CUMBIA EXTENSIBLE PROCESS MODELING: UN MODELO EXTENSIBLE PARA LA CONSTRUCCIÓN DE MOTORES DE WORKFLOW Y SU VALIDACIÓN MEDIANTE PATRONES DE CONTROL DE FLUJO

JAIME DE JESÚS SOLANO ESCOBAR

UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN
BOGOTÁ, D.C.
2005

CUMBIA EXTENSIBLE PROCESS MODELING: UN MODELO EXTENSIBLE PARA LA CONSTRUCCIÓN DE MOTORES DE WORKFLOW Y SU VALIDACIÓN MEDIANTE PATRONES DE CONTROL DE FLUJO

JAIME DE JESÚS SOLANO ESCOBAR

Trabajo de Grado presentado como requisito para optar al título de Magíster en Ingeniería de Sistemas

Director: Jorge Villalobos Salcedo
Profesor Asociado

UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN
BOGOTÁ, D.C.
2005

CONTENIDO

1. INTRODUCCIÓN 8

2. OBJETIVOS 9
2.1 GENERALES 9
2.2 ESPECÍFICOS 9

3. MODELOS EXISTENTES 10
3.1 XPDL 10
3.2 BPEL4WS - BUSINESS PROCESS EXECUTION LANGUAGE FOR WEB SERVICES 12
3.3 JBPM – JAVA BUSINESS PROCESS MANAGEMENT 13
3.4 APEL – ABSTRACT PROCESS ENGINE LANGUAGE 16
3.5 COMPARACIÓN ENTRE LOS MODELOS 17
3.5.1 Facilidad de Expresión 17
3.5.2 Sintaxis Gráfica 18
3.5.3 Herramientas de Soporte 18
3.5.4 Interacción con otros Modelos de Workflows 19
3.5.5 Extensibilidad, Introspección y Reflexión 20
3.5.6 Dependencias Tecnológicas 20
3.5.7 Capacidad de control 21
3.5.8 Granularidad 21
3.5.9 Modelo de Flujo de Control 22
3.5.10 Modelo Flujo de Información 22

4. PATRONES DE CONTROL DE FLUJO 24
4.1 PATRONES DE CONTROL BÁSICO 24
4.2 PATRONES DE RAMIFICACIÓN Y SINCRONIZACIÓN AVANZADA 26
4.3 PATRONES ESTRUCTURALES 28
4.4 PATRONES DE INSTANCIAS MÚLTIPLES 29
4.5 PATRONES BASADOS EN ESTADO 30
4.6 PATRONES DE CANCELACIÓN 32
4.7 ANÁLISIS: APEL Y PATRONES DE CONTROL DE FLUJO 33

5. MOTIVACIÓN 41

6. EL MODELO CUMBIA-XPM 43
6.1 ELEMENTOS DEL MODELO ESTÁTICO 43
6.1.1 Variable 44
6.1.2 Actividad 44
6.1.3 Recurso 44
6.1.4 Workspace 44
6.1.5 Puerto 44
6.1.6 Map 45
6.1.7 Dataflow 45
6.1.8 MultiActividad 46
6.1.9 Proceso 46
6.1.10 Autómata 46

6.1.11 Estado ... 47
6.1.12 Transición ... 47
6.1.13 Evento ... 47
6.1.14 ColaEventos (*eventQueue*) ... 47
6.1.15 Acción ... 47
6.2 ELEMENTOS DEL MODELO DINÁMICO ... 47
6.2.1 Valor (*Value*) ... 47
6.2.2 Dato (*Data*) ... 48
6.2.3 Recurso-i ... 48
6.2.4 Memoria (*Memory*) ... 48
6.2.5 Puerto-i ... 48
6.2.6 Dataflow-i ... 49
6.2.7 Workspace-i ... 50
6.2.8 Actividad-i ... 51
6.2.9 MultiActividad-i ... 52
6.2.10 Proceso-i ... 54
6.3 SEMÁNTICA DE LOS ELEMENTOS ... 55
6.3.1 Autómatas de los Elementos Básicos de C-XPM ... 55
6.4 AUTÓMATAS SINCRONIZADOS ... 59
6.4.1 Representación Gráfica ... 61
6.7 MECANISMOS DE EXTENSIBILIDAD ... 65
6.7.1 Reflexión ... 65
6.7.2 Extensión Simple ... 66
6.7.3 Adaptación ... 68
6.7.4 Especialización ... 69

7. VALIDACIÓN DE CUMBIA-XPM BASADA EN LOS PATRONES DE CONTROL DE FLUJO ... 70
7.1 PATRONES DE CONTROL BÁSICO ... 70
7.2 PATRONES DE RAMIFICACIÓN Y SINCRONIZACIÓN AVANZADA ... 72
7.3 PATRONES ESTRUCTURALES ... 77
7.4 PATRONES DE INSTANCIAS MÚLTIPLES ... 78
7.5 PATRONES BASADOS EN ESTADO ... 79
7.6 INTERPRETACIÓN DE LOS RESULTADOS OBTENIDOS ... 83

8. CONCLUSIONES ... 85

9. TRABAJO FUTURO ... 86

REFERENCIAS ... 88

LISTA DE FIGURAS

Diagrama del Proceso de subasta 14
Diagrama del Patrón 'Sequence' 24
Diagrama del Patrón 'Parallel Split' 24
Diagrama del Patrón 'Synchronization' 25
Diagrama del Patrón 'Exclusive Choice' 25
Diagrama del Patrón 'Simple Merge' 26
Diagrama del Patrón 'Multi-Choice' 26
Diagrama del Patrón 'Synchronizing Merge' 27
Diagrama del Patrón 'Multi-Merge' 27
Diagrama del Patrón 'Discriminator' 28
Diagrama del Patrón 'Arbitrary Cycles' 28
Diagrama del Patrón 'Implicit Termination' 29
Diagrama del Patrón 'MI without Synchronization' 29
Diagrama del Patrón 13 30
Diagrama del Patrón 14 30
Diagrama del Patrón 15 30
Diagrama del Patrón 'Deferred Choice' 31
Diagrama del Patrón 'Interleaved Parallel Routing' 31
Diagrama del Patrón 'Milestone' 32
Diagrama del Patrón 'Cancel Activity' 32
Diagrama del Patrón 'Cancel Case' 32
Diagrama del Patrón 'Sequence' en APEL 33
Diagrama del Patrón 'Parallel Split' en APEL 33
Diagrama del Patrón 'Synchronization' en APEL 34
Diagrama del Patrón 'Exclusive Choice' en APEL 34
Diagrama del Patrón 'Simple Merge' en APEL 35
Diagrama del Patrón 'Multiple Choice' en APEL 35
Diagrama del Patrón 'Synchronizing Merge' en APEL 36
Diagrama del Patrón 'Multiple Merge' en APEL 36
Diagrama del Patrón 'Arbitrary Cycles' en APEL 37
Diagrama del Patrón 13 en APEL. Las actividades X1 y X2 son iguales (instancias). 38
Diagrama del Patrón 'Deferred Choice' en APEL, usando puertos asíncronos 38
Diagrama del Patrón 'Milestone' en APEL 39
Representación gráfica de una Actividad 44
Representación gráfica de un Puerto. 45
Representación gráfica de un Dataflow 45
Representación gráfica de una MultiActividad 46
Representación gráfica de un Proceso 46
Autómata Básico de un Puerto 56
Autómata Básico de un Dataflow 56
Autómata Básico de una Actividad 57
Autómata Básico de un Workspace 58
Autómata Básico de una MultiActividad 58
Autómata Básico de un Proceso 59
Representación Gráfica de Sincronización de Autómatas 61

Autómata Sincronizado de un Puerto 61
Autómata Sincronizado de un Dataflow 62
Autómata Sincronizado de una Actividad 63
Autómata Sincronizado de un Workspace 64
Autómata Sincronizado de una MultiActividad 64
Autómata Sincronizado de un Proceso 65
Ejemplo de Reflexión: Vista en tiempo de diseño 66
Ejemplo de Reflexión: Vista en tiempo de ejecución 66
Ejemplo de Extensión: Acción 'cargarDocumento' 67
Ejemplo de Adaptación: Autómata del nuevo tipo de Actividad 68
Diagrama del Patrón 'Sequence' en C-XPM 70
Diagrama del Patrón 'Parallel Split' en C-XPM 70
Diagrama del Patrón 'Synchronization' en C-XPM 71
Diagrama del Patrón 'Exclusive Choice' en C-XPM 71
Diagrama del Patrón 'Simple Merge' en C-XPM 72
Diagrama del Patrón 'Multiple Choice' en C-XPM 72
Diagrama del Patrón 'Synchronizing Merge' en C-XPM 73
Modificación al Autómata de una Actividad, para crear el 'Sync. Merge' 74
Diagrama del Patrón 'Multiple Merge' en C-XPM 75
Diagrama del Patrón 'Discriminator' en C-XPM 75
Modificación al Autómata de una Actividad, para crear el 'Discriminator' 75
Diagrama del Patrón 'Arbitrary Cycles' en C-XPM 77
Diagrama del Patrón 'Implicit Termination' en C-XPM 77
Diagrama del Patrón 'MI without Synchronization' en C-XPM 78
Diagrama de los Patrones 'MI with Synchronization 13 y 14' en C-XPM 78
Modificación al Autómata de una MultiActividad, para soportar el Patrón 15 79
Diagrama del Patrón 'Deferred Choice' en C-XPM 79
Diagrama del Patrón 'Interleaved Parallel Routing' en C-XPM 80
Diagrama del Patrón 'Milestone' en C-XPM 83

LISTA DE TABLAS

Proceso de subasta en jPdl....................15
Elementos de la estructura de un Valor....................48
Operaciones de un Puerto-i....................48
Operaciones de un Dataflow-i....................49
Operaciones de un Workspace-i....................51
Operaciones de una Actividad-i....................51
Operaciones de una MultiActividad-i....................53
Operaciones de un Proceso-i....................54
Ejemplo de Implementación de la Acción 'cargarDocumento'....................68
Ejemplo de implementación del Workspace 'Exclusive-Choice'....................71
Ejemplo de implementación del Workspace 'Multi-Choice'....................73
Ejemplo de la operación 'checkRegisteredPaths'....................74
Ejemplo de la operación 'checkPaths'....................76
Ejemplo de la implementación del Workspace 'Deferred-Choice'....................79
Ejemplo de implementación del Workspace 'Start-Interleaving'....................80
Ejemplo de la Acción 'paths'....................81
Ejemplo de implementación del Workspace 'End-Interleaving'....................81
Ejemplo de la Acción 'executions'....................82
Comparación de C-XPM, APEL, XPDL y BPEL utilizando los Patrones de Control de Flujo....................83

1. INTRODUCCIÓN

La automatización de procesos de negocio como alternativa al desarrollo de aplicaciones no es un tema nuevo. Muchos son los avances realizados actualmente sobre este tema, motivado principalmente por su importancia y aplicación al mundo de los negocios. La idea de que un Analista de negocios trabaje mano a mano con un Programador en la creación de un proceso automatizado, es lo más llamativo y fundamental de este enfoque. Empresas como Sun, IBM y Microsoft han realizado grandes aportes a este campo, ratificando así la importancia que tiene esta área en el mundo informático y empresarial.

