

IBM Cúram Social Program Management  
Version 7.0.0

*Cúram Workflow Reference Guide*



**Note**

Before using this information and the product it supports, read the information in "Notices" on page 129

**Edition**

This edition applies to IBM Cúram Social Program Management v7.0.0 and to all subsequent releases unless otherwise indicated in new editions.

Licensed Materials - Property of IBM.

© **Copyright IBM Corporation 2012, 2016.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

© Cúram Software Limited. 2011. All rights reserved.

# Contents

|                          |            |
|--------------------------|------------|
| <b>Figures</b> . . . . . | <b>vii</b> |
|--------------------------|------------|

|                         |           |
|-------------------------|-----------|
| <b>Tables</b> . . . . . | <b>ix</b> |
|-------------------------|-----------|

## Cúram Workflow Reference . . . . . 1

|  |    |
|--|----|
| Overview . . . . .                               | 1  |
| Prerequisites . . . . .                          | 1  |
| How to use the guide . . . . .                   | 1  |
| Structure of this Document . . . . .             | 1  |
| Workflow Processes . . . . .                     | 1  |
| Data Flow . . . . .                              | 2  |
| Activities . . . . .                             | 2  |
| Flow Control . . . . .                           | 4  |
| Development and Runtime . . . . .                | 4  |
| Inbox Configuration and Customization . . . . .  | 5  |
| Creating a Workflow Process . . . . .            | 5  |
| Process definition life cycle . . . . .          | 5  |
| Process creation . . . . .                       | 5  |
| Process visualization. . . . .                   | 5  |
| Releasing a process . . . . .                    | 6  |
| Process versions (process editing) . . . . .     | 7  |
| Process import, export and copy. . . . .         | 7  |
| Localization . . . . .                           | 8  |
| Process execution. . . . .                       | 9  |
| Basic engine behavior . . . . .                  | 9  |
| Executing multiple versions . . . . .            | 9  |
| Process Instance Administration . . . . .        | 9  |
| Method Reference Library . . . . .               | 10 |
| Referencing Cúram methods. . . . .               | 10 |
| Method types. . . . .                            | 10 |
| WDO templates . . . . .                          | 11 |
| Metadata . . . . .                               | 11 |
| Import and syncing. . . . .                      | 12 |
| Validations . . . . .                            | 12 |
| Process Definition Metadata . . . . .            | 12 |
| Overview . . . . .                               | 12 |
| Metadata . . . . .                               | 13 |
| Validations . . . . .                            | 15 |
| Description of Context WDOs . . . . .            | 16 |
| Workflow Data Objects . . . . .                  | 16 |
| Overview . . . . .                               | 16 |
| Metadata . . . . .                               | 17 |
| Validations . . . . .                            | 20 |
| List of Context WDOs . . . . .                   | 21 |
| Runtime Information . . . . .                    | 22 |
| Process Enactment . . . . .                      | 23 |
| Overview . . . . .                               | 23 |
| Code enactment (enactment service API) . . . . . | 23 |
| Metadata . . . . .                               | 24 |
| Validations . . . . .                            | 25 |
| Code . . . . .                                   | 26 |
| Event enactment. . . . .                         | 26 |
| Configuration data . . . . .                     | 27 |
| Validations . . . . .                            | 28 |
| Base Activity . . . . .                          | 29 |

|                                       |    |
|---------------------------------------|----|
| Overview . . . . .                    | 29 |
| Metadata . . . . .                    | 29 |
| Localized Text . . . . .              | 30 |
| Validations . . . . .                 | 30 |
| Basic Activity Types . . . . .        | 30 |
| Route Activity . . . . .              | 30 |
| Start/End Process Activity . . . . .  | 31 |
| Automatic. . . . .                    | 31 |
| Prerequisites . . . . .               | 31 |
| Overview . . . . .                    | 31 |
| Cúram Business Methods. . . . .       | 31 |
| Metadata . . . . .                    | 32 |
| Validations . . . . .                 | 32 |
| Code . . . . .                        | 32 |
| Input Mappings . . . . .              | 33 |
| Metadata . . . . .                    | 33 |
| Validations . . . . .                 | 38 |
| Runtime Information . . . . .         | 39 |
| Output Mappings . . . . .             | 39 |
| Metadata . . . . .                    | 39 |
| Validations . . . . .                 | 43 |
| Runtime Information . . . . .         | 43 |
| Description of Context WDOs . . . . . | 43 |
| Event Wait . . . . .                  | 44 |
| Prerequisites . . . . .               | 44 |
| Overview . . . . .                    | 44 |
| List of events. . . . .               | 45 |
| Metadata . . . . .                    | 45 |
| Validations . . . . .                 | 46 |
| Code . . . . .                        | 47 |
| Runtime Information . . . . .         | 47 |
| Deadline . . . . .                    | 48 |
| Prerequisites . . . . .               | 48 |
| Metadata . . . . .                    | 49 |
| Validations . . . . .                 | 50 |
| Code . . . . .                        | 51 |
| Runtime Information . . . . .         | 51 |
| Description of Context WDOs . . . . . | 51 |
| Output Mappings . . . . .             | 52 |
| Metadata . . . . .                    | 52 |
| Validations . . . . .                 | 53 |
| Runtime Information . . . . .         | 53 |
| Description of Context WDOs . . . . . | 53 |
| Reminders. . . . .                    | 53 |
| Metadata . . . . .                    | 54 |
| Validations . . . . .                 | 54 |
| Code . . . . .                        | 55 |
| Runtime Information . . . . .         | 55 |
| Manual. . . . .                       | 55 |
| Prerequisites . . . . .               | 55 |
| Overview . . . . .                    | 55 |
| Task details . . . . .                | 56 |
| Metadata . . . . .                    | 57 |
| Validations . . . . .                 | 60 |
| Code . . . . .                        | 61 |
| Runtime Information . . . . .         | 62 |

|   |    |   |     |
|---|----|---|-----|
| Description of Context WDOs . . . . .               | 62 | Notification Details . . . . .                    | 93  |
| Allocation strategy . . . . .                       | 62 | Metadata . . . . .                                | 93  |
| Prerequisites . . . . .                             | 63 | Validations . . . . .                             | 95  |
| Metadata . . . . .                                  | 63 | Code . . . . .                                    | 97  |
| Validations . . . . .                               | 67 | Runtime Information . . . . .                     | 97  |
| Code . . . . .                                      | 69 | Notification Allocation Strategy . . . . .        | 97  |
| Runtime Information . . . . .                       | 70 | Prerequisites . . . . .                           | 97  |
| Description of Context WDOs . . . . .               | 71 | Code . . . . .                                    | 97  |
| Business Object Associations . . . . .              | 71 | Transitions . . . . .                             | 100 |
| Metadata . . . . .                                  | 71 | Overview . . . . .                                | 100 |
| Validations . . . . .                               | 72 | Metadata . . . . .                                | 100 |
| Code . . . . .                                      | 72 | Validations . . . . .                             | 102 |
| Runtime Information . . . . .                       | 72 | Runtime Information . . . . .                     | 102 |
| Event Wait . . . . .                                | 72 | Conditions . . . . .                              | 103 |
| Prerequisites . . . . .                             | 73 | Overview . . . . .                                | 103 |
| Description of Context WDOs . . . . .               | 73 | Metadata . . . . .                                | 103 |
| Decision . . . . .                                  | 73 | Validations . . . . .                             | 106 |
| Prerequisites . . . . .                             | 73 | Split/Join . . . . .                              | 107 |
| Overview . . . . .                                  | 73 | Introduction . . . . .                            | 107 |
| Task Details . . . . .                              | 73 | Choice XOR Split . . . . .                        | 108 |
| Metadata . . . . .                                  | 74 | Metadata . . . . .                                | 108 |
| Validations . . . . .                               | 75 | Parallel AND split . . . . .                      | 108 |
| Runtime Information . . . . .                       | 77 | Metadata . . . . .                                | 108 |
| Question Details . . . . .                          | 77 | Workflow Structure . . . . .                      | 109 |
| Metadata . . . . .                                  | 78 | Overview . . . . .                                | 109 |
| Validations . . . . .                               | 80 | Graph Structure . . . . .                         | 109 |
| Runtime Information . . . . .                       | 81 | Block Structure . . . . .                         | 110 |
| Description of Context WDOs . . . . .               | 81 | An Analogy for Blocks . . . . .                   | 110 |
| Subflow . . . . .                                   | 82 | Block Types Supported by Workflow . . . . .       | 110 |
| Prerequisites . . . . .                             | 82 | Structural Rules . . . . .                        | 111 |
| Overview . . . . .                                  | 82 | Graph Structure Rules . . . . .                   | 111 |
| Subflow Process . . . . .                           | 82 | Block Structure Rules . . . . .                   | 112 |
| Metadata . . . . .                                  | 82 | Validations . . . . .                             | 112 |
| Validations . . . . .                               | 83 | Simple Syntactic Checks . . . . .                 | 112 |
| Input Mappings . . . . .                            | 83 | Graph Checks . . . . .                            | 113 |
| Metadata . . . . .                                  | 83 | Block Checks . . . . .                            | 113 |
| Validations . . . . .                               | 84 | Workflow Web Services . . . . .                   | 114 |
| Output Mappings . . . . .                           | 84 | Overview . . . . .                                | 114 |
| Metadata . . . . .                                  | 84 | Exposing a workflow web service . . . . .         | 114 |
| Validations . . . . .                               | 85 | Process Enactment . . . . .                       | 114 |
| Loop Begin and Loop End . . . . .                   | 86 | Process completion callback . . . . .             | 115 |
| Prerequisites . . . . .                             | 86 | Invocation from BPEL processes . . . . .          | 115 |
| Overview . . . . .                                  | 86 | File Locations . . . . .                          | 116 |
| Loop Type . . . . .                                 | 86 | Overview . . . . .                                | 116 |
| Metadata . . . . .                                  | 86 | Workflow Process Definition Files . . . . .       | 116 |
| Loop Begin Activity . . . . .                       | 86 | Customizing Workflow Process Definition           |     |
| Loop End Activity . . . . .                         | 87 | Files . . . . .                                   | 117 |
| Runtime Information . . . . .                       | 88 | Event Definition Files . . . . .                  | 117 |
| Description of Context WDOs . . . . .               | 88 | Configuration . . . . .                           | 118 |
| Parallel . . . . .                                  | 88 | Overview . . . . .                                | 118 |
| Prerequisites . . . . .                             | 88 | Application Properties . . . . .                  | 118 |
| Overview . . . . .                                  | 88 | JMSLite . . . . .                                 | 119 |
| Metadata . . . . .                                  | 89 | What JMSLite Does . . . . .                       | 119 |
| Generic Metadata for a Parallel Activity . . . . .  | 89 | Why JMSLite? . . . . .                            | 120 |
| Metadata for a Parallel Manual Activity . . . . .   | 89 | Using JMSLite . . . . .                           | 120 |
| Metadata for a Parallel Decision Activity . . . . . | 90 | Debugging workflows . . . . .                     | 120 |
| Validations . . . . .                               | 92 | Inbox and Task Management . . . . .               | 121 |
| Runtime Information . . . . .                       | 92 | Overview . . . . .                                | 121 |
| Description of Context WDOs . . . . .               | 92 | Inbox Configuration . . . . .                     | 121 |
| Activity Notifications . . . . .                    | 92 | Inbox List Sizes Configuration Settings . . . . . | 121 |
| Overview . . . . .                                  | 92 | Get Next Task Configuration Settings . . . . .    | 122 |

Task Redirection and Allocation Blocking  
Settings . . . . . 123  
Inbox Customization . . . . . 124  
How to customize the Inbox . . . . . 125

**Notices . . . . . 129**  
Privacy Policy considerations . . . . . 130  
Trademarks . . . . . 131



---

## Figures

|   |                                       |     |
|---|---------------------------------------|-----|
| 1. Visualization of Close Case Workflow Process | 3. Callback Port Type . . . . .       | 116 |
| Definition . . . . .                            | 4. WSDL extensions for BPEL . . . . . | 116 |
| 2. Process Enactment Port Type . . . . .        |                                       | 115 |



---

## Tables

|    |   |     |    |   |     |
|----|---|-----|----|---|-----|
| 1. | Description of the ProcEnactmentEvt Table | 27  | 5. | Inbox List Sizes Configuration Settings     | 122 |
| 2. | Description of the ProcEnactEvtData Table | 27  | 6. | Get Next Task Configuration Settings        | 123 |
| 3. | Subject Text Data Conversion . . . . .    | 58  | 7. | Security Identifiers and Associated Actions | 123 |
| 4. | Condition Expression Operators . . . . .  | 105 | 8. | Customization Points . . . . .              | 124 |



---

## Cúram Workflow Reference

The Cúram Workflow Management system is used to define processes to achieve certain business goals. A process definition is the central component that describes the business process. Process definition metadata is the top-level concept in a process definition. It contains information to identify and describe the process definition. In-depth descriptions of the workflow metadata can be entered. The effects of that metadata at run time can be set.

---

### Overview

The Workflow Reference Guide is intended to provide detailed explanations of the concepts of the Cúram Workflow Management System (WMS). It aims to describe how to define a process to achieve certain goals by giving in-depth descriptions of the workflow metadata as well as the effects of that metadata at runtime. The guide is not intended as a tutorial document but rather a concise description of all the features available in Cúram workflow.

### Prerequisites

The guide assumes some familiarity with workflow concepts and how they are realized in the Cúram WMS.

In particular it assumes that you have at least read the *Business Analyst Guides: Cúram Workflow Overview Guide*.

### How to use the guide

As the guide is a reference manual, the sections are as independent of others as possible. The intent is that a reader is aware of a concept that they want to get further details on, finds the relevant section in the guide and need read only that section. While it is not expected that this document be read from cover to cover it is structured in such a way as to make such a reading possible and productive.

Some parts of the Cúram WMS itself draw heavily on each other and as such the documentation reflects this. These external references come in two flavors: prerequisites, which are pointers to information that is indispensable in understanding the section at hand and general links, which highlight related but not required information.

### Structure of this Document

The guide can be also viewed in a number of distinct sections each of which reflects an area of the Cúram WMS and how these interact with each other. The following sections include a summary of what these logical sections are, what other sections are included in those logical sections and what areas of the Cúram WMS are covered within those related sections.

#### Workflow Processes

The *Workflow Processes* section of the document describes the metadata that is associated with a workflow process definition. The lifecycle of a process definition is also described.

“Creating a Workflow Process” on page 5 describes how to create and visualize a workflow process by using the Cúram workflow system. Releasing a process is also described while its effect on the versioning associated with process definitions is also detailed. Importing and exporting process definitions is discussed while the localization of the text contained within a process is outlined. Running a workflow process by using the Cúram workflow engine is described in detail. A description of the method library and the workflow data object (see “Workflow Data Objects” on page 16) template library is also provided.

“Process Definition Metadata” on page 12 describes the metadata that is associated with a workflow process definition. Each metadata field is outlined while the validations and context workflow data objects associated with the workflow process as a whole are detailed.

## **Data Flow**

The *Data Flow* section of the document describes how data is stored and manipulated in a process instance. In particular issues of how data is conveyed from the outside world (at process enactment) and between activities and transitions within the process is described.

“Workflow Data Objects” on page 16 describes the objects that are used to maintain and pass data around in the workflow engine. The metadata that constitutes workflow data objects and their attributes is outlined in detail. Validations that pertain to the creation and modification of workflow data objects are discussed. Finally, the context workflow data objects that are made available by the Process Definition Tool and workflow engine are also described in the section.

“Process Enactment” on page 23 describes the starting of a process instance (that is, the performing of the work that is defined in the process definition). The enactment service API is described while the enactment mappings metadata associated with the enactment of a process is discussed. Associated validations and code examples are also provided. It is also possible to start a process in response to an event being raised and this is also described in the section. The configuration data to perform this action is outlined in detail. Any validations that are run when the mappings between events and workflow processes are created is described.

## **Activities**

Activities are central in a workflow process as they are the steps at which the business processing for the workflow takes place. The various activity types that are supported by the Cúram WMS are all described in the *Activities* section of the document. As notifications are also pertinent to each activity type, they are also described in the guide.

“Base Activity” on page 29 describes the metadata details common to all of the supported activity types in the Cúram workflow system. The validations that are run when creating or modifying an activity are also outlined. Finally, some of the more simple activity types are described including the route activity and the start and end process activities.

“Automatic” on page 31 describes the metadata details associated with an automatic activity. Both the input and output mappings that are specified for the method that is associated with the automatic activity are discussed in detail. The validations run when creating or modifying the metadata for an automatic activity are outlined. Finally, the `Context_Result` and `Context_Error` workflow data objects that are available for use in transitions from automatic activities are also described in the section.

“Event Wait” on page 44 describes the metadata details associated with an event wait activity. This includes the list of events, the deadline details (including any deadline reminders) associated with an event wait and also any output mappings that can be specified. The validations that are run when creating or modifying event wait metadata are also described. The runtime information that is associated with the execution of event wait activities by the workflow engine is also outlined in detail. Finally, the Context\_Event and Context\_Deadline workflow data objects that are available for use in transitions from event wait activities are also detailed in the section.

“Manual” on page 55 describes the metadata details associated with a manual activity. This includes the manual task details, the allocation strategy, the business object associations, and the event wait associated with the manual activity. The validations that are run when creating or modifying manual activity metadata are also described. The runtime information that is associated with the execution of manual activities by the workflow engine is also outlined in detail. Finally, a description of the Context\_Task workflow data object that is available for use in the various mappings that are associated with a manual activity is also provided in the section.

“Decision” on page 73 describes the metadata details associated with a decision activity. This metadata includes the decision task details (which is similar to the manual activity task details) and the question details for multiple choice and free text questions. The various validations that are run when creating or modifying the task or question details that are associated with a decision activity are outlined. The section also includes a description of the runtime information that is present when the workflow engine ran a decision activity. A description of the Context\_Decision workflow data object is also provided in the section.

“Subflow” on page 82 describes the metadata details associated with a subflow activity. This includes details of the subflow process that is associated with the subflow activity and any input mappings that are required to enact that subflow process. The various validations that are run when creating or modifying this metadata and a description of these is also provided in the section.

“Loop Begin and Loop End” on page 86 describes the metadata details associated with a loop begin and loop end activity. The loop type, loop condition, and end loop activity reference of a loop begin activity are described. The section also includes a description of the runtime information that is present when the workflow engine runs a loop in a workflow process definition. A description of the Context\_Loop workflow data object is also provided in this section.

“Parallel” on page 88 describes the metadata details associated with a parallel activity. Parallel activities wrap existing activity types including “Manual” on page 55 activities and “Decision” on page 73 activities. Since the metadata that is associated with these activity types remains the same, it is not described again in the section. The validations run when creating or modifying parallel activity metadata are also described. The runtime information that is associated with the execution of parallel activities by the workflow engine is also outlined in detail. Finally, a description of the Context\_Parallel workflow data object that is available for use in the various mappings that are associated with a parallel activity is also provided in the section.

“Activity Notifications” on page 92 describes the metadata details associated with an activity notification. These details include the delivery mechanism, the subject, the body, the allocation strategy, and actions associated with the notification. A

number of validations that are run when creating or modifying notification metadata are also outlined in the section. A description of the runtime information when the workflow engine creates a notification is also provided. Finally, there are a number of implementation details that are required in the Cúram application to allow notifications to be delivered correctly. These are also discussed in the section.

## **Flow Control**

A workflow process models the flow of information through an organization, passing through steps that are carried out by human agent or computer software to achieve a business goal. The *Flow Control* section of the document details how such information flow (between activities) is specified in and managed by the Cúram WMS.

“Transitions” on page 100 describes the links between activities. The metadata that is associated with transitions is described in detail. Validations that pertain to the creation and modification of transitions are also discussed. The runtime information that is associated with the processing of transitions by the workflow engine is also described.

“Conditions” on page 103 describes the process definition metadata construct that represents a condition. Validations that pertain to the creation and modification of conditions are also discussed.

“Split/Join” on page 107 describes the metadata that is associated with activity splits and joins, when they are used and the various types available.

“Workflow Structure” on page 109 describes the structure of a workflow process as determined by the activities in the process and the transitions between them. The constraints present when a process definition is constructed to ensure that it is a valid block structure are outlined while validations that are executed as part of these constraints are discussed.

## **Development and Runtime**

The *Development and Runtime* section of the document describes the specifics of the development and runtime environment for Cúram workflows. Specifically, it details how to run, configure, and debug workflows.

“Workflow Web Services” on page 114 describes the steps necessary to allow process enactment through web services by exposing Cúram workflow process as a web services.

“File Locations” on page 116 details where the various outputs of such utilities as the Process Definition Tool and other administration user interfaces are exported to and version controlled. These outputs include process definition metadata files and also the source files that are associated with events.

“Configuration” on page 118 describes the workflow-related application properties, their names, their default settings and what they are used for in the Cúram workflow system.

“JMSLite” on page 119 details the Cúram lightweight JMS server that can run alongside the RMI testing environment in a supported Integrated Development Environment. The steps that are required to start the JMSLite server are outlined while a detailed description of how to debug workflows by using JMSLite is also discussed.

## **Inbox Configuration and Customization**

The *Inbox Configuration and Customization* section of the document describes the configuration and customization options that are available in the Inbox and Task Management section of the Cúram WMS. Specifically, it details how to configure the number of tasks that are displayed on the various lists that are displayed in the Inbox and also how to customize the various Inbox and Task Management actions that are available in the system.

“Inbox and Task Management” on page 121 describes the configuration options available to be used in the Inbox. It also details how to customize the available Inbox and Task Management functions by using the Google Guice framework.

---

## **Creating a Workflow Process**

### **Process definition life cycle**

The process definition is the central concept in any workflow system so naturally how it is created and used is of critical importance. This chapter describes the facilities provided by the Cúram workflow system to create and administer process definitions.

#### **Process creation**

The Cúram workflow system provides a *Process Definition Tool* (PDT) for creating and maintaining process definitions which can then be interpreted by the workflow engine. Creating a process definition involves using the Process Definition Tool to describe the desired process behavior in terms of activities and transitions.

A number of utilities are provided as part of the Process Definition Tool that can aid in process creation. The PDT allows a process definition to be visualized during design. Processes can also be copied, imported, and exported using the PDT.

#### **Process visualization**

A read-only graphical utility is provided as part of the Process Definition Tool which enables process administrators to visualize processes as they are being created or modified. This tool allows administrators view all activities and transitions in a process definition and provides a high level view of all the possible paths through the workflow process during execution. The following figure shows an example of a graphical representation of a workflow process definition.

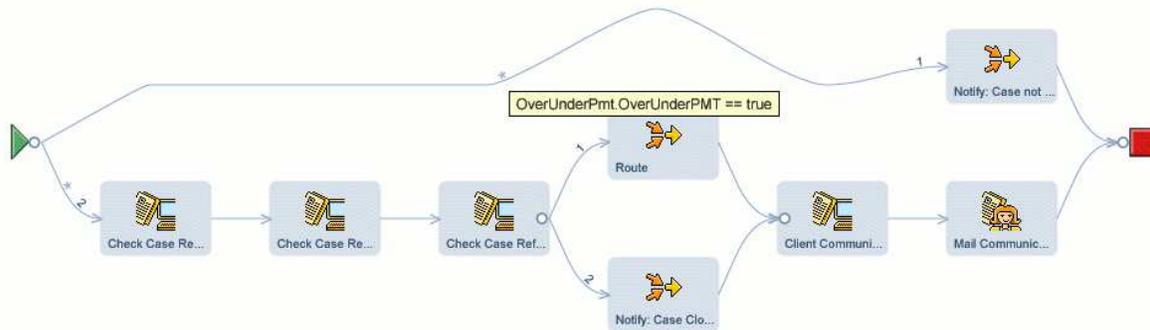


Figure 1. Visualization of Close Case Workflow Process Definition

The visualized process comprises a number of nodes on a graph representing the activities in the process. The nodes are linked by graph edges and these reflect the transitions defined in the process definition. Clicking on an activity in the graph displays the details of the activity in the PDT. Similarly, clicking on a transition between activities on the graph displays the details of the transition in the PDT.

The graphical tool displays the following information for each process visualized:

- The type and name of each activity. Each activity type is identified by a specific icon.
- The notifications defined for each activity (See “Activity Notifications” on page 92). If an activity has an associated notification, it will be represented as an envelope which is click-able through to the associated activity notification page.
- The split/join type (See “Split/Join” on page 107) for each activity. A split or join type of “choice” on an activity is represented as a circle, while a split or join type of “parallel” is represented as a square.
- The transitions between activities. Where a transition between activities has an associated transition condition (See “Conditions” on page 103), this is represented as an asterisk. The details of the condition are displayed when the mouse is placed over that asterisk.
- The ordering of each choice split (See “Split/Join” on page 107) from an activity. As the ordering of a choice split from an activity is important (the first eligible transition in the list will be followed), the order of each transition from the activity is displayed as a number on that transition.

### Releasing a process

Once a process definition has been created and is ready for use, it must be released before it can be executed by the workflow engine (See “Process execution” on page 9). As a process is being released using the PDT, it is examined to ensure all the information the engine needs to execute the process is present and internally consistent. The validations required to release a process are described in the various metadata sections of this document.

Only processes that have passed all of the required validations can be released and made available to the workflow engine. Once a process definition has been released it becomes read-only and can no longer be edited by the Process Definition Tool without creating a new version.

## Process versions (process editing)

Changes may be required to a released process over time, but as a released process is read-only, a new version is required before any modifications can be applied. Attempting to edit a released process in the PDT will automatically create a new unreleased version of that process.

There can only be one unreleased version of a process at any time. If the administrator wishes to edit a released process, any existing unreleased versions must first be released or deleted.

## Process import, export and copy

The import and export functionality allows developers move process definitions as required. For example, a process definition might be developed on a development system and only moved to a production system after testing has been completed.

Exporting a process will export the process metadata to the file system. This metadata can then be imported using the import process option in the PDT. A process imported in this way will be assigned the highest version number available, and will be unreleased regardless of its released state when imported. This is to ensure that imported process definitions are subject to the same release validations as other definitions developed locally. An overwrite option is available when importing that ensures any existing unreleased version of the process will be overwritten with the imported version.

There may be situations where a process definition differs only slightly from another in the workflow system. A copy process option is available which allows an existing process to be copied to a new process when required. The new process will always be unreleased when copied with a version set to 1, regardless of the status of the original process.

### Validations:

- A process definition cannot be imported if an unreleased version of a process already exists with the same name, and the overwrite option has not been selected.
- A process definition cannot be imported if a name for that process has not been specified
- A process definition cannot be imported if a process already exists with the same name and different process identifier. This validation ensures that an imported definition cannot inadvertently overwrite an existing process definition unless the process identifiers match.
- When copying an existing process, the name of the new process must be unique within the workflow system.
- The length of the name of the workflow process definition being imported must not exceed the maximum length allowed for such a name. This length is 254 characters.
- The length of the names of any of the workflow data objects contained in the workflow process definition being imported must not exceed the maximum length allowed for such a name. This length is 75 characters.
- The length of the names of any of the workflow data object attributes contained in the workflow process definition being imported must not exceed the maximum length allowed for such a name. This length is 75 characters.
- Any code table values that are contained in the workflow process definition being imported must be valid (i.e. the code table must exist and the specified code must exist in that codetable).

- For each localizable text in the process definition being imported, there must exist at least an entry for the English (i.e. "en") locale. Entries in other locales can also exist (e.g. the different user locales supported by the application) but each translation must be accompanied by an entry for the English locale.
- The identifiers for activities, transitions, transition condition expressions, loop condition expressions, events and reminders must be unique in the workflow process definition being imported.

## Localization

Workflow process definitions contain metadata text that needs to be viewed in different languages by different users. For example, when a manual activity is executed, it creates a task which has an associated subject. The Process Definition Tool enables the process developer to localize this subject string for each of the locales supported by the application.

Localizable strings can be identified in a process definition by the metadata specified in “Localized Text” on page 30. Any localizable text strings that have been specified in a process definition must at least have a corresponding entry for the English (i.e. "en") locale. When adding a localized string to a process definition, the PDT by default adds the string to both the user's and the English locales. Any subsequent changes to localized text (i.e. additions, deletions or modifications) can be made via the localization screen of the PDT.

The following is a list of the localizable text strings that may be specified in a process definition.

- Process display name
- Process description
- Workflow Data Object display name
- Workflow Data Object description
- Workflow Data Object attribute display name
- Activity name
- Activity description
- Manual Activity Task message
- Manual Activity Task Action message
- Parallel Manual Activity Task message
- Parallel Manual Activity Task Action message
- Decision Activity Action message
- Decision Activity Question message
- Decision Activity Secondary Action message
- Decision Activity Answer display value
- Parallel Decision Activity Action message
- Parallel Decision Activity Question message
- Parallel Decision Activity Secondary Action message
- Parallel Decision Activity Answer display value
- Activity Notification Subject message
- Activity Notification Body message
- Activity Notification Action message
- Reminder Notification Subject message
- Reminder Notification Body message
- Reminder Notification Action message

The `LocalizableStringResolver` API provides routines that resolve and return the various localizable strings for tasks and notifications that exist in a workflow process definition for the locale of the current user. Where a text string has not been localized for the current user locale, the text for the English (i.e. "en") locale is returned instead.

## Process execution

A workflow process definition describes the tasks and flow of a business process in terms understood by the Cúram Workflow Management System. To perform the work described in the specified process definition, an instance of it must be created and executed by the workflow engine. The mechanism by which this is done is described in this section. A process instance may be considered as the runtime data for an enacted workflow process definition.

### Basic engine behavior

The Cúram Workflow Management System includes a workflow engine which provides the runtime execution environment for a process instance. There are various mechanisms available to enact a workflow process and these are discussed in "Process Enactment" on page 114. When a process is enacted, the workflow engine examines the relevant database table and uses the *latest released version* of the specified process definition to create the process instance to run.

As each activity is executed, an associated activity instance record is created and managed by the workflow engine. This record contains the runtime data for an activity instance in the enacted workflow. As the workflow progresses, the engine evaluates the transitions (see "Transitions" on page 100) for the various activities to decide which path through the process to take. This involves determining the types of splits and joins (see "Split/Join" on page 107) that the activity possesses and also executing any conditions (see "Conditions" on page 103) that the various transitions in the process may have. Transition instance records (which contain the runtime data for a workflow transition) for each transition followed in the workflow process are also created and managed by the engine.

### Executing multiple versions

Modifying and releasing a new version of a process will not affect any currently executing instances of that process. A process will run to completion in the workflow engine with the version that it was started with, regardless of any subsequent versions that may have been released.

### Process Instance Administration

A workflow administrator has the ability to influence the execution of a running process instance through the Cúram Workflow Administration interface. The following functions are available for this purpose:

#### Suspend a Process Instance

Any currently executing process instance may be suspended. When this occurs, the workflow engine will allow all activity instances that are *in progress* within that process instance to complete. However, the next set of activities that are required to be executed for that process instance are started by the workflow engine and immediately suspended. Any synchronous subflow processes (see "Subflow" on page 82) associated with the process instance being suspended will also be suspended by the workflow engine.

