer can make informed decisions about how to best design loaders to populate RDOs with data.

4.2.1 Data Categorization

An important feature of loader design is data categorization. This is the separation into different loaders of different types of information based on the data storage structure.

In the event that a single category of data is unavailable (e.g., Dependents' information), the loader that retrieves this data will fail. This means that the loader will not deliver this category of data to the RDO and will also fail to deliver any other data. However, if different data retrieval tasks are implemented using separate loaders, the RDO can still receive other data that is present. Thus, the careful design of which loaders retrieve which information has a fundamental bearing on the ability of Curam to successfully manipulate product information.

This also has an impact on the quality of output information received by the

client. When a claim is processed by the client, Curam outputs information according to what criteria failed and why; careful loader design ensures that as much accurate information as possible can be provided to the client.

For example, an unemployment benefit claimant may have personal details in the database, but no payment history details. Separate loaders would be used to populate the RDO. One loader would succeed in retrieving data (personal details), while the other would fail (payment history details). If all the data retrieval for the RDO was implemented in a single loader, the whole retrieval operation would fail, as would the ability of the final Curam server application to process this particular claim. Additionally, the client would not be able to view the personal details for this claim even though the information would be available in the database.

4.2.2 Loader Invocation

Loader invocation has to be kept in mind when designing the loaders. If a loader is invoked at the wrong moment, it will overwrite data that was already loaded using other means, e.g., using data item assignments in a rule set. In a data item assignment, a boolean indicator attribute ("Load Target") instructs whether the data item should be loaded before execution or not. By default, this indicator is set to true and, if this is so, the loader will be invoked prior to executing the data item assignment.

When a rule tries to access a value of an RDO attribute, the attribute first checks if its value has already been loaded. If the value has not been loaded yet, the attribute will call the loader. The loader will load the values of all the attributes to which it was assigned, thus possibly overwriting the values which were already set for other attributes.

4.2.3 Loaders and BPOs

Loaders usually have one-to-one correspondences with BPOs. BPOs handle data retrieval from the database. As part of the loader implementation process, BPOs must be added to the Curam application model and the selection code must be hand-crafted. For this reason, the design and implementation of loaders and BPOs go hand-in-hand. For more information on BPOs in Curam Modeling Reference Guide and Curam Server Developer's Guide.

Important

While the majority of loaders are paired with BPOs to retrieve information from the database, loaders are simply Java classes and can retrieve information from a range of other sources such as other RDOs.

4.2.4 Implementing a Loader and its BPO

The sequence of steps required for adding a loader/BPO pair can be sum-

marized as follows:

1. Define the loader in the Curam application model.
2. Define structs for passing information based on the data required.
3. Create a BPO with required operations in the Curam application model.
4. Create the appropriate structs in the model.
5. Add attributes to the structs.
6. Add the structs to the operations in the BPO.
7. Write a specification document for the BPO and processing required.
8. Run the Curam Generator to create the shell of the BPO.
9. Create loader by extending class `curam.util.rules.Loader`
10. In the loader:
    a. Obtain an instance of the RDO to be used in the loader.
    b. Initialize any data required by the BPO.
    c. Call the Appropriate method on the BPO using the initialized structs.
    d. Implement the population of the RDO with the data returned from the BPO.
11. Implement the selection code in the BPO.
12. Build the server application.
13. Test the server application and the implementation of the rules.

These steps are described in more detail in the following sections.

4.2.5 Defining the Loader in the Curam Application Model

A Loader is added to a Curam application model in RSA by means of the Curam Palette or the project explorer context menu. Ensure the name of the loader is identical to the loader specified for the RDO or RDO dataitem. The loader can be added by following the steps below:

1. Select any package in the model where the loader needs to added.
2. Right click on it, select "Add Class" in the menu and Select "Loader".
3. Name the loader by ensuring that the name of the loader is identical to the loader specified for the RDO or RDO dataitem.
4. Save the model after adding the loader.

4.2.6 Defining the Structs Based on the Data Required

This topic is covered in the *Curam Modeling Reference Guide*.

4.2.7 Creating a BPO in the Curam Application Model

This topic is covered in the *Curam Modeling Reference Guide*.

4.2.8 Implementing BPOs

BPOs are used to retrieve data from external data sources such as databases. In some cases, loaders do not require BPOs. For example loaders without an associated BPO can be used to load constants to RDOs. The procedure of designing and implementing BPOs is described in the *Curam Modeling Reference Guide* and the *Curam Server Developer's Guide*.

4.2.9 Creating the Loader

Loaders are hand-crafted but follow a standard structure and are straightforward to code. Writing a loader typically involves creating the subclass of `curam.util.rules.Loader`, then implementing the method `load(curam.util.rules.RulesParameters dtls)` by adding the selection logic which will be passed to the BPOs for the actual database access.

Loaders that are used to retrieve data from the database must have a corresponding BPO. This is to provide a process in the Curam application model that can access the DAL objects and implement the actual selection and retrieval of the data required by the loader. Data is passed between the loader and the BPO via purpose-defined structs. These structs must be defined in the model using the Class Wizard on the Merlin Toolbar.

When a BPO, its operations, and the relevant structs have been added to the Curam application model, the Curam Generator will generate a skeleton for the BPO, which subsequently requires hand-crafted code for selection operations.

The Curam Generator does not generate skeletons for the loader files. In order to implement a loader you have to extend the class `curam.util.rules.Loader`.