Sin embargo, a pesar de toda la atención que recibe hoy en día, aún no existe la opción que ofrezca todo lo que el desarrollador y el empresario necesitan. ¿Por qué para ambos? Porque la principal meta en la creación de este tipo de modelos es que, tal como se mencionó anteriormente, puedan ser utilizadas tanto para Ingenieros de Sistemas como para Analistas de Negocios. Un Modelo que de una vez por todas logré integrar estas dos ramas de una manera más transparente y práctica.

Debido al afán de grandes empresas por posicionar su producto rápidamente en el mercado (la carrera de siempre), hoy en día se puede observar como no existe aún un lenguaje de modelación de procesos que cumpla con todos los requerimientos del mundo actual (tales como la integración de aplicaciones ya existentes), especialmente, el mundo de los negocios. Es este apresuramiento el que hace que el enfoque hacia el desarrollo de una solución caiga sobre el lenguaje a utilizar, y no en el modelo que está detrás de la solución misma. Si las bases no son las mejores, cualquier cosa que se construya sobre éstas tendrá las mismas imperfecciones.

Cumbia-XPM (C-XPM de ahora en adelante) es un modelo para la definición de procesos, cuya principal característica es su gran poder expresivo, basado en la alta capacidad de extensibilidad que posee (de ahí sus siglas XPM: *Extensible Process Modeling*). Es el primer y más importante paso hacia el desarrollo de un Motor de Workflows que permita integrar de la mejor manera los tres grandes campos que conforman la automatización de Procesos de Negocios: Workflow, BPM (*Business Process Management*) y Orquestación de Servicios.

En el presente documento se analizarán los conceptos más importantes relacionados con la automatización de procesos, así como aquellas tecnologías o estándares que sobresalen como las principales alternativas actualmente. Se presentará ampliamente el Modelo C-XPM y se mostrará, mediante la representación de los Patrones de Control de Flujo, su capacidad de extensibilidad, comprobando así su gran poder de expresión.

2. OBJETIVOS

2.1 GENERALES

1. Crear y validar un modelo minimal altamente extensible, que sirva de especificación para la construcción de Sistemas Manejadores de Procesos Automatizados.

2.2 ESPECÍFICOS

1. Presentar las características principales de tecnologías y lenguajes para la definición de procesos de negocio usados actualmente.

2. Analizar los inconvenientes que implica utilizar las tecnologías actuales, con el fin de resaltar los puntos a abarcar por el modelo a realizar.

3. Identificar, definir y explicar cada uno de los elementos que hacen parte del modelo de Cumbia-XPM.

4. Definir y describir mediante ejemplos cada una de los mecanismos de extensibilidad de C-XPM.

5. Validar la capacidad expresiva y extensible del modelo C-XPM mediante la representación de los Patrones de Control de Flujo.

3. MODELOS EXISTENTES

A continuación, se presentan varios modelos de las principales tecnologías que actualmente se postulan como las más importantes dentro del área de la automatización de procesos en general. Cabe anotar que aunque algunos son conocidos como lenguajes, o como Sistemas Manejadores de Workflow, la idea fue realizar la comparación entre los modelos que se encuentran 'detrás' de cada uno de estas tecnologías. Por eso, de ahora en adelante, se hará referencia a cada uno de ellos como 'Modelo'.

Es importante aclarar que estos modelos fueron escogidos porque, además de su aceptación actual, cada uno representa un campo de acción diferente. Actualmente, existen muchísimos modelos de Workflows, por lo que entrar en detalle a analizar cada uno de ellos está fuera del alcance de este documento.

En el final del capítulo se presentará un análisis comparativo sobre los modelos descritos, teniendo en cuenta un conjunto específico de criterios, previamente definidos.

3.1 XPDL

XPDL es el acrónimo de XML Process Definition Language, lenguaje de definición de procesos impulsado por la WfMC como forma de estandarizar la Interfaz 1 de su modelo de referencia [WMC1995].

A continuación se presentan los principales conceptos en los que esta basado XPDL.

3.1.1 Definición del proceso. La definición del proceso provee información contextual que aplica a todas las entidades que se encuentran dentro de éste. Actúa como un contenedor del proceso como tal y provee información asociada con la administración del proceso, tales como autor y fecha de creación. También posee información que será utilizada durante la ejecución del proceso (parámetros de ejecución, prioridad entre otros [WMC2002]).

3.1.2 Actividad. Una actividad se define en XPDL como una unidad de trabajo lógica y auto contenida dentro del proceso [WMC2002]. Representa un trabajo a ejecutar, el cual puede ser realizado por una combinación de recursos o por un conjunto de aplicaciones.

Existen tres clases de actividades dentro de XPDL:

- **Route**: una actividad de tipo *Route* representa una actividad vacía, es decir, que su único objetivo es servir como elemento para expresar lógica de control de flujo. No tiene la capacidad de hacer cambios sobre los datos relevantes del proceso.

- **BlockActivity**: una actividad de tipo *BlockActivity* es una actividad que ejecuta un conjunto de actividades y transiciones, conocida en XPDL como *ActivitySet*. En otros modelos se conocen a estas actividades como subprocesos. Una actividad de tipo *BlockActivity*, cuando va a ser ejecutada, puede recibir parámetros de entrada y puede retornar parámetros de salida.

- **Implementation**: una actividad de *Implementation* es una actividad atómica dentro de la definición del proceso. Una actividad atómica puede ser ejecutada manualmente o automáticamente, o bien puede delegar su trabajo a otro proceso (otro flujo). De acuerdo a la característica de ejecución de la actividad, una actividad *Implementation* puede ser subdivida en tres subtipos. El primero de estos subtipos es el *No-Implementation*, que indica que una actividad va a ser realizada mediante un procedimiento manual. El segundo subtipo es del *Implementation-Tool*, que indica que la actividad va a ser ejecutada de forma automática mediante una aplicación; dicha aplicación debe ser referenciada en las aplicaciones pertenecientes a la definición del proceso. Por último, el subtipo *Implementation-Subflow* permite instanciar otros procesos.

3.1.3 Transiciones. Las actividades de XPDL están relacionadas entre si mediante elementos llamados transiciones. Una transición posee dos elementos básicos: la actividad fuente y la actividad destino. Por otro lado, una transición XPDL puede ser condicional o incondicional. Una transición incondicional es tomada cuando su actividad fuente termina su ejecución; por el contrario, una transición condicional será tomada dependiendo de una condición a evaluar.

Dependiendo del control de flujo de un proceso, se pueden ejecutar actividades de forma secuencial y/o paralela. En una actividad se puede definir la semántica asociada con el seguimiento de varias transiciones salientes de la misma (split), así como la semántica asociada con la sincronización de varias transiciones entrantes a dicha actividad (join). La semántica de estas transiciones se define a través de restricciones que indican su comportamiento.

Una restricción Split-AND indica que las transiciones se toman de forma paralela o concurrente, es decir, creando un hilo de control por cada transición presente en la restricción. Un Split-XOR continúa sólo por la primera de las transiciones que cumpla con la condición establecida.

Una restricción de tipo Join-AND indica que se debe esperar por el control de todas las transiciones presentes en la restricción. Una restricción de tipo Join-XOR indica que sólo se debe esperar por una de las transiciones presentes en la restricción.

Para expresar transiciones más complejas, la especificación de XPDL [WMC2002] indica que se pueden utilizar transiciones condicionales combinadas con actividades de tipo Route.

3.1.4 Participantes. La declaración de participantes en una definición de proceso, provee la descripción de los recursos que pueden llevar a cabo una o varias actividades. El modelo de recursos en XPDL no se refiere exclusivamente a personas. Existen otros tipos o clases de recursos, tales como roles y unidades organizacionales.

En tiempo de diseño solo se conoce de manera abstracta el recurso que va a ejecutar la actividad. En otras palabras, el recurso sólo se declara. Durante la ejecución del proceso se hace un enlace de esta definición con el recurso concreto y se definen las políticas de asignación de dicho recurso.

A través de la definición de los recursos en forma abstracta, en XPDL no se definen repositorios de recursos.

3.1.5 Aplicaciones. En la definición de un proceso se realiza una declaración de aplicaciones o interfaces que pueden ser invocadas por el motor de ejecución a fin de soportar la automatización de actividades.

Similar al modelo de recursos, las aplicaciones también se declaran en abstracto: se define el nombre de la aplicación y los parámetros de entrada y de salida, pero no se indica el tipo de aplicación. El enlace con la aplicación es responsabilidad del motor de ejecución, que debe tener la capacidad de realizar el llamado a la misma.

3.1.6 Datos Relevantes. Son los datos creados y utilizados por una instancia de proceso durante su ejecución [WMC2002].

Los datos pueden ser utilizados para mantener información persistente, durante y después de la ejecución del proceso. También pueden ser usados en las expresiones condicionales definidas en las transiciones.

En XPDL los datos no tienen un alcance definido, es decir, cualquier actividad tiene la capacidad de acceder a los datos relevantes especificados dentro de la definición del proceso.

3.1.7 Tipos de Datos. Los datos en XPDL son tipados. El esquema de XPDL incluye la definición de varios tipos básicos y complejos de datos, pero es posible definir nuevos tipos de datos complejos a partir de los existentes o completamente nuevos.

Para definir nuevos tipos de datos se debe crear un esquema XML e incluirlo en la definición de proceso. O también, según dice la especificación [WMC2002], se pueden referenciar de una fuente externa.

3.2 BPEL4WS - BUSINESS PROCESS EXECUTION LANGUAGE FOR WEB SERVICES

BPEL4WS [CUR2002] (Business Process Execution Language for Web Services - conocido también sólo como BPEL) es un lenguaje creado como una iniciativa de estandarización en el área de orquestación de Web Services. Por esta razón, es el resultado de la unión de WSFL (Web Services Flow Language [LEY2001]) de IBM, y de XLANG (Web Services for Business Process Design [THA2001]) de Microsoft, dos de los lenguajes de orquestación más importantes existentes antes de la especificación de BPEL4WS. A partir de la fusión de estas propuestas, BPEL4WS combina los enfoques de programación estructurada de XLANG, con los de grafos dirigidos de WSFL.

Este lenguaje permite modelar dos tipos de procesos: el abstracto y el ejecutable. En un proceso abstracto se especifica el intercambio de mensajes entre varios participantes, sin describir el comportamiento de cada uno. En un proceso ejecutable se especifica el orden de ejecución de actividades, los participantes que intervienen en el proceso, los mensajes intercambiados entre éstos, y el manejo de errores y excepciones en caso de algún fallo en la ejecución del proceso. En otras palabras, un proceso abstracto se limita a describir el protocolo de comunicación entre varios grupos (coreografía), mientras que el ejecutable es el que especifica la orquestación entre los Web Services del proceso a modelar [PEL2003].

El elemento fundamental de un proceso BPEL (nombre reducido con el cual también se le conoce a BPEL4WS) es la actividad. Las actividades están divididas en dos grupos: primitivas (o de acción simple) y estructuradas (o compuestas).

Entre las actividades primitivas se encuentran:

- <receive>: actividad que espera la llegada de un mensaje. Esta actividad se bloquea hasta que el mensaje sea recibido.
- <reply>: actividad que permite responder a un mensaje que haya sido recibido a través de un <receive>.
- <invoke>: actividad que permite invocar una operación de un Web Service.