#### Resume a Process Instance

Any workflow process instance that has been suspended may be resumed. When this occurs, the activity instances that were previously suspended for

that process instance are restarted by the workflow engine. Any suspended synchronous subflow processes (see “Subflow” on page 82) associated with that process instance will also be resumed by the workflow engine.

#### **Aborting a Process Instance**

Any currently executing or suspended process instance may be aborted. All activities that are *in progress* in the aborted process instance are completed. If the process contains any manual or decision activities that are in progress, the associated tasks will be closed by the workflow engine when the process instance is aborted. No new activities associated with an aborted process instance will be started by the workflow engine. Any synchronous subflow processes (see “Subflow” on page 82) associated with the process instance will also be aborted. An aborted process instance cannot be resumed.

## **Method Reference Library**

Several situations exist in the Cúram Workflow Management System where it is necessary to interact with the Cúram application by calling some business process or entity methods (see “Cúram Business Methods” on page 31 for one example of such an interaction). Any business process object (BPO) or entity method in the application can be called by the workflow engine. However there are far too many such methods to present to a process designer for use in their process definitions in an acceptable way. The purpose of this library is to allow an administrator to assign methods that are likely to be of use in process definitions to a more manageable list for use in process design. Of course it is not necessary to pre-populate the library with all methods that could be used in the future. New methods can be added to the library as required.

#### **Referencing Cúram methods**

Business process object (BPO) or entity methods must be added to the Method Reference Library before they can be referenced in a process definition. The method type defined when adding to the library will dictate where that method will be available for use within a process definition.

Note that removing a method reference from the Method Reference Library will not remove it from any process definitions that reference it. As long as the method is still a valid Cúram application method any process definitions that reference it will remain valid.

#### **Method types**

A Cúram business process object (BPO) or entity method must be added to the Method Reference Library with one of the three defined method types. A method can be associated with more than one method type, but the method will have to be added repeatedly with the different method type each time. Detailed below are different method types in the Method Reference Library, along with the restrictions on their use.

##### **General**

Methods with a type of *General* are only available as application methods to be invoked from automatic activities (see “Cúram Business Methods” on page 31). The Process Definition Tool restricts access to only these methods when selecting a method to be invoked from an automatic activity.

##### **Allocation**

Methods in the library with an *Allocation* type are only available for use as allocation strategy functions associated with manual activities, decision activities, parallel activities and activity notifications. (See “Allocation

strategy” on page 62). All methods specified with an allocation method type must have a return type of `curam.util.workflow.struct.AllocationTargetList`.

### Deadline

Methods of type *Deadline* in the method library can only be referenced as deadline handler methods associated with event-wait, manual, decision and parallel activities. (See “Deadline” on page 48).

## WDO templates

Data is maintained and passed around in the workflow engine as workflow data objects (see: “Workflow Data Objects” on page 16) The workflow data objects that a process can use are defined within the process definition itself. However it is conceivable that some workflow data objects will be useful in many process definitions. Therefore, it would be convenient if they could be imported from some pool instead of having to be recreated in each individual process. This is the purpose of this library.

### Metadata

```
<wdo is-list-wdo="false" initialize-attributes="false">
  <wdo-name>TaskCreateDetails</wdo-name>
  <display-name>
    <localized-text>
      <locale language="en">TaskCreateDetailsName</locale>
    </localized-text>
  </display-name>
  <description>
    <localized-text>
      <locale language="en">The Task Create Details WDO
        Template</locale>
    </localized-text>
  </description>
  <attributes>
    <attribute>
      <attribute-name>subject</attribute-name>
      <display-name>
        <localized-text>
          <locale language="en">Task Subject</locale>
        </localized-text>
      </display-name>
      <type>STRING</type>
      <required-at-enactment>>false</required-at-enactment>
      <process-output>>false</process-output>
      <constant-value/>
    </attribute>
    <attribute>
      <attribute-name>dueDate</attribute-name>
      <display-name>
        <localized-text>
          <locale language="en">Task Due Date</locale>
        </localized-text>
      </display-name>
      <type>DATE</type>
      <required-at-enactment>>false</required-at-enactment>
      <process-output>>false</process-output>
      <constant-value/>
    </attribute>
  </attributes>
</wdo>
```

The metadata defined for workflow data object templates is exactly the same as that defined for workflow data objects. For a full description of this metadata, see

“Workflow Data Objects” on page 16. The workflow data object template library is stored on the WDOTemplateLibrary database table.

Note that the initialize-attributes element of a workflow data object and the required-at-enactment, process-output and the constant-value elements of a workflow data object attribute are not available for editing in workflow data object templates and are automatically initialized to their default values in the associated metadata.

### **Import and syncing**

The templates defined in the workflow data object template library are available for use when creating process definitions. Importing a workflow data object template from the library will add the workflow data object and all its attributes to the current process definition.

Once a workflow data object template has been imported into a process definition, it can be synchronized with its corresponding entry in the workflow data object template library at any time. Synchronizing the template for a process definition will force the name and display name of the workflow data object to be updated from the template library. Along with this, any new attribute entries that exist in the template library entry will automatically be added to the workflow data object in the process definition. The user can optionally decide to override existing attributes in the workflow data object with those from the template library when synchronizing. It should be noted that overriding existing attributes might invalidate the process definition and require updates where the old attribute values have been used.

### **Validations**

- A workflow data object cannot be imported from a template if one already exists in the associated workflow process definition with the same name.

---

## **Process Definition Metadata**

### **Overview**

The process is the top level concept in a process definition. Primarily, it contains information to identify and describe the process definition. This information includes the identifier and the version of the process definition, its name and a brief description. It also includes a description of the failure allocation strategy that may be specified for a process. The following sections describe this top-level information.

## Metadata

```
<workflow-process id="100" process-version="2"
  language-version="1.0"
  released="false" category="PC5"
  createdBy="testuser"
  creationDate="20050812T135800">
  <name>ApprovePlannedItem</name>
  <process-display-name>
    <localized-text>
      <locale language="en">Approve Planned Item</locale>
    </localized-text>
  </process-display-name>
  <description>
    <localized-text>
      <locale language="en">This workflow process may be
        enacted to approve a planned item.</locale>
    </localized-text>
  </description>
  <documentation>Refer to the approve planned
    item documentation.
  </documentation>
  <web-service expose="true">
    <callback-service>wsconnector.ApprovePlannedItem
  </callback-service>
  </web-service>
  <failure-allocation-strategy>
    <allocation-strategy type="target"
      identifier="FAILUREALLOCATIONSTRATEGY" />
  </failure-allocation-strategy>
  ...
</workflow-process>
```

### **workflow-process**

This is the parent tag of all process definition metadata.

**id** This is a 64-bit identifier supplied by the Cúram key server when a process is created in the process definition tool. The process identifier is required to be unique in the Cúram workflow system. The reason for this is that the process identifier in conjunction with the process version number is how the workflow engine distinguishes one process definition record from another for the purpose of database reads.

### **process-version**

This number represents the version of a workflow process definition. A workflow process definition record is uniquely identified by its identifier and version number. A process definition may have many released versions and one version that is in edit. Once a process definition has been released, a new version is created and it can no longer be updated. Any subsequent updates will require a new version to be created and this version will not be active until it is released. When a process is enacted the highest released version number is used. Process instances that begin with a given version number remain bound to that version until completion.

### **language-version**

The process definition metadata is the Cúram workflow language. As new features and enhancements are added, this language may

change. This version number will allow either the workflow engine to run old language versions different to newer ones or more likely upgrade tools to convert old process definitions to new language versions.

**released**

This represents a boolean flag indicating whether or not the process definition has been released. Only process definitions that have been released can be enacted or selected as sub-processes in a subflow activity (see: “Subflow” on page 82).

**category**

A process definition must be placed into a category. The category must be selected in the Process Definition Tool and is taken from the ProcessCategory code-table. This attribute is intended to be used for process definition search functionality and has no functional effect on the process in the workflow engine.

**createdBy**

This represents the name of the user that created the workflow process definition. This attribute is intended to be used for process definition search functionality and has no functional effect on the process in the workflow engine.

**creationDate**

This represents the date and time that the workflow process definition was created. This attribute is intended to be used for process definition search functionality and has no functional effect on the process in the workflow engine.

**process-display-name**

This is the display name of the process definition. This is the name of the process that the user sees in the PDT. It is presented in the user's locale. The process display name is localizable and can be edited in the localization screen.

**name** This is the technical identifier of the process definition. It is the means by which the process is identified for the purpose of enactment. The enactment service (the API used to enact a process in code) identifies the process to enact by its name. As such this name is required to be unique within the workflow system and cannot be changed once the process is created. Since the process name is effectively a constant it is not localizable like an activity name.

**description**

A process can also have an optional description that briefly specifies what the process does for the benefit of those editing the process definition in the future. This is localizable text field in the same format as all localizable fields in a process definition (see: “Localized Text” on page 30).

**documentation**

A process can also have a link to some documentation that may explain the process in a more descriptive fashion. This is a free-form text field where the developer can enter the name of a document pertinent to the workflow process or indeed a link to such a document.

**web-service**

This optional element describes the web service details of a workflow process. A process can be marked as a Web Service by setting this metadata value which indicates that the process should be exposed as a

Web Service. This allows the process to be able to participate in a BPEL (Business Process Execution Language) orchestrated process and means that the process can be called from a BPEL process. Further details on this functionality may be seen in “Workflow Web Services” on page 114.

#### **expose**

This attribute represents a boolean flag indicating whether or not the process definition should be exposed as a Web Service. A workflow process definition is not exposed as a Web Service by default.

#### **callback-service**

This is an optional element because not all invocations from a BPEL process require a callback. The value is the fully qualified name of a class extending the `org.apache.axis.client.Service` class (which is part of the Service (Axis API) of the Apache Axis project). The `org.apache.axis.client.Service` class is generated by the Cúram web services connector functionality for outbound web services.

#### **failure-allocation-strategy**

A process can also have an optional failure allocation strategy specified for it. When allocating a task (associated with a “Manual” on page 55 or “Decision” on page 73 activity), the workflow engine invokes the associated allocation strategy to retrieve the list of allocation targets. If no allocation targets are returned from this invocation, the workflow engine will then check for the presence of a failure allocation strategy and will use this strategy to attempt to allocate the task. Since the allocation strategy of type *TARGET* specifies an allocation target directly there is never a need to fall back to the failure allocation strategy. The failure allocation strategy is a process-wide strategy and if specified will be used for all the manual and decision activities in the process when required.

#### **allocation-strategy**

This describes the failure allocation strategy being used for the process. The failure allocation strategy must be of type *TARGET*. If the work resolver cannot assign the task to a user, an organizational object (e.g. organization unit, position or job) or a work queue using the specified allocation target the task will be assigned to the default work queue. The identifier attribute represents the identifier of the allocation target being used as the failure allocation strategy.

## **Validations**

- A workflow process must have a unique process name. This means that a process cannot be created if the process name is empty or if a process with the same name already exists.
- A workflow process must have a process display name in the English (i.e. "en") locale. A display name in the user's locale should also be specified, but this is optional.
- A workflow process is required to specify a category.
- A released version of workflow process cannot be deleted once it has been enacted. This is required as even if a newer version of a process exists, process instances that are in progress when the new version becomes available run to completion with the version that they started with. Process definitions are also a necessary historical record that is drawn upon to create auditing information.

- A released version of workflow process cannot be deleted if it is referenced by a subflow activity in a released version of another process, where that released version is the latest released version.
- If a failure allocation strategy has been specified for the workflow process, then its type must be *TARGET*.
- The callback service class name cannot be specified if the workflow process has not been exposed as a webservice.
- The callback service class name must represent a class that can be found on the application classpath.
- The callback service class name must represent a class that extends the `org.apache.axis.client.Service` class.

## Description of Context WDOs

Certain generic system runtime information about the workflow engine is required to be made available to the activities and the transitions during the lifetime of a process instance. Details of the `Context_RuntimeInformation` workflow data object that provides this information can be seen in the following location: “List of Context WDOs” on page 21.

---

## Workflow Data Objects

### Overview

Data is maintained and passed around in the workflow engine as workflow data objects and list workflow data objects. These are logical objects defined in the process definition that have a name and a list of attributes of various types to which data can be assigned. They are conceptually similar to objects in programming languages although their manifestation in the workflow system is of course quite different. Workflow data object values may be written at process enactment or from the output of various activity types.

Workflow data object instances and list workflow data object instances exist as soon as the process is enacted and exist until the process completes. As such they are available to be used in the activities and the transitions throughout the lifetime of that process instance. It is therefore the responsibility of the process designer to ensure that attributes of workflow data objects are populated before they are used. Attempts to use workflow data object attributes before they are populated will result in failures at runtime.

## Metadata

```
<workflow-process id="32456" ..... >
  <name>CreateManualTask</name>
  .....
  </description>
  <enactment-mappings>
    .....
  </enactment-mappings>
  <wdos>
    <wdo is-list-wdo="false" initialize-attributes="true">
      <wdo-name>TaskCreateDetails</wdo-name>
      <display-name>
        <localized-text>
          <locale language="en">Task Create Details</locale>
        </localized-text>
      </display-name>
      <description>
        <localized-text>
          <locale language="en">This workflow data object
            contains the attributes required for the
            manual creation of a task.</locale>
        </localized-text>
      </description>
      <attributes>
        <attribute>
          <attribute-name>subject</attribute-name>
          <display-name>
            <localized-text>
              <locale language="en">Task subject</locale>
            </localized-text>
          </display-name>
          <type>STRING</type>
          <required-at-enactment>true</required-at-enactment>
          <process-output>true</process-output>
        </attribute>
        <attribute>
          <attribute-name>participantRoleID</attribute-name>
          <display-name>
            <localized-text>
              <locale language="en">Participant Role ID</locale>
            </localized-text>
          </display-name>
          <type>INT64</type>
          <required-at-enactment>true</required-at-enactment>
          <process-output>true</process-output>
        </attribute>
        <attribute>
          <attribute-name>deadlineDateTime</attribute-name>
          <display-name>
            <localized-text>
              <locale language="en">Deadline date</locale>
            </localized-text>
          </display-name>
          <type>DATETIME</type>
          <required-at-enactment>true</required-at-enactment>
          <process-output>>false</process-output>
        </attribute>
        <attribute>
          <attribute-name>deadlineDuration</attribute-name>
          <display-name>
            <localized-text>
              <locale language="en">Deadline Duration</locale>
            </localized-text>
          </display-name>
          <type>INT32</type>
          <required-at-enactment>>false</required-at-enactment>
          <process-output>>false</process-output>
          <initial-value>300</initial-value>
        </attribute>
        <attribute>
          <attribute-name>priority</attribute-name>
```

**wdos** This is optional (as a workflow process definition does not have to contain any workflow data objects) and contains the details of all the workflow data objects defined for the workflow process definition.

**wdo** This contains the details of one the workflow data objects defined for the workflow process definition. This includes the generic details of the workflow data object itself and also details of each of it's attributes. The metadata that describe a workflow data object and it's attributes are described below:

**is-list-wdo**

This contains a BOOLEAN value which indicates whether the specified workflow data object is a list workflow data object or not. When set to true, the specified workflow data object will act as a list and thus can be used to make lists of data available throughout the workflow.

**initialize-attributes**

This contains a BOOLEAN value which indicates whether the attributes associated with the workflow data object should be initialized when the workflow data object is first used. The default values used are the same as would be set in a Cúram struct.

**wdo-name**

This contains the name of the workflow data object.

**display-name**

This contains the display name of the workflow data object. This name represents a short description of the workflow data object and is displayed throughout the Process Definition Tool. It is a localizable string that does not contain any parameters. For more details on the localized text and associated metadata, see "Localized Text" on page 30.

**description**

This contains a more detailed description of the workflow data object. It is also a localizable string with no parameters. For more details on the localized text and associated metadata, see "Localized Text" on page 30.

**attributes**

This contains the details of all of the attributes associated with the workflow data object.

**attribute**

This contains the details of one of the attributes associated with the workflow data object. The following metadata described below make up a workflow data object attribute:

**attribute-name**

This contains the name of the workflow data object attribute.

**display-name**

This represents the display name of the workflow data object attribute. This name represents a short description of the workflow data object attribute. It is a localizable string that does not contain any parameters. For more details of the localized text and associated metadata, see "Localized Text" on page 30.

**type**

Each workflow data object attribute defined must specify a type which must be a valid Cúram base domain. When creating a workflow data object attribute in the Process Definition Tool this type is selected from the DomainType codetable. This codetable should be consulted to obtain the full list of types available for

workflow data object attributes. The type of a workflow data object attribute is utilized to ensure that the data mappings contained within a workflow process are compatible and will not cause failures at runtime. An example of this would be that if a business process object method parameter field was of type STRING, then the workflow data object attribute used to map the data into that field must also be of type STRING.

#### **required-at-enactment**

Enactment mappings represent the minimum amount of data that the workflow requires in order to be enacted. They must contain an entry for each workflow data object attribute that has its required at enactment flag set to true. Conversely, setting this flag to false (the default) means that this workflow data object attribute is not required for the enactment of the associated process. The Process Definition Tool is used to create these enactment mappings and it does so by examining each workflow data object attribute that has been defined and creating a mapping for those that have the required at enactment flag set to true. When a released workflow process definition has been selected as a subflow process in a subflow activity (see “Subflow” on page 82), all of the workflow data objects that have been marked as required for enactment in the subflow process must be mapped before that parent process definition can be released.

#### **process-output**

A workflow process can be marked as a Web Service by setting a metadata value which indicates that the process should be exposed as a Web Service. This allows the process to be able to participate in a BPEL (Business Process Execution Language) orchestrated process and means that the process can be called from a BPEL process either synchronously or asynchronously. It may also be necessary to map data out from a workflow process back into the BPEL process that called it. When set to true, this optional element indicates that the data from this workflow data object attribute should be passed back to the calling BPEL process when the Cúram workflow process completes. The default for this element is false.

#### **constant-value**

This optional element indicates if the workflow data object attribute represents a constant value. In numerous places throughout a workflow process definition, workflow data object attributes are used in input mappings (i.e. allocation function mappings, deadline function mappings etc.). In some of these cases, it is required to be allowed to use constants in some of these mappings. By providing a constant value, workflow data object attributes of this type may be used for this purpose. A workflow data object attribute cannot have its required for enactment flag set to true and also contain a constant value. Data that is passed in as enactment data is deemed to be dynamic and subject to change. The data specified in a constant workflow data object attribute is not suitable for this purpose as its value is already known.

#### **initial-value**

This element indicates if the workflow data object attribute has an initial value. This feature can be useful in the situations where a workflow data object attribute is used in the workflow before it

has been populated by an automatic activity or otherwise (i.e. to prevent having to use an automatic activity to populate workflow data object attributes just to ensure that these attributes are not null when they are used as part of transition conditions later in the workflow). When this element has been populated, the workflow data object attribute is initialized to the specified value the first time it is used. The initial value of a workflow data object attribute can be overwritten later by the various output mappings that exist in a workflow process. A workflow data object attribute cannot have both a constant value and an initial value specified for it.

## Validations

- A workflow process must contain one and only one `Context_RuntimeInformation` workflow data object.
- A workflow data object name must be unique in the context of the containing workflow process definition.
- The name of a workflow data object must be a valid Java™ identifier.
- A user-defined workflow data object name cannot contain the prefix `Context_` as this is a reserved prefix in the Cúram workflow system.
- Each workflow data object specified in the workflow process definition must contain at least one associated attribute.
- The workflow data object attribute name must be a valid Java identifier.
- A workflow data object attribute cannot be created with the name "value". This is a reserved attribute name in the Cúram workflow system.
- The type of a workflow data object attribute must be a valid Cúram base domain and must be contained in the `DomainType` codetable.
- A workflow data object attribute cannot be both marked as required for enactment and also marked as a constant value.
- A workflow data object attribute cannot have both a constant value and an initial value specified for it.
- If a workflow data object attribute has been marked as a constant, then a constant value must be supplied. Conversely, if the attribute has not been marked as a constant, then no such value should be specified.
- If the workflow data object attribute has been marked as a constant, then a blank value can only be specified for that attribute if the type of the attribute is a `STRING`.
- If the workflow data object attribute has been specified with an initial value, then a blank initial value can only be specified for that attribute if the type of the attribute is a `STRING`.
- If the workflow data object attribute has been marked as a constant, then the value specified as that constant must be compatible with the type of the associated attribute.
- If the workflow data object attribute has been specified with an initial value, then the value specified as that initial value must be compatible with the type of the associated attribute.
- The process output flag can only be set to true for a specified workflow data object attribute if the associated workflow process has been exposed as a webservice.

## List of Context WDOs

Context workflow data objects are those that are not explicitly defined in the workflow process definition metadata but are made available by the Process Definition Tool and workflow engine at various places during the execution of a process. The following is a brief description of these context workflow data objects and links are provided to where further information may be found about them.

### **Context\_RuntimeInformation Workflow Data Object**

The Context\_RuntimeInformation workflow data object is a workflow data object that is made available and maintained by the workflow engine. It contains information that is pertinent throughout the life-cycle of a workflow process instance and the attributes available reflect this. These attributes are as follows:

- `processInstanceID` : The system generated identifier of the process instance (taken from the Cúram key server using the workflow key set).
- `enactingUser` : The username of the user whose actions in the application resulted in the workflow process being enacted.
- `enactmentTime` : The date and time at which the process was enacted.

### **Context\_Result Workflow Data Object**

A transition from an automatic activity should be able to use the return value of the invoked method in it's condition directly without the need for mappings to workflow data object attributes. However due to the transactional model of the workflow engine this data has to persist outside the transaction of the business process object method invocation. In order to achieve this, a workflow data object definition is created at runtime if the return value is used in outbound transition conditions. These return value definitions never need to be persisted as they are inferred wherever needed in the workflow engine. The actual workflow data object data is persisted until after the transitions from the activity instances in question have been evaluated, at which point they are deleted. For more details on the Context\_Result workflow data object, see "Description of Context WDOs" on page 43

### **Context\_Event Workflow Data Object**

The Context\_Event workflow data object is available for use in a data item or function conditions (see "Conditions" on page 103) for a transition from an activity containing an event wait. It makes available certain information (e.g. the event class and event type of the event raised, the time the event was raised etc.) contained in the event raised to complete that activity instance. This information can then be used to model the path from that specified activity. For more details on the Context\_Event workflow data object, see "Description of Context WDOs" on page 53.

### **Context\_Decision Workflow Data Object**

The Context\_Decision workflow data object is available for use in a data item or function condition (see "Conditions" on page 103) for a transition from a decision activity. The attributes available will depend on the answer format defined for the decision activity. For more details on the Context\_Decision workflow data object, see "Description of Context WDOs" on page 81

### **Context\_Task Workflow Data Object**

The Context\_Task workflow data object is available for use in various mappings associated with a manual activity task (e.g. Allocation Function Input mappings, Deadline Function Input mappings, Manual Activity Action Link parameters). This context workflow data object makes

available the identifier of the task created as a result of the execution of the containing activity. For more details on the Context\_Task workflow data object, see “Description of Context WDOs” on page 62.

#### **Context\_Loop Workflow Data Object**

The Context\_Loop workflow data object is available for use when creating the loop condition associated with a loop-begin activity. It is also available for creating outgoing transition conditions for any activity within a loop, and for when specifying input mappings, text parameters and action link parameters for some activities and functions contained within a loop. This context workflow data object makes the number of times that a loop has been iterated over available for such mappings. For more details on the Context\_Loop workflow data object, see “Description of Context WDOs” on page 88.

#### **Context\_Deadline Workflow Data Object**

The Context\_Deadline workflow data object is available for use when creating a data item or function condition (see “Conditions” on page 103) for a transition from an activity that has an event wait with a deadline specified for it. It is available to allow a developer to model different paths of execution from an activity containing a deadline depending on whether that deadline has expired. For more details on the Context\_Deadline workflow data object, see “Description of Context WDOs” on page 51.

#### **Context\_Parallel Workflow Data Object**

The Context\_Parallel workflow data object is available for use in the various mappings associated with a parallel manual activity (e.g. task subject and task action text parameters, allocation strategy mappings etc.) and a parallel decision activity (e.g. decision action text parameters, secondary action text parameters, question text parameters etc.). It makes available the index of the item from the Parallel Activity List Workflow Data Object that is used to create the specified instance of the wrapped activity. For more details on the Context\_Parallel workflow data object, see “Description of Context WDOs” on page 92.

#### **Context\_Error Workflow Data Object**

The Context\_Error workflow data object is available for use in a data item or function condition (see “Conditions” on page 103) for a transition from an automatic activity. It allows a process developer to model an exception path out of an automatic activity i.e. a transition that is followed if the automatic activity fails due to an un-handled exception. For more details on the Context\_Error workflow data object, see “Description of Context WDOs” on page 43

## **Runtime Information**

Instances of workflow data objects and list workflow data objects exist as soon as a workflow process is enacted and exist until the process completes. These workflow data object instances are thus available to be used in the activities (e.g. pass data to a BPO method) and the transitions (e.g. make data available in the evaluation of transition conditions) throughout the lifetime of that process instance.

The `enactingUser` attribute of the `Context_RuntimeInformation` Workflow Data Object is set to the username of the user whose actions in the application resulted in the workflow process being enacted. This does *not* result in the same value being assigned to the transaction when a BPO method is subsequently invoked in the workflow process instance. This is due to the transaction demarcation in the workflow engine when automatic activities (i.e. BPO methods) are invoked in the

application server. Due to the asynchronous nature of this invocation and the requirement to ensure that the call to the application code is in its own transaction, the BPO method is invoked by the workflow engine (SYSTEM user) rather than the user who enacted the workflow process in the first place. Indeed in a real business sense, the person who enacted the workflow may not even know they have invoked that BPO method.

In a similar fashion, it should be noted that the enacting user of a workflow process instance is not passed into any of the subflow process instances that may be invoked from the parent process. If the enacting user of the parent process instance is required in any of the subflow process instances, it should be passed explicitly using a workflow data object attribute in the input mappings for that subflow process.

Care should also be taken when updating workflow data object attribute instance data when executing parallel automatic activities in a workflow process instance. If such automatic activities invoke the same BPO method and that method attempts to update the data for the exact same workflow data object attribute, then a database record deadlock situation may occur. The workflow process designer should alleviate such situations occurring by designing the workflow process definition to ensure automatic activities executed in parallel do not update the same workflow data object attribute.

---

## Process Enactment

### Overview

A process definition defines the structure of a business process and to start performing the work defined in that process definition an instance of the process must be created. The starting of a process instance is referred to as *process enactment*. Most process definitions require a minimum set of initial data which is used primarily to identify the specific business objects the process instance will operate on. All enactment mechanisms must have a way to accept the input data for starting a given process. This input data is known as the *enactment data* for a process.

Currently there are four enactment mechanisms supported by Cúram workflow:

- Enactment from code
- Enactment from an event
- Enactment as a subflow
- Enactment via a web service

The first two mechanisms are described in this chapter. The sub-flow enactment mechanism is described in “Subflow” on page 82. The web service enactment mechanism is described in “Workflow Web Services” on page 114.

### Code enactment (enactment service API)

The most direct way of enacting a process is by identifying a location in the application from which a process instance must be started. Code must then be inserted at that point to call the enactment service API. This API allows the developer to specify the name of the process to start and to supply the enactment data required by the process.

While enacting a process in this way is simple and intuitive, it does have the drawback of being hard coded in the application logic. This being the case, alterations such as removing the enactment, changing the process to start or indeed even minor changes to the required enactment data will require code changes and redeployment of the application.

## Metadata

```
<enactment-mappings>
  <mapping>
    <source-attribute
      struct-name="curam.core.sl.struct.TaskCreateDetails"
      name="subject" />
    <target-attribute
      wdo-name="TaskCreateDetails"
      name="subject" />
  </mapping>
  <mapping>
    <source-attribute
      struct-name="curam.core.sl.struct.GroupMemberDetails"
      name="dtls.memberName" />
    <target-attribute
      wdo-name="MemberCreateDetails"
      name="memberName" />
  </mapping>
  <mapping>
    <source-attribute
      struct-name="curam.core.sl.struct.ChildDetailsList"
      name="dtls.identifier" />
    <target-attribute
      wdo-name="ChildDetails"
      name="identifier" />
  </mapping>
  ...
</enactment-mappings>
```

### **enactment-mappings**

Contains a list of mappings that can be used as initial data in enacting the associated process instance. A process definition is not required to have enactment mappings defined in order for it to be enacted.

### **mapping**

A mapping represents a data item supplied from a Cúram struct attribute to be used in enacting the associated process instance.

### **source-attribute**

This represents a Cúram struct attribute to be used in populating the enactment data for the process and is mandatory in an enactment mapping.

### **struct-name**

The name of a Cúram struct that contains an attribute required to enact the workflow process. Aggregated and list structs may also be used to pass enactment data into a workflow process, as illustrated in the metadata snippet above.

**name** The name of the attribute of a Cúram struct required to enact the associated workflow process. Where a field from an aggregated struct or list struct is being used, this name represents the fully qualified name of that field. In such a case, the name consists of

the role name from the association between the parent and child struct in addition to the actual field name. This is illustrated in the metadata snippet above.

**target-attribute**

This represents a workflow data object attribute which is to be populated with enactment data for the process and is mandatory in an enactment mapping.

**wdo-name**

The name of a Cúram workflow data object containing the target attribute to be mapped. (See “Workflow Data Objects” on page 16).

**name** The name of a Cúram workflow data object attribute that is marked as being required for enactment. The value of the corresponding Cúram struct source attribute will be mapped to this attribute when the process is enacted.

**Validations**

- The Cúram struct attribute used as a source attribute in an enactment mapping must be valid and be of the correct type for the associated target workflow data object attribute.
- The target workflow data object attribute in an enactment mapping must be valid and must be marked as being required for enactment.
- If the target attribute of the enactment mapping is from a list workflow data object, then the source attribute must be a field from a list struct.

## Code

```
// Create the list we will pass to the enactment service.
final List enactmentStructs = new ArrayList();

final TaskCreateDetails taskCreateDetails =
    new TaskCreateDetails();

taskCreateDetails.subject = "The subject of a Task";
taskCreateDetails.reservedBy = "someUser";

enactmentStructs.add(taskCreateDetailsStruct);

// An aggregated struct.
GroupMemberDetails groupMemberDetails
    = new GroupMemberDetails();

groupMemberDetails.dtls.memberName = "Test User";

enactmentStructs.add(groupMemberDetails);

// A list struct.
ChildDetailsList childDetailsList
    = new ChildDetailsList();

ChildDetails recordOne = new ChildDetails();
recordOne.identifier = 1;
childDetailsList.dtls.add(recordOne);

ChildDetails recordTwo = new ChildDetails();
recordTwo.identifier = 2;
childDetailsList.dtls.add(recordTwo);

enactmentStructs.add(childDetailsList);

EnactmentService.startProcess(
    "TASKCREATEWORKFLOW", enactmentStructs);
```

- The EnactmentService API is provided to allow for the enacting of workflow processes from application code. The list of Cúram structs provided to the startProcess() method must be sufficient to fully populate the enactment mappings of the associated process. Note that enacting a process in this way is asynchronous and the process will get kicked off once the current application transaction completes.
- The startProcessInV3CompatibilityMode method is provided for the use of the core application Task API only. Direct use of this method in custom code is not supported and may hamper future upgrades.