The following method has to be implemented for the new loader:

- `protected void load(RulesParameters dtls) throws AppException, InformationalException`

The new loader has to be placed in the package, `<applicationname>.rules.loaders`

A sample loader skeleton is detailed below.

```
package testapp.rules.loaders;

import curam.util.rules.ItemGroupGlobals;
import curam.util.exception.AppException;
import curam.util.exception.InformationalException;
import curam.util.rules.Loader;
import curam.util.rules.RulesParameters;

public class SampleLoader extends Loader {

  protected void load(RulesParameters dtls)
    throws AppException, InformationalException {
  }
}
```

Example 4.1 Loader Skeleton

A class stereotype 'loader' is also supported in the model, and loaders specified by rules data items must exist as classes in the model. The reasons for this are:

1. Loaders specified by rules data items can be validated at generation time rather than run time.
2. Java interfaces can be generated for loaders.
3. Developers can model inheritance between loaders.

Loaders support a restricted form of subclassing in the UML model which allows the developer to model a replacement for a Loader class. i.e. all Loader classes in the model behave as if their 'Replace Superclass' has been set - although this option is not available to the developer on these classes.

For more information on the 'Replace Superclass' option see the Curam Modeling Reference Guide.

4.2.10 Adding Hand-crafted Code to the Loader Body

This generally consists of three tasks, namely:

1. Initializing any data required by the BPO,
2. Calling the BPO,
3. Populating the RDO with the data returned from the BPO.

To see how these tasks are implemented, see the sample below.

```
package server.testapp.rules.loaders;

import curam.util.rules.ItemGroupGlobals;
import curam.util.exception.AppException;
import curam.util.exception.InformationalException;
import curam.util.rules.Loader;
import curam.util.rules.RulesParameters;
import sampleapp.rules.rdo.DemographicDataRDOGroup;
import sampleapp.struct.*;

public class SampleLoader extends Loader {
```

```
  protected void load(RulesParameters dtls)
    throws AppException, InformationalException {
    ItemGroupGlobals globals =
      ItemGroupGlobals.getCurrentInstance(dtls);
    DemographicDataRDOGroup rdo =
      DemographicDataRDOGroup.getCurrentInstance(dtls);

    // Initializing the data required by the BPO.
    MaintainPersonKey maintainPersonKey =
      new MaintainPersonKey();
    PersonDetails personDetails = null;

    maintainPersonKey.personID =
      globals.getPersonReferenceNumber().getValue(dtls);

    // Calling the BPO.
    sampleapp.intf.MaintainPerson
      maintainPersonObj =
        sampleapp.fact.MaintainPerson.newInstance();

    personDetails =
      maintainPersonObj.readDetails(maintainPersonKey);

    // Populating the RDO with the data returned from the BPO.
    rdo.getName().setValue(personDetails.fullName);
  }
}
```

Example 4.2 Sample Loader Implementation

Implementation of list group loaders is almost similar to the above implementation except that the startLoader() and endLoader() methods are to be called at the begining and end of the list RDO processing respectively. It has to be noted that for every startLoader() method call there should be an endLoader() method call immediately after the list RDO processing is completed. Every time when startLoader() method is called it saves the current position of the list RDO and endLoader() restores it, so that the rule set execution is not affected by the list RDO processing.

```
package server.testapp.rules.loaders;

      import curam.util.rules.ItemGroupGlobals;
      import curam.util.exception.AppException;
      import curam.util.exception.InformationalException;
      import curam.util.rules.Loader;
      import curam.util.rules.RulesParameters;
      import sampleapp.rules.rdo.HouseHoldDetailsGroup;
      import sampleapp.intf.HouseEligibilty;
      import sampleapp.intf.EligibilityFactory
      import sampleapp.struct.*;

      public class SampleListLoader extends Loader {

        protected void load(RulesParameters dtls)
          throws AppException, InformationalException {

          ItemGroupGlobals globals =
            ItemGroupGlobals.getCurrentInstance(dtls);

          HouseEligibilty eligibilityObj =
            EligibilityFactory.newInstance();

          HholdDetailsList readHholdDetailsList =
            eligibilityObj.readHholdDetails(
              globals.getPersonReferenceNumber().getValue(dtls));
```

```
        HouseHoldDetailsGroup houseListRDO =
          HouseHoldDetailsGroup.getCurrentInstance(dtls);

        /* startLoader() is called to save the current
           position of the houseListRDO.
         */
        houseListRDO.startLoader();
        for (int i = 0;
           i < readHholdDetailsList.hholdEvid.size();
              i++) {

          houseListRDO.insertIfRequired(i);
          houseListRDO.setCurrentPos(i);

          houseListRDO.current().getName()
            .setValue(
               readHholdDetailsList.hholdEvid.item(i).name);

        }

        /* endLoader() is called to restores the houseListRDO
           position, so that the rule set execution is not
           affected by the list RDO processing.
         */
        houseListRDO.endLoader();
      }
    }
```

Example 4.3 Sample List Group Loader Implementation

4.2.11 Extending Loaders

It is possible to extend Loaders in the Curam application model. This feature allows the sub-classing of loaders. The Rules Engine picks up the new sub class loader rather than the original super class loader.

Loaders support a restricted form of sub-classing in the Curam application model, which allows the developer to model a replacement for a loader class. To do this create a new loader in a custom component to subclass the appropriate loader. All Loader classes in the model behave as if their 'Replace Superclass' has been set although this option is not available to the developer.

For example in the Curam application model, if a loader class `LoaderB` extends `LoaderA` and a data item references `LoaderA` then `LoaderB` will be invoked instead of `LoaderA`.

For more information on the 'Replace Superclass' option see the *Curam Modeling Reference Guide*.