Por otro lado, las actividades estructuradas se asemejan mucho a las ya conocidas en el campo de la programación estructurada. Las más significativas son:

- <secuence>: ejecuta un grupo de actividades en el orden léxico en el que están definidas.
- <switch>: permite escoger sólo uno de los caminos entre un conjunto de opciones.
- <while>: esta actividad funciona como un ciclo condicional.
- <flow>: permite que una o más actividades sean ejecutadas de forma concurrente.
- <scope>: define un alcance en el que se pueden definir variables y actividades de compensación.

Podría decirse que las actividades primitivas son aquellas que interactúan con componentes que son externos al proceso en sí, mientras que las estructuradas son las encargadas de establecer el flujo de ejecución del proceso [PEL2003].

En BPEL existe el concepto de participante. Este concepto no está asociado a personas, puesto que todos los participantes en un proceso BPE son Web Services. Por otro lado, los participante poseen roles, los cuales se pueden declarar dentro de la definición del proceso.

El concepto de datos también existe en BPEL. Éstos se utilizan para mantener el estado de las instancias de los procesos. Los datos en BPEL son tipados, y se declaran a través de variables. Los tipos de datos se deben especificar de dos formas: utilizando esquemas XML o definiendo tipos de mensajes a través de WSDL.

3.3 JBPM – JAVA BUSINESS PROCESS MANAGEMENT

El Java Business Process Management (jBPM) es un Sistema Manejador de Workflows (Workflow Management System – WFMS) flexible y extensible, perteneciente a la compañía JBoss [BAY2005]. Toda la funcionalidad se encuentra encerrada en una librería Java, la cual puede ser usada también en ambientes J2EE.

En jBMP, un proceso se representa mediante un grafo dirigido, el cual se compone de **nodos** y **transiciones**. Los nodos poseen diversos comportamientos (dependiendo de su tipo) y son los encargados del flujo de control a través del proceso. Dicho flujo de control se representa mediante un **token**, al cual se le indica que fluya mediante una señal. Esta señal debe ser dada por el nodo, indicando por cuál transición debe seguir el token.

Como se mencionó anteriormente, existen varios tipos de nodos. Sin embargo, antes de mencionarlos, es importante conocer las dos responsabilidades que todo nodo posee:

- Ejecutar código Java.
- Propagar la ejecución del proceso, es decir, el flujo de control.

Para esta última, existen varias alternativas:

- No propagar la ejecución.

- Propagar la ejecución por una de las transiciones que salen del nodo.
- Crear nuevos caminos de ejecución.
- Finalizar los caminos de ejecución.

Entre los tipos de nodos que ofrece jBPM se encuentran:
- **nodo-tarea:** Representa una o más tareas que serán ejecutadas por humanos. Cuando el nodo toma la ejecución, crea el número determinado de tareas, y dependiendo del tipo de sincronización, propaga la ejecución. Los tipos de sincronización son:
    - ***last:*** La ejecución continúa cuando la última tarea haya terminado. Si al llegar al nodo no se crean tareas, la ejecución continúa.
    - ***last – wait:*** La ejecución continúa cuando la última tarea haya terminado. Si al llegar al nodo no se crean tareas, la ejecución espera hasta que éstas hayan sido creadas.
    - ***first:*** La ejecución continúa cuando la primera tarea haya terminado. Si al llegar al nodo no se crean tareas, la ejecución continúa.
    - ***first – wait:*** La ejecución continúa cuando la primera tarea haya terminado. Si al llegar al nodo no se crean tareas, la ejecución espera hasta que éstas hayan sido creadas.
    - ***unsyncrhonized:*** la ejecución continúa, sin importar si existen o no tareas sin finalizar.
    - ***never:*** la ejecución nunca continúa, sin importar si existen o no tareas sin finalizar.
- **estado:** no realiza ninguna tarea. Puede ser usado para representar un estado que espera una señal externa para reanudar la ejecución del proceso.
- **decisión:** representa un punto en el que sólo una de las transiciones salientes puede ser tomada. La elección de dicha transición puede hacerse mediante condiciones sobre cada transición o por medio de una señal externa que indique por cuál transición seguir.
- ***fork:*** divide el camino de ejecución en varios. En otras palabras, crea un token por cada transición que sale del nodo.
- ***join:*** une varios caminos de ejecución en uno sólo, asumiendo que todos los tokens que le llegan fueron generados por el mismo *fork*. Una vez reciba todos los caminos, sólo un token es propagado por la <u>única</u> transición de salida.
- **nodo:** representa un nodo que ejecuta una acción definida por el usuario. Dicha acción puede ser cualquier cosa que el usuario desee, sin olvidarse de propagar la ejecución.

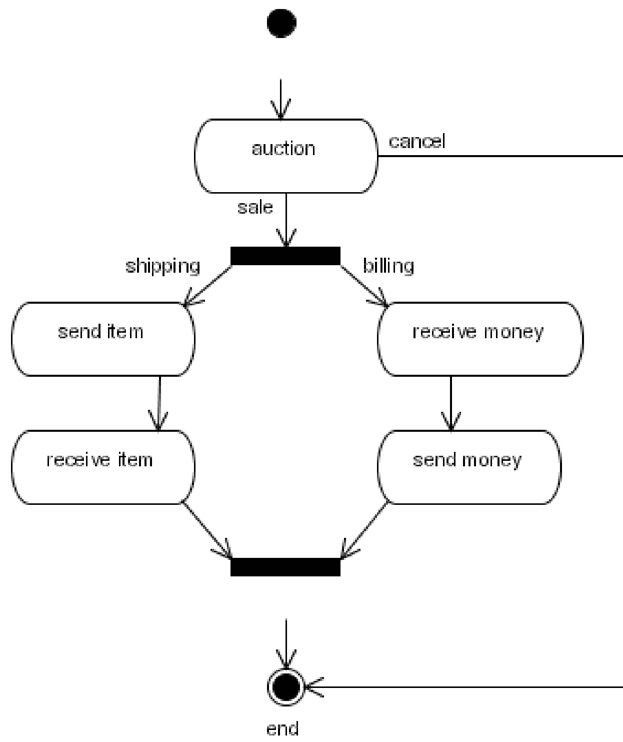
Debe quedar claro que estos tipos son los predefinidos por jBPM, puesto que dicho modelo permite al desarrollador crear sus propios nodos.

Un grupo de nodos puede ser agrupado para formar un **super-estado**. Dicho elemento puede estar contenido a su vez en otro super-estado, permitiendo así agregarle jerarquías a la definición de un proceso. Un super-estado también posee transiciones, e inclusive permite que transiciones salientes de nodos externos lleguen directamente a nodos internos, y viceversa.

Un concepto interesante que incluye jBPM es el de **acción** (*action*). Una acción es (específicamente hablando) un pedazo de código Java que se ejecuta después de la ocurrencia de un evento. Los principales eventos en la ejecución de un proceso jPBM son: entrar a un nodo, salir de un nodo y tomar una transición. Todo esto lo que busca es poder añadirle funcionalidad técnica a un proceso, sin necesidad de modificar el diseño del mismo.

El lenguaje de definición de procesos de jBPM es jPdl, el cual está basado en XML. A continuación se presenta un diagrama de un proceso de subasta. Seguidamente, se muestra su representación jBPM en jPdl:

Figura 3.1. Diagrama del Proceso de subasta



Tomado de jbpm.org/3/processmodelling.html.

Tabla 3.1. Proceso de subasta en jPdl

```xml
<process-definition>

  <start-state>
    <transition to="auction" />
  </start-state>

  <state name="auction">
    <transition name="auction ends" to="salefork" />
    <transition name="cancel" to="end" />
  </state>

  <fork name="salefork">
    <transition name="shipping" to="send item" />
    <transition name="billing" to="receive money" />
  </fork>

  <state name="send item">
    <transition to="receive item" />
  </state>

  <state name="receive item">
    <transition to="salejoin" />
  </state>

  <state name="receive money">
    <transition to="send money" />
  </state>

  <state name="send money">
    <transition to="salejoin" />
  </state>

  <join name="salejoin">
    <transition to="end" />
  </join>

  <end-state name="end" />

</process-definition>
```

Tomada de jbpm.org/3/processmodelling.html.

3.4 APEL – ABSTRACT PROCESS ENGINE LANGUAGE

APEL es un framework de definición de procesos desarrollado por el laboratorio LSR Francia como resultado de varios años de investigación en esta área. APEL esta basado en un modelo extensible que define dos tipos de extensiones; el primer tipo apunta a agregar nuevos conceptos del dominio de problema o extensiones horizontales, el otro tipo logra definir conceptos en una maquina abstracta para ejecutarla en maquinas concretas estas son las extensiones verticales [EST2003].

APEL esta basado en los conceptos de actividad, puerto, producto y dataflow. Una actividad es unidad de trabajo auto contenida. Las actividades en APEL son compuestas es decir que pueden tener otras actividades como parte de su definición. Las actividades define su cardinalidad, la cual determinará el máximos de instancias que puede tener la actividad durante la ejecución de un proceso. Cada actividad posee un Desktop que es el lugar donde se consumen y crean los productos [VIL2002].

Los productos son elementos de información producida, transformada y consumida por una actividad. Cada producto es único, por lo que se manejan versiones para diferenciar a productos que puedan ser iguales en algún momento de la ejecución de un proceso.

Los dataflows son canales por los cuales fluyen los productos, desde y hacia las actividades. Los dataflows, además de conducir los productos hacia los puertos, establecen el control de flujo la definición del proceso.

Un Puerto es un punto que define y controla la comunicación entre una actividad y un dataflow. En otras palabras, es el punto de llegada o salida de toda actividad. Los puertos definen un conjunto determinado de productos. Una vez un puerto reciba de un dataflow todos los productos que espera, será "liberado", es decir, podrá trasmitir dichos productos al destino correspondiente a través de los dataflows. Mientras no reciba todos los productos del conjunto que espera, el puerto no puede ser liberado.

Una actividad APEL puede tener varios puertos de entrada o salida. Lo importante a tener en cuenta es que sólo un puerto de salida puede ser liberado mientras la actividad se encuentre activa, y una vez liberado, todos los demás puertos de salida son inicializados, es decir, se eliminan los productos existentes en dichos puertos.

APEL es un lenguaje cuyo modelo de control de flujo está basado en grafos dirigidos. Este tipo de grafo se representa mediante nodos y enlaces. En APEL un nodo corresponde a una actividad, mientras que un enlace corresponde a un dataflow.

APEL define un modelo de recursos básico, donde cada actividad posee un responsable y un ejecutor. El modelo de recursos de APEL esta diseñado para ser enriquecido a través de extensiones. De esta forma, se puede utilizar un modelo de recursos más sofisticado en la definición de un proceso.

3.5 COMPARACIÓN ENTRE LOS MODELOS

A continuación se presenta un análisis comparativo entre los modelos descritos en la sección anterior. Dicho análisis está basado en un conjunto de criterios, los cuales pueden aplicar tanto a los modelos como a sus implementaciones. Cabe anotar que el objetivo del presente estudio no es encontrar el mejor modelo, sino por el contrario, presentar cada una de sus falencias y fortalezas. La elección de la mejor propuesta queda a criterio del lector.