## Event enactment

It is possible to start a process in response to an event being raised. This requires the setup of some configuration data (either through an administration interface or as pre-configured database entries). The configuration specifies the process/processes to start in response to a specific event being raised. Mappings of event data to the enactment data required by the process can also be configured in this way.

Process enactment event configuration is stored on the database and a user interface is supplied to allow the manipulation of this data. As such process enactment created in this way can be enabled, disabled, changed and even removed at runtime. The main drawback of this approach is that since events have

a finite amount of information, only process definitions that require such a small amount of enactment data can be enacted in this way.

A Process Enactment Event Handler is supplied with Cúram and is automatically registered to listen for events raised in the application. Where a process has been configured to be enacted from an event, the data from the event is mapped into the enactment data of the process, and the process is started.

### Configuration data

Enabling an event to enact a process requires an event-process association to be configured. Every event raised in the application checks to see if any processes have been associated and are required to be enacted. The latest released version of a process will always be enacted for an associated event.

The registration of an event to trigger a process is stored as a record on the ProcEnactmentEvt table. The process enactment event handler searches a cached representation of this table for matching entries when an event is raised in the application and enacts any matching processes. The following table describes the data required to populate the ProcEnactmentEvt table.

*Table 1. Description of the ProcEnactmentEvt Table*

| Entity Field Name | Description of Field   |
|-------------------|--|
| procStartEventID  | The unique identifier of the event-process association.  |
| eventClass        | The event class of the event that has been specified to enact the workflow process.  |
| eventType         | The event type of the event that has been specified to enact the workflow process.   |
| processToStart    | If an event containing the specified event class and type describe above is raised, the latest released version of the workflow process specified by this name will be enacted.  |
| enabled           | This boolean flag indicates if the event-process association is enabled. This allows the enactment of a workflow process by a specified event to be enabled/disabled at runtime. |

The ProcEnactEvtData table stores the data to be mapped from a business event to a the workflow being enacted when that specified event is raised. The following table describes the data required to populate the ProcEnactEvtData table.

*Table 2. Description of the ProcEnactEvtData Table*

| Entity Field Name  | Description of Field   |
|--------------------|--|
| procEventMappingID | The unique identifier of the process enactment event data mapping. |

Table 2. Description of the ProcEnactEvtData Table (continued)

| Entity Field Name | Description of Field   |
|-------------------|--|
| procStartEventID  | The unique identifier of the event-process association. This field is the unique key on the associated ProcEnactmentEvt table and is used to associate all of the data required to enact the workflow process when a specified event is raised.  |
| eventField        | This indicates which of the three fields of an event will be used to populate the workflow data object attribute. The values for this field are taken from the EventField codetable and are described in more detail below.  |
| wdoAttribute      | The fully qualified name of a workflow data object attribute to populate with data from the given event field when a process is enacted. This table will include an entry for each workflow data object attribute that has been marked as required for enactment in the process being enacted by the raised event. |

There are three fields of an event may be used as enactment mappings. These are enumerated in the EventField codetable and are described below.

#### primary event data

A unique identifier related to the event class from which the event is raised. For example, where the business object type specified for an event is equal to 'Case', the event data could be case identifier.

#### secondary event data

This can be any numeric value and is intended for events that must represent an association between two entities.

#### raised by user

The Cúram username of the user who raised the event.

### Validations

- The data available from an event must be sufficient to fully populate the enactment data for the associated process definition.
- Where a process has already been configured for event-based enactment, subsequent modifications to the processes enactment data must satisfy the existing event data mappings.
- Where a process has been configured to be enacted from an event, it cannot have its latest released version deleted if the next latest released version is unable to have its enactment data fully populated from the event.

---

## Base Activity

### Overview

All the activity types supported by Cúram workflow have some base details in common. This information allows them to be uniquely identified by the workflow engine and displayed both textually and graphically in the Process Definition Tool. Every activity has a name and an optional description, both of which are localizable. This allows various administration user interfaces to display the information in the appropriate locale.

This base level uniformity allows activities to be identified and executed by the workflow engine without the knowing the specific type of the activity. Each activity type knows it's own metadata and how to behave when executed. This arrangement will allow the addition of new activity types, if required, without affecting the core behavior of the workflow engine.

### Metadata

```
<automatic-activity id="1" category="AC1">
  <name>
    <localized-text>
      <locale language="en">ApproveCase</locale>
    </localized-text>
  </name>
  <description>
    <localized-text>
      <locale language="en">This automatic activity
        will be executed to approve a case.</locale>
    </localized-text>
  </description>
  ...
</automatic-activity>
```

**id** This is a 64-bit identifier supplied by the Cúram key server when activities are created in the process definition tool. The activity identifier is required to be unique within a process definition but global uniqueness within all of the process definitions on the system is not required.

**category**

An activity can optionally be placed into a category. The category must be selected in the Process Definition Tool and is taken from the ActivityCategory code-table. This attribute is intended to be used for searching functionality based on activities and has no functional effect on the activity.

**name** The name of the activity is the means by which the activity is identified for the purpose of display. This is in contrast to the activity identifier which is used to identify the activity for the purpose of execution by the workflow engine.

**description**

An activity can also have an optional description that briefly specifies what the activity does for the benefit of those editing the process definition in the future.

## Localized Text

As shown in the XML fragment above, the activity name and description are not just text fields, but are defined in terms of a `localized-text` element. This is general purpose element used throughout the process definition metadata where ever text is required to be localizable.

A valid `localized-text` element must have at least one `locale` child element. With the exception of the localization screen, any localizable text that is entered in the process definition tool is saved under both the user's and the English (i.e. "en") locales.

```
<localized-text>
  <locale language="en">ApproveCase</locale>
  <locale language="en" country="US">ApproveCase</locale>
  <locale language="fr">ApprouverAffaire</locale>
  <locale language="fr" country="CA">ApprouverAffaire</locale>
</localized-text>
```

**locale** This contains the text for the locale specified by the language and country attributes. Note: A locale is uniquely identified by both the language and the country meaning that *en*, *en\_US* and *en\_GB* all represent different locales.

### language

This is mandatory and is the two letter ISO language code.

### country

This is optional and is the two letter ISO country code.

## Validations

- The activity name is mandatory and must be unique within a specified workflow process definition. However, the activity name is also a localizable string. This validation also ensures that a specified activity name is also unique for each locale specified.
- An activity must be one of the permitted activity types. In practice this rule is self-satisfying as there is no way to create activities without selecting an appropriate type in the process definition tool. Even when constructing process definitions manually in a text editor, the activity type names correspond to the metadata element names making it impossible to create valid markup that represents a nonexistent activity type.

## Basic Activity Types

Some activity types namely route, start-process and end-process activities have no additional metadata other than that common to all activity types. Their behavior is also sufficiently intuitive to be described here. All of the other activity types have dedicated chapters.

### Route Activity

A route activity is an activity that performs no business functionality. It can be considered a null activity as its execution does not affect the application data nor the business process in any way.

The primary purpose of the route activity is to assist in flow control. Route activities are often used as branch (split) and synchronization (join) points. They are also useful when the activities required by a business process do not naturally form a valid block structure that the workflow engine can execute.

Since all activity types can have notifications associated with them (see: “Activity Notifications” on page 92), route activities can be used to provide the effect of a pure notification that is not connected to any other functionality.

### **Start/End Process Activity**

The start-process and end-process activities provide markers for the beginning and end of a process. They are anchor points to which other activities can be attached using transitions thus creating a series of steps from the start to the end of the process. In a valid process definition traversing all the transitions between activities starting from the start-process activity should lead to end-process activity (note that in a running process instance not all paths will necessarily be traversed, for example if a split (see “Split/Join” on page 107) is encountered only some of the paths may actually be followed depending on the evaluation of transition conditions). As such the simplest (and incidentally the most useless) process definition is one that contains only these two activities and a transition from the start-process to the end-process activity.

Every process definition must have exactly one start-process and exactly one end-process activity. When defining a process using the Process Definition Tool these two activities are created automatically on process creation and are not required to be (in fact cannot be) explicitly created by the user.

The start-process and end-process activities form the outermost block of a validly block structured process definition as required by Cúram workflow.

---

## **Automatic**

### **Prerequisites**

- The base details common to all the activity types supported by Cúram workflow are described in “Base Activity” on page 29 and are applicable to the automatic activity described here.

### **Overview**

An automatic activity is a step in a workflow process that is wholly automated and under normal circumstances no human intervention is required for the completion of such a step. An automatic activity step invokes a method in the application to perform some processing required as part of the overall business process. Typical uses for automatic activities include: performing calculations, updating entities in the application and pulling data into the workflow engine.

### **Cúram Business Methods**

Much of the processing for an automatic activity is performed in the application code that is invoked. Automatic activities do their work by invoking Cúram business methods (both BPO (business process object) and entity methods are supported). Technically these are public methods on Cúram business process objects and entities. A critical part of the automatic activity definition is the method to invoke and the parameters to pass to it. The following sections describe these.

## Metadata

```
<automatic-activity id="1" category="AC1">
  ...
  <bpo-mapping
    interface-name="curam.sample.facade.intf.SampleBenefit"
    method-name="createAssociatedProductDeliveryForPlannedItem">
    <formal-parameters>
      ...
    </formal-parameters>
  </bpo-mapping>
</automatic-activity>
```

### **bpo-mapping**

This contains the details of the Cúram business method that will be invoked when the associated automatic activity is executed. These details include the name of the interface and associated method and also any input and return mappings associated with the method being invoked. The input and output mappings are described in the following sections. The mandatory attributes of a business process object (BPO) mapping are described below.

#### **interface-name**

This represents the fully qualified name of the Cúram interface containing the method associated with the automatic activity.

#### **method-name**

This represents the method on the specified Cúram interface that will be invoked when the automatic activity is executed.

## **Validations**

- Both the interface and method names must be specified for the automatic activity business process object method mapping.
- The interface name specified must be a valid class and this class must exist on the Cúram application classpath.
- The method name must be a valid method name and must exist on the specified interface.

## **Code**

As stated previously, any valid public Cúram business method (BPO or entity) may be associated with an automatic activity in a workflow process and hence be invoked when that activity is executed. In general, a failure of such a method when an automatic activity is executed will cause the Workflow Error Handling strategy to be invoked. This may cause, for example, the activity associated with the failed method to be retried a number of times. Based on this fact, the methods associated with automatic activities should in general not throw exceptions. If the modeled exceptions feature is being used, then when a BPO method throws an exception and has been retried the required number of times, all of the transitions from the automatic activity that contain the Context\_Error workflow data object are evaluated. If any of these transitions evaluate to true, their paths are followed and in this way, remedial processing may take place after the automatic activity BPO method failed.

## Input Mappings

There must be a way to supply the parameters required by a method in order to invoke it in the workflow engine. The workflow engine has a pool of data at its disposal in the form of workflow data objects (see “Workflow Data Objects” on page 16). Input mappings are used to declare which workflow data object attributes will be used to populate the values of the specific method parameters when the method is invoked. Input mappings are optional where struct fields have been specified as method parameters. However, primitive base type parameters must be mapped.

### Metadata

The following metadata is common to all three types of parameter input mappings (base type, struct and aggregated structs) and hence will not be described again.

#### formal-parameters

This contains the list of formal parameters as defined in the automatic activity business method signature.

#### formal-parameter

This contains the details of one formal parameter input mapping as defined in the associated business method signature. In this instance, a formal parameter mapping entry will exist for each parameter defined in the associated business method.

**index** This represents the position of the formal parameter in the list of formal parameters defined for the specified method. It is a zero-based index.

**Input mappings for base type parameters:** Base type parameters provide the simplest type of input mapping. In this instance, input mappings are created for each base type formal parameter contained in the business method associated with the automatic activity. A base type parameter in a Cúram business method represents a domain definition (see the *Cúram Modeling Reference Guide* for details on domain definitions).

```
<automatic-activity id="1" category="AC1">
  ...
  <bpo-mapping
    interface-name="curam.sample.facade.intf.SampleBenefit"
    method-name="createDelivery">
    <formal-parameters>
      <formal-parameter index="0">
        <base-type type="STRING">
          <wdo-attribute wdo-name="SPPProductDeliveryPI"
            name="description"/>
        </base-type>
      </formal-parameter>
      <formal-parameter index="1">
        <base-type type="INT64">
          <wdo-attribute wdo-name="SPPProductDeliveryPI"
            name="plannedItemID"/>
        </base-type>
      </formal-parameter>
    </formal-parameters>
  </bpo-mapping>
</automatic-activity>
```

#### base-type

This contains the details of one base type input mapping. A base type mapping indicates that the field being mapped to is primitive (unlike the

struct and nested struct mappings described below). A base type input mapping contains the following mandatory attribute:

**type** This describes the type of the primitive field being mapped to. For a base type input mapping, this is the type of the domain definition specified as the formal parameter in the method.

#### **wdo-attribute**

This contains the details of the workflow data object (see “Workflow Data Objects” on page 16) attribute containing the data that will be used to populate the associated base type parameter when the automatic activity business method is invoked. The mandatory attributes are described below:

#### **wdo-name**

This describes the name of the workflow data object used in the input mapping.

**name** This describes the name of the attribute on the specified workflow data object used in the input mapping.

**Input mappings for struct parameters:** Structs may be specified as parameters to business process object methods. This section describes the metadata of the input mappings associated with such parameters.

```
<automatic-activity id="1" category="AC1">
  ...
  <bpo-mapping
    interface-name="curam.sample.facade.intf.SampleBenefit"
    method-name="createAssociatedProductDeliveryForPlannedItem">
    <formal-parameters>
      <formal-parameter index="0">
        <struct
          type="curam.struct.SampleBenefitPlanItemDetails">
            <field name="description">
              <base-type type="STRING">
                <wdo-attribute wdo-name="SPProductDeliveryPI"
                  name="description"/>
              </base-type>
            </field>
            <field name="plannedItemIDKey">
              <base-type type="INT64">
                <wdo-attribute wdo-name="SPProductDeliveryPI"
                  name="plannedItemID"/>
              </base-type>
            </field>
            <field name="plannedItemName">
              <base-type type="STRING" />
            </field>
          </struct>
        </formal-parameter>
      </formal-parameters>
    </bpo-mapping>
  </automatic-activity>
```

**struct** This contains the details of one struct input mapping, including the type of the struct and mappings for each field defined in that struct. A struct input mapping contains the following mandatory attribute:

**type** This describes the type of the struct that has been specified as the formal parameter in the method. This is represented as the fully qualified name of the struct specified as the formal parameter.

**field** This contains the details of the input mapping for one of the fields defined

in the struct parameter. A field contains the details of the input mapping for the primitive base type associated with that field as well as the following mandatory attribute:

**name** This describes the name of the field as defined in the struct specified as the formal parameter.

**base-type**

This contains the details of one base type input mapping for the specified field. A base type input mapping contains the following mandatory attribute:

**type** This describes the type of the primitive field being mapped to.

**wdo-attribute**

This contains the details of the workflow data object (see “Workflow Data Objects” on page 16) attribute containing the data that will be used to populate the associated base type field when the method is invoked. This will not be present if the user has not specified an input mapping for this method parameter. This element, when specified, contains the following mandatory attributes:

**wdo-name**

This describes the name of the workflow data object used in the input mapping.

**name** This describes the name of the attribute on the specified workflow data object used in the input mapping.

**Input mappings for aggregated struct parameters:** Aggregated structs (see the *Cúram Modeling Reference Guide* for details on struct aggregation) may be specified as parameters to business methods. In this instance, the metadata is similar to that described above for struct formal parameters (see “Input mappings for struct parameters” on page 34). The subtle difference is, however, that a field in the struct parameter defined may resolve down to another struct and not to a primitive type as seen in the struct mappings example. In this scenario, the field name is not the name of the field being mapped associated with the struct parameter but is the name of the role contained in the association between the specified struct and the struct it aggregates. The following metadata snippet provides an example of such input mappings. The metadata elements have been previously described above in the struct input mappings section.

```

<automatic-activity id="1" category="AC1">
  ...
  <bpo-mapping
    interface-name="curam.sample.facade.intf.SampleBenefit"
    method-name="createBenefit">
    <formal-parameters>
      <formal-parameter index="0">
        <struct type="curam.struct.PlannedItemDetails">
          <field name="description">
            <base-type type="STRING">
              <wdo-attribute wdo-name="SPPProductDeliveryPI"
                name="description"/>
            </base-type>
          </field>
          <field name="plannedItemID">
            <base-type type="INT64">
              <wdo-attribute wdo-name="SPPProductDeliveryPI"
                name="plannedItemID"/>
            </base-type>
          </field>
          <field name="dtls">
            <struct type="curam.struct.PlannedItemKey">
              <field name="subject">
                <base-type type="STRING">
                  <wdo-attribute wdo-name="SPPProductDeliveryPI"
                    name="subject"/>
                </base-type>
              </field>
              <field name="concernRoleID">
                <base-type type="INT64">
                  <wdo-attribute wdo-name="SPPProductDeliveryPI"
                    name="concernRoleID"/>
                </base-type>
              </field>
            </struct>
          </field>
        </struct>
      </formal-parameter>
    </formal-parameters>
  </bpo-mapping>
</automatic-activity>

```

**Input mappings for list struct parameters:** Input mappings for list structure parameters may now also be specified. In this instance, the metadata is similar to that described above for aggregate formal parameters (see “Input mappings for aggregated struct parameters” on page 35). The type of the struct specified in the metadata for a list struct parameter is the name of the list structure. The name of the first field specifies the name of the role contained in the association between the specified list struct and the child struct it aggregates. Typically, this field then resolves down to another struct (the child struct contained within the list struct). The workflow data object specified in such a mapping is a list workflow data object. The following metadata snippet provides an example of such input mappings. The metadata elements have been previously described above in the struct input mappings section.

```

<automatic-activity id="1" category="AC1">
...
<bpo-mapping
  interface-name="curam.sample.facade.intf.SampleBenefit"
  method-name="processClaimantDependents">
  <formal-parameters>
    <formal-parameter index="0">
      <struct type="curam.sample.struct.
        ClaimantDependentDetailsList">
        <field name="dtls">
          <struct type="curam.sample.struct.
            ClaimantDependentDetails">
            <field name="identifier">
              <base-type type="INT64">
                <wdo-attribute wdo-name="ClaimantDependent"
                  name="identifier"/>
              </base-type>
            </field>
            <field name="firstName">
              <base-type type="STRING">
                <wdo-attribute wdo-name="ClaimantDependent"
                  name="firstName"/>
              </base-type>
            </field>
            <field name="surname">
              <base-type type="STRING">
                <wdo-attribute wdo-name="ClaimantDependent"
                  name="surname"/>
              </base-type>
            </field>
          </struct>
        </field>
      </struct>
    </formal-parameter>
  </formal-parameters>
</bpo-mapping>
</automatic-activity>

```

**Input mappings and indexed items from list workflow data objects:** For activities contained within loops, an item from a list workflow data object can be used in an input mapping to populate a formal parameter field. When this type of input mapping is used, each time the loop containing the activity is iterated over, the formal parameter field will be populated with the next value from that list workflow data object. This is highlighted here as the metadata syntax for such a mapping is subtly different than that of the other input mapping types. The metadata snippet provides an example of such input mappings. The name of the list workflow data object used to populate the formal parameter field is qualified with the [Context\_Loop.loopCount] syntax. This is used by the workflow engine at runtime to determine which iteration of the loop is being executed and hence which item from the list workflow data object to retrieve the data to populate the formal parameter field with.

```

<automatic-activity id="1" category="AC1">
...
<bpo-mapping
  interface-name="curam.sample.facade.intf.SampleBenefit"
  method-name="retrieveClaimantDependentDetails">
  <formal-parameters>
    <formal-parameter index="0">
      <struct type="curam.sample.struct.
        ClaimantDependentDetails">
        <field name="identifier">
          <base-type type="INT64">
            <wdo-attribute name="identifier"
              wdo-name=
                "ClaimantDependent[Context_Loop.loopCount]"/>
          </base-type>
        </field>
        <field name="fullName">
          <base-type type="STRING">
            <wdo-attribute name="fullName"
              wdo-name=
                "ClaimantDependent[Context_Loop.loopCount]"/>
          </base-type>
        </field>
      </struct>
    </formal-parameter>
  </formal-parameters>
</bpo-mapping>
</automatic-activity>

```

## Validations

- The workflow data object attributes specified in the input mappings must be valid. The criteria that defines a valid workflow data object attribute may be seen in “Validations” on page 20
- The type of the formal parameter being mapped to and the type of the workflow data object attribute being used in that input mapping must be compatible. For example, if the input mapping being created is a struct field that has a type of STRING, then the workflow data object attribute being used for that mapping must also be of type STRING.
- The Context\_Task workflow data object cannot be used in an input mapping if the associated activity is not a manual or decision activity.
- The Context\_Loop workflow data object cannot be used in an input mapping if the associated activity is not contained within a loop.
- A validation warning will be displayed if all struct parameters defined in the business process object method do not contain an associated input mapping.
- All primitive base type formal parameters defined in the business process object method which must contain an associated input mapping.
- If the formal parameter field being mapped is a base type parameter, then an attribute from a list workflow data object cannot be used.
- If the formal parameter field being mapped is from a list structure, then it must be mapped to an attribute from a list workflow data object.
- If the indexed item from a list workflow data object (i.e. ClaimantDependent[Context\_Loop.loopCount]) is being used in an input mapping, then the associated workflow data object must be a list workflow data object and the activity containing the input mappings must be contained within a loop.

## Runtime Information

The values of the workflow data object attributes defined in the input parameter mappings are provided as input data to the specified method before it is invoked when the associated automatic activity is executed.

## Output Mappings

Workflow data objects (see “Workflow Data Objects” on page 16) are the workflow engines data store. Some of the attributes on the specified workflow data objects are populated when the process is enacted. It is useful, however, to update or set the values of workflow data object attributes as the workflow process is executed. To support this, some activity types can map data back into the workflow engine. This is particularly useful for automatic activities as the business methods they invoke could conceivably access data stored on any entity in the application and return it for use in subsequent activities in the workflow process. These return mappings from a business process object method associated with an automatic activity are optional.

### Metadata

In a similar fashion to input mappings (see “Input Mappings” on page 33), output mappings are supported for primitive return types, struct return types, nested (aggregated) struct return types and list struct return types. If the return type is a primitive type, one return mapping entry may be specified. If the return type is a struct, an aggregated struct or a list struct, return mappings for one or more of the fields in the specified struct may be created. The following metadata snippets provide examples of such mappings:

#### Primitive return type:

```
<automatic-activity id="1" category="AC1">
  ...
  <bpo-mapping
    interface-name="curam.sample.facade.intf.SampleBenefit"
    method-name="createAssociatedProductDeliveryForPlannedItem">
    <formal-parameters>
      <formal-parameter index="0">
        ...
      </formal-parameter>
    </formal-parameters>
    <return>
      <base-type>
        <wdo-attribute wdo-name="SPPProductDeliveryPI"
          name="plannedItemID"/>
      </base-type>
    </return>
  </bpo-mapping>
</automatic-activity>
```

### Struct return type:

```
<automatic-activity id="1" category="AC1">
  ...
  <bpo-mapping
    interface-name="curam.sample.facade.intf.SampleBenefit"
    method-name="createAssociatedProductDeliveryForPlannedItem">
    <formal-parameters>
      <formal-parameter index="0">
        ...
      </formal-parameter>
    </formal-parameters>
    <return>
      <struct>
        <field name="description">
          <base-type>
            <wdo-attribute wdo-name="SPPProductDeliveryPI"
              name="description"/>
          </base-type>
        </field>
        <field name="subject">
          <base-type>
            <wdo-attribute wdo-name="SPPProductDeliveryPI"
              name="subject"/>
          </base-type>
        </field>
      </struct>
    </return>
  </bpo-mapping>
</automatic-activity>
```

### Aggregated struct return type:

```
<automatic-activity id="1" category="AC1">
...
<bpo-mapping
  interface-name="curam.sample.facade.intf.SampleBenefit"
  method-name="createAssociatedProductDeliveryForPlannedItem">
  <formal-parameters>
    <formal-parameter index="0">
      ...
    </formal-parameter>
  </formal-parameters>
  <return>
    <struct>
      <field name="description">
        <base-type>
          <wdo-attribute wdo-name="SPProductDeliveryPI"
            name="description"/>
        </base-type>
      </field>
      <field name="subject">
        <base-type>
          <wdo-attribute wdo-name="SPProductDeliveryPI"
            name="subject"/>
        </base-type>
      </field>
      <field name="dtls">
        <struct>
          <field name="concernRoleID">
            <base-type>
              <wdo-attribute wdo-name="SPProductDeliveryPI"
                name="concernRoleID"/>
            </base-type>
          </field>
          <field name="participantID">
            <base-type>
              <wdo-attribute wdo-name="SPProductDeliveryPI"
                name="participantID"/>
            </base-type>
          </field>
        </struct>
      </field>
    </struct>
  </return>
</bpo-mapping>
</automatic-activity>
```

## List struct return type:

```
<automatic-activity id="1" category="AC1">
...
<bpo-mapping
  interface-name="curam.sample.facade.intf.SampleBenefit"
  method-name="readClaimantDependentDetails">
  <formal-parameters>
    <formal-parameter index="0">
      ...
    </formal-parameter>
  </formal-parameters>
  <return>
    <struct>
      <field name="dtls">
        <struct>
          <field name="identifier">
            <base-type>
              <wdo-attribute wdo-name="ClaimantDependent"
                name="identifier"/>
            </base-type>
          </field>
          <field name="firstName">
            <base-type>
              <wdo-attribute wdo-name="ClaimantDependent"
                name="firstName"/>
            </base-type>
          </field>
          <field name="surname">
            <base-type>
              <wdo-attribute wdo-name="ClaimantDependent"
                name="surname"/>
            </base-type>
          </field>
        </struct>
      </field>
    </struct>
  </return>
</bpo-mapping>
</automatic-activity>
```

**return** This contains the details of the output mappings specified for the business method associated with the automatic activity. For a primitive return type, one entry of the base type metadata will be present as shown in the example above (see “Primitive return type” on page 39). For a struct, aggregated struct and list struct return types, the struct metadata tag is specified and contains fields whose base types are mapped using workflow data object attributes.

**struct** This contains the details of the struct output mapping. A struct output mapping contains the following mandatory attribute.

**field** This contains the details of the output mapping for one of the fields defined in the struct return type. A field contains the details of the output mapping for the primitive base type associated with that field as well as the following mandatory attribute:

**name** This represents the name of the field as defined in the struct specified as the return type. For non-aggregated struct return types, this simply represents the name of the field on the specified return struct that is being mapped. For aggregated struct and list struct return types, the field name represents the name of the role contained in the association between the specified struct and the struct it aggregates.

**base-type**

This contains the details of one base type output mapping for the specified field or a primitive return type.

**wdo-attribute**

This contains the details of the workflow data object (see “Workflow Data Objects” on page 16) attribute that the data present in the associated return type field will be mapped into and persisted. The mandatory attributes are described below:

**wdo-name**

This represents the name of the workflow data object used in the output mapping.

**name** This represents the name of the workflow data object attribute used in the output mapping.

**Validations**

- No duplicate output parameter mappings are allowed. In other words, a workflow data object attribute can only be specified once in any list of output return mappings.
- All of the workflow data object attributes specified in the output mappings must be valid workflow data object attributes in the context of the containing workflow process definition.
- The type of the return field being mapped from and the type of the workflow data object attribute being mapped to must be compatible.
- Output mappings cannot be created for workflow data object attributes that have been marked as constant workflow data object attributes. Constant workflow data object attributes represent data that should remain constant for the lifetime of the process instance (see “Metadata” on page 17). If these attributes were allowed to be utilized in output mappings, this data would be overwritten with that specified in the output mappings.
- If the return struct is a list return struct, then the workflow data object used in the return mapping must be a list workflow data object.

**Runtime Information**

The values of the return type fields defined in the output parameter mappings are persisted using the specified workflow data object attributes after the associated automatic activity has been executed.

**Description of Context WDOs**

There are two context workflow data objects that are available when creating data item and function conditions for transitions from an automatic activity. These are described below.

**Context\_Result Workflow Data Object**

The Context\_Result workflow data object is available for use in a data item or function conditions (see “Conditions” on page 103) for a transition from an automatic activity. This allows the use of the return value of the invoked method in the said conditions. The conventions for the attributes available for the Context\_Result workflow data object are as follows:

- If the return type is a base type, the attribute available is called `value` (i.e. `Context_Result.value`).
- If the return value is a struct then the Context\_Result attribute values available are all the fields present on the struct return class (i.e. `Context_Result.description` etc.).

- If the return value is a nested (aggregated struct) then the Context\_Result attribute values available will be the fields available in the containing struct (i.e. Context\_Result.description etc.) and also the fully qualified names of those fields in the nested structs (i.e. Context\_Result.dtls:concernRoleID etc.). Regardless of the depth of the nesting of the return value struct, there is only one Context\_Result workflow data object available with the names of the nested structs forming part of the attribute names. The separator between a nested struct and it's fields is a colon as seen in the example above.
- If the return type is a list struct, the Context\_Result workflow data object is not available.

### Context\_Error Workflow Data Object

A BPO method being called by an automatic activity can sometimes fail (i.e. throw an exception that causes the activity transaction to roll-back). When this happens, it can be useful to be able to model follow-on actions after the failure. The Context\_Error workflow data object enables this type of "error path" modeling. It is available for use in a data item or function conditions (see "Conditions" on page 103) for a transition from an automatic activity.. The Context\_Error workflow data object has one attribute exceptionOccurred which is described below:

- The exceptionOccurred attribute is a boolean value indicating if the BPO method associated with an automatic activity failed. It defaults to false and is set to true if the associated BPO method fails.

At runtime, if the BPO method being called in an automatic activity fails (and is re-tried the prerequisite number of times and still fails), the workflow engine will set the exceptionOccurred attribute of Context\_Error to true. Any transitions using the Context\_Error workflow data object are then evaluated and followed if they resolve to true. This enables a workflow process instance to proceed along the defined error path even though the automatic activity failed.

If the BPO method being called fails and there are *no* transitions using the Context\_Error workflow data object, then the activity is halted and an entry is created in the Failed Messages Admin console.