4.2.12 Building the Application

When the loader(s) and BPO(s) have been completed, as described above, the application can be built.

A description of how to do this can be found in the *Curam Server Developer's Guide*.

4.2.13 Testing the Implementation of Rules

For a description of the testing and debugging processes, see Chapter 6, *Debugging the Rules*.

Chapter 5

Rule Set Best Practices

5.1 Introduction

Between packages, objective groups, rule groups and rules themselves, and the multi-level rules structure, it is possible to develop quite substantial rule sets, and, in the process, be faced with choices as regarding how best to structure them.

This chapter is designed to highlight several of the key areas where attention at design time can sometimes greatly enhance optimum rule set development.

5.2 What is an Optimally Designed Rule Set

When developing rule sets, there are two criteria which the designer seeks to maximize:

1. End-user clarity of information, and
2. System efficiency.

In some cases, enhancing one of these has a cost in terms of the other. Readability and accuracy of information for the client is most important. For the sake of clarity and explanatory feedback, some techniques referred to below actually increase processing required beyond simply establishing eligibility.

5.3 Highest Value Rate Groups

If a client is familiar with a criterion “at least 39 payments must have been made”, then it is good practice to show this explicitly in the output for the processing for this claim, even if the claimant has made, say, over 50 payments. There may be several rates coded, e.g., “<10 payments”, “10-20 payments”, “<39 payments”, “>=39 payments”, “>50 payments”, and if the

highest value has been chosen, then processing will stop once the “>50 payments” condition has been met. For the purposes of full and useful client feedback information, it is preferable to select the “all objectives” option, so that the client can see the familiar “>=39 payments” rule succeed.

5.4 Informational Rules

Generally, rules are implemented purely as succeed/fail tests against the criteria for eligibility for a product. Under certain circumstances, however, rules are implemented that will always evaluate to `true`, purely to enhance the output information for the client.

For example, in an unemployment benefit product there may be a product-level rule group with a “>50 contributions” rule that succeeds. There may also be an informational rule of “>35 contributions” that also gets evaluated, even though the “>50 contributions” rule has succeeded, purely so that the user can see the success output of “>35 contributions”, that may be useful or clarifying for the client.

5.5 Shared Sub-Rule Sets

Sub-Rule Sets are a re-usable subset of rules, i.e., where one copy of the sub-rule set is called, or linked to, from more than one place in the main rule set. Sub-rule sets replace an earlier, more laborious, process of copying and pasting rule sets.

Sub-rule sets are used in cases where rule structures are similar across different products. For example, the repayment mechanisms for different types of loan products. There is virtually zero runtime overhead for utilizing sub-rule sets, while careful use of sub-rule sets during design time can increase clarity, speed of development, and reliability.

Usage of sub-rule sets can be handy when developing large rule sets. It allows multiple developers to work on separate parts of the rule set without interfering with each other. This can also make source control management easier.

Chapter 6

Debugging the Rules

6.1 Introduction

Once the RDOs, loaders and rule sets have been designed and added to the Cúram application, the Cúram application must be tested to check the implementation of the rules.

The user should be able to enter any possible claim eventuality and observe the appropriate response from Cúram when compared to the raw legislation. In practice, unfortunately, this scenario is very rare. There is typically an involved process of testing, debugging, modifying, rebuilding, and re-testing.

This chapter describes process of testing and debugging of rules implementation.

In order to do preliminary testing of rules you can use the Rules Simulation Environment. For more information see the `Cúram Rules Editor Guide`.

6.2 Typical Testing Phases

There are several main areas where problems can occur, with the ultimate result that Cúram fails to process (consistently or accurately) claims according to the rules. These areas are:

- Errors in the criteria list (the list from which the rules are coded);
- Errors in the Cúram application model (RDOs, other entities);
- Errors in the rule sets;
- Building/Compiling errors (syntax errors).

Assuming that the developer has succeeded in getting the application to build and compile, subsequent testing is primarily aimed at identifying

design or implementation errors in:

- RDOs;
- Loaders;
- BPOs for the loaders;
- Rule set definition.

6.3 Rules Engine Outputs

When applying rules while processing a claim, the output of the Rules Engine is:

- Decision;
- Evidence information;
- Decision information.

Both of these are displayed to the user in the client application as a result of entering a claim for processing.

6.3.1 Decision

Rules Engine outputs a decision as boolean value to define whether a claim succeeded or failed.

6.3.2 Evidence Information

An evidence information consists of list of all attributes of RDOs which were used during rules execution, with their values. The datatype and description/comment fields are also returned.

Rules Engine generates the evidence information in a compressed format and returns it in the form of a String. This compressed result can be decompressed by the Evidence text decoder to a presentable format suitable for display or other purpose.

6.3.3 Decision Information

Every rule that is defined in Cúram has to have a success text and a failure text, that are entered at the point of rule definition in the Rules Editor (see the *Cúram Rules Editor Guide*). This is designed to provide a natural language result for each claim.

Rules Engine generates the rules decision information in a compressed format of bits (Binary digits) and returns it in the form of two byte arrays. Compressed result comprises of rules execution flow bits and status (success/failure) bits. This compressed result can be decompressed by the

Rules decoder to a presentable format suitable for display or other purpose. This enables a user to know specifically which rules have succeeded and which rules have failed. This can often form the basis for a simple modification for another claim attempt, e.g., entering all required demographic details.

6.4 Runtime Rules Logging

The rules system allows runtime logging to be enabled on demand. This is based on a property `curam.trace.rules`. This allows logging to be enabled as required, without the constant overhead of unnecessary logging.