3.5.1 Facilidad de Expresión. Hace referencia a qué tan fácil o qué tan complicado es modelar un proceso cualquiera. Para este criterio nos en la bibliografía existente sobre la evaluación de cada uno de los modelos frente a los patrones de control de flujo definidos en [VDA2003]. Por otro lado, en [DEN2000] mencionan varios aspectos que caracterizan un modelo simple, tales como:

- Soluciona un problema eficientemente, con una mínima cantidad de recursos y energía.
- Usualmente es más fácil de explicar, entender y mantener.

BPEL, al ser el resultado de la unión de XLANG y WSFL, soporta dos tipos de flujo de control: lenguaje estructurado y grafos dirigidos no cíclicos. Esto, aunque puede ser considerado como una ventaja por permitir mayor libertad a la hora de modelar, se convierte en un problema, ya que la

unión de ambos enfoques puede dificultar el entendimiento (y por ende el mantenimiento) de un modelo, a la vez que ofrece demasiadas construcciones que se traslapan [PWO2002]. Sin embargo, al poseer las construcciones propias de lenguajes estructurados, es bastante intuitivo y familiar para los diseñadores y desarrolladores.

APEL por su parte, con los conceptos de puertos, dataflows y productos, permite un buen flujo tanto de control como de información. Sin embargo, a medida que el número de caminos a tomar en cierto punto de un proceso va incrementando, el modelo se torna complejo, ya que es necesario definir los respectivos puertos de interacción. Por ejemplo, para expresar el patrón Synchronizing Merge se debe utilizar un puerto por cada combinación de caminos posibles. Esto quiere decir que si se tienen tres caminos, se necesitan 3 puertos, y para cuatro caminos, se necesitan 7 puertos, logrando una relación de 2^n-1 puertos. Por otro lado, el hecho de tener que definir en todos los puertos y en todos los dataflows los productos que espera, puede convertirse en una tarea bastante tediosa de realizar.

jBPM posee construcciones sencillas e intuitivas que permiten facilidad de modelación. Sin embargo, al no contar con herramientas de administración de procesos, la ejecución del mismo debe ser controlada mediante codificación en Java, realizada por el programador. Por otro lado, actualmente no existen estudios sobre soporte de patrones de flujo que permitan tener una mejor perspectiva de este modelo.

XPDL, al pretender ser un lenguaje de intercambio, se enfoca en ofrecer la intersección de las funcionalidad de los sistemas actuales, más que la unión. Al hacer esto, hay varios patrones importantes que no permite modelar [WVA2003], restándole así expresividad al lenguaje. Por otro lado, la especificación no es precisa en la semántica de ciertas construcciones, dejando espacio para varias interpretaciones sobre un mismo concepto, y dificultando así el proceso de modelación.

3.5.2 Sintaxis Gráfica. Todos los modelos de workflows poseen una sintaxis textual que permite expresar y ejecutar cualquier modelo. Una sintaxis gráfica corresponde a la representación gráfica de los conceptos que hacen parte del modelo, guardando a la vez un mapeo directo con la sintaxis textual. Es dicha sintaxis el primer paso para convertir un modelo en algo más intuitivo y práctico de utilizar.

La especificación de BPEL sólo ofrece las construcciones textuales del lenguaje. No presenta una definición gráfica para los elementos que hacen parte dicho lenguaje. Si embargo, algunas herramientas, tales como Oracle BPEL Process Design, incluyen una sintaxis gráfica para la definición de procesos.

jBMP no posee una sintaxis gráfica propia en su documentación. Sin embargo, utiliza los diagramas de actividad de UML 2 para visualizar sus ejemplos.

En la especificación de XPDL no se incluye sintaxis gráfica. Algunos editores definen su propia sintaxis gráfica, tal como hace JAWE de ObjectWeb [BOJ2004].

APEL define una sintaxis gráfica a nivel de lenguaje. En dicha sintaxis cada elemento textual del lenguaje tiene asociado un elemento gráfico.

3.5.3 Herramientas de Soporte. Corresponde al conjunto de herramientas que de alguna forma colaboran con la creación, administración y ejecución de modelos, así como aquellas que

permiten extender la capacidad expresiva de cada lenguaje a analizar. En este conjunto se pueden encontrar editores, monitores, administradores, listas de tareas o agendas, entre otras.

BPEL es un intento de estandarización de lenguajes de orquestación de Web Services. Por esta razón existen varias compañías que han implementado un conjunto de herramientas para soportar dicho lenguaje. Microsoft incluye el motor de ejecución de BPEL dentro de su Servidor *Biztalk Server 2004* y un plugin que se adapta en Visual Studio .Net 2003 que permite realizar edición grafica de los procesos. Oracle por su parte creó el motor de ejecución *Oracle BPEL Process Manager* que será incluido dentro de la próxima versión de su servidor de aplicaciones. Oracle también creo la herramienta de diseño de procesos Oracle BPEL Process Design que funciona como un plugin del IDE Eclipse.

XPDL es el lenguaje estándar de definición de procesos propuesto por la WFMC. Existen varias herramientas que soportan este lenguaje. Dentro de los editores que soportan XPDL se encuentra JAWE de Object Web, que brinda la capacidad de edición gráfica de procesos. Object Web también desarrolló un motor de ejecución para XPDL basado en CORBA conocido como Shark, en el cual se incluyen herramientas de monitoreo de procesos y manejadores de listas de tareas.

jBMP se distribuye como una librería Java de tal forma que el motor puede ser utilizado en una aplicación *Standalone* o dentro del servidor de aplicaciones JBoss a través de un *Wrapper* que lo presenta como un EJB de Sesión. jBMP recientemente acaba de crear una herramienta para edición gráfica (*JBoss jBPM Graphical Process Designer* [JBP2005]) pero no posee herramientas de administración de procesos. Sólo es posible interactuar con dichos procesos utilizando el API suministrado por jBMP.

APEL posee un motor de ejecución independiente de plataforma. Además, incluye una herramienta de edición grafica de procesos y una agenda para controlar la ejecución de las instancias y las tareas asignadas a los diferentes usuarios. Otras herramientas (como edición de Aspectos) son incluidas con APEL debido a las características del modelo utilizado.

3.5.4 Interacción con otros Modelos de Workflows. Es la capacidad de un modelo de interactuar con otros modelos creados bajo metamodelos diferentes. Esta interacción se puede dar en dos niveles: a nivel de lenguaje (Mapeo o traducción entre lenguajes) y a nivel de motor de ejecución (un motor de Workflow usa los servicios de otro motor).

BPEL no fue concebido con la idea que tuviera que interactuar con otros modelos de Workflow a nivel de lenguaje. Esto quiere decir que un proceso definido en BPEL4WS no tiene establecido ningún tipo de mapeo a otro lenguaje. Por otro lado, la interacción con otro sistema workflow depende estrictamente de la implementación realizada. Por ejemplo, al exponer un proceso BPEL como un Web Service, dicho proceso podría ser utilizado por otro tipo de modelo que posea la capacidad de interactuar con Web Services.

jPdl, el lenguaje de definición de procesos de JBMP, no ofrece ningún mapeo a otro lenguaje. Por otro lado, jBMP no puede interactuar con otro sistema workflow en su implementación básica, pero provee mecanismos de extensión que podrían posibilitar la interacción con otros motores de workflow.

XPDL fue diseñado como lenguaje común de definición de procesos, de tal forma que existen mapeos de otros lenguajes hacia XPDL pero no de forma inversa. La interacción de un sistema

basado en XPDL con otro sistema workflow depende también de la implementación realizada para XPDL.

En APEL no existen mapeos del lenguaje utilizado hacia otros lenguajes de definición de procesos. En cuanto al sistema manejador de workflow, éste no provee una interfaz para interactuar con otros sistemas workflow.

3.5.5 Extensibilidad, Introspección y Reflexión. Extensibilidad es la capacidad de modificar el comportamiento de los elementos que hacen parte del metamodelo. Introspección es la capacidad de consultar (en tiempo de ejecución) las características de los elementos que hacen parte de un modelo. Reflexión es la capacidad de modificar (en tiempo de ejecución) las características mencionadas anteriormente.

BPEL provee un mecanismo de extensión a través del cual se pueden incluir nuevos elementos en el lenguaje. Los nuevos elementos utilizados en la especificación de un proceso deben ser definidos dentro de un espacio de nombres XML y la semántica de comportamiento de estos elementos dependerá del motor en el cual se ejecute. La especificación de BPEL es clara en que la semántica de los elementos correspondientes al espacio de nombres no puede ser modificada. Por otro lado, el modelo no presenta en su especificación las características de introspección y reflexión. Sería necesario analizar si las implementaciones realizadas podrían soportar estas características.

Para extender el modelo de jBPM se define el concepto de *Acción*. Una Acción es un pedazo de código Java que puede ser ejecutado después de la ocurrencia de un evento. Dicha acción sólo puede acceder a las variables de un proceso, más no a la estructura del mismo. No existe reflexión en jBPM ya que en tiempo de ejecución no es posible modificar la especificación de un proceso.

XPDL define un conjunto de atributos extendidos a través de los cuales una implementación puede agregar nuevos atributos a una definición de proceso. Esta extensión es dependiente entonces del motor de ejecución del lenguaje, que puedan soportar las extensiones presentes. Se aclara que si un motor no soporta atributos extendidos, simplemente los debe ignorar. Se considera que es un mecanismo de extensión muy pobre.

El motor de APEL permite la extensibilidad del mismo mediante la definición de Aspectos. Sin embargo, este tipo de extensión puede resultar muy complicado, sobre todo a la hora de hacer seguimiento de depuración. Por otro lado, provee una API que permite en tiempo de ejecución modificar un modelo (Introspección y Reflexión).

3.5.6 Dependencias Tecnológicas. Tecnologías de las cuales depende un lenguaje de definición de procesos para ser funcional. Entre las principales tecnologías actualmente se encuentran Java, C++, J2EE, .Net y Web Services.

BPEL fue creado con el objetivo de realizar orquestación de Web Services. Por esta razón, está completamente ligado a esta tecnología, creando así una dependencia funcional con la misma.

jBMP fue creado exclusivamente para trabajar utilizando el lenguaje de programación Java. Esta dependencia se puede notar en las capacidades que brinda el sistema para extenderse a través de la implementación de interfaces Java. Además, el mecanismo utilizado para realizar llamados a servicios sólo se puede mediante utilizar llamados a clases que cumplan con una interfaz Java.

XPDL es un intento de estandarización, por tal razón no tiene dependencias tecnológicas. Los servicios que utiliza XPDL para realizar tareas automatizadas son declarados a un nivel abstracto y es el motor de ejecución el que se encarga de realizar el mapeo.

APEL tampoco presenta dependencias tecnológicas ya que puede utilizar cualquier clase de servicios para que sean coordinados. El llamado de estos servicios se realiza a través de aspectos, donde se intercepta la ejecución del proceso y se ejecuta un código determinado. Una de las características de APEL es su capacidad de extensión vertical, la cual buscar mantener separado la máquina de ejecución de los conceptos tecnológicos involucrados [EST2003].

Vale la pena anotar que cuando hablamos de sistema manejador de workflow todos tienen una dependencia con respecto a la tecnología que es utilizada para construirlos. El análisis realizado sobre BPEL y XPDL (siendo éstos lenguajes) se basa en cuáles son las dependencias tecnológicas de dichos lenguajes y no de los sistemas manejadores que se implementan para utilizarlos.