**Note:** The Context\_Error workflow data object takes no account of the *cause* of the failure, only whether or not there was one.

## Event Wait

### Prerequisites

- The base details common to all the activity types supported by Cúram workflow are described in "Base Activity" on page 29 and are applicable to the event wait activity described here.

### Overview

The Cúram application has the ability to raise events at various points informing any registered listeners of what has happened. A number of different event listeners may be registered to listen for a specified event. These event listeners are application functions that implement the `curam.util.events.impl.EventHandler` interface. When a specified event is raised, the workflow engine invokes the associated event handler function (see the *Cúram Server Developers Guide* for more details on events and event handlers).

Workflow uses this facility in a slightly different way through event wait activities. An event wait activity pauses the execution of a particular branch of a process instance until a particular event has occurred.

## List of events

It is not completely accurate to say that an event wait activity pauses a workflow process until a particular event is raised. An event wait can in fact specify any number of events to wait for. If it has been specified not to wait for all of these events to be raised to complete the activity, the first event that matches one of the specified event waits will complete the activity and progress the workflow. In this scenario, whether or not the rest of the events ever get raised has no effect on the process. It is also possible to specify that all of the event waits must be matched by associated raised events before completing the activity and continuing the workflow process.

## Metadata

```
<event-wait-activity id="1" category="AC1">
  ...
  <event-wait wait-on-all-events="true">
    <events>
      <event event-class="Task" event-type="Close"
        identifier="1">
        <event-match-attribute name="taskId"
          wdo-name="Context_Task"/>
      </event>
      <event event-class="Parent" event-type="Approve"
        identifier="1">
        <event-match-attribute name="identifier"
          wdo-name="ParentList[Context_Loop.loopCount]"/>
      </event>
      <event event-class="Child" event-type="Approve"
        identifier="2">
        <event-match-attribute name="identifier"
          wdo-name="ChildDetails"/>
        <multiple-occurring-event>
          <list-wdo-name>ChildDetails</list-wdo-name>
        </multiple-occurring-event>
      </event>
    </events>
  </event-wait>
  ...
</event-wait-activity>
```

### event-wait

This contains the details of the event wait associated with the specified activity. This includes the details of all the events for the event wait.

### wait-on-all-events

The value of this flag indicates to the workflow engine if it should wait for events to be raised for all of the specified event waits before completing the associated activity. If set to false, the first event that matches one of the specified event waits will result in the completion of associated activity and the workflow progressing. When set to true, an event must be raised for each of the event waits specified for the activity before the activity is completed and the workflow progressed.

**events** This contains the details of all of the events that the specified activity is waiting on.

**event** This contains the details of one specific event that this activity is waiting on. The event details contain the following mandatory attributes:

**event-class**

This represents the class of business event that this process is waiting on.

**event-type**

This represents the type of business event that this process is waiting on. The combination of event-class and event-type will denote the business event required.

**identifier**

This represents the unique identifier of this event. The identifier is required to be unique only within the list of events for this activity.

**event-match-attribute**

This represents the workflow data object attribute (see “Workflow Data Objects” on page 16) that is used to match the required instance of the specific event. For example, in the first event specified in the metadata above, the workflow data object attribute would refer to the task identifier associated with the closing of a specific task. When this event is raised, the workflow engine will use the data in the event match attribute to uniquely identify the task to close.

**multiple-occurring-event**

This signifies that this event will represent a multiple occurring event. This means that if this metadata is specified for an event, the workflow engine will create one event wait record for each item in the list workflow data object specified as the multiple occurring event when that activity is executed. This allows the workflow engine to wait on multiple occurrences of the same event.

It should be noted that when the multiple occurring event is specified for an event, then an attribute from the associated list workflow data object must be used as the event match data for the event. This will ensure that each event generated by the workflow engine for the multiple occurring event will be unique.

**list-wdo-name**

This represents the name of the list workflow data object to be used as the multiple occurring event.

## Validations

- A least one event must be defined for the event wait information associated with an event wait activity.
- The event class and type specified for each business event must be valid entries on the relevant event database tables.
- An event and associated event match attribute can only be defined once in an event wait activity. That is, the same event class, event type and event match attribute can only be used once as a specific event being waited on for an event wait activity.
- The workflow data object attribute mapped to the event match attribute for an event must be valid, and as it is used as a unique identifier in the event matching mechanism, it must be of type LONG to reflect the 64-bit identifiers used in Cúram.

- The `Context_Task` workflow data object can only be used as the event match data workflow data object attribute if the activity is either a manual or parallel manual activity and the event is not a multiple occurring event.
- If an indexed item from a list workflow data object (i.e. `ParentList[Context_Loop.loopCount]`) is used as the event match data, then the workflow data object must be a list workflow data object and the activity containing the event mapping must be contained within a loop.
- If an indexed item from the Parallel List Workflow Data Object is used as the event match data, then the activity containing the mapping must be a Parallel Activity (i.e. `ParallelListWDO[Context_Parallel.occurrenceCount]`). The workflow data object being indexed by the `Context_Parallel` Workflow Data Object must be the Parallel Activity List Workflow Data Object.
- If the multiple occurring event list workflow data object has not been specified for the event and the activity containing the event mapping is not a parallel activity, then an attribute from a list workflow data object cannot be used as the event match data for that event.
- If the multiple occurring event list workflow data object has been specified for the event, then an attribute from this list workflow data object must be used as the event match data for that event.
- The workflow data object attribute mapped as the multiple occurring event must be valid. It must also be a list workflow data object.

## Code

A Workflow Event Handler is supplied with Cúram and is automatically registered to listen for events raised in the application. Multiple event waits may be registered for a particular activity instance in a workflow process. If the *waitOnAllEvents* flag is set to false for the specified event wait data, only one of these event waits is required to be matched to complete that activity instance. The Workflow Event Handler will process that event by completing the specified activity instance and driving the process forward by starting the next set of activities in the process. All of the other event wait records that were registered for the completed activity instance are then removed. If output mappings (see “Output Mappings” on page 52) have been specified for the event wait, they will be persisted by the workflow engine and may be used in subsequent activities and transitions in the process.

When the *waitOnAllEvents* is set to true, all of the event waits specified for the activity instance must be matched by raised events to complete the activity and progress the workflow. For each raised event that matches an associated event wait for the activity instance, the Workflow Event Handler will process the event by deleting the associated event wait record and persisting any output mappings (see “Output Mappings” on page 52) that have been specified for the event wait. This processing continues until all of the associated event waits have been matched by raised events. It is only then that the Workflow Event Handler will complete the specified activity instance and drive the process forward by starting the next set of activities in the process.

## Runtime Information

An event raised in the application can only cause a process instance to continue if the event matches that being waited on and the event match attribute specified for the event wait matches the primary event data of the event.

## Deadline

An event wait pauses a workflow process in lieu of some event being raised. However, in many cases it is not desirable for a process to wait indefinitely. It is possible for a chain of events to occur that mean the event the process is waiting on may never be raised. For example, by chance the event could be raised before the process reaches the event wait activity. To mitigate against this risk it is possible to optionally specify a deadline for an event to be raised after which a deadline handler will be invoked.

### Prerequisites

- Deadline handler methods specified for an event wait deadline are Cúram business process object methods. The input mappings for the formal parameters of these methods and their associated metadata are described in “Automatic” on page 31. This chapter should therefore be referenced for a description of these mappings.

## Metadata

```
<event-wait-activity id="1" category="AC1">
...
<deadline complete-activity="true">
  <duration>
    <mapped-duration>
      <wdo-attribute wdo-name="TaskCreateDetails"
        name="deadlineDuration" />
    </mapped-duration>
  </duration>
  <deadline-handler interface-name=
    "curam.core.sl.intf.WorkflowDeadlineFunction"
    method-name="defaultDeadlineHandler">
    <formal-parameters>
      <formal-parameter index="0">
        <struct type="curam.core.struct.TaskKey">
          <field name="taskID">
            <base-type type="INT64">
              <wdo-attribute wdo-name="Context_Task"
                name="taskID" />
            </base-type>
          </field>
        </struct>
      </formal-parameter>
      <formal-parameter index="1">
        <struct type="curam.core.struct.ChildKey">
          <field name="identifier">
            <base-type type="INT64">
              <wdo-attribute wdo-name=
                "ClaimantDependents[Context_Loop.loopCount]"
                name="identifier" />
            </base-type>
          </field>
        </struct>
      </formal-parameter>
    </formal-parameters>
  </deadline-handler>
  <deadline-output-mappings>
    <duration-expired wdo-name="TaskDeadlineDetails"
      name="booleanValue" />
    <deadline-expiry-time wdo-name="TaskDeadlineDetails"
      name="dateTimeValue" />
  </deadline-output-mappings>
</deadline>
...
</event-wait-activity>
```

### complete-activity

This represents a boolean flag which indicates whether the activity should complete if the deadline duration expires. The default for this flag is false.

### duration

This represents the duration of time that can elapse before the deadline handler method will be invoked. The duration can be represented in either of the formats below which will subsequently be used to calculate the deadline date time for the event wait:

**seconds**

The number of seconds that can elapse before the deadline handler will be invoked

**mapped-duration**

The attribute of a workflow data object that can be mapped as representing the number of seconds that can elapse before the deadline handler will be invoked.

**deadline-handler**

This represents the method that is to be invoked once the deadline duration has expired. The following metadata must be specified for a deadline handler:

**interface-name**

This represents the fully qualified name of the deadline handler interface class name.

**method-name**

This represents the required method in the deadline handler interface required to be invoked when the deadline expires.

**formal-parameters**

This contains a list of the deadline handler method parameters and associated workflow data object attributes that are mapped to those parameters when the deadline handler is invoked. For details on method parameter mappings see “Input Mappings” on page 33.

**deadline-output-mappings**

This contains the deadline output data which can be optionally mapped to workflow data object attributes. This data indicates whether or not the deadline duration expired and the date and time the deadline duration expired.

**Validations**

- If a deadline handler is specified it must reference a valid Cúram business method that exists on the application's classpath.
- The workflow data object attributes specified in the input mappings must be valid. The criteria that defines a valid workflow data object attribute may be seen in “Validations” on page 20
- The type of the formal parameter being mapped to and the type of the workflow data object attribute being used in that input mapping must be compatible. For example, if the input mapping being created is a struct field that has a type of STRING, then the workflow data object attribute being used for that mapping must also be of type STRING.
- If the indexed item from a list workflow data object (i.e. `ClaimantDependent[Context_Loop.loopCount]`) is being used in an input mapping, then the associated workflow data object must be a list workflow data object and the activity containing the input mappings must be contained within a loop.
- If the `Context_Parallel` workflow data object is being used in an input mapping, then the activity containing the input mappings must be a `Parallel` activity.
- If an indexed item from the Parallel List Workflow Data Object is being used in an input mapping, then the activity containing the mapping must be a `Parallel Activity` (i.e. `ParallelListWDO[Context_Parallel.occurrenceCount]`). The workflow data object being indexed by the `Context_Parallel` Workflow Data Object must be the `Parallel Activity List Workflow Data Object`.

- The deadline duration may be specified by using a deadline duration in seconds or a workflow data object attribute mapping, but not both.
- If the deadline duration has been specified using a workflow data object attribute, the attribute must be valid and be of type INTEGER.
- If a deadline has been specified for an activity, then a deadline handler function must be specified and/or the complete activity flag must be set to true. If this is not the case the workflow would not do anything when the deadline is reached.
- If the duration expired value of the deadline output mappings has been mapped to a workflow data object attribute, then the attribute must be valid and of type BOOLEAN.
- If the deadline expiry time value of the deadline output mappings has been mapped to a workflow data object attribute, then the attribute must be valid and of type DATETIME.
- The complete activity flag cannot be set to true if the activity containing the deadline is a parallel activity. This is due to the fact that parallel activities do not support modeled deadlines.

## Code

- Any return parameters associated with the deadline handler method are not used in the workflow engine and are therefore irrelevant.
- The Workflow Deadline Scanner API function `DeadlineScanner.scanDeadlines()` is provided to allow the scanning of event wait deadlines that have exceeded their specified duration. Any such event waits will be processed and their associated handler function invoked or the associated activity completed.

## Runtime Information

When the workflow engine executes an activity that contains deadline metadata, it creates the deadline date time as follows:

- If the duration has been specified in seconds, then the calculation is the current date time + seconds defined in metadata = deadline date time.
- If the duration has been specified as a workflow data object attribute then the calculation is the current date time + the value as defined in workflow data object attribute = deadline date time

Deadlines that have expired are processed by invoking the `ScanTaskDeadlines` batch job. This batch job in turn invokes the Workflow Deadline Scanner API described above which retrieves a list of all of the deadlines that have expired and processes them. If a deadline handler method has been specified for the deadline, the values of the workflow data object attributes defined in the parameter mappings are provided as input parameters to the deadline handler method and it is invoked. If the complete activity flag has been set to true, then the associated activity is completed. Any deadline output mappings (duration expired and deadline expiry time) that may have been specified are persisted here. The attributes of the `Context_Deadline` workflow data object are also persisted during this processing to allow them to be used in transitions emanating from the activity containing the deadline.

## Description of Context WDOs

The `Context_Deadline` workflow data object is available for use in a data item or function condition (see “Conditions” on page 103) for a transition from an activity with an event wait that has a deadline. The `Context_Deadline` workflow data object attributes available are:

**Context\_Deadline.durationExpired**

Represents a boolean indicating if the deadline duration associated with the activity has expired.

**Context\_Deadline.expiryTime**

An attribute containing the date and time at which the deadline duration expires.

## Output Mappings

The event raised has some information in it that may be worth mapping back into the workflow engine. The event has both primary and secondary event data. The primary event data is what was used to match the event in the first place so there is little point in mapping this back into the process. The secondary event data however may be unknown to the workflow engine and so can be mapped in. Also since an event wait activity can wait on any number of events, the actual event that was raised may be of interest and so can also be mapped into the workflow engine. Finally, the Cúram user raising the event might be of interest and so this can also be mapped into the workflow engine.

It should be noted that if an activity instance should wait for all of its associated event waits to be matched, any event output mappings that exist for the activity instance will be processed each time an event is raised that matches one of the event waits.

### Metadata

```
<event-wait-activity id="1" category="AC1">
  ...
  <event-output-mappings>
    <event-type wdo-name="CaseEventResult"
      name="eventType" />
    <output-data wdo-name="TaskCreateDetails"
      name="concernRoleID" />
    <raised-by wdo-name="CaseEventResult"
      name="eventRaisedBy" />
    <time-raised wdo-name="CaseEventResult"
      name="timeRaised" />
  </event-output-mappings>
  ...
</event-wait-activity>
```

**event-output-mappings**

This contains the data that can be optionally mapped to the workflow engine from the event that was raised.

**event-type**

This contains the business event that was raised which the activity instance was waiting on.

**output-data**

This contains the secondary event data that is to be mapped into the workflow engine.

**raised-by**

This contains the username of the Cúram user that caused the event to be raised.

### **time-raised**

This contains the date and time that the event was raised.

### **Validations**

- The event type event output mapping, if specified, must be a valid workflow data object attribute and must be of type STRING.
- The raised by user name event output mapping, if specified, must be a valid workflow data object attribute and must be of type STRING.
- The output data event output mapping, if specified, must be a valid workflow data object attribute and must be of type LONG.
- The time raised output mapping, if specified, must be a valid workflow data object attribute and must be of type DATETIME.

### **Runtime Information**

When an event is raised in the application that an activity instance is waiting on, any workflow data object attributes contained in event output mappings that have been defined for the event wait are populated and persisted with the relevant data from the event.

### **Description of Context WDOs**

The Context\_Event workflow data object is available for use in a data item or function condition (see “Conditions” on page 103) for a transition from an activity with an event wait. The Context\_Event workflow data object attributes available are:

#### **Context\_Event.raisedByUserName**

The username of the Cúram user who raised the event.

#### **Context\_Event.timeRaised**

The time at which the event was raised.

#### **Context\_Event.fullyQualifiedEventType**

The fully qualified (both event class and event type) name of the business event that was raised.

#### **Context\_Event.outputData**

The secondary event data associated with the raised event.

## **Reminders**

A reminder can be set on any deadline associated with a manual, decision, event wait, parallel manual or parallel decision activity. An arbitrary amount of reminders can be specified. Reminders utilize the notification metadata described in the activity notification (see “Activity Notifications” on page 92) chapter. This means that the typical notification subject, body, allocation strategy and actions can be specified for a reminder.

## Metadata

```
<reminders>
  <reminder id="1" delivery-offset="D01">
    <delivery-time>
      <seconds>93660</seconds>
    </delivery-time>
  or...
    <delivery-time>
      <mapped-delivery-time>
        <wdo-attribute wdo-name="CaseWDO"
          name="caseID"/>
      </mapped-delivery-time>
    </delivery-time>
  ...
  <notification delivery-mechanism="DM1">
    ...standard notification metadata
  </notification>
</reminder>
</reminders>
```

### reminders

This is optional and encapsulates all reminder tags for the deadline.

### reminder

This contains all reminder metadata for the deadline including the associated notification metadata.

### delivery-offset

This refers to a value from the codetable `ReminderDeliveryOffset` indicating what the seconds or mapped-delivery-time will be offset from. In the case of a deadline, it is offset from the deadline expiry time. This is the currently the only offset supported.

### delivery-time

This contains either the seconds or mapped-delivery-time tag depending on which has been specified.

### seconds

This tag represents the seconds prior to the deadline expiry time that the reminder will be sent.

### mapped-delivery-time

This tag represents a workflow data object containing the seconds prior to the deadline expiry time that the reminder will be sent.

## Validations

- A reminder cannot be created if a deadline has not been associated with the relevant activity. In addition, if a deadline does exist, but the deadline handler has not been set, or the complete activity indicator has been set to false, a reminder cannot be created.
- Each reminder has an identifier. This must be unique to the deadline upon which it is associated.
- Either a mapped-delivery-time or seconds must be specified for a reminder.
- If a seconds is specified, it must be prior to the deadline expiry time.
- The workflow data object attribute referenced by the mapped-delivery-time must be of type INTEGER.

- All existing validations for activity notifications (see “Activity Notifications” on page 92) are applicable to the notification metadata associated with reminders.

### Code

The Workflow Deadline Scanner API function `DeadlineScanner.scanDeadlines()` includes a call to the function `deliverReminders()` which processes and delivers any reminders that have reached their delivery time.

### Runtime Information

When an activity containing reminders has been executed, the reminders are persisted onto the Reminders entity. The time that a reminder is due to be sent on is calculated as follows:

- The delivery duration for the reminder is retrieved in seconds. This may be specified directly in seconds or in a workflow data object attribute.
- The duration for the deadline associated with the reminder is retrieved in seconds. This may be specified directly in seconds or in a workflow data object attribute.
- If the delivery duration for the reminder is a positive number and this number is less than the deadline duration (reminder deliveries cannot be specified for times that are greater than the deadline date time for obvious reasons), then the time to deliver the reminder notification is calculated as the deadline duration - the reminder delivery duration. This duration in seconds is then converted into a date time and added to the date time the reminder is being created on. This is then stored on the reminder record as the date time that the reminder notification is due to be sent on.

Reminders that have been configured for deadlines are processed and sent by invoking the `ScanTaskDeadlines` batch job. This batch job invokes the `DeadlineScanner.scanDeadlines()` function which scans for reminders that are due and sends the associated reminder notifications (using the reminder notification allocation strategy to determine the users to send the notifications to). The reminders that have been sent are removed from the Reminders entity to ensure that they are not sent again. When the activity completes any reminders that were configured for that activity but which were not sent are removed.

---

## Manual

### Prerequisites

- The base details common to all the activity types supported by Cúram workflow are described in “Base Activity” on page 29 and are applicable to the manual activity described here.

### Overview

In any automated business process there is a need to interact with human agents to make decisions, supply additional data or to perform tasks in the real world such as telephoning a client. In Cúram workflow, such steps in a process are modeled using manual activities. A manual activity specifies where in the business process human intervention is required. It also specifies the information the user will get when notified that they must perform a task and also the selection of the agents to which the work will be assigned.

## Task details

To notify a user that they are required to do some work as part of some automated business process, a task is assigned to them. A task is a message that appears in the users inbox. This inbox specifies the work the user is expected to do. The task can also have a list of actions associated with it. Actions are links to Cúram application pages where the work required to perform the task may be performed.

## Metadata

```
<manual-activity id="1">
  ...
  <task>
    <message>
      <message-text>
        <localized-text>
          <locale language="en">The following
            case %1n for %1s must be approved</locale>
        </localized-text>
      </message-text>
      <message-parameters>
        <wdo-attribute wdo-name="TaskCreateDetails"
          name="caseID"/>
        <wdo-attribute wdo-name=
          "Claimant[Context_Loop.loopCount]"
          name="caseID"/>
      </message-parameters>
    </message>
    <actions>
      <action page-id="Case_viewHome" principal-action="false"
        open-modal="false">
        <message>
          <message-text>
            <localized-text>
              <locale language="en">
                Case Home Page for case: %1n</locale>
            </localized-text>
          </message-text>
          <message-parameters>
            <wdo-attribute wdo-name="TaskCreateDetails"
              name="caseID"/>
          </message-parameters>
        </message>
        <link-parameter name="childID">
          <wdo-attribute wdo-name="ChildDependents"
            name="identifier"/>
        </link-parameter>
        <link-parameter name="fullName">
          <wdo-attribute wdo-name="ChildDependents"
            name="fullName"/>
        </link-parameter>
        <multiple-occurring-action>
          <list-wdo-name>ChildDependentList</list-wdo-name>
        </multiple-occurring-action>
      </action>
      <action page-id="Person_confirmPersonDetails"
        principal-action="true"
        open-modal="true">
        <message>
          <message-text>
            <localized-text>
              <locale language="en">
                Confirm Person Details for
                person: %1s</locale>
            </localized-text>
          </message-text>
          <message-parameters>
            <wdo-attribute wdo-name=
              "PersonDetailsList[Context_Loop.loopCount]"
              name="fullName"/>
          </message-parameters>
        </message>
        <link-parameter name="identifier">
          <wdo-attribute wdo-name="
            PersonDetailsList[Context_Loop.loopCount]"
            name="identifier"/>
        </link-parameter>
      </action>
    </actions>
    <task-priority>
      <priority ID="1" />
    </task-priority>
  </task>
</manual-activity>
```

**task** This contains all of the details of a task including the message and details of the associated actions. The various metadata associated with a task are described below

**message**

This contains the details of the parameterized message. When a manual activity is executed, a task is created. When a user views their tasks in the inbox, this message represents the subject of that task.

**message-text**

This contains the details of the message text. The text of the subject can contain replaceable strings (%k), which will be replaced with the associated text parameters. A text parameter is a mapping to a workflow data object attribute. Parameter k in the list will replace %k in the text string, where k is the order of the parameter in the list. %k can be repeated within the string and thus each workflow data object attribute must only be mapped once. A format for the replaceable strings can optionally be specified by placing another letter after the replaceable string, e.g. %1d, where d will format the value as a date.

*Table 3. Subject Text Data Conversion*

| Formatting Letter | Format As |
|-------------------|-----------|
| s                 | string    |
| n                 | numeric   |
| d                 | date      |
| z                 | date/time |
| t                 | time      |

**localized-text**

This contains details of the localizable task message text. For more details of the localized text and associated metadata, see “Localized Text” on page 30.

**message-parameters**

A task message may have parameters associated with it. This contains the details of the workflow data object attribute parameters used to replace the placeholders in the associated text. For details on workflow data objects and workflow data object attributes see “Workflow Data Objects” on page 16.

**actions**

This contains the details of all of the actions associated with the manual activity task. These actions are links to Cúram application pages where the work required to perform the task may be performed.

**action** This contains the definition of a hyperlink to a Cúram page on which a task can be performed. The following fields associated with the task action are described below:

**page-id**

This represents the identifier of the target Cúram page on which a user can begin to perform the required action.

**principal-action**

Actions may be defined as primary or secondary actions. Principal actions usually contain the links to the Cúram pages on which a user can begin to perform the actual required work. Secondary

actions usually contain links to supporting information that the user assigned to do the work can refer to while carrying out the assigned task.

**open-modal**

The pages linked from a task action may be specified to open in a modal dialog. If this indicator is set to true, then the page specified by the action link will be opened in a modal dialog. If set to false (the default) then the client infrastructure will decide how to open the link in the same fashion as it does with any other link in the application (i.e. if the page is part of a tab configuration, then it will open the appropriate tab - if not then it will just replace the action link home page in the content area of the current tab).

**message**

This contains the details of the parameterized message that is associated with the action to be performed, including the message text and the optional parameters that may be associated with the text.

**link-parameter**

The links to the Cúram pages where the actual work for the task will be performed must contain a page identifier (described above) and optional page parameters. These page parameters are described by this metadata and they represent a name/value pair where the name attribute is the name of a link parameter (the page parameter name in the associated Cúram client page) and the value is provided by a workflow data object attribute. The following field associated with the link parameter is described below:

**name** The name of the link parameter.

**multiple-occurring-action**

This signifies that this action will represent a multiple occurring action. This means that if this metadata is specified for an action, the workflow engine will create one action record for each item in the list workflow data object specified as the multiple occurring action, when that activity is executed.

It should be noted that when the multiple occurring action is specified for an action, then an attribute from the associated list workflow data object must be used as a link parameter for the action.

**list-wdo-name**

The name of the list workflow data object for use with the multiple occurring action.

**wdo-attribute**

The value used in the action link parameter is provided by the workflow data object attribute mapping specified in this piece of metadata.

**task-priority**

A task can optionally contain a priority and this metadata contains those details. The priority of a task be represented in either of the formats below:

**priority**

In this instance, the priority is selected in the Process Definition Tool and is taken from the TaskPriority code-table.

**mapped-priority**

The priority of a manual task can be mapped using a workflow

data object attribute. The following metadata snippet provides an example of how this can be achieved:

```
<manual-activity id="1">
  ...
  <task>
    <message>
      .....
    </message>
    <actions>
      <action page-id="Case_viewHome" principal-action="true">
        .....
      </action>
    </actions>
    <task-priority>
      <mapped-priority>
        <wdo-attribute wdo-name="WorkflowTestWDO"
          name="taskPriority"/>
      </mapped-priority>
    </task-priority>
    .....
  </task>
  ...
</manual-activity>
```

#### **allow-deadline-override**

This represents is a boolean flag used to indicate if the deadline (see “Deadline” on page 48) associated with the manual activity task may be overridden. Setting the value of this flag to true (the default is false) indicates that the deadline time can be changed after the task has been created by the workflow engine.

#### **allow-task-forwarding**

This is a boolean flag used to indicate if the task generated due the execution of the associated manual activity can be forwarded to another user. When a task is generated, it is allocated to an agent to carry out the work. Setting this flag to true (the default is true) allows that agent to forward that task to another agent to carry out the specified work.

#### **administration-sid**

This field allows an administration security identifier to be specified for a manual task. This allows a user in a group associated with the specified security identifier to modify the task details, although the task may be reserved by another user in the application.

#### **initial-comment**

This allows an initial comment mapping to be specified for the manual task. The value of the workflow data object attribute used in this mapping is used to place a record in the TaskHistory table when the associated manual activity is executed.

### **Validations**

- A subject must be defined for the manual activity task. This is a localizable string.
- All of the workflow data objects used as subject text parameters in the manual activity task subject message must be valid workflow data object attributes in the context of the containing workflow process definition.
- If an indexed item from a list workflow data object (i.e. PersonDetailsList[Context\_Loop.loopCount]) is used as a subject text parameter,

then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.

- If the `Context_Parallel` workflow data object is used as a subject text parameter, then the activity containing the mapping must be a `Parallel` manual activity.
- If an indexed item from the `Parallel List Workflow Data Object` is used as a subject text parameter, then the activity containing the mapping must be a `Parallel Activity` (i.e. `ParallelListWDO[Context_Parallel.occurrenceCount]`). The workflow data object being indexed by the `Context_Parallel Workflow Data Object` must be the `Parallel Activity List Workflow Data Object`.
- If actions have been specified for the manual activity task, any workflow data object attributes used as mappings for action text parameters must be valid in the context of the containing workflow process definition.
- If actions have been specified for the manual activity task, any workflow data object attributes used in the action link parameter mappings of a manual activity action must be valid in the context of the containing workflow process definition.
- If an indexed item from a list workflow data object (i.e. `PersonDetailsList[Context_Loop.loopCount]`) is used in the action text or action link parameter mappings, then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.
- If the `Context_Parallel` workflow data object is used in the action text or action link parameter mappings, then the activity containing the mapping must be a `Parallel` manual activity.
- If an indexed item from the `Parallel List Workflow Data Object` is used in the action text or action link parameter mappings, then the activity containing the mapping must be a `Parallel Activity` (i.e. `ParallelListWDO[Context_Parallel.occurrenceCount]`). The workflow data object being indexed by the `Context_Parallel Workflow Data Object` must be the `Parallel Activity List Workflow Data Object`.
- The number of placeholders used in the subject text and action text of the manual activity task must equal the number of mapped workflow data object attributes for all the locales defined.
- The priority of a manual task may be specified by using a codetable code value or a workflow data object attribute mapping, but not both.
- If a mapped priority has been specified for the manual activity task, the workflow data object attribute specified for it must be valid in the context of the containing workflow process definition. It must also be of type `STRING`.
- If an initial comment mapping has been specified for the manual activity task, the workflow data object attribute specified for it must be valid in the context of the containing workflow process definition. It must also be of type `STRING`.
- The workflow data object specified for use in the multiple occurring action must be a valid workflow data object in the context of the containing workflow process definition. It must also be a list workflow data object.
- At least one attribute from the multiple occurring action list workflow data object must be utilized in the link parameters specified for a multiple occurring action.

## Code

### Action Pages and Action Page Parameters

The actions specified for the manual activity task are links to Cúram application pages where the work required to perform the task may be

performed. The pages specified in the task actions must be valid Cúram pages and must be available in the Cúram application. The parameters in these pages must match the parameters specified as action link parameters in the associated task actions.

#### **LocalizableStringResolver TaskStringResolver API**

The task subject and associated task action messages are displayed in the user's inbox to inform them of the work required to be completed for the associated task. The `LocalizableStringResolver.TaskStringResolver` API contains the functions to resolve both the task subjects and action messages for the correct user locale. The replacement of the placeholders with the associated workflow data object attribute values specified in the associated mappings is also carried out as part of these functions.

#### **Task Admin API**

A number of functions have been provided on the `TaskAdmin` class to allow the manipulation of tasks. For further details of the functions available, see the associated Javadoc specification for the `TaskAdmin` class.

#### **Task History Admin API**

Various life cycle events for a task (i.e. when a task is created; when a task is allocated; when a task is closed) are written to the `TaskHistory` table during the lifetime of a task. A number of search functions have been provided on this API class to allow these entries to be examined. For further details of the functions available, see the associated Javadoc specification for the `TaskHistoryAdmin` entity.