Note

A change to the rules logging property does not affect the servers that are already running.

6.4.1 How Logging Works

The rules logging works by capturing the details at all the core points inside the rules engine to provide comprehensive information on the operation of the rules and a clear picture of how a given decision is reached. It is envisaged that this functionality will only be used in the development environment to analyze any issues that arise during execution.

6.4.2 What Gets Logged?

The logging is placed in a large number of areas inside the Rules Engine including:

- **Loader.** Each piece of evidence is associated with a loader, and a loader may load several pieces of evidence. Every call to the loader during Rules Engine execution can be logged.
- **Data Item.** Start of data item value retrieval, end of data item value retrieval including the value retrieved and setting a value for a data item are logged.
- **Rule Item.** Start of execution of any rule item, end of execution of any rule item along with the name, value, condition, formula (if available), and result. Rule item can be any one of the rule, rule group, rule list group, objective group, objective list group, objective tag, sub-rule set, sub-rule set link, product and data item assignment are logged.
- **Rule Set.** Start of execution of a rule set and end of rule set execution are logged.

6.4.3 Designing for Logging

The major component of the logging refers to the name of the components

involved and the their values in the case evidence. With this in mind the names of components should be meaningful in their own right, even when stripped of the associated failure and success texts. This also is a very good reason to ensure that component names are unique inside a rule set.

Chapter 7

Advanced Topics

7.1 Introduction

This chapter addresses a selection of more advanced topics, including

- Performance
- Rules Results Decompression
- Execution Modes
- Custom Functions
- Mathematical Operations
- Missing evidence
- Multi Language Support
- Error reporting

7.2 Performance

The Cúram Rules Engine is a high performance rules engine capable of evaluating thousands of rules per second. However to maximize the capabilities of the Rules Engine there are several considerations that must be evaluated when designing the rule set and associated RDOs. These come under the following headings.

- Keep it Simple.
- Separate Business Rules from Business Logic.
- Rules Data Retrieval.
- Rule Execution Optimization.

7.2.1 Keep it Simple

By focusing on design and keeping the overall rule set as simple as possible you should benefit not only from rules which are easier to maintain and easier to read but also this will typically yield better performance than unnecessarily complex rule sets.

7.2.2 Separate Business Rules from Business Logic

The Cúram rules engine is designed to support business and legislative rules which are typically volatile. The surrounding business logic which is more static, is better suited to BPOs, which are designed to handle this type of processing. Coding business logic in rules typically leads to more complex rule sets and unnecessary rule executions

7.2.3 Rules Data Retrieval

As retrieving the rules data is significantly slower than rule execution it is essential that this process is as efficient as possible. The rules data retrieval processing should minimize database or legacy application access. One possible way to achieve this could be to load related evidence data into a single BPO allowing related RDOs to be populated from it.

While RDOs are cached for the duration of a rule set execution, they are not cached across rule set executions. So caching should be considered in the BPOs to ensure that for subsequent rule set executions, this data can be retrieved from the cache. Cúram v6.0 includes a Transaction SQL Cache, where identical database queries in the same transaction are cached for the duration of the transaction; if this is disabled or if there is significant processing on the data after the read, application level caches should be considered to improve performance.

7.2.4 Rule Execution Optimization

The performance of a rule set can be strongly affected by ensuring that rules are executed in an efficient and prioritized fashion. The number of rules executed also directly influences decisions. Unnecessarily large decisions are more difficult to interpret and will take longer to display as there is more processing involved on both the server and the client. Effective use of the Summary Item option and clear success and failure text, will increase the probability the user will get the information from the summary design view and thus avoid viewing the full decision.

Prioritizing and Ordering of Rules

The ordering of the rules and the design of the conditions applied can influence the number of rules executed. It is important that the rules which are most likely to fail are placed before those that are less likely to fail. Remem-

ber that the rule set is a tree and placing the rules most likely to fail closer to the top of the branch will help to avoid the rules contained in the remainder of the branch.

Avoid Repeated Rules Executions

Rule sets can often have repeating blocks of rules. One technique to minimize the number of rules executions in an instance, is to run the rules once and set a dataitem to hold the outcome. This dataitem can be managed and used through the ruleset.

A general design aim is to reduce the number of repeated rules in the rule set to ensure the most efficient execution of the rules. This may result in the introduction of rule groups at levels in the rule tree where they would not logically be located. However, the benefits of the introduction of such rule groups should be weighed against the extra difficulty for the user in deciphering the details of the conditions being applied, as well as the potential for confusion in the reporting of the decision reasons.

Use Rule List Groups Wisely

Rule list groups can be nested inside other rule list groups and objective list groups. Objective list groups can also be nested inside other objective list groups. However it is important to remember that list groups are effectively loops, where the rules in the group will be executed for each element in the list. So if list groups are nested then the inner list group will be executed for each iteration of the outer list group. Complex nesting of rule list groups can make it difficult for a case worker to navigate through as well as create potential performance concerns.

To alleviate these concerns you should consider the following when designing nested rule list groups.

- Keep the number of nested rule list groups to a minimum.
- Order the rules so that the rules exit as soon as possible.
- Minimize the number of elements in the list RDO so that only the elements required for the list group are loaded.
- Pay attention to the order of the elements in the list. Sorting the elements in the list RDO can help to exit from list group in certain circumstances.

Top Level Rule Group

This group is important, as it is usually added and executed before any objectives are evaluated and hence can serve several important purposes. In fact, if this group fails then execution is halted immediately, which can prevent errors occurring at lower levels in the rules by ensuring that the required evidence exists.