3.5.7 Capacidad de control. La forma en que un lenguaje interactúa con diferentes servicios (elementos estructuradores), tal como la invocación de aplicaciones y la comunicación con el exterior (intercambio de mensajes).

La interfaz que utiliza BPEL para comunicarse con sistemas externos es a través de definiciones WSDL. Para la comunicación sincrónica hacia los Web Services, BPEL utiliza la actividad *invoke*, la cual posee un esquema Request-Reply: la actividad se bloquea hasta obtener la respuesta. Cuando la comunicación es desde los Web Services hacia los procesos, se utiliza el par de actividades (*receive* y *reply*) para recibir y contestar respectivamente. Para la comunicación asincrónica, es posible enviar sólo un Request y continuar con la ejecución del proceso, utilizando la actividad *invoke* únicamente con variables de entrada.

JBMP tiene la capacidad de comunicarse con servicios externos utilizando el lenguaje de programación Java. En JPDL, un elemento Action indica la ejecución de un código Java. jBMP tiene entonces la capacidad de utilizar cualquier servicio que pueda ser llamado con Java. La comunicación siempre va desde el motor de jBMP hacia los servicios, y la comunicación puede ser sincrónica o asincrónica, dependiendo de la implementación de la acción.

En XPDL se define un conjunto de aplicaciones que puede ser utilizada por un proceso o por un paquete (grupo de procesos). Para cada una de estas aplicaciones se define un conjunto de parámetros. Todas estas definiciones se realizan de forma abstracta, es decir, sin indicar de qué tipo es el proceso y cómo se va a realizar el llamado. Luego, en ejecución, el motor debe tener la capacidad de 'mapear' estas definiciones a tipos de procesos concretos (Web Services, EJB's etc.). La comunicación siempre se realiza desde los procesos hacia los servicios, de manera sincrónica.

En APEL cuando se quiere invocar un servicio se debe implementar un aspecto que intercepte la ejecución de alguna actividad. La implementación actual provee un mecanismo para que el código que se ejecuta en el aspecto sea realizado en Java. Hay que tener en cuenta que la comunicación también es asincrónica, y solamente en el sentido del motor de APEL hacia los servicios y/o aplicaciones.

3.5.8 Granularidad. Es el nivel de abstracción del elemento básico de un modelo (actividad, estado, etc). Una granularidad "gruesa" simplifica la especificación e incrementa la flexibilidad del modelo, mientras que la granularidad "fina" simplifica la ejecución del mismo.

BPEL maneja los conceptos de actividades primitivas y estructuradas. Las primeras permiten la interacción con los elementos externos al proceso (Web Services) y las segundas permiten el flujo de control. Una actividad estructurada puede estar compuesta de otras actividades estructuradas y de actividades primitivas. El nivel de profundidad de las mismas es el definido por el modelador.

APEL maneja el concepto de actividades simples y compuestas. Una actividad simple es el paso más elemental en la ejecución del proceso, mientras que las compuestas pueden contener otras actividades (compuestas o simples). En sí, un proceso APEL es concebido como una gran actividad que contiene a todas las actividades del proceso.

En XPDL, además del concepto de actividad (el elemento básico), se encuentran los conceptos de Conjunto de Actividades (ActivitySet) y de Bloque de Actividades (BlockActivity). El primero, como su nombre lo indica, es un conjunto de actividades agrupadas que pueden ser referenciadas en cualquier parte del proceso. El Bloque de Actividades es la forma de referenciar un proceso dentro de otro proceso. Esto quiere decir que un proceso definido en XPDL puede ser referenciado en otro proceso y ser visto como una actividad.

JBPM, en su lenguaje jPdl, incorpora el concepto de *superstate*. Este concepto se asemeja al de BlockActivity, en XPDL. Un *superstate* es un conjunto de nodos que representa la ejecución de un proceso, es decir, un subproceso. Cuando la ejecución del proceso llega a un *superstate*, se ejecutan los nodos definidos dentro de él.

3.5.9 Modelo de Flujo de Control. Actualmente existen dos tipos de enfoque en este concepto. Modelos cuyo flujo se comporta como bloques estructurados y modelos cuyo flujo se comporta como un grafo dirigido. El primero guarda estrecha relación a los lenguajes de programación estructurados (uso de estructuras especiales), mientras que el segundo se asemeja más a lo que busca representar un modelo de workflow (Nodos, enlaces, etc).

BPEL, al ser el resultado de la unión de XLANG y WSFL, soporta ambos tipos de flujo de control. Como herencia de XLANG soporta el enfoque de bloques estructurados. WSFL le brinda a BPEL su capacidad de expresión utilizando grafos dirigidos no cíclicos.

APEL y jBPM utilizan un enfoque de grafos dirigidos, soportando ciclos arbitrarios. En APEL, la decisión sobre el enrutamiento se encuentra en los puertos; en jBPM esta decisión es tomada en los nodos.

Por su parte, XPDL, en su intento de estandarización, soporta tres niveles de conformidad con su modelo. El nivel non-blocked soporta la definición de grafos dirigidos; el nivel loop-blocked soporta grafos dirigidos no cíclicos; y el nivel full-blocked soporta grafos dirigidos no cíclicos, con una relación de uno a uno entre los splits y joins del mismo tipo (orientado a estructuras).

3.5.10 Modelo Flujo de Información. Es la representación de los caminos posibles que puede tomar el intercambio de información. Algunos modelos proveen una representación separada del flujo de control, mientras que otros los unen. Esto último quiere decir que el flujo de información está implícito en el flujo de control.

En BPEL el flujo de información no esta explícito en el flujo de control. El intercambio de datos se hace a través de variables, cuyo acceso puede variar dependiendo del alcance en el que ésta haya

sido definida. Dicho alcance puede ser a nivel de procesos (variables globales) o a nivel de bloques utilizando la construcción <scope> (variables locales).

En APEL, la información se representa mediante Productos. Los productos son transportados a través de los dataflows, por lo que el flujo de información y el de control están unidos. Además, cada actividad posee un desktop donde residen los productos a ser manipulados.

En XPDL existe el concepto de datos relevantes, los cuales son expresados a través de variables. Sólo existen dos alcances para las variables definidas en XPDL: alcance de paquete y alcance de proceso. El alcance de paquete está disponible para cualquier elemento dentro del paquete; el alcance de proceso está definido para cualquier elemento en el proceso.

jBPM también define variables para manejar su información, sólo que todas las variables definidas en un proceso son consideradas como globales (sólo existe un alcance).

Como se pudo observar en el estudio, cada propuesta posee ventajas y desventajas sobre los demás. Sin embargo, como se mencionó al comienzo de la sección, queda a criterio de cada persona dar su opinión sobre el que considere el mejor ante los demás.

4. PATRONES DE CONTROL DE FLUJO

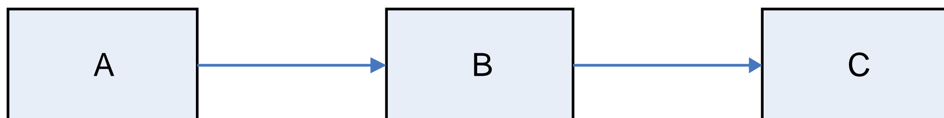
Los Patrones de Control de Flujo [VDA2003] son a la teoría de Workflows, lo que los Patrones de Diseño [GAM1995] son al Diseño Orientado por Objetos. Son soluciones generalizadas a problemas que muy frecuentemente aparecen en el diseño de flujo de tareas. Actualmente es uno de los principales frameworks a la hora de evaluar un modelo de Workflow, por lo que es fundamental su presencia en este trabajo de investigación.

Los Patrones de Control de flujo se dividen en cinco grandes grupos: Control Básico, Ramificación y Sincronización Avanzada, Estructurales, Instancias Múltiples, Basados en Estado y de Cancelación. A continuación se presentan cada uno de los patrones pertenecientes a estos grupos, dando su definición, un ejemplo y el diagrama que representa su solución. Seguidamente, se mostrará un estudio de APEL basado en estos patrones. Cabe anotar que sólo se incluyó este análisis, puesto que actualmente ya existen otros estudios sobre los demás lenguajes descritos en el capítulo anterior [PWO2002], [WVA2003].

4.1 PATRONES DE CONTROL BÁSICO

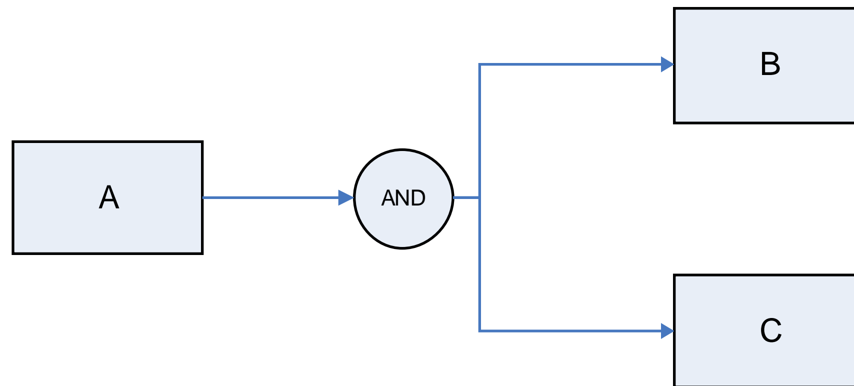
PF1 ***Sequence.*** Una actividad en un proceso es habilitada después de la culminación de otra actividad en el mismo proceso.
Ejemplo: Después de pagar por un producto, el cliente recibe dicho producto.

Figura 4.1. Diagrama del Patrón 'Sequence'



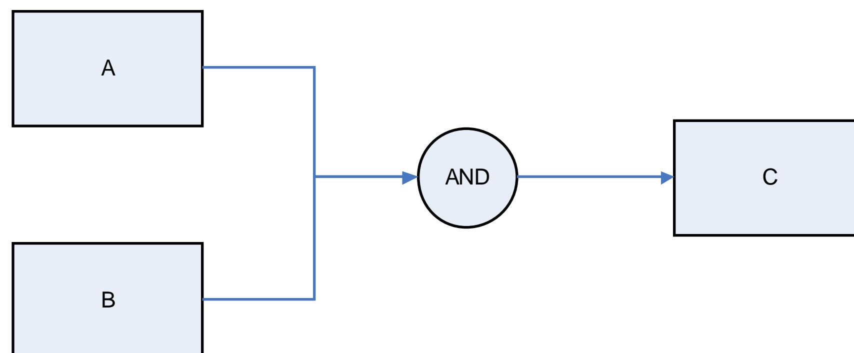
PF2 ***Parallel Split.*** Un punto en el proceso donde un único camino de control se divide en varios caminos de control, los cuales pueden ser ejecutados en paralelo, permitiendo así que todas las actividades puedan ser ejecutadas simultáneamente o en cualquier orden.
Ejemplo: Después de que la actividad *pago* se ha ejecutado las actividades *informar a cliente* y *entregar producto* pueden ser ejecutadas

Figura 4.2. Diagrama del Patrón 'Parallel Split'



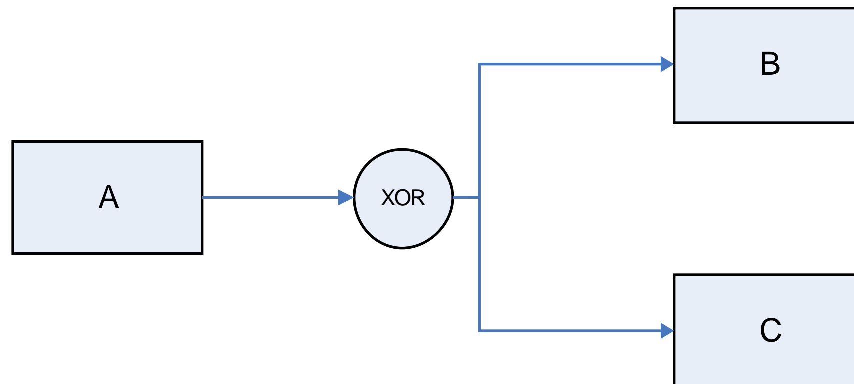
PF3 ***Synchronization.*** Un punto en el proceso donde múltiples subprocesos y/o actividades convergen en un único camino de control, sincronizando los múltiples caminos de los subprocesos y/o actividades entrantes. Este patrón asume que cada camino de llegada se ejecuta sólo una vez.
Ejemplo: Después de ejecutar las actividades *enviar_tiquetes* y *recibir_pago* la actividad *archivar* se habilita.