#### **Workflow Deadline Admin API**

A number of functions have been provided on the `WorkflowDeadlineAdmin` class to allow the manipulation of workflow deadlines. For further details of the functions available, see the associated Javadoc specification for the `WorkflowDeadlineAdmin` class.

### **Runtime Information**

When a manual activity is executed by the workflow engine, a task is created and is allocated to an agent to perform that work (see "Allocation strategy").

#### **Description of Context WDOs**

The `Context_Task` workflow data object allows the unique identifier of the task created as part of the execution of the associated manual activity to be available for use in the various metadata mappings associated with a manual activity. Examples of some of these mappings include event match data mappings (see "List of events" on page 45) and deadline function input mappings (see "Deadline" on page 48). The one attribute available on this workflow data object is:

##### **Context\_Task.taskID**

The `taskID` attribute represents the unique identifier of the task created when the associated manual activity is executed.

## **Allocation strategy**

An organization will typically have many human agents at various levels of responsibility that can perform work for a given process definition. To select a specific agent or group of agents that can do the work for a specific manual activity, an allocation strategy is assigned to the activity. There are four types of allocation strategies currently supported by Cúram workflow: function, Classic rules, Cúram Express rules (CER) and target. When an allocation strategy of type target is selected, the agent or group of agents to assign the work to are named directly. Selecting a function allocation strategy results in the invocation of the

specified allocation function when the associated activity is executed by the workflow engine. Finally, if a classic or Cúram Express rules (CER) allocation strategy is selected, the specified ruleset is executed when the associated activity is executed.

### **Prerequisites**

- If the allocation strategy associated with a manual activity is of type Function, these allocation functions are Cúram business methods with a specific signature. The input mappings for the formal parameters of these methods and their associated metadata are described in “Automatic” on page 31. This chapter should therefore be referenced for a description of these mappings.

### **Metadata**

As described previously, there are four types of allocation strategies. The required metadata for each of these types is described in the following sections.

#### **allocation-strategy**

This contains the details of the allocation strategy defined for the manual task. The following fields associated with an allocation strategy are described below:

**type** This contains the type of the allocation strategy. The four types of allocation strategies currently supported by Cúram workflow are function, classic rules, CER rules and target.

#### **identifier**

This represents the identifier of the allocation strategy. For an allocation strategy of type function, this identifier represents the fully qualified name of the allocation function being used. For an allocation strategy of type rule or curam express rule, this identifier represents the identifier of the ruleset being used. Finally, when an allocation strategy of type target is selected, this identifier represents the identifier of the allocation target being used.

## Function Allocation Strategy:

```
<manual-activity id="1" category="AC1">
  ...
  <task>
    ...
  </task>
  <allocation-strategy
    identifier="curam.core.sl intf.
      WorkflowAllocationFunction.manualAllocationStrategy"
    type="function">
    <function-mappings>
      <formal-parameters>
        <formal-parameter index="0">
          <base-type type="INT32">
            <wdo-attribute wdo-name="Context_Task"
              name="taskID"/>
          </base-type>
        </formal-parameter>
        <formal-parameter index="1">
          <base-type type="INT64">
            <wdo-attribute
              wdo-name="Context_RuntimeInformation"
              name="processInstanceID"/>
          </base-type>
        </formal-parameter>
        <formal-parameter index="2">
          <struct type="curam.struct.TaskDetails">
            <field name="taskID">
              <base-type type="INT64">
                <wdo-attribute wdo-name="Context_Task"
                  name="taskID"/>
              </base-type>
            </field>
            <field name="category">
              <base-type type="STRING">
                <wdo-attribute wdo-name="TaskCreateDetails"
                  name="category"/>
              </base-type>
            </field>
          </struct>
        </formal-parameter>
        <formal-parameter index="3">
          <struct type="curam.struct.PersonDetails">
            <field name="identifier">
              <base-type type="INT64">
                <wdo-attribute wdo-name=
                  "PersonDetailsList[Context_Loop.loopCount]"
                  name="identifier"/>
              </base-type>
            </field>
            <field name="fullName">
              <base-type type="STRING">
                <wdo-attribute wdo-name=
                  "PersonDetailsList[Context_Loop.loopCount]"
                  name="fullName"/>
              </base-type>
            </field>
          </struct>
        </formal-parameter>
      </formal-parameters>
    </function-mappings>
  </allocation-strategy>
  <event-wait>
    ...
  </event-wait>
</manual-activity>
```

## function-mappings

This contains the details of the input mappings for the formal parameters of the specified allocation function. Allocation functions are Cúram business methods (similar to those that are specified for automatic activities) that have a distinct return signature (allocation functions must have a return type of `curam.util.workflow.struct.AllocationTargetList`). Therefore, the metadata used for these mappings are the same as those used for the input mappings for the business process object methods that are associated with automatic activities. The reader should refer to the “Input Mappings” on page 33 section of the automatic activity chapter for further details of this metadata and its meaning.

## Classic Rules Allocation:

```
<manual-activity id="1" category="AC1">
  ...
  <task>
    ...
  </task>
  <allocation-strategy type="rule"
    identifier="PRODUCT_1">
    <ruleset-mappings>
      <rdo-mapping>
        <source-attribute wdo-name="TaskCreateDetails"
          name="caseID" />
        <target-attribute rdo-name="TaskDetails"
          name="caseID" />
      </rdo-mapping>
      <rdo-mapping>
        <source-attribute wdo-name="TaskCreateDetails"
          name="concernRoleID" />
        <target-attribute rdo-name="TaskDetails"
          name="concernRoleID" />
      </rdo-mapping>
    </ruleset-mappings>
  </allocation-strategy>
  <event-wait>
    ...
  </event-wait>
  ...
</manual-activity>
```

## ruleset-mappings

This contains the details of all the mappings for the *ruleset* specified in the allocation identifier. It is not required to map all of the rules data object attributes specified in the ruleset (mappings for a subset of them may be created).

## rdo-mapping

This contains the details of one mapping between a rules data object attribute specified in the allocation ruleset and its associated workflow data object attribute. The following metadata constitute a valid mapping:

### source-attribute

This contains the details of the source attribute in the mapping (i.e. where the data will be provided from at runtime). A source attribute consists of a workflow data object name and its associated attribute name (see “Workflow Data Objects” on page 16).

### target-attribute

This contains the details of the target attribute in the mapping (i.e.

where the data will be mapped into at runtime). A target attribute consists of a rules data object name and it's associated attribute name.

### CER Rules Allocation:

```

<manual-activity id="1" category="AC1">
  ...
  <task>
    ...
  </task>
  <allocation-strategy type="curam express rule"
    identifier="Sample Allocation Rules">
    <cer-set-mappings primary-class="SampleAllocationClass">
      <cer-class-mapping>
        <source-attribute wdo-name="TaskCreateDetails"
          name="caseID" />
        <target-attribute cer-class-name="SampleAllocationClass"
          name="caseID" />
      </cer-class-mapping>
      <cer-class-mapping>
        <source-attribute wdo-name="TaskCreateDetails"
          name="subject" />
        <target-attribute cer-class-name="RuleClassA"
          name="subject" />
      </cer-class-mapping>
      <cer-class-mapping>
        <source-attribute wdo-name="ListTaskDetails"
          name="employerIDs" />
        <target-attribute cer-class-name="RuleClassA"
          name="listOfEmployerIDs" />
      </cer-class-mapping>
      <cer-class-mapping>
        <source-attribute wdo-name="ListTaskDetails"
          name="concernRoleID" />
        <target-attribute cer-class-name="SampleAllocationClass"
          name="listConcernRoleIDs" />
      </cer-class-mapping>
      <cer-class-mapping>
        <source-attribute wdo-name="ListTaskDetails"
          name="participantIDs" />
        <target-attribute cer-class-name="SampleAllocationClass.listRuleClassB.RuleClassB"
          name="participantIDs" />
      </cer-class-mapping>
    </cer-set-mappings>
  </allocation-strategy>
  <event-wait>
    ...
  </event-wait>
  ...
</manual-activity>

```

#### cer-set-mappings

This contains the details of all the mappings for the *CER rule set* specified in the allocation identifier. The *primary-class* metadata tag must point to a CER rule class that contains an attribute called *targets*. This is required as the workflow engine uses an attribute of this name to determine the list of allocation targets for the specified allocation strategy. Mappings must be created for all of the attributes that have been marked as specified in all of the CER rule classes used for the allocation strategy.

#### cer-class-mapping

This contains the details of one mapping between a rule class attribute specified in the CER rule set and it's associated workflow data object

attribute. One of these mappings must exist for each CER rule class attribute that has been marked as *specified* being used in the allocation strategy. The following metadata constitutes a valid mapping:

#### **source-attribute**

This contains the details of the source attribute in the mapping (i.e. where the data will be provided from at runtime). A source attribute consists of a workflow data object name and its associated attribute name (see “Workflow Data Objects” on page 16). Attributes of list workflow data objects may also be used here if the mapping being created is related to a CER rule class attribute list type.

#### **target-attribute**

This contains the details of the target attribute in the mapping (i.e. where the data will be mapped into at runtime). A target attribute consists of a CER class name and its associated attribute name. Some but not all CER rule class attribute types are supported for use in allocation strategy mappings. The supported types include *String*, *Boolean*, *Number*, *Date* and *DateTime*. A list of rule classes may also be specified as well as lists of the base types outlined previously.

### **Target Allocation Strategy:**

```
<manual-activity id="1" category="AC1">
  ...
  <task>
  ...
  </task>
  <allocation-strategy type="target"
    identifier="HEARINGSCHEDULER"/>
  <event-wait>
  ...
  </event-wait>
  ...
</manual-activity>
```

No further metadata is required to describe an allocation strategy of type *target*. As stated previously, the identifier in this case is the identifier of the allocation target containing the agent or group of agents that the task will be assigned to.

### **Validations**

- An allocation strategy must be defined for a manual task.
- If the allocation strategy is of type function, the function specified must be a valid and must exist on the Cúram application classpath.
- If the allocation strategy is of type function, the return type of the function must be `curam.util.workflow.struct.AllocationTargetList`.
- If the allocation strategy is of type function, any of the input parameters of the specified function that are mapped must be to valid workflow data object attributes and the type of the workflow data object attribute must match the type of the input parameter field.
- If the allocation strategy is of type function and an indexed item from a list workflow data object is used in an input mapping, then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.
- If the allocation strategy is of type classic or CER rule, the specified ruleset must be valid.

- If the allocation strategy is of type CER rule, a primary CER rule class name must be specified.
- If the allocation strategy is of type CER rule, the specified primary CER rule class must exist in the specified CER ruleset.
- If the allocation strategy is of type CER rule, the specified primary CER rule class must extend the required abstract Workflow Allocation CER rule class.
- If the allocation strategy is of type CER rule, the specified primary CER rule class must contain an attribute named *targets*.
- If the allocation strategy is of type CER rule, if CER rule classes other than the primary CER rule classes have been specified in the input mappings, then the CER primary class must contain attributes that refer to those classes, one for each class.
- If the allocation strategy is of type CER rule, all of the source attributes specified in the mappings must be valid workflow data object attributes in the context of the containing workflow process definition. All of the target attributes must be valid CER class attributes in the context of the specified ruleset. The type of the workflow data object attribute specified as the source attribute must match the type of the CER class attribute specified as the target attribute in the mapping.
- If the allocation strategy is of type CER rule, no duplicate target attribute mappings are allowed. In other words, a CER rule class attribute can only be specified once in any list of CER class mappings.
- If the allocation strategy is of type CER rule, all of the attributes that have been marked as *specified* for all of the CER rule classes used for the allocation strategy must contain an input mapping. CER class attributes that have not been marked as *specified* must not contain an input mapping.
- If the allocation strategy is of type classic rule, all of the source attributes specified in the mappings must be valid workflow data object attributes in the context of the containing workflow process definition. All of the target attributes must be valid rules data object attributes in the context of the specified ruleset. The type of the workflow data object attribute specified as the source attribute must match the type of the rules data object attribute specified as the target attribute in the mapping.
- If the allocation strategy is of type classic rule, no duplicate target attribute mappings are allowed. In other words, a rules data object attribute can only be specified once in any list of ruleset mappings.
- If an indexed item from a list workflow data object (i.e. `PersonDetailsList[Context_Loop.loopCount]`) is used in the function, classic rule or CER rule allocation strategy mappings, then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.
- If the `Context_Parallel` workflow data object is used in the function, classic or CER rule allocation strategy mappings, then the activity containing the mapping must be a `Parallel` activity.
- If an indexed item from the Parallel List Workflow Data Object is used in the function, classic or CER rule allocation strategy mappings, then the activity containing the mapping must be a `Parallel Activity` (i.e. `ParallelListWDO[Context_Parallel.occurrenceCount]`). The workflow data object being indexed by the `Context_Parallel` Workflow Data Object must be the `Parallel Activity List Workflow Data Object`.

## Code

As stated previously, any business process object method specified as an allocation function must return a structure of type `curam.util.workflow.struct.AllocationTargetList`.

As is the case with business methods associated with automatic activities, a failure of the allocation function when a manual activity is executed will cause the Workflow Error Handling strategy to be invoked. This may cause, for example, the activity associated with the failed method to be retried a number of times. Based on this fact the allocation functions associated with the allocation strategies of manual or decision activities should in general not throw exceptions unless an unrecoverable situation occurs.

The application must implement the `curam.util.workflow.impl.WorkResolver` callback interface to determine how tasks will be allocated in the application. The application property `curam.custom.workflow.workresolver` must refer to the `curam.util.workflow.impl.WorkResolver` implementation class in the application as the workflow engine will use this property to determine the correct function to allocate the task.

The `curam.util.workflow.impl.WorkResolver` class has an overloaded method `resolveWork` because the various allocation strategy types return the allocation targets in different formats. However this is an implementation detail that developers of custom work resolver classes should not have to deal with especially since the business processing for all versions of the method should be the same.

```
package curam.util.workflow.impl;

...

public interface WorkResolver {

    void resolveWork(
        final TaskDetails taskDetails,
        final Object allocationTargets,
        final boolean previouslyAllocated);

    void resolveWork(
        final TaskDetails taskDetails,
        final Map allocationTargets,
        final boolean previouslyAllocated);

    void resolveWork(
        final TaskDetails taskDetails,
        final String allocationTargetID,
        final boolean previouslyAllocated);

    ...
}
```

To mitigate this issue the `curam.core.sl.impl.DefaultWorkResolverAdapter` provides a more convenient mechanism for implementing a work resolver. This class implements the different methods and converts their input parameters into allocation target lists allowing developers of custom work resolution logic to extend this class and implement one method that is called regardless of the source of the allocation targets.

```

package curam.core.sl.impl;

...

public abstract class DefaultWorkResolverAdapter
    implements curam.util.workflow.impl.WorkResolver {

    public abstract void resolveWork(
        final TaskDetails taskDetails,
        final AllocationTargetList allocationTargets,
        final boolean previouslyAllocated);

    ...
}

```

In addition to this adapter class, the application ships with a work resolver implementation that is used out-of-the-box. This class is called `curam.core.sl.impl.DefaultWorkResolver` and it also serves as an example of how to extend the adapter.

## Runtime Information

When a manual activity is executed, the workflow engine processes the allocation strategy defined in the metadata to retrieve the list of allocation targets for that task.

If the allocation strategy is of type function, the workflow engine processes the input mappings defined for the associated allocation function and invokes it to retrieve the list of allocation targets.

If the allocation strategy is of type classic rule, the workflow engine processes the mappings for the specified ruleset and calls the rules engine to run the ruleset to retrieve the list of allocation targets.

If the allocation strategy is of type CER rule, the workflow engine processes the CER class mappings that have been specified for the allocation strategy. The data from the workflow data object attributes is mapped into the corresponding CER class rule attributes. The primary class is then retrieved and the *targets* attribute is queried to retrieve the list of allocation targets.

If the allocation strategy is of type target, the allocation target is simply the one specified in the metadata and no further processing is required.

As described in the metadata for a workflow process (see “Process Definition Metadata” on page 12), a failure allocation strategy may be specified for a process. This strategy will be processed and used if the invocation of the allocation strategy associated with the task results in no allocation targets being returned.

The workflow engine then uses the `curam.custom.workflow.workresolver` property to determine the implementation of the function used to allocate tasks in the application. This function is then called by the workflow engine passing to it the list of allocation targets as determined by the allocation strategy and also details of the task to be allocated.

After the work resolver has been called for the task, the workflow engine makes a call to the method `checkTaskAssignment` in the `curam.core.sl.impl.TaskAssignmentChecker` class. This function will check the assignment status of the task (i.e. to ensure that it has been assigned to at least one user or organizational object (organization unit, position or job) or to a work

queue). If the task has not been assigned, the application property `curam.workflow.defaultworkqueue` is examined to see what has been specified as the default work queue for workflow. The task is then assigned to that work queue.

If the task has been assigned to one user and only one user after the work has been resolved, the system checks the value of the application property `curam.workflow.automaticallyaddtasktouser`. This flag controls whether the system will automatically add the specified task being processed to the list of that user's tasks to allow them to work on it. The default value for the property is NO but if it has been specified as YES, then the system will automatically add that task to the user's My Tasks list in their Inbox to allow them to work on it.

## Description of Context WDOs

The `Context_Task` workflow data object is available for both allocation function, classic and CER ruleset mappings. This context workflow data object and its attribute have already been described above: (see "Description of Context WDOs" on page 62).

## Business Object Associations

Manual activities, and indeed workflow in general, perform operations on entities that exist in the application. For this reason, it may be useful to associate a task with the entities that are related to it for that process. Business object associations essentially provide links between a task and any application entities of interest for that process. The quintessential examples in Cúram include the Case and Concern entities.

### Metadata

```
<manual-activity id="1" category="AC1">
  ...
  <task>
    ...
  </task>
  <allocation-strategy type="target"
    identifier="1"/>
  <event-wait>
    ...
  </event-wait>
  <biz-object-associations>
    <biz-object-association biz-object-type="BOT1">
      <wdo-attribute wdo-name="TaskCreateDetails"
        name="caseID"/>
    </biz-object-association>
    <biz-object-association biz-object-type="BOT2">
      <wdo-attribute wdo-name="
        PersonDetailsList[Context_Loop.loopCount]"
        name="identifier"/>
    </biz-object-association>
  </biz-object-associations>
</manual-activity>
```

#### **biz-object-associations**

This contains the details of all the business object associations that have been specified for the manual activity.

#### **biz-object-association**

This contains the details of one business object association that has been specified for that manual activity. This includes the business object type and the workflow data object attribute mapping associated with that type.

This workflow data object attribute mapping represents the unique identifier of the business object in the association (i.e. for a business object association of type Case, this would represent the unique identifier of the case being linked to the task).

#### **biz-object-type**

This details the actual business object type for the business object association for the manual activity. The business object type must be selected in the Process Definition Tool and is taken from the BusinessObjectType code-table.

### **Validations**

- The business object type specified must be a valid codetable code contained within the BusinessObjectType codetable.
- The workflow data object attribute mapped to the business object type of a manual activity business object association must be valid. This attribute type must be assignable to a type LONG as this represents a mapping to a unique identifier (e.g. a case identifier or participant identifier).
- If an indexed item from a list workflow data object (i.e. PersonDetailsList[Context\_Loop.loopCount]) is used in a business object association mapping, then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.
- If the Context\_Parallel workflow data object is used in a business object association mapping, then the activity containing the mapping must be a Parallel manual activity.
- If an indexed item from the Parallel List Workflow Data Object is used in a business object association mapping, then the activity containing the mapping must be a Parallel Activity (i.e. ParallelListWDO[Context\_Parallel.occurrenceCount]). The workflow data object being indexed by the Context\_Parallel Workflow Data Object must be the Parallel Activity List Workflow Data Object.

### **Code**

#### **Business Object Association Admin API**

A number of functions have been provided on the BusinessObjectAssociationAdmin class to allow the manipulation of business object associations. For further details of the functions available, see the associated Javadoc specification for the BusinessObjectAssociationAdmin class.

### **Runtime Information**

Business object associations have no functional impact on the execution of a manual activity. The workflow engine simply examines the metadata and places a record on the BizObjAssociation entity for each business object association specified. The business object type, the value of the workflow data object attribute mapping and the identifier of the newly created task associated with the manual activity are all used in the creation of this record.

## **Event Wait**

Since a manual activity requires some action to be taken by a user before it can be completed and the process can continue, there must be some way to notify the workflow engine when the work required has been performed. Since this semantic is similar to that of the event wait activity the event wait mechanism is re-used for manual activities.

## Prerequisites

- The details of an event wait and its associated metadata (which are also applicable to a manual activity) may be found in “Event Wait” on page 44.

## Description of Context WDOs

The Context\_Task workflow data object is available for use in the input mappings for deadline functions associated with the event wait of a manual activity. It is available for the input mappings associated with allocation function, classic or CER rule input mappings. It is also available to use as a mapping for the event match data of a specified event wait associated with a manual activity. This context workflow data object and its attribute have already been described above (see “Description of Context WDOs” on page 62).

---

## Decision

### Prerequisites

- The base details common to all the activity types supported by Cúram workflow are described in “Base Activity” on page 29 and are applicable to the decision activity described here.
- There are also workflow metadata constructs that are common between manual activities and decision activities (i.e. allocation strategy, task subject, task deadline etc). The details of these may be found in “Manual” on page 55.

### Overview

A typical requirement in business processes is to have a human agent make decisions that have simple answers. An example of such a decision is to approve or reject a case or to supply some simple information such as the age of the claimant. Using manual activities to solicit such information would require that a different user interface screen for each question be available in the application. This is cumbersome and since process definitions can change over time, such user interface screens would be somewhat temporary.

The Decision activity is a specialization of a Manual activity that drives a metadata driven user interface for asking simple questions. The questions and possible answers are in the activity metadata thus allowing a single user interface to be used for a wide range of questions. Two types of questions are currently supported. These are multiple choice type questions and questions requiring an answer that can be supplied in one field on the user interface.

### Task Details

Similar to a “Manual” on page 55, decision activities will notify users that they are required to do some work, and assign a task to them based on the allocation strategy defined. The task will automatically link to a user interface page in the application that assembles the decision question from the decision activity question metadata and moves the workflow forward once the decision answer has been provided. A decision activity, therefore, can have only one associated task action and requires no action page to be defined for that action.

In addition to the task action, a decision activity can have zero or more secondary actions associated with it. Secondary actions contain a link to a page which may provide supplementary information to help the user answer the decision question.

## Metadata

```
<decision-activity id="1">
...
  <allocation-strategy type="target" identifier="1" />
  <message>
    <message-text>
      <localized-text>
        <locale language="en">
          Decide the age of the user %1s for Case %2n.</locale>
        </localized-text>
      </message-text>
    </message-parameters>
    <message-parameters>
      <wdo-attribute wdo-name="TaskCreateDetails"
        name="userName" />
      <wdo-attribute wdo-name=
        "CaseList[Context_Loop.loopCount]"
        name="identifier" />
    </message-parameters>
  </message>
  <decision-action>
    <message>
      <message-text>
        <localized-text>
          <locale language="en">
            Participant Home Page %1n for Case %2n.
          </locale>
        </localized-text>
      </message-text>
    </message-parameters>
    <message-parameters>
      <wdo-attribute wdo-name="TaskCreateDetails"
        name="concernRoleID" />
      <wdo-attribute wdo-name=
        "CaseList[Context_Loop.loopCount]"
        name="identifier" />
    </message-parameters>
  </message>
</decision-action>
<secondary-actions>
  <secondary-action page-id="Case_viewDetails">
    <message>
      <message-text>
        <localized-text>
          <locale language="en">View case details.</locale>
        </localized-text>
      </message-text>
    </message>
  </secondary-action>
  <secondary-action page-id="Case_viewUserDetails">
    <message>
      <message-text>
        <localized-text>
          <locale language="en">View details for user %1s.
        </locale>
        </localized-text>
      </message-text>
      <message-parameters>
        <wdo-attribute wdo-name=
          "ChildDependents[Context_Loop.loopCount]"
          name="userName" />
      </message-parameters>
    </message>
    <link-parameter name="userName">
      <wdo-attribute wdo-name="ChildDependents"
        name="childName" />
    </link-parameter>
  </secondary-action>
</secondary-actions>
</multiple-occurring-action>
</secondary-action>
</secondary-actions>
```

**allocation-strategy**

This describes the allocation strategy used to determine the user assigned to the associated task. For details on allocation strategies, see “Allocation strategy” on page 62.

**message**

This represents the parameterized subject message of the task created. For full details on parameterized messages, see “Manual” on page 55.

**decision-action**

This represents the parameterized action text message associated with the task. The user will click on this action text to bring up the auto-generated user interface decision screen with the relevant question.

**deadline**

This describes the deadline details for the decision activity. If an answer has not been provided for the decision activity within the deadline duration specified, the associated deadline handler method is invoked. For more details on deadlines, see “Deadline” on page 48

**secondary-actions**

This describes any optional secondary actions which may be included with the decision activity.

**secondary-action**

A secondary action contains a parameterized message and a parameterized link to supporting information to help the user answer the decision question. For details of parameterized messages and parameterized links within actions, see “Metadata” on page 57

**page-id**

This represents the identifier of the target Cúram page which contains the supplementary information being linked to by the secondary action.

**multiple-occurring-action**

This signifies that this secondary action will represent a multiple occurring action. This means that if this metadata is specified for a secondary action, the workflow engine will create one secondary action record for each item in the list workflow data object specified as the multiple occurring action, when that activity is executed.

It should be noted that when the multiple occurring action is specified for a secondary action, then an attribute from the associated list workflow data object must be used as a link parameter for the secondary action.

**list-wdo-name**

The name of the list workflow data object for use with the multiple occurring action.

**Validations**

- An activity subject must be defined.
- Every workflow data object attribute mapped to the decision activity subject must be a valid workflow data object attribute.
- If an indexed item from a list workflow data object (i.e. CaseList[Context\_Loop.loopCount]) is used as a decision subject text parameter,

then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.

- If the `Context_Parallel` workflow data object is used as a decision subject text parameter, then the activity containing the mapping must be a `Parallel` decision activity.
- If an indexed item from the `Parallel List Workflow Data Object` is used as a decision subject text parameter, then the activity containing the mapping must be a `Parallel Activity` (i.e. `ParallelListWDO[Context_Parallel.occurrenceCount]`). The workflow data object being indexed by the `Context_Parallel Workflow Data Object` must be the `Parallel Activity List Workflow Data Object`.
- The number of placeholders used in the subject text and action text must equal the number of mapped workflow data object attributes (for all locales).
- If an indexed item from a list workflow data object (i.e. `CaseList[Context_Loop.loopCount]`) is used as a decision task action text parameter, then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.
- If the `Context_Parallel` workflow data object is used as a decision action text parameter, then the activity containing the mapping must be a `Parallel` decision activity.
- If an indexed item from the `Parallel List Workflow Data Object` is used as a decision action text parameter, then the activity containing the mapping must be a `Parallel Activity` (i.e. `ParallelListWDO[Context_Parallel.occurrenceCount]`). The workflow data object being indexed by the `Context_Parallel Workflow Data Object` must be the `Parallel Activity List Workflow Data Object`.
- An allocation strategy must be defined.
- The allocation target, function, classic or CER rule set specified as an allocation strategy must be valid. If the allocation type is function, it must be a valid Cúram business method and must exist on the application classpath. If the allocation type is classic or CER rule, it must be a valid ruleset.
- The optional deadline handler, if specified, must be a valid Cúram business method.
- All deadline handler input mappings must be valid. This means that all the input parameter fields required by the specified method are mapped to valid workflow data object attributes of the correct type.
- Each secondary action must have a page link specified, which cannot contain white spaces.
- Each secondary action must have a message specified.
- Secondary action message text must contain a number of placeholders equal to the number of message parameters specified.
- Secondary action message parameters must be mapped to valid workflow data object attributes of the correct type.
- Secondary action page link parameters must be mapped to valid workflow data object attributes.
- If an indexed item from a list workflow data object (i.e. `ChildDependents[Context_Loop.loopCount]`) is used in the secondary action text or secondary action link parameter mappings, then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.
- If the `Context_Parallel` workflow data object is used in the secondary action text or secondary action link parameter mappings, then the activity containing the mapping must be a `Parallel` decision activity.

- If an indexed item from the Parallel List Workflow Data Object in the secondary action text or secondary action link parameter mappings, then the activity containing the mappings must be a Parallel Activity (i.e. ParallelListWDO[Context\_Parallel.occurrenceCount]). The workflow data object being indexed by the Context\_Parallel Workflow Data Object must be the Parallel Activity List Workflow Data Object.
- The workflow data object specified for use in the multiple occurring action must be a valid workflow data object in the context of the containing workflow process definition. It must also be of type List.
- At least one attribute from the multiple occurring action list workflow data object must be utilized in the link parameters specified for a multiple occurring action.

### **Runtime Information**

When a decision activity is executed, the workflow engine creates the associated task. A snapshot of the workflow data object data required for the decision activity subject and action text parameters, and any secondary action message text and link parameters, is taken and stored. The allocation strategy associated with the decision activity is invoked to determine the user(s) who will be assigned the decision task. The workflow engine also creates an event wait for the DECISION.MADE event with the associated task identifier as the event match data. The workflow is then paused at this point, awaiting the raising of this event which will indicate the result of the decision made.

### **Question Details**

The decision activity currently supports both multiple choice and free text questions as question formats. The auto-generated decision page examines the question format required and generates the relevant question from the question metadata once the user clicks on the action associated with the task.

## Metadata

### Multiple Choice:

```
<decision-activity id="1">
...
<question>
  <message>
    <message-text>
      <localized-text>
        <locale language="en">
          Is the claimant, %1s, for Case %2n, over 18?
        </locale>
      </localized-text>
    </message-text>
    <message-parameters>
      <wdo-attribute wdo-name="Participant"
        name="userName" />
      <wdo-attribute wdo-name=
        "CaseList[Context_Loop.loopCount]"
        name="identifier" />
    </message-parameters>
  </message>
  <answers multiple-selection="false">
    <answer name="yesAnswer">
      <answer-text>
        <localized-text>
          <locale language="en">Yes</locale>
        </localized-text>
      </answer-text>
      <choice-output-mapping>
        <wdo-attribute wdo-name="DecisionResult"
          name="ageBracket" />
        <selected-value>18-65</selected-value>
        <not-selected-value>0-17</not-selected-value>
      </choice-output-mapping>
    </answer>
    <answer name="noAnswer">
      <answer-text>
        <localized-text>
          <locale language="en">No</locale>
        </localized-text>
      </answer-text>
      <choice-output-mapping>
        <wdo-attribute wdo-name="DecisionResult"
          name="ageBracket" />
        <selected-value>0-17</selected-value>
        <not-selected-value>18-65</not-selected-value>
      </choice-output-mapping>
    </answer>
  </answers>
</question>
...
</decision-activity>
```

#### question

This represents the question associated with the decision activity, which for a multiple choice question contains the metadata outlined below.