The top level rule group can also be used to ensure that cases that cannot

qualify for the product delivery are caught quickly, and no more processing then necessary is done. However, it should also be noted that the benefit of early detection of failing cases may be outweighed by the repetition of rules through the rule set at lower levels of the tree where, for example, the failing of a case according to a rule may be preferred in order to give the user the failure message. The benefits of each rule in the top level rule group should be carefully assessed in these terms.

Using the correct Execution Mode

The correct execution mode is an important factor when trying to reduce the number of rules executed inside rule groups and rule list groups. The execution of rules inside a rule group or rule list group can stopped early by using the execution mode to stop when a successful result for the rule group or rule list group has been achieved. This can be used as long as it is not mandatory to run all the rules in the group. For rule groups you can use the "Stop On Result" execution mode and for rule list groups you can use either the "Succeed All Stop" or "Succeed One Stop" execution modes.

7.3 Rules Results Decompression

The Rules Engine generates rules decision information in a compressed format of bits (Binary digits) and returns it in the form of two byte arrays. Compressed results are comprised of rules execution flow bits and status (success/failure) bits. This compressed result can be decompressed by the `ResultTextDecoder` class to a presentable format suitable for display or other purposes. This enables a user to know specifically which rules have succeeded and which rules have failed for a given decision.

Evidence information is also returned in a compressed format. This information can be decompressed using the `EvidenceTextDecoder` class.

Note

In order to decompress decision or evidence information into a format suitable for processing, the rule set that was used for the construction of the decision must be provided to the decompression method.

This means that no modifications can be made to a rule set once it has been executed and the results of any such execution need to be referenced (for example, by displaying a decision record using the Cúram Client application). Any modifications to such a rule set must be carried out on a cloned copy of the rule set.

7.4 Execution Modes

An execution mode allows to alter the Rules Engine's runtime behavior. The execution mode allows a developer to use the same rule set under different

circumstances. For example some rules can be excluded when running in kReassessment or kQuotation mode.

7.4.1 Execution Mode kNormal

kNormal - default rules execution mode. In this mode all rules are executed, all loaders are invoked and execution fails if the value for a required data item cannot be loaded.

7.4.2 Execution Mode kQuotation

The special rules engine behavior in this mode is that the rules marked as excluded "From Simulation"* are not executed. It is expected that the evidence to be used will be passed in to the Rules Engine when running in this mode rather than relying on the loaders to load the data from the external source.

7.4.3 Execution Mode kReassessment

The rules marked as excluded "From Reassessment" will not be executed in this mode, otherwise it is exactly the same as kNormal.

7.4.4 Execution Mode kSimulation

kSimulation - only used by simulation environment. Special behavior for this mode is that the rules marked as excluded "From Simulation"* are not executed. The loaders are not invoked to determine the size of a list RDO.

7.5 ObjectiveTag Type

The behavior of ObjectiveTag value evaluation depends on the ObjectiveTag type. The type is an attribute of the rule set ObjectiveTag element which can be used to specify one of the following tag type codes from the RulesTagType codetable.

- RTT1 - Product delivery recommendation.
- RTT2 - Assessment recommendation.
- RTT3 - Money.
- RTT4 - Double.

The value of the ObjectiveTag with the type RTT3 or RTT4 is evaluated, but its result must be always a numerical value as it would be used to calculate the value of its parent Objective.

Apart from the above mentioned types in the RulesTagType codetable, the ObjectiveTag can also have the type code that starts with the word "EVAL" (For example: EVAL1, EVAL_A or EVAL). The value of the ObjectiveTag

with the type that starts with "EVAL" will be evaluated by the Rules engine and returned as a String but its value is not used for calculating the value of its parent Objective.

The following example shows a sample ObjectiveTag with the type as EVAL.

```
<Objective deductionallowable="true" description="sample"
  id="11" ratetarget="RC1" ratetype="PC1" recordid="11">
    <Label highlightonfailure="true">
        <RuleName>
            <Text locale="en_US" value="Sample Objective" />
        </RuleName>
        <SuccessText>
            <Text locale="en_US" value="Sample Objective
            Succeeded" />
        </SuccessText>
        <FailureText>
            <Text locale="en_US" value="Sample Objective
            Failed" />
        </FailureText>
    </Label>

    <ObjectiveTag id="2" type="EVAL_Sample" name="Sample"
    recordid="12"  value="SampleRDO.dataItem" />

 </Objective>
```

Example 7.1 Sample ObjectiveTag with EVAL type

All the ObjectiveTags having the types other than RTT3, RTT4 and the types that does not start with EVAL will not be evaluated. Their values will be returned after the execution in the same form as they were specified in the rule set.

7.6 Get Dynamic Rate

The Rules Engine provides a method `getDynamicRate` to calculate the rate value of a given Objective for a given date range. The application can determine the rate for each financial objective by calling this method.

7.6.1 The getDynamicRate API

The `getDynamicRate` method gets the parameters of rule set ID, objective ID, objective Name, date range and tag values. The tag values parameter is calculated during the initial eligibility check.

7.6.2 The Processing Performed By getDynamicRate

The following processing is done when the `getDynamicRate` method is invoked:

- The rule set is loaded and the objective is located in the rule set

- The objective tags related to the objective are identified.
- The values of the objective tags are summed in such a way, that the date range specified in parameters of the methods is covered by the smallest possible number of frequency patterns associated with the objective tags identified in the previous steps.
- The value is then returned from the method.