Figura 4.3. Diagrama del Patrón 'Synchronization'



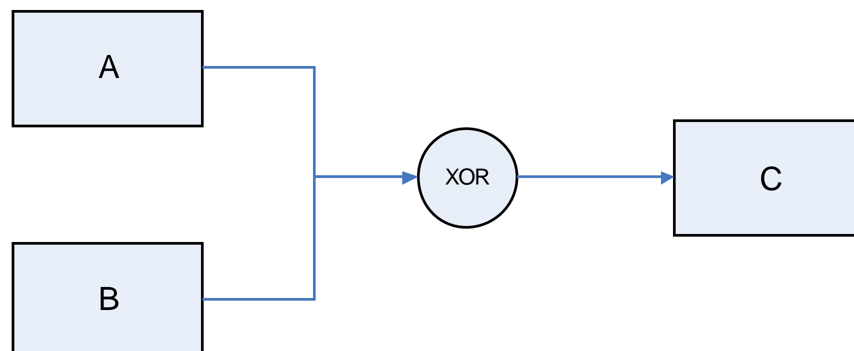
PF4 ***Exclusive Choice.*** Un punto en el proceso en el que, con base a una decisión o información de control de flujo, es escogido un camino entre varios.
Ejemplo: Dependiendo si un retiro supera los US$10.000, el director del banco es notificado o no.

Figura 4.4. Diagrama del Patrón 'Exclusive Choice'



PF5 ***Simple Merge.*** Punto en el proceso en que uno o varios caminos alternativos se unen sin necesidad de sincronización. Se supone que ninguno de dichos caminos alternativos se ejecuta en paralelo.
Ejemplo: Después que una película es comprada o alquilada, ésta es entregada al cliente.

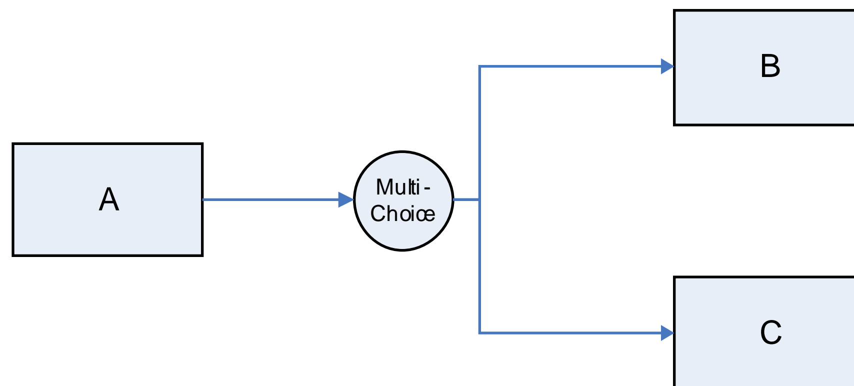
Figura 4.5. Diagrama del Patrón 'Simple Merge'



4.2 PATRONES DE RAMIFICACIÓN Y SINCRONIZACIÓN AVANZADA

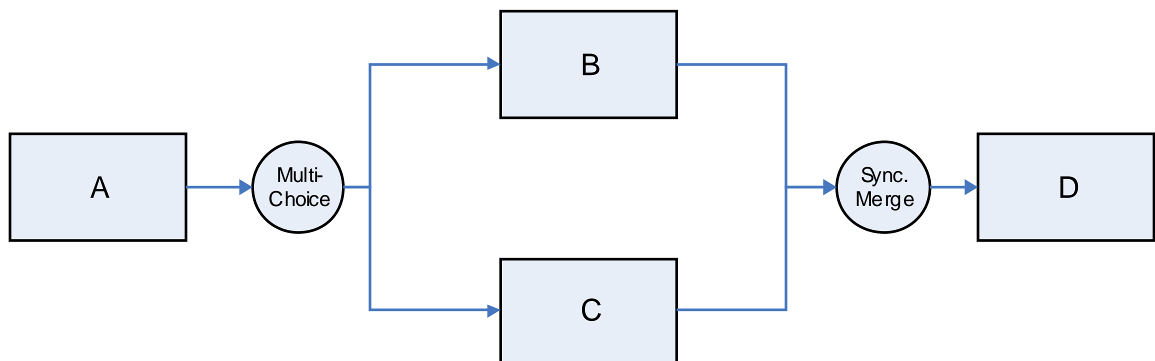
PF6 ***Multiple Choice.*** Punto en el proceso donde, con base a una decisión o información de control, se escoge un número de caminos que son ejecutados en paralelo.
Ejemplo: Después de hacer la reserva de un vuelo, se puede reservar una habitación en un hotel y/o alquilar un carro.

Figura 4.6. Diagrama del Patrón 'Multi-Choice'



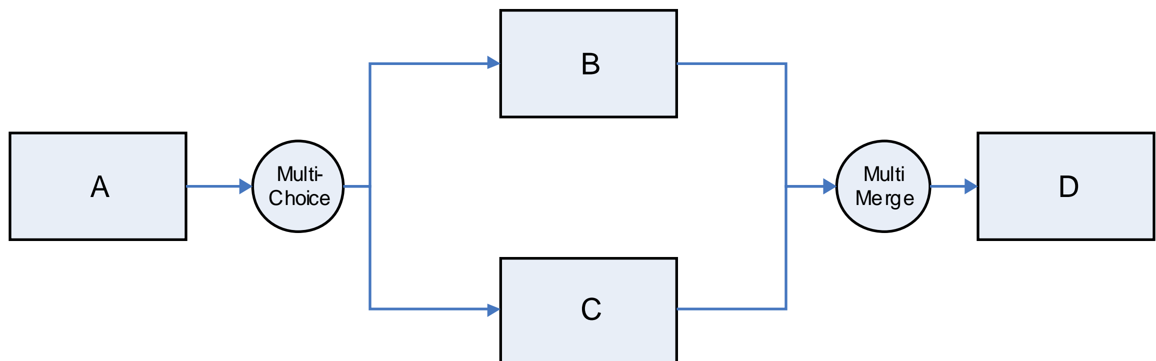
PF7 ***Synchronizing Merge.*** Punto en el proceso donde varios caminos se unen en uno solo. Si sólo uno de los caminos está siendo ejecutado, la actividad siguiente a la unión es activada una vez termine dicho camino. Si no, se deben sincronizar todos los caminos que se hayan ejecutado. Se supone que un camino que fue activado no puede volver a ser activada mientras la unión está esperando que otros caminos terminen.
Ejemplo: Después de reservar una habitación y/o alquilar un carro, se debe generar una factura por los servicios solicitados.

Figura 4.7. Diagrama del Patrón 'Synchronizing Merge'



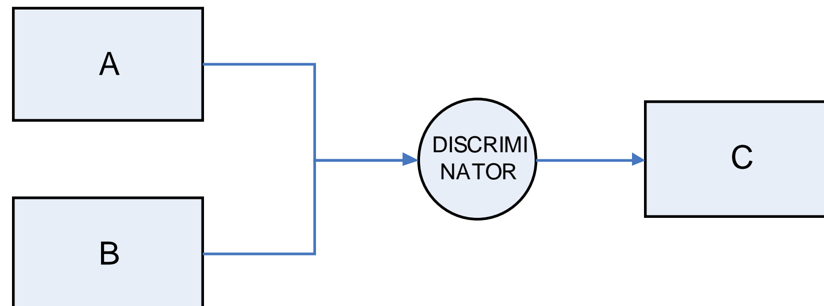
PF8 ***Multiple Merge.*** Punto en el proceso donde se unen varios caminos sin sincronizarse. Si más de un camino es activado, entonces la actividad que sigue a la unión será ejecutada por cada una de estas activaciones.
Ejemplo: Una modificación del ejemplo anterior. Después de reservar una habitación de hotel y/o alquilar un carro, se debe generar una factura por cada uno de estos servicios.

Figura 4.8. Diagrama del Patrón 'Multi-Merge'



PF9 *Discriminator.* El discriminador es un punto en el proceso que espera que se complete la llegada de un camino de control antes de continuar con la siguiente actividad. En el momento en que llega ese camino de control, se ejecuta la siguiente actividad y se ignoran los demás caminos que llegan al discriminador. Una vez todos los caminos han llegado al discriminador, éste se "resetea" para que pueda activarse nuevamente.
Ejemplo: Una consulta es enviada a dos servicios de base de datos en Internet, y sólo la primera respuesta es procesada. La otra es ignorada.

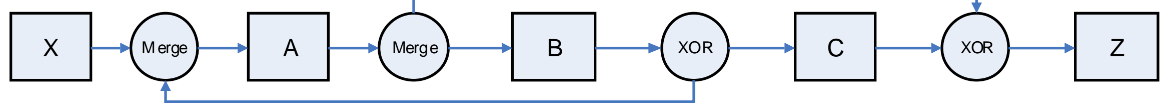
Figura 4.9. Diagrama del Patrón 'Discriminator'



4.3 PATRONES ESTRUCTURALES

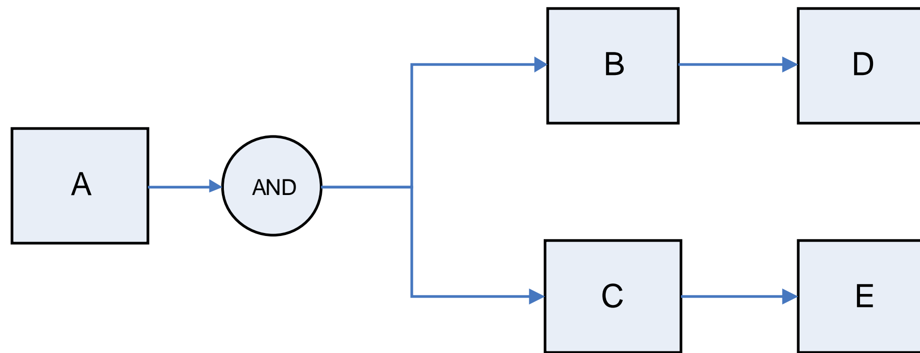
PF10 *Arbitrary Cycles.* Un punto en el proceso donde una o más actividades pueden ser ejecutadas de forma repetida.
Ejemplo: La actividad *revisar_documento* es ejecutada hasta que la actividad *aprobar_documento* termina con un visto bueno.