#### message

This represents the parameterized text of the question to be asked for all locales.

**answers**

This represents a list of answers the user can choose from for the multiple choice question.

**multiple-selection**

This represents a flag that indicates if the user can select multiple answers from those supplied, or whether only one can be selected.

**answer**

This represents an answer that the user can select. There must be at least one answer supplied for a multiple choice question.

**name** This represents the name of the answer. Once the user selects an answer or answers, the names of the selected answers are passed to the workflow engine and the process is progressed. As the engine treats these answers similar to workflow data object attributes, answer names must be valid Java identifiers.

**answer-text**

This represents the answer text that the user can select for all locales.

**choice-output-mapping**

This tag encloses the metadata that describes how the output from a multiple choice answer will be persisted.

**wdo-attribute**

The name of the workflow data object attribute used to store the value of the multiple choice answer.

**selected-value**

If specified, the value in this element will be persisted to the workflow data object attribute if that answer has been selected by the user. If the workflow data object attribute is a boolean type this value need not be specified, it will get a default value of true.

**not-selected-value**

If specified, the value in this element will be persisted to the workflow data object attribute if that answer has not been selected by the user. If the workflow data object attribute is a boolean type this value need not be specified, it will get a default value of false.

## Free Text:

```
<decision-activity id="1">
  ...
  <question>
    <message>
      <message-text>
        <localized-text>;
        <locale language="en">
          What is the age of the claimant, %1s?
        </locale>
      </localized-text>
    </message-text>
    <message-parameters>
      <wdo-attribute wdo-name="Participant"
        name="userName" />
    </message-parameters>
  </message>
  <free-text type="INT32">
    <wdo-attribute wdo-name="DecisionResult"
      name="ageOfClaimant" />
  </free-text>
</question>
  ...
</decision-activity>
```

### question

This represents the question associated with this decision activity, which for a free text question contains the metadata outlined below.

### message

This represents the parameterized text of the question to be asked for all locales.

### free-text

This contains the details of the free text answer that the user must supply.

**type** This represents the required data type of the free text answer that must be supplied.

### wdo-attribute

This represents the workflow data object attribute that maps the free text answer back into the workflow engine.

## Validations

- The answer format and question text must be specified for a decision activity.
- The number of placeholders used in question text must equal the number of mapped workflow data object attributes (for all locales).
- If an indexed item from a list workflow data object (i.e. CaseList[Context\_Loop.loopCount]) is used as a question text parameter, then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.
- If the Context\_Parallel workflow data object is used as a question text parameter, then the activity containing the mapping must be a Parallel decision activity.
- If an indexed item from the Parallel List Workflow Data Object is used as a question text parameter, then the activity containing the mapping must be a

Parallel Activity (i.e. ParallelListWDO[Context\_Parallel.occurrenceCount]). The workflow data object being indexed by the Context\_Parallel Workflow Data Object must be the Parallel Activity List Workflow Data Object.

- For a question with a Free Form Text answer format, the answer data type must be specified and the workflow data object attribute mapped must be valid and match the answer data type. The workflow data object attribute mapped cannot be a constant workflow data object attribute.
- For a question with a List answer format at least one answer option must be listed. All answer names must be valid Java attribute names.

## Runtime Information

When an answer for a decision activity question has been supplied, the DECISION.MADE event is raised with the task identifier of the decision activity task used as the event match data. The workflow event handler will process this event and this will cause the workflow process to be progressed.

If the answer supplied is a free text answer it will be mapped to the specified workflow data object attribute for use later in the process where required.

## Description of Context WDOs

The Context\_Decision workflow data object is available for use in a data item or function condition (see “Conditions” on page 103) for a transition from a decision activity. The attributes available will depend on the answer format defined for the activity.

### Free Text Answer

If the answer format is a free text answer the attribute available is:

#### **Context\_Decision.value**

The value of the free text answer supplied. This can be used in transition conditions and can be mapped to a specified workflow data object attribute.

### Multiple Choice Answer

In this instance, the Context\_Decision workflow data object will be populated with attributes for each of the answers available, each being of type boolean. This indicates whether that answer had been selected or not. In the multiple choice answer metadata snippet above, (“Multiple Choice” on page 78, if the user selected the first answer (Yes), this would be reflected with the following Context\_Decision workflow data object attribute being set to true:

#### **Context\_Decision.yesAnswer**

This represents a boolean indicating whether the yes answer for the question had been selected. This can only be used in transition conditions from the decision activity.

Alternatively, if the user selected the second answer (No), this would be reflected with the following Context\_Decision workflow data object attribute being set to true:

#### **Context\_Decision.noAnswer**

This represents a boolean indicating whether the no answer for the question had been selected. This also can only be used in transition conditions from the decision activity.

---

## Subflow

### Prerequisites

- The base details common to all the activity types supported by Cúram workflow are described in “Base Activity” on page 29 and are applicable to the subflow activity described here.

### Overview

When designing a complex business process it may become too large to manage as one monolithic process definition. A sub-flow activity allows another process definition to be enacted as part of another process.

It may be a prudent decision to design process definitions as a set of sub-flows regardless of whether there are concerns over size. This would allow sections of the business process to change without affecting others. Also the subflow processes could act as reusable components that customers can reuse in building their own higher order process definitions.

### Subflow Process

To enact a process as a subflow, the subflow activity must identify the process that will be enacted by name. As with the other process enactment mechanisms, the latest released version of the process is the one that will be enacted.

Subflows can be enacted *synchronously*. This means that the branch of the parent workflow containing the subflow activity that started the subflow process waits for that subflow process to finish before continuing.

Alternatively, a subflow may be enacted *asynchronously*. This means that once the subflow activity starts the subflow process, the branch containing that subflow activity continues immediately with the outcome of the subflow process having no effect on the parent process.

### Metadata

```
<subflow-activity id="1">
  ...
  <subflow workflow-process="ApproveCase" synchronous="true"/>
  ...
</subflow-activity>
```

#### **subflow**

##### **workflow-process**

The name of the workflow process to start when the activity is executed. Process names are case sensitive and the process name specified here must exactly match that of the process to start as a subflow.

##### **synchronous**

A flag to indicate whether the sub-flow should be executed synchronously or not (see: “Subflow Process”) relative to its parent process.

## Validations

- A workflow process for the subflow activity must be specified.
- The workflow process specified as the subflow must have at least one released version.

## Input Mappings

Data is supplied to the subflow when it is enacted from the parent process workflow data objects. The subflow activity defines the mapping between the parent process's workflow data objects and the subflows enactment data.

## Metadata

```
<subflow-activity id="1">
  ...
  <input-mappings>
    <mapping>
      <source-attribute wdo-name="MaintainCase"
        name="caseID" />
      <target-attribute wdo-name="ApproveCase"
        name="caseID" />
    </mapping>
    <mapping>
      <source-attribute wdo-name="MaintainCase"
        name="concernRoleID" />
      <target-attribute wdo-name="ApproveCase"
        name="concernRoleID" />
    </mapping>
    <mapping>
      <source-attribute wdo-name=
        "PersonDetailsList[Context_Loop.loopCount]"
        name="identifier" />
      <target-attribute wdo-name="PersonDetails"
        name="identifier" />
    </mapping>
    <mapping>
      <source-attribute wdo-name="ChildDetailsList"
        name="identifier" />
      <target-attribute wdo-name="ClaimantDependentList"
        name="identifier" />
    </mapping>
  </input-mappings>
</subflow-activity>
```

### input-mappings

This specifies how data is mapped from the currently executing process to a sub-process as enactment data when the sub-process is started. The process specified as a subflow may not have any workflow data object attributes marked as required at enactment in which case no input mappings are required.

### mapping

A mapping represents the data that will be pushed from a workflow data object attribute to an attribute in the process being enacted as a subflow. If a list of data is required to enact the subflow process, attributes from list workflow data objects can be used for this purpose. The number of mappings specified is governed by how many attributes are marked as required at enactment in the subflow process, since all such attributes must be populated when the process starts.

**source-attribute**

This represents a workflow data object attribute from the parent process to use to populate the associated attribute in the subflow when it is enacted.

**target-attribute**

This represents a workflow data object attribute from the subflow to be populated with data from the associated attribute in the parent process at enactment time.

**source/target-attribute****wdo-name**

This represents the name of a Cúram workflow data object as described in “Workflow Data Objects” on page 16).

**name** This represents the name of a Cúram workflow data object attribute as described in “Workflow Data Objects” on page 16).

**Validations**

- Every workflow data object attribute marked as *required for enactment* in the subflow must be specified in the input mappings. If no workflow data object attributes have been marked as required for enactment in the subflow process, then no input mappings should be specified.
- The data type of the workflow data object attribute specified by the target-attribute tag must match or be assignable from the attribute specified by the source-attribute tag.
- If an indexed item from a list workflow data object (i.e. PersonDetailsList[Context\_Loop.loopCount]) is specified in the source-attribute tag of the subflow input mapping, then that workflow data object must be a list workflow data object and the subflow activity containing the input mapping must be contained within a loop. The data type of the workflow data object attribute specified by the target-attribute tag must match or be assignable from the attribute specified by the source-attribute tag.
- If the specified subflow input mapping uses a list workflow data object then the workflow data object attributes for both the parent source-attribute and subflow process target-attribute must be list workflow data objects.

**Output Mappings**

Output Mappings are only applicable to *synchronous* subflow activities as asynchronous subflows can continue without completing the activity. Data is supplied to the parent process from the subflow activity after it completed. The subflow activity defines the mapping between a subflow workflow data object attribute and the parent process's workflow data object attribute.

**Metadata**

```

<subflow-activity id="1">
...
  <output-mappings>
    <mapping>
      <source-attribute wdo-name="SubflowCaseWDO"
                       name="participantName" />
      <target-attribute wdo-name="CaseWDO"
                       name="participantName" />
    </mapping>
    <mapping>
      <source-attribute wdo-name="SubflowChildDetailsList"
                       name="identifier" />
      <target-attribute wdo-name="ChildDetailsList"
                       name="identifier" />
    </mapping>
  </output-mappings>
...
</subflow-activity>

```

### output-mappings

This specifies how data is mapped from the invoked sub-process to the parent process when the sub-process has completed. The process specified as a subflow may not have any output mappings defined, in which case the subflow completes as normal.

### mapping

This represents the data that will be pushed from a subflow workflow data object attribute to an attribute in the parent process. If a list of data is being pushed from the subflow process to the parent process, attributes from list workflow data objects can be used for this purpose. The number of mappings specified is governed by the number of output mappings specified.

### source-attribute

This represents a workflow data object attribute from the subflow process which is used to populate the associated attribute in the parent process upon completion.

### target-attribute

This represents a workflow data object attribute from the parent to be populated with data from the associated attribute in the subflow process when completed.

### source/target-attribute

#### wdo-name

This represents the name of a Cúram workflow data object (as described in “Workflow Data Objects” on page 16).

**name** This represents the name of a Cúram workflow data object attribute (as described in “Workflow Data Objects” on page 16).

## Validations

- The parent target-attribute and subflow source-attribute workflow data object attributes used in the subflow output mapping must be valid within the context of the containing process definition.
- The data type of the workflow data object attribute specified by the parent target-attribute tag must match or be assignable from the attribute specified by the subflow source-attribute tag.

- If the specified subflow output mapping uses a list workflow data object then the mapped workflow data object attributes for both the parent target-attribute and subflow process source-attribute must be of type list.

---

## Loop Begin and Loop End

### Prerequisites

- The base details common to all the activity types supported by Cúram workflow are described in “Base Activity” on page 29 and are applicable to the loop begin/loop end activities described here.

### Overview

Many business processes are required to *repeat* until some condition is met. In Cúram, this is implemented using the loop-begin and loop-end activities. All activities that are between a loop-begin and its associated loop-end activity are repeated until the loop completes.

In a process definition, loop begin and loop end activities come in pairs, and the metadata allows each loop-begin to know its associated loop-end and vice versa. To add a sequence of activities to a loop, a transition is created from the loop-begin activity to the first activity to be repeated. Subsequent activities in the sequence are linked using transitions as would normally be done outside a loop; however, the last activity in the sequence has a transition to the loop-end activity. A common impulse is to also add a transition from the loop-end activity to the start to create the cycle; however, this is incorrect and results in an invalid process definition.

A loop must also specify the criteria the loop will use to determine whether or not to terminate. To support this, a loop in Cúram workflow has a loop-exit condition.

Loops can contain other loops as long as they are fully nested and do not interleave each other. This ensures that the loops and therefore the process definition remains a valid block structure as required by the Cúram workflow engine (see “Workflow Structure” on page 109).

### Loop Type

In addition to the loop-exit condition, a loop also specifies whether the condition should be tested before the loop executes (a while loop) or at the end of a loop execution (a do-while loop). A while loop may never execute the activities in the loop and jump to the activity following the loop if the exit condition is met at the start of the loop, whereas a do-while loop will execute the activities in the loop at least once.

## Metadata

### Loop Begin Activity

```

<loop-begin-activity id="1">
  ...
  <loop-type name="do-while"/>
  ...
  <condition>
    <expression id="1" data-item-lhs="Context_Loop.loopCount"
      operation="&lt;" data-item-rhs="UserAccountWDO.size()"/>
  </condition>
  <block-endpoint-ref activity-id="5"/>
</loop-begin-activity>

```

### loop-type

The `loop-type` specifies how the loop will be executed as detailed in “Loop Type” on page 86. The only two valid values for the `name` attribute are `while` and `do-while`.

### condition

The `condition` tag specifies the condition that will be evaluated based on Workflow Data Object values (see: “Workflow Data Objects” on page 16). When list workflow data objects are present in the workflow, two attributes that are not part of that workflow data object metadata are made available when creating a loop condition expression using a list workflow data object. These are as follows:

- `size()` : This will evaluate to a number (of type INTEGER) to indicate the number of items in the list.
- `isEmpty()` : This will evaluate to a BOOLEAN flag to indicate if the list contains any elements or not.

The actual condition metadata is used in other places in the process definition metadata and is thus described in the dedicated chapter, “Conditions” on page 103.

### block-endpoint-ref

The `block-endpoint-ref` in this context allows the `loop-begin-activity` to recognize its associated `loop-end-activity`. This information is useful to the workflow engine when executing the loop. For example, when a `while` loop's exit condition evaluates to `true` before the loop executes, the `block-endpoint-ref` tells the workflow engine which activity to jump to and continue the execution of the process.

## Loop End Activity

```

<loop-end-activity id="3">
  ...
  <block-endpoint-ref activity-id="1"/>
</loop-end-activity>

```

### block-endpoint-ref

The `block-endpoint-ref` in this context allows the `loop-end-activity` to recognize its associated `loop-begin-activity`. This information is useful to the workflow engine when executing the loop. For example, if after the execution of a loop the exit condition evaluates to `false`, the `block-endpoint-ref` tells the workflow engine which activity to jump to in order to begin another iteration of the loop.

## Runtime Information

It is expected that any activity within a loop will be executed more than once during the execution of a process instance. To prevent the process instance data for the activity becoming corrupted by subsequent iterations, each activity instance is associated with a specific iteration and so is uniquely identifiable by the workflow engine regardless of the number of times the loop is executed.

## Description of Context WDOs

The Context\_Loop workflow data object is available on the following occasions:

- When creating the loop condition associated with a loop-begin activity.
- When creating the outgoing transition conditions from a loop-begin activity, or from any activity contained within a loop (see “Conditions” on page 103).
- When creating the input mappings for any automatic activity or subflow activity within a loop.
- When creating the input mappings for any allocation strategy function or deadline handler function present in an activity within a loop.
- When specifying a subject text parameter for a Manual or Decision Activity that is contained within a loop, or for a notification attached to an activity that is contained within a loop.
- When specifying action text parameters and action link parameters for a Manual or Decision Activity that is contained within a loop, or for a notification attached to an activity that is contained within a loop.
- When specifying the identifier for a business object association for a Manual Activity that is contained within a loop.
- When specifying a question text parameter for both a free-form or multiple choice question for a Decision Activity that is contained within a loop.
- When specifying a body text parameter for a notification attached to an activity that is contained within a loop.

The Context\_Loop workflow data object attributes available are:

### **Context\_Loop.loopCount**

The number of times that a loop has been iterated over.

---

## Parallel

### Prerequisites

- The base details common to all the activity types supported by Cúram workflow are described in “Base Activity” on page 29 and are applicable to the parallel activity described here.
- As parallel activities wrap existing activities in a workflow process definition, the metadata described in “Manual” on page 55 and “Decision” on page 73 is also relevant to the parallel activity described here.

### Overview

In business processes, it may be required to send multiple tasks to different human agents at the same time to expedite the progress of the overall process. When the number of parallel paths are known at development time this can easily be achieved using a split. However in some cases the number of paths will not be known until runtime. Such situations can be modeled using parallel activities.

A parallel activity acts as a wrapper around existing activities. The effect of using one of these new activities at runtime is that multiple instances of the wrapped activity are executed in parallel. To date, the only supported types of wrapped activity are Manual (“Manual” on page 55) and Decision (“Decision” on page 73) activities. Therefore, executing a parallel activity currently equates to the creation and allocation of multiple tasks in parallel.

## Metadata

A parallel activity must specify the type of activity it wraps. A list workflow data object must also be associated with the parallel activity. The number of items in this list workflow data object will then determine the number of instances of that wrapped activity that will be created by the workflow engine at runtime.

### Generic Metadata for a Parallel Activity

```
<parallel-activity id="1" category="AC1">
  <list-wdo-name>EmployerDetailsListWDO</list-wdo-name>
  <manual-activity>
    <name>
      <localized-text>
        <locale language="en">
          CheckEmployerDetailsTasks</locale>
        </localized-text>
      </name>
      .....
    </manual-activity>
  </parallel-activity>
```

or .....

```
<parallel-activity id="1" category="AC1">
  <list-wdo-name>ChildDetailsListWDO</list-wdo-name>
  <decision-activity>
    <name>
      <localized-text>
        <locale language="en">ValidateChildDetails</locale>
      </localized-text>
    </name>
    .....
  </decision-activity>
</parallel-activity>
```

#### manual-activity/decision-activity

This reflects the type of activity wrapped by the parallel activity. Currently, two types of wrapped activities are supported, “Manual” on page 55 and “Decision” on page 73 activities. The types of activity that may be wrapped by a parallel activity can be seen in the `ParallelActivityType` codetable.

#### list-wdo-name

Each parallel activity must have a list workflow data object associated with it. The number of instances of the wrapped activity that are created at runtime is determined by the number of items in this list workflow data object.

### Metadata for a Parallel Manual Activity

The example below illustrates the metadata associated with the wrapped activity of type Manual. This metadata is *exactly* the same that as that seen for a manual activity described in “Manual” on page 55 and hence will not be described here again. Any validations pertaining to the parallel manual activity mappings are also

described in “Manual” on page 55. The Context\_Parallel Workflow Data Object and an indexed item from the Parallel Activity List WDO can be used in all the available mappings for a Parallel Manual Activity. Examples of such usage can be seen below:

```
<parallel-activity id="1" category="AC1">
  <list-wdo-name>EmployerDetailsListWDO</list-wdo-name>
  <manual-activity>
    ...
    <task>
      <message>
        <message-text>
          <localized-text>
            <locale language="en">Check employer
              details for %1s. This is employer number: %1n.
            </locale>
          </localized-text>
        </message-text>
        <message-parameters>
          <wdo-attribute
            wdo-name=
"EmployerDetailsListWDO[Context_Parallel.occurrenceCount]"
            name="fullName" />
          <wdo-attribute
            wdo-name=
"Context_Parallel" name="occurrenceCount" />
        </message-parameters>
      </message>
    </task>
    ...
    <event-wait wait-on-all-events="false">
      <events>
        <event identifier="1" event-class="EMPLOYER"
          event-type="DETAILSCHECKED">
          <event-match-attribute wdo-name=
"EmployerDetailsListWDO[Context_Parallel.occurrenceCount]"
            name="identifier" />
        </event>
      </events>
    </event-wait>
    <biz-object-associations>
      <biz-object-association biz-object-type="BOT2">
        <wdo-attribute
          wdo-name=
"EmployerDetailsListWDO[Context_Parallel.occurrenceCount]"
          name="identifier" />
      </biz-object-association>
    </biz-object-associations>
  </manual-activity>
</parallel-activity>
```

### Metadata for a Parallel Decision Activity

The example below illustrates the metadata associated with the wrapped activity of type Decision. This metadata is *exactly* the same that as that seen for a decision activity described in “Decision” on page 73 and hence will not be described here again. Any validations pertaining to the parallel decision activity mappings are also described in “Decision” on page 73. The Context\_Parallel Workflow Data Object and an indexed item from the Parallel Activity List WDO can be used in all the available mappings for a Parallel Decision Activity. Examples of such usage can be seen below:

```

<parallel-activity id="1" category="AC1">
  <list-wdo-name>ChildDetailsListWDO</list-wdo-name>
  <decision-activity>
    ...
    <message>
      <message-text>
        <localized-text>
          <locale language="en">In this task the details
            for child %1s must be validated. This is child
            number: %1n.
          </locale>
        </localized-text>
      </message-text>
      <message-parameters>
        <wdo-attribute
          wdo-name=
"ChildDetailsListWDO[Context_Parallel.occurrenceCount]"
          name="fullName" />
        <wdo-attribute
          wdo-name=
"Context_Parallel" name="occurrenceCount" />
      </message-parameters>
    </message>
    <decision-action>
      <message>
        <message-text>
          <localized-text>
            <locale language="en">Validate the child details
              for %1s associated with this case %2n.</locale>
          </localized-text>
        </message-text>
        <message-parameters>
          <wdo-attribute
            wdo-name=
"ChildDetailsListWDO[Context_Parallel.occurrenceCount]"
            name="fullName" />
          <wdo-attribute wdo-name="CaseDetails"
            name="identifier" />
        </message-parameters>
      </message>
    </decision-action>
    ...
    <question>
      <message>
        <message-text>
          <localized-text>
            <locale language="en">Are the details for this
              child whose first name is %1s and second name
              %2s correct?</locale>
          </localized-text>
        </message-text>
        <message-parameters>
          <wdo-attribute
            wdo-name=
"ChildDetailsListWDO[Context_Parallel.occurrenceCount]"
            name="firstName" />
          <wdo-attribute
            wdo-name=
"ChildDetailsListWDO[Context_Parallel.occurrenceCount]"
            name="surname" />
        </message-parameters>
      </message>
      <answers multiple-selection="false">
        <answer name="answerYes">
          <answer-text>
            <localized-text>
              <locale language="en">Yes</locale>
            </localized-text>
          </answer-text>
        </answer>
        <answer name="answerNo">
          <answer-text>
            <localized-text>

```

## Validations

- A workflow data object must be specified for a parallel activity. This must be a list workflow data object and it must be valid in the context of the containing workflow process definition.
- All of the other validations pertaining to parallel activities are described in the chapters that describe the activities that a parallel activity can wrap (i.e. “Manual” on page 55 and “Decision” on page 73).

## Runtime Information

The workflow engine loads the instance data for the list workflow data object associated with the parallel activity. For each item in the list workflow data object, a new instance of the wrapped activity is created and executed. The details of what occurs when these instances of the wrapped activity are executed may be found in the relevant chapters describing the activities that a parallel activity can wrap (“Manual” on page 55 and “Decision” on page 73).

At runtime, the Workflow Engine treats a Parallel Activity as if it were multiple activities, contained within a *Parallel (AND) Split/Join* block. One Activity Instance is created per item in the Parallel Activity List WDO (e.g. if that list contains three items, then three Activity Instances will be created). This ensures that all of the activity instances associated with the parallel activity must be completed before the actual parallel activity is deemed to be complete and the workflow can progress.

In order to resolve the mappings associated with a Parallel Activity, each instance of the wrapped activity is associated with one item from the Parallel Activity List WDO. The item is indexed using the Context\_Parallel Workflow Data Object (e.g. ChildDetailsListWDO[Context\_Parallel.occurrenceCount]).

## Description of Context WDOs

Each Parallel Activity Instance is associated with one item from the Parallel Activity List WDO. This item is accessed by using the Context\_Parallel Workflow Data Object to index the Parallel Activity List WDO (e.g. ChildDetailsListWDO[Context\_Parallel.occurrenceCount]). Indexed items can then be used to map data in the usual way. Examples of such mappings may be seen in the metadata examples shown above (see “Metadata for a Parallel Manual Activity” on page 89 and “Metadata for a Parallel Decision Activity” on page 90. The one attribute available on this workflow data object is:

### **Context\_Parallel.occurrenceCount**

Each Parallel Activity Instance is associated with one item from the Parallel Activity List WDO. The occurrenceCount attribute is the index of that item within the Parallel Activity List WDO. It is of type INTEGER and is a zero-based index.

---

## Activity Notifications

### Overview

The workflow engine is able to notify interested users about the progress of a workflow process instance. Essentially the workflow engine can raise a notification when an activity executes if the notification has been specified in the associated process definition metadata. A notification is specified for an activity as additional metadata that can be attached to any activity type.

When the workflow engine executes an activity it checks whether a notification has been configured for that activity. If one exists, a notification is created by the workflow engine detailing that a particular step in the workflow process has been performed. The delivery of these notifications to the user is determined by the notification delivery mechanism configured in the Cúram application. Notifications may be delivered using emails, as alerts sent to a user's inbox, or using both emails and alerts.

## **Notification Details**

A notification is simply information that is sent to a human agent when a step in the process executes. Notifications manifest themselves as alerts in a user's inbox or as emails. The agents to which the notification must be sent are determined by the allocation strategy (see “Notification Allocation Strategy” on page 97) specified for the notification. The details that are displayed to the user in the alert or email are specified as part of the notification metadata.

### **Metadata**

```

<manual-activity id="1" category="AC1">
...
<notification delivery-mechanism="DM1">
  <subject>
    <message>
      <message-text>
        <localized-text>
          <locale language="en">
            The case number %1n for Claimant %2s has
            been closed.
          </locale>
        </localized-text>
      </message-text>
      <message-parameters>
        <wdo-attribute wdo-name=
          "CaseList[Context_Loop.loopCount]"
          name="identifier" />
        <wdo-attribute wdo-name="PersonDetails"
          name="userName" />
      </message-parameters>
    </message>
  </subject>
  <body>
    <message>
      <message-text>
        <localized-text>
          <locale language="en">
            This case concerned %1n and claimant %2s.
          </locale>
        </localized-text>
      </message-text>
      <message-parameters>
        <wdo-attribute wdo-name=
          "CaseList[Context_Loop.loopCount]"
          name="identifier" />
        <wdo-attribute wdo-name="PersonDetails"
          name="fullName" />
      </message-parameters>
    </message>
  </body>
  <allocation-strategy type="target" identifier="1" />
  <actions>
    <action page-id="viewTaskHome" principal-action="false">
      <message>
        <message-text>
          <localized-text>
            <locale language="en">
              View the task associated with the %1n case.
            </locale>
          </localized-text>
        </message-text>
        <message-parameters>
          <wdo-attribute wdo-name="TaskCreateDetails"
            name="caseID" />
        </message-parameters>
      </message>
      <link-parameter name="childID">
        <wdo-attribute wdo-name="ChildDependents"
          name="childID" />
      </link-parameter>
      <multiple-occurring-action>
        <list-wdo-name>ChildDependents</list-wdo-name>
      </multiple-occurring-action>
    </action>
    <action page-id="viewCaseHome" principal-action="false">
      <message>
        <message-text>
          <localized-text>
            <locale language="en">
              View the case details for %1n.
            </locale>
          </localized-text>
        </message-text>
      </message>
    </action>
  </actions>
</notification>

```

**delivery-mechanism**

This describes the mechanism used to deliver the notification. The delivery mechanisms available are specified in the application codetable `DeliveryMechanism`. Both the Cúram application and customers can extend this codetable and add further delivery mechanisms if required. The delivery mechanism specified plays no functional role in the workflow engine as it simply calls the delivery mechanism configured in the application to deliver the newly created notification.

**subject**

This represents a parameterized text message detailing the subject of the notification for all locales. This subject will be displayed in the user's inbox for the notification alert. For details on parameterized messages, see "Manual" on page 55.

**body** This represents a parameterized text message representing the body of the text associated with this notification for all locales. When the user clicks on the notification subject in the inbox, this body text will be displayed as the full text of the notification.

**allocation-strategy**

This represents the allocation strategy used to determine the agents to which this notification will be sent to (see "Notification Allocation Strategy" on page 97).

**actions**

In the same way a "Manual" on page 55 can have actions associated with its task, a notification can have associated actions the notified user can take. This piece of metadata represents the details of these notification actions and the metadata details for actions is detailed in "Task details" on page 56.

**multiple-occurring-action**

This signifies that this notification action will represent a multiple occurring action. This means that if this metadata is specified for a notification action, the workflow engine will create one action record for each item in the list workflow data object specified as the multiple occurring action, when that activity is executed.

It should be noted that when the multiple occurring action is specified for a notification action, then an attribute from the associated list workflow data object must be used as a link parameter for the notification action.

**list-wdo-name**

The name of the list workflow data object for use with the multiple occurring action.

**Validations**

- A subject must be defined for the notification.
- Every workflow data object attribute mapped to a notification subject must exist in the containing process definition and be a valid workflow data object.
- If an indexed item from a list workflow data object (i.e. `CaseList[Context_Loop.loopCount]`) is used as a notification subject text parameter, then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.
- If the `Context_Parallel` workflow data object is used as a notification subject text parameter, then the activity containing the notification must be a `Parallel` activity.

- If an indexed item from the Parallel List Workflow Data Object is used as a notification subject text parameter, then the activity containing the mapping must be a Parallel Activity (i.e. ParallelListWDO[Context\_Parallel.occurrenceCount]). The workflow data object being indexed by the Context\_Parallel Workflow Data Object must be the Parallel Activity List Workflow Data Object.
- A notification body must be defined.
- Every workflow data object attribute mapped to a notification body must exist in the containing process definition and be a valid workflow data object.
- If an indexed item from a list workflow data object (i.e. CaseList[Context\_Loop.loopCount]) is used as a notification body text parameter, then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.
- If the Context\_Parallel workflow data object is used as a notification body text parameter, then the activity containing the notification must be a Parallel activity.
- If an indexed item from the Parallel List Workflow Data Object is used as a notification body text parameter, then the activity containing the mapping must be a Parallel Activity (i.e. ParallelListWDO[Context\_Parallel.occurrenceCount]). The workflow data object being indexed by the Context\_Parallel Workflow Data Object must be the Parallel Activity List Workflow Data Object.
- An allocation strategy must be defined for an activity notification.
- If a function is specified as the notification allocation strategy, it must be a valid Cúram business method that returns an AllocationTargetList object.
- If the allocation type is classic or CER rule, then the specified ruleset must be valid.
- A delivery mechanism must be defined for an activity notification.
- The workflow data object attributes mapped to the notification action text and notification action link parameters for a notification action must exist in the containing process definition.
- If an indexed item from a list workflow data object (i.e. PersonDetailsList[Context\_Loop.loopCount]) is used as a notification action text or notification action link parameter mapping, then the workflow data object must be a list workflow data object and the activity containing the mapping must be contained within a loop.
- If the Context\_Parallel workflow data object is used as a notification action text or notification action link parameter mapping, then the activity containing the notification must be a Parallel activity.
- If an indexed item from the Parallel List Workflow Data Object is used as a notification action text or notification action link parameter mapping, then the activity containing the mapping must be a Parallel Activity (i.e. ParallelListWDO[Context\_Parallel.occurrenceCount]). The workflow data object being indexed by the Context\_Parallel Workflow Data Object must be the Parallel Activity List Workflow Data Object.
- The number of placeholders used in the notification subject text, notification action text and notification body text must equal the number of mapped workflow data object attributes (for all locales).
- The workflow data object specified for use in the multiple occurring action must be a valid workflow data object in the context of the containing workflow process definition. It must also be of type List

- At least one attribute from the multiple occurring action list workflow data object must be utilized in the link parameters specified for a multiple occurring action.