7.7 Built-in Functions

7.7.1 The not() Function

The `not()` function acts as a logical inversion operator. In the normal case this is applied to a Boolean value, for example:

```
not(ClientDeceasedInd)
```

When using this function, you should be aware of double inversion.

7.7.2 The isNothing() Function

The `isNothing()` function provides a core piece of functionality inside the rules environment. The behavior of the function is best explained by first describing what happens in the case where a piece of evidence which is unavailable from the database is accessed from the rules. In this situation, the database returns an error code and a “Data Item Missing” exception is generated inside the rules. Execution is stopped at this point and a message is stored with the decision noting the missing piece of evidence.

However, there are some circumstances when different behavior is required, for example where multiple pieces of evidence are missing it may be important to inform the client of them all. Another situation would be where the absence of a piece of evidence is in itself evidence. In these circumstances the `isNothing()` function can be applied to a piece of evidence. The function will first invoke the loader if the loader was not invoked yet and return a Boolean value of `true` if the evidence is not available from the database (or business process that it called).

Any data item can be passed as a parameter to this function. For example `isNothing(ClientRDO.clientAddress)`. It returns `true` if clientAddress is not set.

7.7.3 The IsZero() Function

The `IsZero()` function provides an easy type independent way to check if the data item's value is zero. It is possible to check if the value is zero for any numerical data item or date. The function will return `true` if the value is set to a zero or equivalent. It will return `false` otherwise.

7.7.4 The Date() Function

The `Date()` function provides a way to create instances of type curam.util.type.Date.

Three int values are passed in to represent each of day, month and year respectively, for example:

Date(31, 12, 2006)

7.7.5 The DateAddOne() Function

The `DateAddOne()` takes Date or DateTime as input parameter and returns DateTime after adding a day to the input Date or DateTime

7.7.6 The DateAdd() Function

The `DateAdd()` takes Date and an integer that denotes number of days as input parameters and returns Date after adding the given number of days to the input Date

7.7.7 The subDates() Function

The `subDates()` function is passed in two dates and returns the number of days difference between the two dates. This function will return a negative value if the second parameter is greater than the first. For example:

`subDates(SampleRDO.endDate, SampleRDO.startDate)` returns 3 if SampleRDO.endDate is 04 April 2008 and SampleRDO.startDate is 01 April 2008.

7.7.8 The ceiling() Function

The `ceiling()` function rounds the specified number up, and return the smallest number that is greater than or equal to the specified number. This function takes a double and an integer as input parameters and calculates the ceiling of the first value based on the precision of the second value. For example:

```
ceiling(23.1, 1) returns 24
```

7.7.9 The floor() Function

The `floor()` function rounds the specified number down, and returns the largest number that is less than or equal to the specified number.This function takes a double and an integer as input parameters and calculates the floor of the first value based on the precision of the second value. For example:

```
floor(23.5,1) returns 23
```

7.7.10 The round() Function

The `round()` function returns a number rounded to a specified number of decimal places.This function takes a double and an integer as input parameters and rounds the first value based on the precision of the second value. For example:

```
round(23.7,1) returns 24
```

7.8 Custom Functions

7.8.1 Custom functions in the Rules expressions

Rules expressions have access to a number of "built-in" functions which deal with the requirements of rules expressions. However to allow the user to address a greater number of issues, there is the possibility to expand the number of functions available to these expressions. Custom functions allow the user to add their own code to the Rules Parser and implement this functionality whenever an expression is evaluated.

In order to implement a new custom function a user must:

- Create the custom function class.
- Populate `CustomFunctionMetaData.xml` with the meta data of the function.

7.8.2 Writing the custom function

In order to create a custom function the user must extend `curam.util.rules.functor.CustomFunctor` . The new class name must also have the prefix `CustomFunction` and implement the method:

```
public Adaptor getAdaptorValue(final RulesParameters rulesParameters)
```

```
throws AppException, InformationalException
```

These implementations should be stored in the package: `<SERVER_DIR>/components/<COMPONENT_NAME>/source/curam/rules/functions` where COMPONENT_NAME is the name of the component in which you are working.

A `java.util.List` of passed parameters can then be retrieved by calling the `getParameters()` method. Each element in the List will be of type `curam.util.rules.functor.Adaptor` representing a parameter in the order that they were supplied. These can be cast from `java.lang.Object` as per the entry in the `CustomFunctionMetaData.xml` (described below).

The example below accepts an int or a String as its parameter and multiplies it by 5. e.g. multByFive("2")

```
import curam.util.exception.AppException;
...  // List of imports.
import curam.util.rules.RulesParameters;

public class CustomFunctionmultByFive extends CustomFunctor {

  public Adaptor getAdaptorValue(
    final RulesParameters rulesParameters)
    throws AppException, InformationalException {

    // Get the passed parameters list.
    final List parameters = getParameters();
    //get the adaptor representing the passed in value.
    final Adaptor inputAdaptor = (Adaptor) parameters.get(0);
    int inputValue;

    // Evaluate the adaptor.
    if (inputAdaptor instanceof IntegerAdaptor) {
       inputValue = ((IntegerAdaptor) inputAdaptor)
         .getIntegerValue(rulesParameters);
    } else if (inputAdaptor instanceof StringAdaptor) {
      final String stringValue = ((StringAdaptor) inputAdaptor)
        .getStringValue(rulesParameters);
      inputValue = Integer.getInteger(stringValue).intValue();
    }

    final int returnValue = inputValue * 5;
    // Create an IntegerAdaptor representing the returnValue.
    return AdaptorFactory.getIntegerAdaptor(
      new Integer(returnValue));
  }

}
```