Figura 4.10. Diagrama del Patrón 'Arbitrary Cycles'



PF11 *Implicit Termination.* Un subproceso termina cuando no existen más actividades por ejecutar.

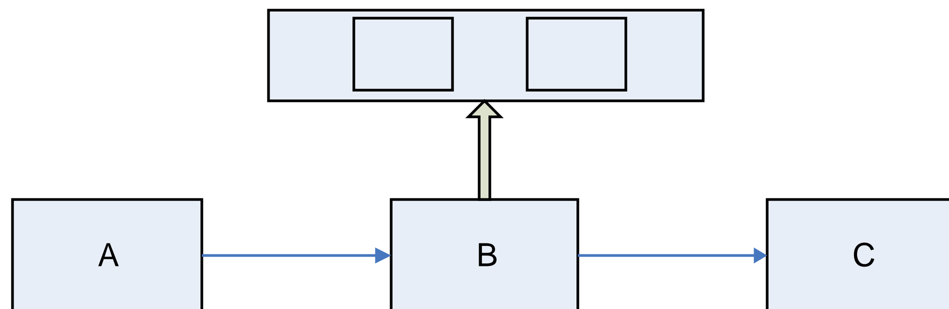
Figura 4.11. Diagrama del Patrón 'Implicit Termination'



4.4 PATRONES DE INSTANCIAS MÚLTIPLES

PF12 MI *without Synchronization.* En el contexto de un proceso, varias instancias de una actividad son creadas (posiblemente de manera consecutiva), con la posibilidad de ejecutarse en paralelo.
Ejemplo: Al reservar un viaje, se pueden reservar varios vuelos si así lo requiere dicho viaje.

Figura 4.12. Diagrama del Patrón 'MI without Synchronization'

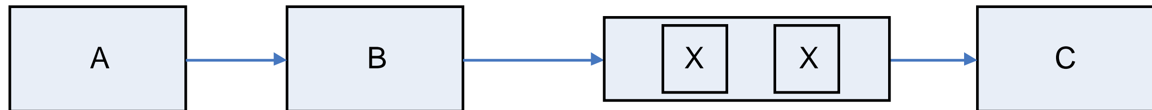


PF13-PF15 *MI with Synchronization.* Punto en un proceso donde varias instancias de una actividad son iniciadas, para luego ser sincronizadas antes de seguir con el resto del proceso. En

PF13 el número de instancias se conoce en tiempo de diseño; en PF14 el número es conocido en tiempo de ejecución, pero antes de que las instancias hayan sido inicializadas; en PF14 el número de instancias no es conocido de antemano: nuevas instancias son creadas a medida que se necesiten.

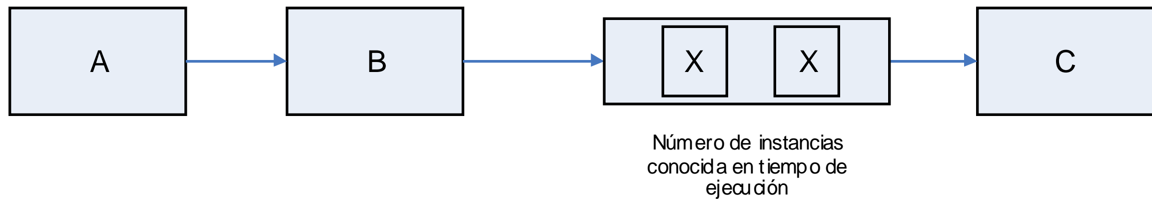
Ejemplo de PF13: Al reservar un viaje Cartagena-Madrid, se sabe que se deben hacer 2 escalas (3 vuelos): Cartagena-Bogotá, Bogotá-Nueva York y NuevaYork-Madrid. Estas 3 actividades se pueden ejecutar en paralelo.

Figura 4.13. Diagrama del Patrón 13



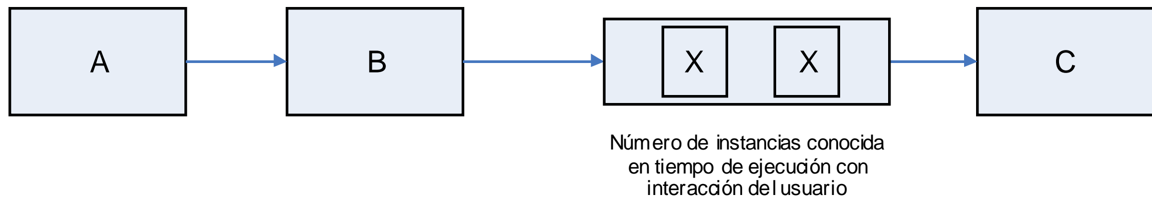
Ejemplo de PF14: Se desea mandar a revisar un texto a varios profesores. El número de profesores se conoce justo antes del envío.

Figura 4.14. Diagrama del Patrón 14



Ejemplo de PF15: Al realizar las reservas de un tour en particular, se pueden realizar varias reservas de viajes. El número de viajes sólo es conocido en tiempo de ejecución mediante la interacción con el usuario.

Figura 4.15. Diagrama del Patrón 15

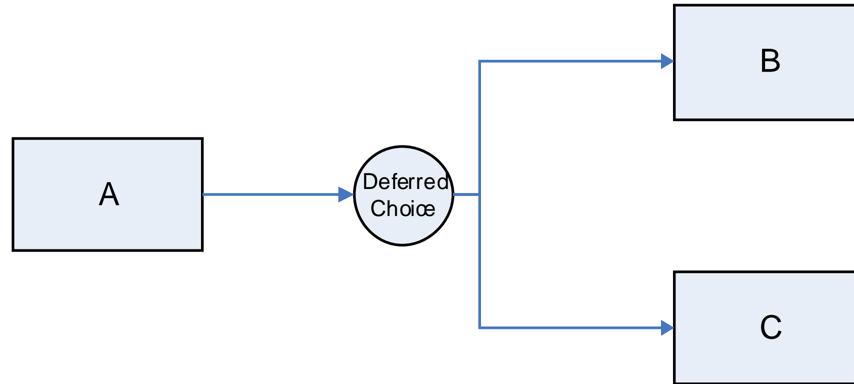


4.5 PATRONES BASADOS EN ESTADO

PF16 *Deferred Choice.* Un punto en el proceso en el cual uno de varios caminos alternativos es escogido. A diferencia del patrón *Exclusive Choice* la decisión no es tomada explícitamente (basado

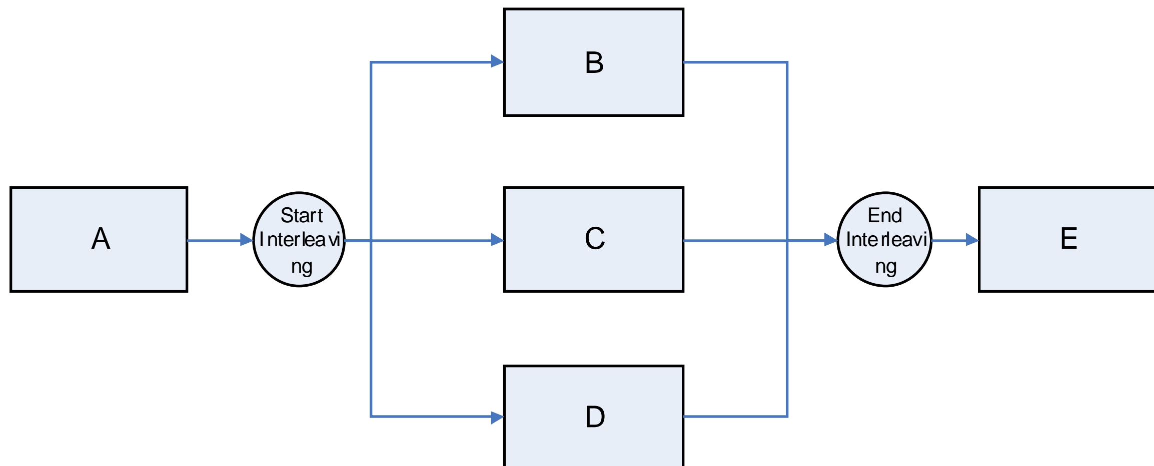
en la información de control), sino que se ofrecen varias alternativas presentes en el ambiente para hacer la elección. Es importante notar que el momento de tomar la decisión del camino a seguir se puede hacer tan tarde como se quiera en la ejecución del proceso.
Ejemplo: Un punto en el proceso donde debe tomarse una decisión por parte de un humano.

Figura 4.16. Diagrama del Patrón 'Deferred Choice'



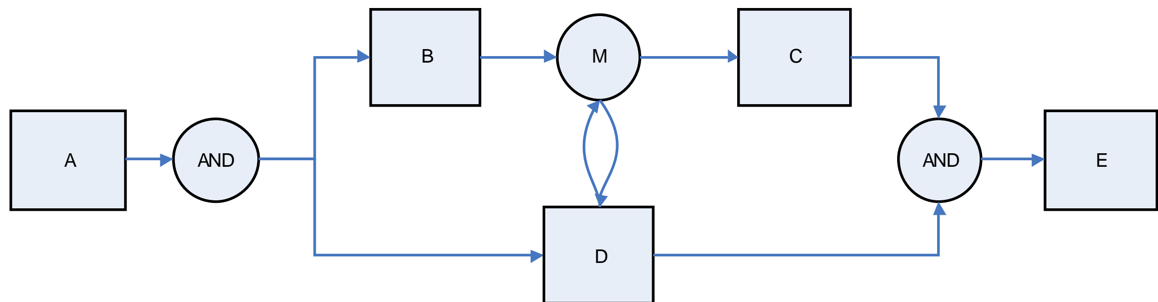
PF17 ***Interleaved Parallel Routing.*** Un conjunto de actividades es ejecutado en un orden arbitrario. Cada actividad del conjunto se ejecuta una sola vez, y mientras lo haga, ninguna otra actividad puede estar ejecutándose. El orden de la ejecución se decide en tiempo de ejecución, y sólo hasta que una actividad del conjunto ha terminado, es tomada la decisión de qué actividad sigue.
Ejemplo: En un examen médico, el paciente debe someterse a un examen físico, un examen de sangre y un examen de rayos X. Las actividades pueden ejecutarse en cualquier orden, pero sólo una vez cada una. Además, dos actividades no se pueden hacer a la vez.

Figura 4.17. Diagrama del Patrón 'Interleaved Parallel Routing'



PF18 *Milestone.* La habilitación de una actividad depende de que la instancia del proceso se encuentre en un estado específico, es decir, la actividad sólo se habilita si un cierto punto en el proceso (milestone) ha sido alcanzado y no ha expirado. Considere tres actividades A, B y C. La actividad A solamente está habilitada si la actividad B ya ha sido ejecuta y la actividad C no ha sido ejecutada aún. De esta forma, A no esta habilitada antes de la ejecución de B y después de la ejecución de C.
Ejemplo: La actividad *reservar_hotel* o *alquilar_vehículo* puede ser ejecutada mientras la actividad *imprimir_solicitud* no se haya realizado.