### **Code**

For each action defined, the action page must refer to a valid Cúram page in the application whose page parameters are fully populated by the action link parameters contained in the notification metadata.

A `LocalizableStringResolver` API is provided to the application which allows for parameterized message strings to be resolved. The methods in this API will resolve and return the specified message for the required locale. Along with this, any workflow data objects to be used in the message placeholders will be resolved and included as part of the string returned.

As part of the `LocalizableStringResolver` API, a `NotificationStringResolver` interface is provided for resolving the parameterized messages associated with notifications. The notification subject, body, and action text can be resolved for use in the application using the methods contained in this API. The application should use these methods to process the notification once the workflow engine invokes the associated notification delivery method in the application.

### **Runtime Information**

After the workflow engine has completed executing an activity, it checks whether an associated notification has been defined for that activity. If one has been defined, the engine determines the users to be notified from the allocation strategy employed and calls the notification delivery method in the application with the notification details.

## **Notification Allocation Strategy**

### **Prerequisites**

The notification allocation strategy determines the user or users to be notified once the associated activity has occurred. Defining the notification allocation strategy to be used is exactly the same as that used for manual activity tasks (see “Allocation strategy” on page 62).

### **Code**

The application must implement the `NotificationDelivery` callback interface to determine how notifications are handled in the application.

The workflow engine will call the `deliverNotification` method in the `curam.util.workflow.impl.NotificationDelivery` implementation class in order to process the notification. The engine will pass both the list of allocation targets determined by the allocation strategy and the details of the required notification to this application method.

The application property `curam.custom.notifications.notificationdelivery` defines what implementation of the `NotificationDelivery` interface will be used by the workflow engine to process the notification.

The `deliverNotification` method in this default implementation class is overloaded. This is because the various allocation strategy types return the allocation targets in different formats. However this is an implementation detail that developers of custom notification delivery classes should not have to deal with especially since the business processing for all versions of the method should

be the same.

```
package curam.util.workflow.impl;

...

public interface NotificationDelivery {

    boolean deliverNotification(
        final NotificationDetails notificationDetails,
        final Object allocationTargets);

    boolean deliverNotification(
        final NotificationDetails notificationDetails,
        final Map allocationTargets);

    boolean deliverNotification(
        final NotificationDetails notificationDetails,
        final String allocationTargetID);

    ...
}
```

To mitigate against this issue the `curam.core.sl.impl.DefaultNotificationDeliveryAdapter` provides a more convenient mechanism for implementing a work resolver. This class implements the different methods and converts their input parameters into allocation target lists allowing developers of custom notification delivery logic to extend this class and implement one method that is called regardless of the source of the allocation targets.

```
package curam.core.sl.impl;

...

public abstract class DefaultNotificationDeliveryAdapter
    implements curam.util.workflow.impl.NotificationDelivery {

    public abstract boolean deliverNotification(
        final NotificationDetails notificationDetails,
        final AllocationTargetList allocationTargets);

    ...
}
```

In addition to this adapter class the application ships with a notification delivery implementation that is used out-of-the-box. This class is called `curam.core.sl.impl.DefaultNotificationDelivery` and it also serves as an example of how to extend the adapter.

The notification delivery strategies are listed in the `DELIVERYMECHANISM` code table. Adding a new strategy is simply a matter of extending this code table with a new strategy (for example SMS) and implementing a delivery strategy that recognizes this code and performs the appropriate logic. However since the notification delivery class is set using a single application property, replacing the `curam.core.sl.impl.DefaultNotificationDelivery` class would disable out-of-the-box delivery mechanisms. If the goal is to extend rather replacing the out-of-the-box delivery mechanisms, custom classes should extend the `curam.core.sl.impl.DefaultNotificationDelivery` in a way that preserves the original functionality. The `curam.core.sl.impl.DefaultNotificationDelivery` class

has been implemented with this in mind.

```
package curam.core.sl.impl;

public class DefaultNotificationDelivery
    extends DefaultNotificationDeliveryAdapter {

    public boolean deliverNotification(
        NotificationDetails notificationDetails,
        AllocationTargetList allocationTargetList) {
        return selectDeliveryMechanism(
            notificationDetails, allocationTargetList);
    }

    protected boolean selectDeliveryMechanism(
        NotificationDetails notificationDetails,
        AllocationTargetList allocationTargetList) {

        boolean notificationDelivered = false;
        if (notificationDetails.deliveryMechanism.equals(
            curam.codetable.DELIVERYMECHANISM.STANDARD)) {
            notificationDelivered = standardDeliverNotification(
                notificationDetails, allocationTargetList);
        } else if (
            ...
            return notificationDelivered;
        }

        ...
    }
}
```

The `curam.core.sl.impl.DefaultNotificationDelivery` class implements the `deliverNotification` method from the abstract adapter but immediately delegates the identification of the mechanism to use to a protected method. The protected `selectDeliveryMechanism` method can be overridden by subclasses to identify any custom delivery mechanisms and perform the appropriate operations as shown in the example below:

```
public class CustomNotificationDeliveryStrategy
    extends DefaultNotificationDelivery {

    protected boolean selectDeliveryMechanism(
        NotificationDetails notificationDetails,
        AllocationTargetList allocationTargetList) {

        boolean notificationDelivered = false;
        boolean superNotificationDelivered = false;
        superNotificationDelivered = super.selectDeliveryMechanism(
            notificationDetails, allocationTargetList);
        if (notificationDetails.deliveryMechanism.equals(
            curam.codetable.DELIVERYMECHANISM.CUSTOM)) {
            notificationDelivered = customDeliverNotification(
                notificationDetails, allocationTargetList);
        }
        return (superNotificationDelivered || notificationDelivered);
    }
}
```

Notice that the `selectDeliveryMechanism` method in the custom class first delegates to its super class before executing any of its own logic. Extending the functionality in this way allows custom classes to invoke the out-of-the-box delivery mechanism without having to know the specific codes the parent class recognizes. This approach is also upgrade friendly as if a future version of Cúram supports more

delivery mechanisms out-of-the-box a custom class implemented as shown here will not need to change to avail of the new functionality.

The boolean flag returned from the notification delivery function above is used to indicate to the Workflow Engine if the notification was delivered to at least one user on the system. If it was not, then the engine writes a workflow audit record detailing this fact.

---

## Transitions

### Overview

Transitions provide the links between activities. They are the primary flow control construct and dictate the order in which activities will be executed. Transitions are unidirectional and an activity can have multiple outgoing and incoming transitions forming branch and synchronization points respectively. Since every process definition must have one start and one end activity (see “Base Activity” on page 29), a process definition can be thought of informally as a directed graph in which activities are the vertices, transitions are the arcs and every path from the start activity eventually leads to the end activity.

### Metadata

```

<workflow-process id="32456" .... >
  <name>WorkflowTestProcess</name>
  ...
  <wdos>
  ...
  </wdos>
  <activities>
    <start-process-activity id="512">
      ...
    </start-process-activity>
    <route-activity id="513" category="AC1">
      ...
    </route-activity>
    <route-activity id="514" category="AC1">
      ...
    </route-activity>
    <end-process-activity id="515">
      ...
    </end-process-activity>
  </activities>
  <transitions>
    <transition id="1" from-activity-idref="512"
      to-activity-idref="513" />
    <transition id="2" from-activity-idref="513"
      to-activity-idref="514">
      <condition>
        <expression id="5"
          data-item-lhs="TaskCreateDetails.reservedByInd"
          operation="==" data-item-rhs="true"
          opening-brackets="2" />
        <expression id="6"
          data-item-lhs="TaskCreateDetails.subject"
          operation="&gt;"
          data-item-rhs="&quot;MANUAL&quot;"
          conjunction="and" closing-brackets="1" />
        <expression id="7"
          data-item-lhs="TaskCreateDetails.status"
          operation="!="
          data-item-rhs="&quot;OPEN&quot;"
          conjunction="or" />
        <expression id="8"
          data-item-lhs="TaskCreateDetails.status"
          operation="&lt;="
          data-item-rhs="&quot;INPROGRESS&quot;"
          conjunction="or" closing-brackets="1" />
      </condition>
    </transition>
    <transition id="3" from-activity-idref="514"
      to-activity-idref="515">
  </transitions>
</workflow-process>

```

### transitions

A workflow process definition must contain at least one transition. This contains the details of all of the transitions between the activities in the specified workflow process definition.

### transition

This contains the details of one transition between two activities in the specified workflow process definition. The following mandatory fields that constitute a transition are described below:

**id** This is a 64-bit identifier supplied by the Cúram key server when transitions are created in the Process Definition Tool (PDT). The transition identifier is required to be unique within a process

definition but global uniqueness within all of the process definitions on the system is not required.

**from-activity-idref**

This is the 64-bit identifier of the source activity of the transition.

**to-activity-idref**

This is the 64-bit identifier of the target activity of the transition.

**condition**

Transitions can optionally have a condition to decide whether or not the given transition will be followed. A condition is a list of expressions that perform logical operations on workflow data objects attributes. Conditions are described in more detail in “Conditions” on page 103

## Validations

- The source activity defined for the transition must be a valid activity within the containing workflow process definition.
- The target activity defined for the transition must be a valid activity within the containing workflow process definition.
- The source and target activities defined for a transition cannot be the same activity.
- The start process activity in a workflow process definition must not contain any incoming transitions.
- The end process activity in a workflow process definition must not contain any outgoing transitions.
- All activities defined in the workflow process definition, except for the end process activity, must contain at least one inbound transition.
- All activities defined in the workflow process definition, except for the start process activity, must contain at least one outbound transition.

## Runtime Information

Activities that perform some application related work (as opposed to workflow engine only work such as route and end process activities) require a clear transactional boundary between the engine and application code. It is also useful to have asynchronous invocations between the workflow engine and the application (e.g. a user should not have to wait while workflow transitions to the next activity before control is returned to them in the user interface).

To this end, there are three distinct functions present in a workflow activity, `start()`, `execute()` and `complete()`. After the completion of an activity in the workflow process instance, the workflow engine calls the function to continue the process. This function evaluates the outgoing transitions from that activity to determine which one(s) will be followed.

For each activity to be followed, the corresponding `start()` function is called. The appropriate activity instance data is then set up for that activity. If the activity is to be executed directly with no JMS (Java Message Service (JMS) API is a part of Java EE) messaging required (i.e. a route activity is always executed directly as there is no application related work involved), the `execute()` method is called here. Otherwise, a JMS message is sent to execute the specified activity (i.e. an automatic activity). The workflow message handler resolves the process and activity specified in the message and calls the `execute()` function on the activity.

After calling the application code to carry out the work specified by the activity, another message is sent to complete the activity. Again, the workflow message handler resolves the process and activity specified in the message and calls the complete function for the activity. After marking the activity as complete, the function to continue the process is called again to resolve the set of transitions to be followed from the completed activity and the process begins again.

---

## Conditions

### Overview

The flow control constructs described in “Transitions” on page 100 and “Loop Begin and Loop End” on page 86 require or support the evaluation of conditions to determine how the workflow should proceed. The Loop Begin activities must have some metadata that specifies the loop exit conditions, while transitions can optionally have a condition to decide whether or not the given transition will be followed.

This chapter describes the process definition metadata construct that represents a condition. A condition is a list of expressions that perform logical operations on workflow data objects attributes. The condition itself is a compound whose value is conjunction or disjunction of its constituent expressions. The parent constructs (loops and transitions) are responsible for taking appropriate actions as a result of the evaluation of conditions.

### Metadata

```

<workflow-process id="32456" ..... >
...
<activities>
...
</activities>
<transitions>
  <transition id="1" from-activity-idref="512"
              to-activity-idref="513">
    <condition>
      <expression id="5"
                  data-item-lhs="TaskCreateDetails.reserveToMeInd"
                  operation="==" data-item-rhs="true"
                  opening-brackets="2"/>
      <expression id="6"
                  data-item-lhs="TaskCreateDetails.caseID"
                  operation="&amp;&gt;"
                  data-item-rhs="2" conjunction="and"
                  closing-brackets="1"/>
      <expression id="7"
                  data-item-lhs="TaskCreateDetails.status"
                  operation="!="
                  data-item-rhs="&quot;Completed&quot;"
                  conjunction="or"/>
      <expression id="8"
                  data-item-lhs="TaskCreateDetails.status"
                  operation="&amp;&lt;="
                  data-item-rhs="&quot;Closed&quot;"
                  conjunction="or" closing-brackets="1"/>
    </condition>
  </transition>
  <transition id="2" from-activity-idref="512"
              to-activity-idref="513">
    <condition>
      <expression id="9" function="isNothing"
                  data-item-rhs="TaskCreateDetails.subject"/>
    </condition>
  </transition>
  <transition id="3" from-activity-idref="513"
              to-activity-idref="514">
    <condition>
      <expression id="10"
                  data-item-rhs="TaskCreateDetails.reserveToMeInd"
                  conjunction="and" function="not" />
    </condition>
  </transition>
  <transition id="4" from-activity-idref="514"
              to-activity-idref="515">
    <condition>
      <expression id="6"
                  data-item-lhs
                    ="ClaimantDependents[Context_Loop.loopCount]"
                  operation="&amp;&gt;"
                  data-item-rhs="20"
                  conjunction="and"
                  closing-brackets="1"/>
    </condition>
  </transition>
</transitions>
</workflow-process>

```

### condition

This metadata is mandatory for a loop begin activity (as a loop must have an exit condition specified for it) but optional for a transition (a transition may not have a condition specified for it). It contains the details of all the expressions defined for the condition.

### expression

This contains the details of one expression contained in a condition. There may be one or many expressions specified for an associated condition. Two types of expression may be defined in a condition. These are function expressions (using one of two predefined functions, `not()` and `isNothing()`) and data item expressions (where the condition expression created applies the chosen operator to either two workflow data object attributes, or a workflow data object attribute and a constant). A transition expression consists of the following attributes:

**id** This represents a 64-bit identifier supplied by the Cúram key server when transition expressions are created in the PDT. The expression identifier is required to be unique within a process definition but global uniqueness within all of the process definitions on the system is not required.

### data-item-rhs

This represents the name of the data item to use on the right hand side of the condition expression. In the case of a data item condition expression, it may represent a workflow data object attribute (see “Workflow Data Objects” on page 16) or a constant value that the chosen operator will be applied to. For function condition expressions, this represents a workflow data object attribute that either of the two predefined functions will be used against to evaluate the condition.

### data-item-lhs

This metadata tag is optional as it is not required for a function condition expression. In the case of a data item condition expression, it represents the name of the data item to use on the left-hand side of the condition (i.e. a workflow data object attribute).

### operation

This metadata tag is optional as it is not required for a function condition expression. In the case of a data item condition expression, it represents an identifier for the logical operation that will be applied to either two workflow data object attributes or a workflow data object attribute and a constant value. The following is the list of valid operators that may be used in a data item condition expression:

Table 4. Condition Expression Operators

| Operator | Explanation              |
|----------|--------------------------|
| ==       | equal to                 |
| !=       | not equal to             |
| <=       | less than or equal to    |
| >=       | greater than or equal to |
| <        | less than                |
| >        | greater than             |

### conjunction

This represents an identifier for a logical conjunction that may be used in either a function or data item condition expression. There are two possible values for this attribute, and (the default) and or.

When a condition consists of multiple expressions, the logical conjunction is used in the evaluation of the complete condition.

#### **function**

This is optional as it is only used when specifying a function condition expression. As stated previously, there are two predefined functions, `Not()` and `isNothing()`. The `Not()` function acts as a logical inversion operator. In normal cases, this is applied to a boolean value. The `isNothing()` function may be applied to any workflow data object attribute type other than a boolean value. It may be used to test the scenarios where required data does not exist or has not been provided. The function returns a boolean value of `True` if the workflow data object attribute being examined does not contain any data.

#### **opening-brackets**

This is optional (the default is 0) as it may not be specified for either type of condition expression. It represents the number of opening brackets to insert at the start of the expression.

#### **closing-brackets**

This is optional (the default is 0) as it may not be specified for either type of condition expression. It represents the number of closing brackets to insert at the end of the expression.

The number of opening and closing brackets specified for an individual expression do not have to match (unless of course there is only one expression in the condition). The overall number of opening and closing brackets in the condition as a whole (with all of the expressions included) must be the same. Care should therefore be taken when specifying the number and position of opening and closing brackets within an individual expression, and the condition as a whole, as these brackets help determine how the condition and the individual expressions within that condition are evaluated. The same care should be taken when specifying the conjunction of an expression as failure to do so can lead to unexpected results.

## **Validations**

- The workflow data object attribute specified as the right hand side data item of the condition expression must be a valid workflow data object attribute in the context of the containing workflow process definition.
- The workflow data object attribute specified as the left hand side data item of the condition expression must be a valid workflow data object attribute in the context of the containing workflow process definition.
- The operator specified in a data item condition expression must be a valid and supported operator.
- The function specified in a function condition expression must be a valid and supported function.
- The conjunction specified in a condition expression must be valid and supported conjunction.
- The number of opening brackets and the number of closing brackets must be equal in the context of the overall condition.
- If the function `Not()` is specified for a function condition expression, then the type of the workflow data object attribute specified as the right hand side data item of the expression must be of type `BOOLEAN`.

- If the function `isNothing()` is specified for a function condition expression, then the type of the workflow data object attribute specified as the right hand side data item of the expression must not be of type `BOOLEAN`.
- If the right hand side data item of a data item condition expression is a workflow data object attribute, the type of this attribute must be compatible with the corresponding left hand side data item workflow data object attribute. Likewise, if the right hand side data item has been specified as a constant value, it must be compatible with the type of the corresponding left hand side data item workflow data object attribute.
- If either the right hand side or left hand side of a transition condition expression contains an indexed item from a list workflow data object (i.e. `ChildDependents[Context_Loop.loopCount].age`), then the associated workflow data object must be a list workflow data object and the activities involved in the transition must be contained within a loop.
- For a loop condition expression, if either the right hand side or left hand side of the expression specifies the `size()` attribute for a workflow data object, then that workflow data object must be a list workflow data object.
- For a loop condition expression, if either the right hand side or left hand side of the expression specifies the `size()` attribute for a workflow data object, then the item on the other side of the expression must be assignable to the type `INTEGER`.
- For a loop condition expression, if either the right hand side or left hand side of the expression specifies the `isEmpty()` attribute for a workflow data object, then that workflow data object must be a list workflow data object.
- For a loop condition expression, if either the right hand side or left hand side of the expression specifies the `isEmpty()` attribute for a workflow data object, then the item on the other side of the expression must be assignable to the type `BOOLEAN`.

---

## Split/Join

### Introduction

Transitions link activities in a process definition. In the most basic configuration of activities and transitions, each activity has only one incoming and one outgoing transition. However it is often useful to follow more than one path out of an activity resulting in a split (i.e. multiple transitions emanating from an activity). To support a valid block structure in a process definition (see “Workflow Structure” on page 109), each split must be matched by a join (i.e. multiple transitions meeting at one activity). In general a split allows multiple threads of work to be done at the same time while a join is the reciprocal synchronization point for those threads.

There are two reasons for an activity to have a split (and by extension some other activity down the line to have a join). The first is to allow work that does not have dependencies to be done in parallel while the second is to allow a choice to be made between a number of different paths in the workflow.

At the metadata level, each activity has a split and a join type. When the activity has only one outgoing or incoming transition a type of `none` is assigned to the split or join respectively. The other two split and join types, `choice` (also known as `XOR`) and `parallel` (also known as `AND`), are self explanatory and are the primary subject of this chapter.

# Choice XOR Split

## Metadata

```
<manual-activity id="1" category="AC1">
  ...
  <join type="and"/>
  <split type="xor">
    <transition-id idref="1"/>
    <transition-id idref="2"/>
    <transition-id idref="3"/>
    <transition-id idref="4"/>
  </split>
  <task>
    ...
  </task>
  <allocation-strategy type="target"
    identifier="HEARINGSCHEDULE"/>
  <event-wait>
    ...
  </event-wait>
</manual-activity>
```

**split** This is present for each activity and it contains the details of the split from the activity. This includes a list of the transitions from the specified activity that will be resolved by the workflow engine when the associated activity is completed to examine if they can be followed or not.

The order of the transitions in this list is important for a split type of XOR as it is the first transition that is eligible in the ordered list of transitions that will be followed by the workflow engine. In the metadata example above, if the transition conditions for transition identifiers 2, 3 and 4 are satisfied, it is the transition with the identifier of 2 that will be followed as this is the first eligible transition in the list of ordered transitions.

**type** This represents the type of the split. As described above, there are three possible split types. A split type of none indicates that there is only one outgoing transition from the specified activity. A split type of xor indicates a choice and this means that the first eligible transition from the list of ordered transitions will be followed. A split type of and indicates a parallel path of execution which ensures that all of the eligible transitions listed in the ordered list of transitions will be followed in parallel.

### **transition-id**

This contains a reference to the specified transition. There will be multiple entries of this metadata tag when the split type is xor or and.

**idref** This contains a reference to a transition in the workflow process definition.

# Parallel AND split

## Metadata

```

<manual-activity id="1" category="AC1">
  ...
  <join type="none"/>
  <split type="and">
    <transition-id idref="1"/>
    <transition-id idref="2"/>
    <transition-id idref="3"/>
    <transition-id idref="4"/>
  </split>
  <task>
  ...
  </task>
  <allocation-strategy type="target"
    identifier="HEARINGSCHEDULE"/>
  <event-wait>
  ...
  </event-wait>
</manual-activity>

```

The metadata for the split type of and is similar to the split type of xor (see “Choice XOR Split” on page 108). The difference is that the type of split is specified as and. This ensures that when the workflow engine is determining the list of transitions to follow from a specified activity, the order of the transitions in this list is not important as all eligible transitions in an and split will be followed. The ordered list of transitions is maintained in this instance for this split type to facilitate the changing of the split type from and to an xor, in which case the order of the transitions becomes important again.

---

## Workflow Structure

### Overview

The structure of a workflow process is determined by the activities in the process and the transitions between them. Hence a workflow forms a Graph in which the activities are vertices and the transitions are arcs (the graph formed by a workflow can be viewed using the *Visualize Workflow Process* feature in the Process Definition Tool).

In order for the workflow engine to successfully interpret and execute a process, the graph formed by that process must meet certain criteria. This chapter presents those criteria under two main headings: Graph Structure and Block Structure.

### Graph Structure

Since a the set of activities and transitions in a process form a Graph, Graph Theory can be applied to catch several well-known structural problems before a process is ever executed.

**Graph Theory:** Graph Theory is a branch of mathematics. Fortunately, those parts of graph theory that are relevant to workflow are very simple. Hence, this chapter does not require any prior knowledge of graph theory (a degree in mathematics is definitely not required!). There is a wealth of information about graph theory on the Internet, where further discussion on many of the topics discussed in this chapter can be easily found.

For example: consider a process in which an activity has a transition to another activity, which in turn has a transition back to the first activity. This forms a cycle in the process graph.

If there were no conditions on the transitions, the process would be guaranteed to end up in an infinite loop. These loops are known as informal loops (or 'ad-hoc' loops) and their presence renders several useful structural validations impossible. For this reason (among others), Cúram workflow provides formal constructs for delimiting iterative sections of a process (the loop-begin and loop-end activities). This allows it to detect the presence of ad-hoc loops in processes and prevents such processes from being released.

**Code Analogies:** Many developers will be familiar with the programming-language GOTO statement and the curly braces commonly used to delimit the start ({} and end ({} of a formal loop.

GOTO is analogous to ad-hoc loops in a workflow. The curly braces are analogous to the formal loop-begin and loop-end activities in a workflow.

## Block Structure

There are several workflow elements which can affect the choice of flow path (or paths) through a workflow at runtime. These include:

- "Choice XOR Split" on page 108
- "Parallel AND split" on page 108
- "Loop Begin and Loop End" on page 86

These elements always come in pairs. This is because they demarcate areas where the process should exhibit a specific behavior (one related to the flow of control). These areas are normally referred to as 'blocks', because they have a specific start-point that must have a corresponding end-point.

Consider a process with a structure where all paths emerging from a Choice Split (guaranteed to only follow one outbound path) all converge at a Parallel Join (which will wait until all inbound paths complete before executing the next activity). In this case, the process is guaranteed to stall at the Parallel Join. This is an example of a problem with the block-structure that can be caught by validations before a process is even executed.

### An Analogy for Blocks

A common analogy for how "blocks" work in a workflow is the way that brackets (like this!) work in a sentence. Brackets have an explicit start point '(', which is always matched by a specific end-point ')'. They demarcate an area of the sentence that has a specific meaning.

The way that brackets work in a mathematical expression is a closer analogy. In addition to matching opening and closing brackets, a mathematical expression can use several types of brackets. The bracketed expressions can be nested inside one another, but cannot be interleaved. This is very similar to how blocks work in a workflow.

### Block Types Supported by Workflow

The following sections describe the different types of blocks in Cúram workflow, how they begin/end and what their purpose is.

**'Choice' (XOR) Block:** A *Choice Block* is started at a Choice (XOR) Split and ended at a Choice (XOR) Join (the 'brackets'). It indicates that, of the possible paths within the block, no more than *one* can be followed.

The split has several transitions outbound from it, indicating the possible paths that a process instance could follow. Since this is a Choice block, the paths are mutually exclusive - only one will be followed by a given process instance.

The Choice Split must be matched by a corresponding Choice Join. This indicates the point at which the process ceases to be distinct for each path, so the paths are merged back together (i.e. the remaining process is common).

**'Parallel' (AND) Block:** A *Parallel Block* is started at a Parallel (AND) Split and ended at a Parallel (AND) Join (the 'brackets'). It indicates that, of the possible paths within the block, *many or all* can be followed.

The split has several transitions outbound from it, indicating the possible paths that a process instance could follow. Since this is a Parallel block, any number of the paths can be followed in parallel (assuming their transition conditions are met).

The Parallel Split must be matched by a corresponding Parallel Join. This indicates the point at which all the parallel paths must be synchronized before the workflow can continue.

**'Loop' Block:** A *Loop Block* is started at a loop-begin activity and ended at a loop-end activity (the 'brackets'). It indicates that the section of the workflow delimited by the loop-begin and loop-end activities should be repeated as long as the loop condition is met.

The loop-begin activity marks the point to which execution should return if the loop condition is met (i.e. the place to return to if the engine determines that the loop should iterate). The loop-end activity marks the point to which execution should jump if the loop condition is *not* met.

## Structural Rules

There are certain structural rules that workflow designers should be aware of when constructing process definitions. When a Cúram workflow process is validated, the validations assess whether the structure of the process conforms to these rules. Like all validations, the aim is to ensure that the process can be executed by the workflow engine.

### Graph Structure Rules

A Cúram process must form a graph that has the following properties: directed, connected and acyclic. This may sound complicated, but these are just the technical terms for some very simple graph properties.

- A "directed" graph is one in which each edge only goes one way (it is usually referred to as a digraph). In workflow terms, this means that a transition from activity A to activity B cannot be used to get from B back to A. This is given in Cúram workflow. It is mentioned here only because the 'acyclic' property (see below) is defined differently for graphs and digraphs.
- A "connected" graph is one in which every vertex can be reached. In workflow terms, this means that every Activity in the process must be reachable on at least one path from the start activity to the end activity.

This prevents workflows from having a structure such that one or more activities could never be executed.

- Finally, an "acyclic" digraph is one in which there are no *directed* cycles. In workflow terms, this means there can be no ad-hoc loops (i.e. loops constructed using transitions instead of loop-begin and loop-end activities).

Ad-hoc loops may seem convenient, but (like GOTO statements in programming languages) they can make a process very difficult to read and understand. Using explicit loop constructs leads to clearer, more understandable process definitions.

In addition, it allows the engine to know where looping can occur, so it can keep track of how many times a loop has iterated at runtime.

## Block Structure Rules

As mentioned earlier, the way that brackets work in a mathematical expression is a close analogy for how "blocks" work in a workflow. Recall - there are several types of blocks: Choice, Parallel and Loop. In Cúram workflow:

- Any block-starting constructs (Choice Split, Parallel Split or Loop- Begin Activity) must be terminated by a corresponding block-ending construct (Choice Join, Parallel Join or Loop- End Activity respectively).

In the case of Splits and Joins - all paths outbound from a split must converge at the corresponding Join.

**Rationale:** Requiring Splits and Joins (for example) to be matched improves readability. In a section containing multiple paths, it makes it clear whether a single path (or many) can be followed. This in turn makes it clear whether or not synchronization is required at the point where the paths merge.

If they were not required to match, it would be possible (easy!) to model processes that would be guaranteed to stall, or ones in which the end of the process could be reached before some activities had finished executing.

- Blocks can be nested within each other (e.g. a Choice Split inside a Loop), but they cannot be interleaved (e.g. None of the transitions from the choice split can go to an activity outside the loop).

This helps avoid situations that are difficult for the engine to process and are very unintuitive for workflow developers.

Consider a Loop that contains a Join, where the Join has two inbound transitions: one from an activity inside the loop, the other from an activity outside the loop.

It is very difficult in this situation to decide how the join synchronization should work. One inbound transition can only fire once, the other can fire multiple times. Any rules for handling such a situation in the would be arbitrary and hence unintuitive.

Workflows defined using Choice, Parallel and Loop blocks have a clear, simple structure whose meaning can be understood at a glance.

## Validations

A valid Cúram workflow must form a directed, connected, acyclic graph that is block-structured. For the most part these properties (directed, connected, acyclic) are discrete and so they can be checked independently by the Process Definition Tool (PDT) before releasing a process. The structural validations performed on a process definition are done in a distinct order and these are outlined below.