Example 7.2 Example of Custom Function implementation

7.8.3 CustomFunctionMetaData.xml

In order to facilitate type checking of passed arguments an entry for the new function must be made within `CustomFunctionMetaData.xml`. This file can be found within the `rulesets/functions directory` of the component which is being used. If this file does not exist it must be created. For each permutation of the parameters that may be expected to be passed to the custom function an entry should be made within the parameters section of that function. In the example the function accepts either an `IntegerAdaptor` or a `StringAdaptor` and the entry should resemble the following.

```
<CustomFunctions>
...
  <CustomFunctor name=”CustomFunctionmultByFive”>
    <parameters>
      <parameter>
        curam.util.rules.functor.Adaptor$IntegerAdaptor
      <parameter>
    </parameters>
    <parameters>
      <parameter>
        curam.util.rules.functor.Adaptor$StringAdaptor
      <parameter>
```

```
    </parameters>
    <returns>
      curam.util.rules.functor.Adaptor$IntegerAdaptor
    </returns>
  </CustomFunctor>
...
</CustomFunctions>
```

Example 7.3 Sample CustomFunctionMetaData.xml

7.8.4 Adaptor Types

The parameter and return types available are:

- `curam.util.rules.functor.Adaptor$IntegerAdaptor` contains a value which will be evaluated to an int.
- `curam.util.rules.functor.Adaptor$DoubleAdaptor` contains a value which will be evaluated to a double.
- `curam.util.rules.functor.Adaptor$LongAdaptor` contains a value which will be evaluated to a long.
- `curam.util.rules.functor.Adaptor$BooleanAdaptor` contains a value which will be evaluated to a boolean.
- `curam.util.rules.functor.Adaptor$StringAdaptor` contains a value which will be evaluated to a String.
- `curam.util.rules.functor.Adaptor$DateAdaptor` contains a value which will be evaluated to a curam.util.type.Date.
- `curam.util.rules.functor.Adaptor$DateTimeAdaptor` contains a value which will be evaluated to a curam.util.type.DateTime.
- `curam.util.rules.functor.Adaptor$CharacterAdaptor` contains a value which will be evaluated to a char.
- `curam.util.rules.functor.Adaptor$MoneyAdaptor` contains a value which will be evaluated to a curam.util.type.Money.
- `curam.util.rules.functor.Adaptor` Base type of all adaptors. An entry of this type into parameters denotes that any type of adaptor would be accepted.

7.9 Mathematical Operations

The mathematical operations inside the rules can be used in the objective tags for compound rate evaluation.

7.9.1 Bracketing of Terms

The bracketing of terms can have a significant impact on the result of a cal-

culation. The behavior is as normal for mathematical operations, but the effects of brackets can be combined with operator precedence (see Section 7.9.2, *Operator Precedence*) and may add complexity to the situation. The basic approach is that any operation that should be carried out in advance of another operation should be bracketed, e.g., 5 * (3/4) = 3.75.

7.9.2 Operator Precedence

The precedence of operators is as defined for the Java programming language. These following operators are listed in order of precedence:

Operator	Associatively	Type
()	left to right	parentheses
* /	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equities

Table 7.1 Operator Precedence

This can have a significant impact on the results of calculations unless it is considered from the beginning.

7.9.3 Data Types and Supported Operations

The operations that are explicitly supported between the data types are detailed in the Table 7.2, *Data Types and Supported Operations*.

It is possible to perform operations between the data types not listed in the Table 7.2, *Data Types and Supported Operations* if the underlying data type of an attribute can be converted into one of the types for which an operation is supported.

For example, the addition of SVR_INT8 and SVR_MONEY is possible, because SVR_INT8 is converted into SVR_DOUBLE and the addition of SVR_DOUBLE and SVR_MONEY is supported.

It is possible to add or subtract integers from the dates. Integers represent the number of days to be added or subtracted.

The first parameter type	The second parameter type	Operations supported	Result type
SVR_STRING	SVR_STRING	==, !=	SVR_BOOLEAN
SVR_CHAR	SVR_CHAR	==, !=	SVR_BOOLEAN
SVR_MONEY	SVR_MONEY	==, !=, <, >, <=, >=	SVR_BOOLEAN

The first parameter type	The second parameter type	Operations supported	Result type
SVR_MONEY	SVR_DOUBLE	==, !=, <, >, <=, >=	SVR_BOOLEAN
SVR_DOUBLE	SVR_MONEY	==, !=, <, >, <=, >=	SVR_BOOLEAN
SVR_DOUBLE	SVR_DOUBLE	==, !=, <, >, <=, >=	SVR_BOOLEAN
SVR_DATE	SVR_DATE	==, !=, <, >, <=, >=	SVR_BOOLEAN
SVR_DATE	SVR_DATETIME	==, !=, <, >, <=, >=	SVR_BOOLEAN
SVR_DATETIME	SVR_DATETIME	==, !=, <, >, <=, >=	SVR_BOOLEAN
SVR_DATETIME	SVR_DATE	==, !=, <, >, <=, >=	SVR_BOOLEAN
SVR_MONEY	SVR_MONEY	+, -, /, *	SVR_DOUBLE
SVR_MONEY	SVR_DOUBLE	+, -, /, *	SVR_DOUBLE
SVR_DOUBLE	SVR_MONEY	+, -, /, *	SVR_DOUBLE
SVR_DOUBLE	SVR_DOUBLE	+, -, /, *	SVR_DOUBLE
SVR_FLOAT	SVR_FLOAT	+, -, /, *	SVR_DOUBLE
SVR_INT8	SVR_INT8	+, -, /, *	SVR_INT32
SVR_INT16	SVR_INT16	+, -, /, *	SVR_INT32
SVR_INT32	SVR_INT32	+, -, /, *	SVR_INT32
SVR_INT64	SVR_INT64	+, -, /, *	SVR_INT64
SVR_DATE	SVR_INT32	+, -	SVR_DATE

Table 7.2 Data Types and Supported Operations

7.9.4 Literal Values

The basic rule for literal values is: do not use them. There are numerous maintenance and readability issues, but there can be more serious behavioral problems at run time. These are caused by the fact that any un-typed value will be treated as an integer (SVR_INT32) and makes all operations on this value integer operations, for example 4 / 5 = 0.