### Simple Syntactic Checks

The first set of structural validations carried out are *simple syntactic* checks. These ensure that the activity joins and splits (see "Split/Join" on page 107) in the process definition are set up correctly. These validations include:

- All activities except the *start* and *end* activities must have at least one inbound and one outbound transition.

- Any activity with more than one inbound transition *must* have a join type specified (i.e. a join type not equal to *NONE*).
- Any activity with more than one outbound transition *must* have a split type (i.e. a split type not equal to *NONE*).
- Any activity with *exactly* one inbound transition must have a join type of *NONE*.
- Any activity with *exactly* one outbound transition *must* have a split type *NONE*.
- The split type for a *Parallel* activity must be *NONE*.
- The join type for a *Parallel* activity must be *NONE*.
- A *Parallel* activity must have *exactly* one inbound transition.
- A *Parallel* activity must have *exactly* one outbound transition.
- The split type of the activity on the far side of the incoming transition to a *Parallel* activity *must* be *NONE*.
- The join type of the activity on the far side of the outgoing transition from a *Parallel* activity *must* be *NONE*.

### Graph Checks

The second set of structural validations carried out are *graph* checks. These ensure the flow graph is a directed, connected acyclic graph. These validations include:

- The workflow must form a 'connected' graph. This means that each activity *must* appear on at least one path from the *start* activity to the *end* activity.
- The workflow must form an acyclic digraph. This means that there can be *no* path through the workflow that hits the same activity twice. This validation checks for cycles created by transitions only - cycles created with loop-begin and loop-end activities are perfectly valid.
- Every instance subgraph within the workflow graph must correctly terminate. This means that starting at the *start* activity, every possible path through the workflow must end up at the *end* activity.

### Block Checks

The third set of structural validations carried out are *block* checks. These ensure ensuring that the flow graph is correctly block-structured.

The block-start constructs are: Start Process Activity, Loop Begin Activity, Parallel (AND) Split and Choice (XOR) Split. Their corresponding block-end constructs are: End Process Activity, Loop End Activity, Parallel (AND) Join and Choice (XOR) Join.

Based on these, the following block-structure validations are executed:

- For every block start, there must be a corresponding block end (i.e. if there is a Loop Begin activity in the workflow, then there must be a corresponding Loop End activity).
- The block start/end types must match (i.e. if there is a Parallel (AND) Split present in the workflow graph, then this must be matched by a corresponding Parallel (AND) Join).
- Blocks can be nested but not interleaved.

---

## Workflow Web Services

### Overview

Cúram workflows can inter-operate with other workflow systems through support for specific aspects of the Oasis group's *Business Process Execution Language* (BPEL) standard. BPEL processes can enact Cúram workflow processes and be notified when the process completes.

The Cúram workflow engine is not intended to be a fully fledged BPEL orchestration engine. Instead the Cúram workflows should be able to participate in BPEL orchestrated processes. This is done by providing functionality to expose Cúram workflow processes as web services that can be invoked from BPEL process partner links.

### Exposing a workflow web service

Workflow web services build on top of the existing Cúram web services support. In particular the workflow engine requires a Business Process Object (BPO) modeled as a Document Oriented Web Service (see the *Cúram Inbound Web Services* chapter of the *Cúram Modeling Reference Guide* for details).

The web service BPO is just a front end to the workflow enactment API (`curam.util.workflow.impl.EnactmentService`). This being the case only one such BPO is required per application. An appropriate BPO is already provided in the Cúram application: `Logical View::MetaModel::Curam::Facades::Workflow::WebService::WorkflowProcessEnactment`.

To use workflow web services the BPO named `Logical View::MetaModel::Curam::Facades::Workflow::WebService::WorkflowProcessEnactment` must be assigned a server component of stereotype `webservice`.

Cúram web services can be customized in other ways for example making them secure using WS-Security as described in the *Secure Web Services* chapter of the *Cúram Modelling Reference Guide*. All customizations for workflow web services must be made to this BPO.

**Note:** Since all workflow web services are handled by the same BPO any customizations will affect all process definitions that are exposed as web services.

### Process Enactment

Exposing a Cúram workflow process definition as a web service simply requires marking it as such in the Process Definition Tool (PDT) or directly in the metadata as described in "Process Definition Metadata" on page 12. Once the process definitions have been marked as web services the server, the server EAR and the web services EAR file must be rebuilt.

Like other Cúram web services the WSDL for the service can only be accessed once the web services EAR has been deployed. The name of workflow web service is the same as the process name. Thus the WSDL can be accessed at a URL similar to the following: `http://testserver:9082/CuramWS/services/<ProcessName>?wsdl`

The content of the WSDL is determined in part by the input to the process (the WDO attributes marked as required for enactment) and the process output (the WDO attributes marked as process output) (see "Metadata" on page 17). The

WSDL port type is the process name and the operation to enact a process is always startProcess.

```
<wsdl:portType name="SomeCuramWorkflow">
  <wsdl:operation name="startProcess">
    <wsdl:input message="intf:startProcessRequest"
      name="startProcessRequest"/>
    <wsdl:output message="intf:startProcessResponse"
      name="startProcessResponse"/>
    <wsdl:fault message="intf:InformationalException"
      name="InformationalException"/>
    <wsdl:fault message="intf:AppException"
      name="AppException"/>
  </wsdl:operation>
</wsdl:portType>
```

Figure 2. Process Enactment Port Type

### Process completion callback

An external system (probably but not necessarily a BPEL process) that enacts a Cúram workflow via web services will often require notification that the process completed and possibly some output data from the process definition. Doing this requires a web service that will be invoked when the process completes to be specified for each process definition.

The callback web service is specified in the process definition metadata using the PDT or directly in the metadata as described in “Process Definition Metadata” on page 12.

**Note:** Before use in a workflow process definition the callback web service must be registered as a Cúram outbound web service connector as described in the *Cúram Outbound Web Service Connectors* chapter of the *Cúram Modeling Reference Guide*.

The callback web service must be implemented by an external system but conform to a port type definition specified by the Cúram workflow web service, “Invocation from BPEL processes” has further details.

## Invocation from BPEL processes

The creation of BPEL processes that enact Cúram workflow processes is out of the scope of this document. However the WSDL for each workflow process web service contains information that can be used by BPEL processes.

### Callback Port Type

There is a port type in WSDL for a Cúram workflow web service that is not implemented by the service itself. The name of this port type is the name of the process with the word "Complete" appended to it (<ProcessName>Complete).

The purpose of this unimplemented port type is to define the web service interface that a Cúram workflow web service expects to be implemented by the BPEL process that enacted it. This port type that must be implemented by the callback web service configured in the process definition (see “Process completion callback”).

```

<!--Implemented by the BPEL process-->
<wsdl:portType name="SomeCuramWorkflowComplete">
  <wsdl:operation name="processCompleted">
    <wsdl:input message="intf:processCompletedRequest"
      name="processCompletedRequest"/>
  </wsdl:operation>
</wsdl:portType>

```

Figure 3. Callback Port Type

### Partner Link Type

Technically the only thing necessary to allow a Cúram workflow process to participate in a BPEL orchestrated process is to expose the process as a web service. However it is possible to add some metadata to assist the BPEL process developer by defining the port types involved in the partner link and the roles they play.

The BPEL specification allows partner link types to be defined in the WSDL for the service to be invoked in the partner link using the WSDL extension mechanism. The WSDL generated for a Cúram workflow web service defines the partner link type it expects to participate in and specifies the port types that play each role.

```

<!--Partner link type-->
<partnerLinkType name="CuramWorkflowPartnerLink"
  xmlns="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
  <role name="curamService">
    <portType name="tns1:SomeCuramWorkflow"/>
  </role>
  <role name="partnerService">
    <portType name="tns1:SomeCuramWorkflowComplete"/>
  </role>
</partnerLinkType>

```

Figure 4. WSDL extensions for BPEL

---

## File Locations

### Overview

While there are utilities like the Process Definition Tool PDT and other administration user interfaces, the outputs of such tools often need to be exported and version controlled. Of course these externalized files need to be put back into the runtime system when building or installing Cúram. The pattern in Cúram is to place such files into a predefined source folder from which they are loaded onto the database (perhaps after some pre-processing). This chapter describes the location of workflow related source files.

### Workflow Process Definition Files

Workflow process definitions (both released and unreleased) can be imported onto the relevant database table using the standard **build database** target.

These workflow process definitions must be stored in XML files in a workflow subdirectory under the relevant Cúram server component directory (e.g. `...\EJBServer\components\core\workflow` for the core component or `...\EJBServer\components\Appeal\workflow` for the Appeal component etc.).

Each component in the Cúram application can have a workflow directory containing the process definition XML files relevant to it. Any process definition files stored in these workflow directories will automatically be imported when the

**build database** target is executed. If the process definition files are not valid or if the name and version of the definitions do not match those used in the filenames, the import will fail.

The workflow process definition XML files on the file system must follow a strict naming convention. This is as follows: `Process Name_vProcess Version.xml` where:

- Process Name is the name of the workflow process.
- Process Version is the version of the workflow process.

The same version of a process definition can exist in multiple components in the Cúram application. The version imported will always be taken from the component with the highest component order precedence. Component order precedence is configured using the `COMPONENT_ORDER_PRECEDENCE` environment variable.

Each process definition when imported will be assigned a new process definition identifier that is unique for the database it is imported onto. Different versions of the same process definition will be assigned the same unique identifier and only one unreleased version of a process definition can be imported. To handle invalid workflow process definitions loaded during the build database target, strict validations are in place in the workflow engine. These ensure that a workflow process definition cannot be loaded into the process definition cache and executed unless it passes all of the process validations first. These validations are described in the earlier chapters of this document.

## Customizing Workflow Process Definition Files

**Creating New Workflow Process Definition Files:** All new workflow process definition files must be created in the workflow subdirectory of the `...\EJBServer\components\custom` directory. To create a new process definition file, the PDT may be used to create the required definition and enter all the details. The definition may then be exported to a file by the tool and placed in the location specified above.

**Changing An Existing Workflow Process Definition File:** Using the PDT, view the latest version of the process definition that requires modification. Create a new version of that process definition using the tool. Make the changes, validate it and release the workflow.

Export the newly released workflow process definition using the PDT and place it into the workflow subdirectory of the `...\EJBServer\components\custom` directory.

## Event Definition Files

Events provide a mechanism for loosely-coupled parts of the Cúram application to communicate information about state changes in the system. When one module in the application raises an event, one or more other modules receive notification of that event having occurred provided they are registered as listeners for that event. To make use of this functionality, some events have to be defined, some application code must raise these events, and some event handlers have to be defined and registered as listeners to such events.

Events are defined in Cúram in XML files, that specify both the event classes and the event types. These files are created with a `.evx` extension and are placed in the events of a Curam component (e.g. `...\EJBServer\components\core\events`) from where they are picked up and processed by the build scripts.

There are two types of output generated by the **evgen** command; .java files (for code constants that make the use of events less error prone) and .dmx files (Cúram database scripts for loading event definitions onto the database). The Java artifacts produced from a merged event files are placed in the /build/svr/events/gen/[package] directory, where [package] is the package attribute specified in the event definition file. The database scripts produced from a merged event files are placed in the /build/svr/events/gen/dmx directory.

Chapter 10 of the *Cúram Server Developer's Guide* provides a comprehensive description of events and how they may be used in the Cúram application.

## Configuration

### Overview

For the most part, configuration options are not global across all workflow process definitions. Rather they are specific to each definition and hence are held within the actual process definition itself. That said, there are a small number of application properties that affect the Cúram Workflow Management System as a whole. This chapter describes those properties.

### Application Properties

The following application properties can be set in the Application.prx file:

| Property Name  | Description  |
|--|--|
| curam.custom.workflow.workresolver                           | <p><i>Purpose:</i> The fully-qualified name of the application class that implements the <code>WorkResolver</code> callback interface. See "Allocation strategy" on page 62 for further information.</p> <p><i>Type:</i> String</p> <p><i>Default Value:</i> <code>curam.core.sl.impl.DefaultWorkResolver</code></p>   |
| curam.workflow.automaticallyaddtaskoustasks                  | <p><i>Purpose:</i> After the resolution of the allocation targets for a task, if that task is assigned to one user and one user only and the value of this property is set to <code>yes/true</code>, the system will automatically add this task to a user's My Tasks list in their Inbox to allow them to work on it.</p> <p><i>Type:</i> String</p> <p><i>Default Value:</i> <code>NO</code></p>   |
| curam.custom.notifications.notificationdelivery              | <p><i>Purpose:</i> The fully-qualified name of the application class that implements the <code>NotificationDelivery</code> callback interface. See "Notification Allocation Strategy" on page 97 for further information.</p> <p><i>Type:</i> String</p> <p><i>Default Value:</i> <code>curam.core.sl.impl.NotificationDeliveryStrategy</code></p>   |
| curam.workflow.disable.audit.wdovaluehistory.before.activity | <p><i>Purpose:</i> The process instance WDO data auditing table, 'WDOValuesHistory' is populated by the workflow engine at three distinct points during the execution of a workflow process instance (before the execution of an activity, after the execution of an activity and before the evaluation of the transitions from an activity). When specified to <code>true</code>, this property will ensure that no data is written to the WDO data auditing table before an activity is executed.</p> <p><i>Type:</i> <code>BOOLEAN</code></p> <p><i>Default Value:</i> <code>FALSE</code></p> |

| Property Name   | Description  |
|---|--|
| curam.workflow.disable.audit.wdovalueshistory.after.activity        | <p><i>Purpose:</i> The process instance WDO data auditing table, 'WDOValuesHistory' is populated by the workflow engine at three distinct points during the execution of a workflow process instance (before the execution of an activity, after the execution of an activity and before the evaluation of the transitions from an activity). When specified to true, this property will ensure that no data is written to the WDO data auditing table after an activity has been executed.</p> <p><i>Type:</i> BOOLEAN</p> <p><i>Default Value:</i> FALSE</p>   |
| curam.workflow.disable.audit.wdovalueshistory.transition.evaluation | <p><i>Purpose:</i> The process instance WDO data auditing table, 'WDOValuesHistory' is populated by the workflow engine at three distinct points during the execution of a workflow process instance (before the execution of an activity, after the execution of an activity and before the evaluation of the transitions from an activity). When specified to true, this property will ensure that no data is written to the WDO data auditing table before the transitions from an activity are evaluated..</p> <p><i>Type:</i> BOOLEAN</p> <p><i>Default Value:</i> FALSE</p>                                |
| curam.custom.workflow.processcachesize                              | <p><i>Purpose:</i> The workflow engine caches released versions of process definitions in memory (to minimize overheads when looking up metadata). This property controls the maximum number of process versions stored in the cache. When this number has been reached, the engine will begin ejecting process versions from the cache, using a least-recently-used ejection policy. Runtime modifications to the value of this property will take affect the next time the workflow engine attempts to insert a process version in the cache.</p> <p><i>Type:</i> Integer</p> <p><i>Default Value:</i> 250</p> |
| curam.batchlauncher.dbtojms.notification.batchlaunchermode          | See <i>Cúram Batch Processing Guide, Section 5.3</i> for further information.  |
| curam.batchlauncher.dbtojms.notification.encoding                   | See <i>Cúram Batch Processing Guide, Section 5.3</i> for further information.  |
| curam.batchlauncher.dbtojms.notification.host                       | See <i>Cúram Batch Processing Guide, Section 5.3</i> for further information.  |
| curam.batchlauncher.dbtojms.messagespertransaction                  | See <i>Cúram Batch Processing Guide, Section 5.3</i> for further information.  |
| curam.batchlauncher.dbtojms.notification.port                       | See <i>Cúram Batch Processing Guide, Section 5.3</i> for further information.  |

---

## JMSLite

JMSLite is a Cúram-developed lightweight Java Message Service (JMS) server that runs alongside the RMI-based test environment. Hence it can run inside supported Integrated Development Environments (IDEs).

This allows process definitions to be tested inside an Integrated Development Environment, that is, without requiring the application to be deployed to an EJB server. When used along with the Process Definition Tool, JMSLite allows developers to define, deploy, and enact workflows - all within their Integrated Development Environment.

### What JMSLite Does

JMSLite is a JMS server that implements only those sections of the JMS specification necessary to support Integrated Development Environment based testing of Cúram workflows: namely transactional, point-to-point messaging. This

means that JMSLite supports ACID transactions that involve the application database *and* the infrastructure-defined workflow queue destinations.

It does not support custom (application-defined) queues or the publish-subscribe domain (that is, topics).

So, JMSLite allows the workflow enactment service and workflow engine to send JMS messages asynchronously. This means that application calls to workflow-related infrastructure APIs (such as the enactment service and event service) are non-blocking. The APIs pass messages to the workflow engine, which drives process instances asynchronously (for example, runs automatic activities, creates and allocates Tasks, and so forth).

## Why JMSLite?

The purpose of JMSLite is to make the workflow engine behave in an Integrated Development Environment in the closest possible way to how it behaves when deployed on an application server. This increases the likelihood of detecting problems early (while testing in the Integrated Development Environment) rather than late (when testing on an application server). Both risk and cost are consequently reduced.

For example, consider the following situation: Suppose the WMS (running in an Integrated Development Environment) were to enact workflows *synchronously*.

**Reminder:** In production, workflows are enacted *asynchronously* because they are assumed to be long-lived (on the order of hours, days or weeks) relative to normal user operations (order of seconds or milliseconds).

Suppose also that a developer were to write a method that enacted an automated case-approval workflow and then (immediately after the call to the enactment service) tried to do something with the result (for example, check if the case was automatically approved). Since the test environment operates in a different manner (synchronously) from the production environment - the code would work fine in test, but would fail in production (this is an example of a 'temporal coupling' bug).

However, since JMSLite runs asynchronously - this problem would show up in the Integrated Development Environment in the same way as it would on an application server, consequently allowing the developer to detect it early.

## Using JMSLite

The JMSLite server polls queues and unpacks any messages that it finds on them. These messages result in calls from the JMSLite server to the RMI server that is required for Integrated Development Environment -based testing of Cúram methods (commonly referred to as StartServer). The JMSLite server is launched as a thread when the (StartServer) process is started.

Since the JMSLite server dispatches messages to the workflow engine that runs on the RMI server, it is necessary to start the StartServer in debug mode when workflow methods are debugged.

## Debugging workflows

Normally, Cúram infrastructure methods are started by the application. However, in workflow the call is often made the other way around, that is, the workflow engine (infrastructure) calls an application method (for example, a Work Allocation method).

In these cases, it is not possible for an application developer to step from the call to the `curam.util.workflow.impl.EnactmentService.startProcess()` method into their application (Work Allocation) method. In this case, the developer must set breakpoints within the method they want to debug and then run the method that enacts the workflow. The workflow engine will then (asynchronously) start the application method, therefore causing the breakpoint to be reached. The debugger then suspends execution at the specified breakpoint, so allowing normal debugging.

Application methods that fall into the above category are:

- Automatic Activity methods
- Work Allocation Functions
- The application Notification Delivery Method
- The application Work Resolver Method

---

## Inbox and Task Management

### Overview

Tasks are used to assign and track the work of system users and are generated when “Manual” on page 55, “Decision” on page 73 or “Parallel” on page 88 activities are executed by the Workflow Engine. The Inbox and the associated task management functions are used by the users of the Cúram application to manage these tasks. The following sections describe the configuration and customization options that are available for the Inbox and Task Management areas of the Cúram WMS.

### Inbox Configuration

#### Inbox List Sizes Configuration Settings

There are a number of task list views available in the Inbox. These include the following:

- *My Open Tasks* : A list of tasks that the user is currently working on.
- *My Deferred Tasks* : A list of tasks that the user is working on but has deferred to a later date.
- *Available Tasks* : A list of tasks that are available to the user to work on.
- *Task Query Search Results* : A list of tasks that are the result of running a task query.
- *Work Queue Tasks* : A list of tasks that are assigned to a work queue.

There is also a list in the Inbox that displays the notifications that have been delivered to a user.

- *My Notifications* : A list of notifications that have been delivered to the user.

The Inbox list views can be configured to limit the number of records returned to the user. The following application properties can be set in the `Application.prx` file to effect this change.

Table 5. Inbox List Sizes Configuration Settings

| Property Name                    | Description  |
|----------------------------------|--|
| curam.inbox.max.task.list.size   | <p><i>Purpose:</i> The value of the property controls the number of tasks displayed in the various Inbox task list views. The Inbox task lists pages affected by the value of this property include the following: My Open Tasks; My Deferred Tasks; Available Tasks; Task Query Search; Work Queue Tasks. If the number of tasks to be displayed exceeds the specified value then a message is displayed informing the user that not all the records that match the search criteria of the page are being displayed. This message displays both the number of tasks being displayed and also the total number of tasks that match the search criteria.</p> <p><i>Type:</i> Integer</p> <p><i>Default Value:</i> 100</p> |
| curam.notification.max.list.size | <p><i>Purpose:</i> The value of the property controls the number of notifications displayed in the Inbox My Notifications list view. If the number of notifications to be displayed exceeds the specified value then a message is displayed informing the user that not all the records that match the search criteria of the page are being displayed. This message displays both the number of notifications being displayed and also the total number of notifications that match the search criteria.</p> <p><i>Type:</i> Integer</p> <p><i>Default Value:</i> 100</p>   |

### Get Next Task Configuration Settings

There are a number of shortcut functions available in the Inbox to retrieve the next task to work on. These functions include the following:

- Get Next Task - retrieves the next task from the tasks available to the user.
- Get Next Task From Preferred Org Unit - retrieves the next task assigned to the user's preferred organization unit.
- Get Next Task From Preferred Queue- retrieves the next task assigned to the user's preferred work queue.
- Get Next Task From Queue- retrieves the next task assigned to a work queue that the user selects.

The algorithm used by these shortcut functions to retrieve the next task may be configured by using the following application properties in the Application.prx file:

Table 6. Get Next Task Configuration Settings

| Property Name                                    | Description  |
|--|--|
| curam.workflow.reservenexttaskwithpriorityfilter | <p><i>Purpose:</i> The value of the property controls whether the get next task algorithm uses the priority of a task to determine the next task to retrieve. If set to <i>YES</i>, the default, the priority of the task is used for this purpose (the priorities as specified in the curam.workflow.taskpriorityorder) property. Otherwise, the task to be retrieved is based on tasks that have been assigned to the user for the longest period of time.</p> <p><i>Type:</i> String</p> <p><i>Default Value:</i> Yes</p>   |
| curam.workflow.taskpriorityorder                 | <p><i>Purpose:</i> There are three task priorities specified in the Workflow Management System, namely High, Medium and Low (which correspond to the codetable codes TP1, TP2 and TP3 in the TaskPriority codetable). In some cases, customers may have a requirement to add a new task priority (e.g. Critical with a codetable code value of TP4). Retrieving tasks using the task priority containing this value would therefore ensure that critical tasks would appear after those that have a low priority (when the intention would be that tasks with this priority should be retrieved first). This property allows the task priorities to be specified in whatever order is required to satisfy the customer's requirements.</p> <p><i>Type:</i> String</p> <p><i>Default Value:</i> TP1,TP2,TP3</p> |

## Task Redirection and Allocation Blocking Settings

Task redirection enables the user to redirect tasks to another user, organizational object (organization unit, position or job) or work queue for a specified period of time. Task allocation blocking enables the user to ensure that no tasks are assigned to them for a specified period of time. This functionality is available to the user in the Task Preferences area of the Inbox. However, all users on the system may not require access to set up task redirection or task allocation blocking periods for themselves. To facilitate this requirement, these areas of functionality in the Inbox may be disabled for specific users through the use of security identifiers. The following table details the security identifiers that a user must have in order to avail of this functionality.

Table 7. Security Identifiers and Associated Actions

| Security Identifier Name   | Action Allowed  |
|--|---|
| UserTaskRedirection.listTaskRedirectionHistoryForUser                | Allows a user to view all of the task redirection periods specified for them.         |
| UserTaskRedirection.redirectTasksForUser                             | Allows a user to create a task redirection period for themselves.                     |
| UserTaskRedirection.clearTaskRedirectionForUser                      | Allows a user to clear one of their task redirection periods.                         |
| UserTaskAllocationBlocking.list.TaskAllocationBlockingHistoryForUser | Allows a user to view all of the task allocation blocking periods specified for them. |
| UserTaskAllocationBlocking.blockTaskAllocationForUser                | Allows a user to create a task allocation blocking period for themselves.             |
| UserTaskAllocationBlocking.clearTaskAllocationBlockForUser           | Allows a user to clear one of their task allocation blocking periods.                 |

## Inbox Customization

The default behavior of the Inbox Actions, Task Actions and Task Search functionalities can be changed by using Guice to call custom code which overrides the default behavior.

**Note:** Guice is a framework developed by *Google* and is beyond the scope of this document. For more information on Guice please refer to the Guice user's guide.

The Cúram Workflow Management System contains the following customization points and their corresponding default implementations:

*Table 8. Customization Points*

| Customization Point                   | Interface Class                             | Default Implementation Class                    |
|---------------------------------------|---|---|
| Inbox Actions                         | curam.core.hook.<br>task.impl.InboxActions  | curam.core.hook.<br>task.impl.InboxActionsImpl  |
| Task Actions                          | curam.core.hook.<br>task.impl.TaskActions   | curam.core.hook.<br>task.impl.TaskActionsImpl   |
| Task Search and Available Task Search | curam.core.hook.<br>task.impl.SearchTask    | curam.core.hook.<br>task.impl.SearchTaskImpl    |
| Task Query                            | curam.core.hook.<br>task.impl.TaskQuery     | curam.core.hook.<br>task.impl.TaskQueryImpl     |
| Task Search SQL generation            | curam.core.hook.<br>task.impl.SearchTaskSQL | curam.core.hook.<br>task.impl.SearchTaskSQLImpl |

The following *Inbox Actions* may be customized:

- Get Next Task
- Get Next Task From Preferred Organization Unit
- Get Next Task From Preferred Queue
- Get Next Task From Work Queue
- Subscribe User To Work Queue
- Unsubscribe User From Work Queue

The following *Task Actions* may be customized:

- Add Comment
- Close
- Create
- Defer
- Restart
- Forward
- Modify Time Worked
- Modify Priority
- Modify Deadline
- Reallocate
- Add To My Tasks

The following *Task Search* and *Available Task Search* methods may be customized:

- countAvailableTasks
- countTasks

- searchAvailableTasks
- searchTask
- validateSearchTask

The following *Task Query* methods may be customized:

- createTaskQuery
- modifyTaskQuery
- runTaskQuery
- validateTaskQuery

The following Task Search SQL generation methods may be customized. These methods are used to generate the SQL for all of the above task search functionalities.

- getBusinessObjectTypeSQL
- getCategorySQL
- getCountSQLStatement
- getCreationDateSQL
- getDeadlineSQL
- getFromClause
- getOrderBySQL
- getOrgObjectSQL
- getPrioritySQL
- getReservedBySQL
- getRestartDateSQL
- getSelectClause
- getSQLStatement
- getStatusSQL
- getTaskIDSQL
- getWhereClause

## How to customize the Inbox

The following is a description of how to customize the Inbox action `curam.core.hook.task.impl.InboxActionsImpl.getNextTask`. The same process may be followed to customize any of the other customization points.

A custom hook point class must be created. This class *must* extend the default implementation class:

### CustomTaskActionsImpl class

The `CustomTaskActionsImpl` class implements the `getNextTask` method and it implements the `TaskActionsImpl` class.

### TaskActionsImpl class

The `TaskActionsImpl` class implements the following methods:

- getNextTask
- getNextTaskFromWorkQueue
- getNextTaskFromPreferredWorkQueue
- getNextTaskFromPreferredOrgUnit
- subscribeUserToWorkQueue
- unsubscribeUserFromWorkQueue

The TaskActionsImpl class implements the TaskActions interface class:

#### TaskActions interface class

The TaskActions interface class implements the following methods:

- getNextTask
- getNextTaskFromWorkQueue
- getNextTaskFromPreferredWorkQueue
- getNextTaskFromPreferredOrgUnit
- subscribeUserToWorkQueue
- unsubscribeUserFromWorkQueue

**Note:** The custom class must *never* directly implement the interface class, as this could lead to compile time exceptions during an upgrade if new methods were added to the interface. In this case the custom class would not implement the new methods and hence the contract between the interface class and the implementation class would be broken leading to compile time exceptions.

**Customizing the default implementation:** The signature of the getNextTask function on the curam.core.hook.task.impl.InboxActions interface is as follows:

```
package curam.core.hook.task.impl;

@ImplementedBy(InboxActionsImpl.class)
public interface InboxActions {

    public long getNextTask(String userName);

    .
    .
    .
}
```

The default implementation for the function is specified in the curam.core.hook.task.impl.InboxActionsImpl class

```
package curam.core.hook.task.impl;

public class InboxActionsImpl implements InboxActions {

    public long getNextTask(String userName) {
        // Default implementation code is here....
    }

    .
    .
    .
}
```

To customize getNextTask, the method must be implemented in the new custom class created earlier which extends the default curam.core.hook.task.impl.InboxActionsImpl implementation class.

```
package custom.hook.task.impl;

public class CustomInboxActionsImpl extends InboxActionsImpl {

    public long getNextTask(final String userName) {
```

```
    // Custom implementation code goes here
  }
}
```

To ensure that the application executes the new custom class rather than the default implementation a new class `custom.hook.task.impl.Module.java` which extends `com.google.inject.AbstractModule` must be written with the `configure` method implemented as the following example shows:

```
package custom.hook.task.impl;

public class Module extends com.google.inject.AbstractModule {
  protected void configure() {
    bind(
      curam.core.hook.task.impl.InboxActions.class).to(
      custom.hook.task.impl.CustomInboxActionsImpl.class);
  }
}
```

Finally the `custom.hook.task.impl.Module` class name must be inserted into the `ModuleClassName` column of the `ModuleClassName` database table. This can be inserted by adding an extra row to the `ModuleClassName.DMX` file or directly into the database table if required.

Using this approach, when the application is redeployed, the system will now invoke the customized version of the `getNextTask` function rather than the default implementation.



---

## Notices

This information was developed for products and services offered in the United States.

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, MD-NC119 Armonk, NY 10504-17*

For license inquiries regarding double-byte character set (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. 19-21, Nihombashi*

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 jurisdictions 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 typographical 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide 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:

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.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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.

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 actual people or business enterprises 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.

---

## Privacy Policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings

can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies or other similar technologies that collect each user's name, user name, password, and/or other personally identifiable information for purposes of session management, authentication, enhanced user usability, single sign-on configuration and/or other usage tracking and/or functional purposes. These cookies or other similar technologies cannot be disabled.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at <http://www.ibm.com/privacy> and IBM's Online Privacy Statement at <http://www.ibm.com/privacy/details> the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at <http://www.ibm.com/software/info/product-privacy>.

---

## Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at " Copyright and trademark information " at <http://www.ibm.com/legal/copytrade.shtml>.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other names may be trademarks of their respective owners. Other company, product, and service names may be trademarks or service marks of others.







Printed in USA