7.10 Evidence Missing

The Rules Engine supports two basic approaches to the issue of evidence that is not available:

- an exception-based response under normal circumstances, and
- a logical function-based approach using the `isNothing()` function (see Section 7.7.2, *The isNothing() Function*).

These two approaches should allow the rules to deal appropriately with any given missing evidence situation (providing that the rules and associated conditions have been written to allow these situations).

7.10.1 "DataItem Missing" Exception

This is the default route for evidence that is unavailable when the rules are executing; an exception is thrown and execution is halted at this point. This is a valid approach in situations where it is not possible to calculate eligibility and entitlement in the absence of the evidence. Most evidence will fall into this category and there is no extra effort for the rules developer in this case.

7.10.2 Existence Checking

Existence checking involves the use of the `isNothing()` function (see Section 7.7.2, *The isNothing() Function*). This allows a rule to have a condition based on the existence of a piece of evidence. This means that rules can be written to verify the existence of evidence, where the value of the evidence is not used. If the value of the piece of evidence is directly referenced elsewhere in the rules even if it is initially referenced via `isNothing()` then, should that evidence be unavailable an exception will be thrown. Therefore, any place where both the existence and value of a piece of evidence must be used, it is necessary to "wrap" this processing in a business process that gives a Boolean for the existence of the evidence and a second parameter for its value.

7.11 Data Items Used

The result information for each rule and objective will also contain information about the RDO data items and their values which were used for evaluating the result of that particular rule or objective.

This information is not directly accessible to the application developer, but it might be used to display information about the RDO data items used and their values on the client screen.

7.12 Rules Summary Item

Rule groups, rule list groups and rules have a property Summary Item. It is represented by a check-box in the Rules Editor. Summary Item is used to specify which rules should be displayed on rules execution summary page.

This property allows only the relevant rules execution information to be

shown.

For example, a person is eligible for a study grant if he or she is studying or if he or she has a dependent who is studying.

If you are executing the rules for a person who is studying and does not have any studying dependents, this means that person is eligible for the product and you do not really want to show failure of the rule “Does this person have any dependents who are studying?” as it is not relevant.

In order not to display it in the rules summary view, you have to set Summary Item property of rule group to `true` and set summary item property for both rules to `false`.

7.13 Multi Language Support

It is possible to specify multi-language text for rules. Only the text for the appropriate language is displayed in the Cúram application.

The following items can be localized:

- Rule Name field in the *Rules Editor* (for more information see the *Cúram Rules Editor Guide*);
- Success Text field in the *Rules Editor* (for more information see the *Cúram Rules Editor Guide*);
- Failure Text field in the *Rules Editor* (for more information see the *Cúram Rules Editor Guide*);
- Comments of an attribute on a RDO or a ListRDO in the Merlin Toolbar.
- Display name of a RDO or a ListRDO in the in the Merlin Toolbar.

Multi language text for Rule name, Success text and Failure text can be entered by using the online *Rules editor*. An option *Create Translation* has been provided for every rule set node in the Rules editor to facilitate the user to enter the text in different languages.(for more information see the *Cúram Rules Editor Guide*)

7.14 Error Reporting

The errors reported by the rules system can be viewed in a number of categories.

7.14.1 Rules Runtime Errors

Runtime errors are normally thrown as exceptions that are then caught inside the Cúram server application and an appropriate message returned to the user. The main errors reported by the runtime system are:

- **Data Item Missing.** This error is reported if an attempt is made to use a piece of evidence which is not available. This is typically the case when some required information has not been recorded on the system in relation to a given client.
- **Rule Set Missing.** This error is reported if an attempt is made to execute a rule set that does not exist on the system. This would normally occur if an error has been made during the rule release process, or if data has been corrupted elsewhere in the system.

7.14.2 Error Handling

The error handling mechanism used by the rules is the standard Java exception processing mechanism. Exceptions are then “wrapped” by the Cúram infrastructure to return error messages to the user.

7.14.3 Warnings versus Errors

The rules system cannot produce any warnings, as its operation either succeeds or fails. However, it may be the case that the processing invoked after a decision is returned treats the failure due to evidence being missing as a warning, and continues processing via another route.

Also, in the case where eligibility is being assessed for multiple services, a failure (due to an error or otherwise) on a single product should not prevent the assessments for other services going ahead.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service. IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing

IBM Corporation

North Castle Drive

Armonk, NY 10504-1785

U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing

Legal and Intellectual Property Law.

IBM Japan Ltd.

1623-14, Shimotsuruma, Yamato-shi

Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typograph-

ical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Dept F6, Bldg 1

294 Route 100

Somers NY 10589-3216

U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources.

IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products

should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This publication documents intended programming interfaces that allow the customer to write programs to obtain the services of IBM Cúram Social Pogram Management.

Trademarks