Figura 4.18. Diagrama del Patrón 'Milestone'



4.6 PATRONES DE CANCELACIÓN

PF19 Cancel Activity & PF20 Cancel Case El *Cancel Activity* termina la ejecución de una actividad, mientras que el *Cancel Case* termina la ejecución de todo un proceso.
Ejemplo de PF19: Un cliente cancela la reserva de un vuelo.
Ejemplo de PF20: Un cliente cancela la reserva de un viaje.

Figura 4.19. Diagrama del Patrón 'Cancel Activity'

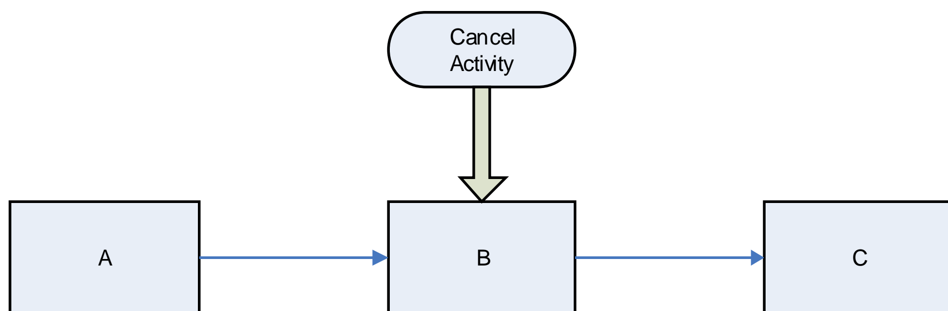
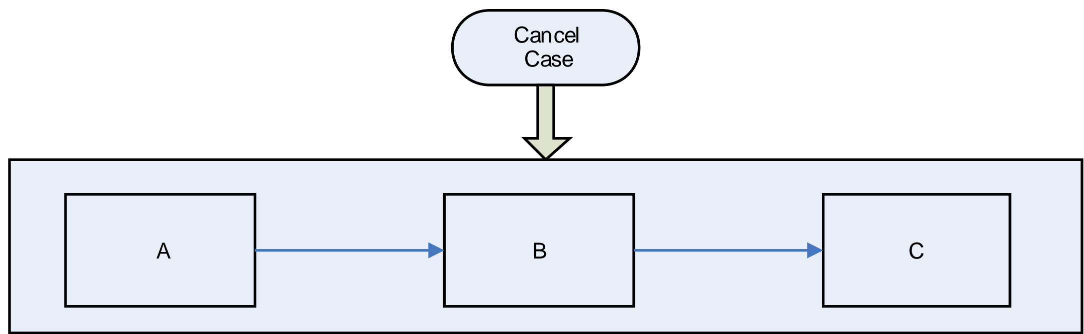


Figura 4.20. Diagrama del Patrón 'Cancel Case'

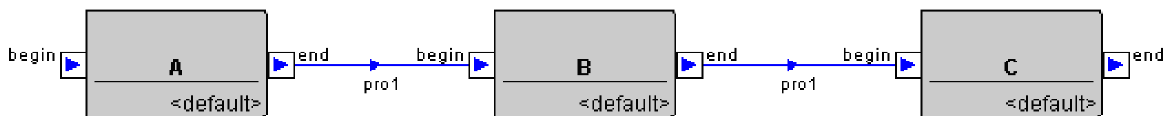


4.7 ANÁLISIS: APEL Y PATRONES DE CONTROL DE FLUJO

A continuación se presenta el análisis realizado a APEL, tomando como referencia los Patrones de Flujo mencionados en las secciones anteriores. Teniendo en cuenta que APEL tiene una notación gráfica definida, se optó por mostrar las soluciones de los patrones mediante dicha representación.

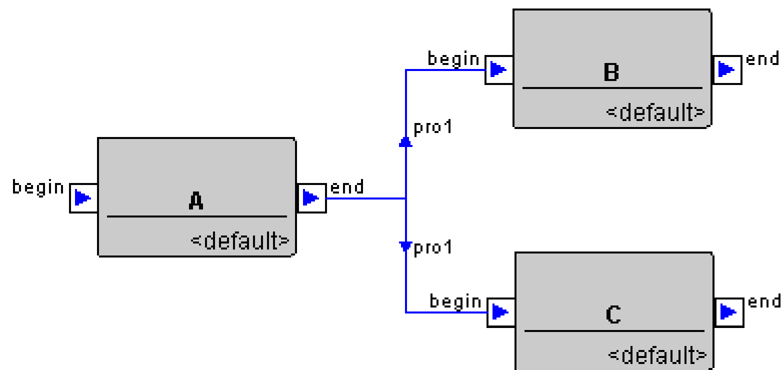
PF1 ***Sequence.*** Es el patrón más sencillo. Consiste en unir el único puerto de salida de una actividad con el único puerto de entrada de otra actividad, mediante un *dataflow*.

Figura 4.21. Diagrama del Patrón 'Sequence' en APEL



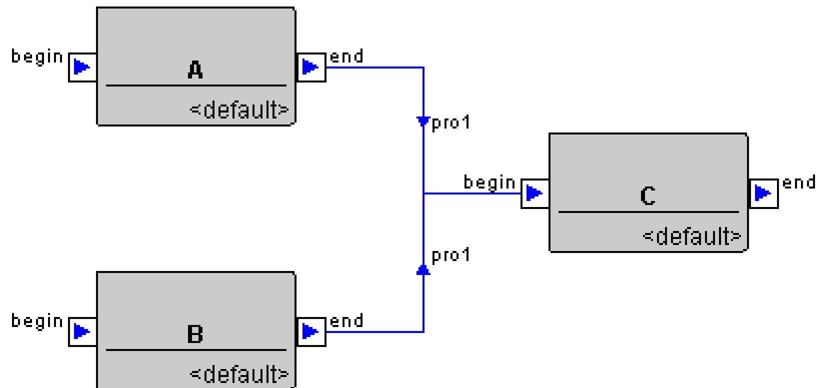
PF2 ***Parallel Split.*** En APEL se soporta este patrón utilizando una actividad que tiene un puerto de salida con varios *dataflows* dirigidos a cada uno de los puertos de entrada de las actividades que van a tomar el control de forma paralela. Una vez el puerto de salida es activado, éste entrega los productos a los *dataflows* correspondientes, los cuales dirigen dichos productos a las nuevas actividades.

Figura 4.22. Diagrama del Patrón 'Parallel Split' en APEL



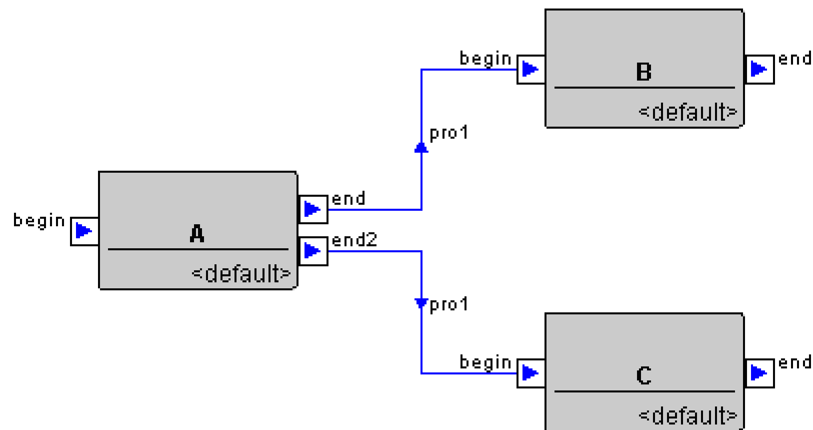
PF3 ***Synchronization.*** En APEL se soporta el patrón *Synchronization* utilizando una actividad donde convergen los diferentes caminos que se van a sincronizar. En esta actividad se coloca un puerto de entrada al cual están conectados todos los *dataflows* que provienen de los puertos de salida de aquellas actividades que se van a sincronizar. Dicho puerto espera todos los productos de los *dataflows* mencionados. Sólo es posible ejecutar la actividad que sincroniza cuando a su puerto de entrada han llegado todos los productos provenientes de las actividades a sincronizar.

Figura 4.23. Diagrama del Patrón 'Synchronization' en APEL



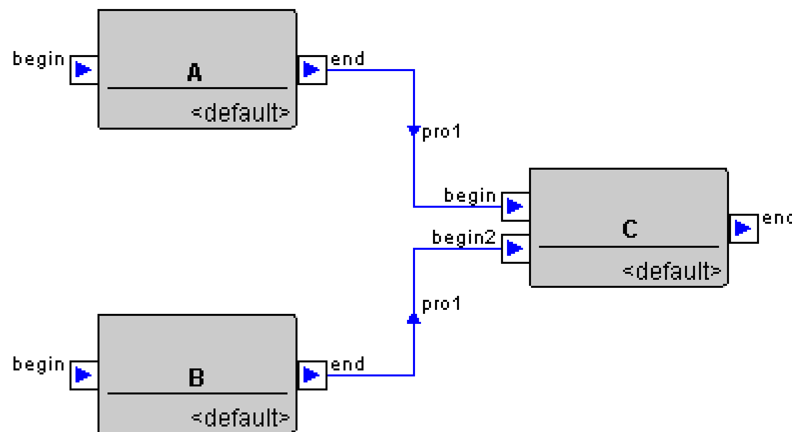
PF4 ***Exclusive Choice.*** El concepto de puerto en APEL es el que permite tener una implementación directa de este patrón. Al definir un puerto de salida para cada camino, se garantiza que sólo uno de dichos caminos será tomado. Esto se debe a que en APEL sólo puede escogerse un puerto de salida, de los que se encuentren activados.

Figura 4.24. Diagrama del Patrón 'Exclusive Choice' en APEL



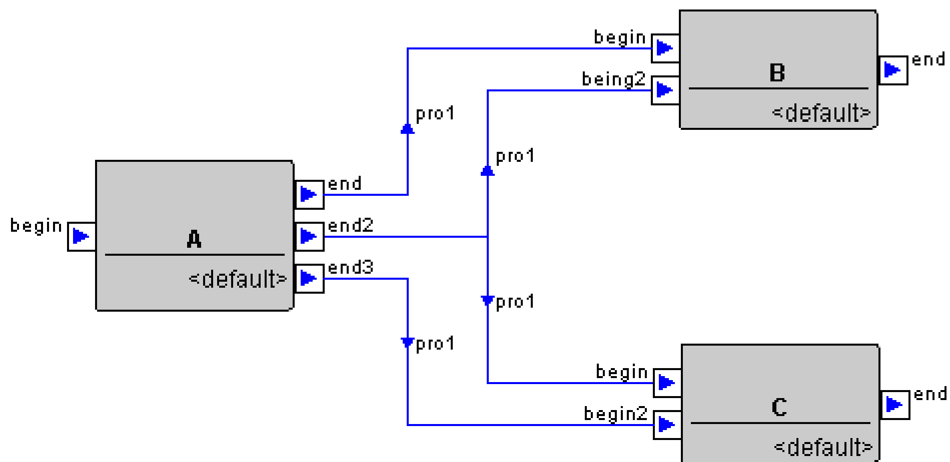
PF5 ***Simple Merge.*** Nuevamente el concepto de Puerto permite la implementación de este patrón. Al definir un puerto por cada flujo de entrada, la actividad C será ejecutada una vez se "llene" uno de sus puertos de entrada.

Figura 4.25. Diagrama del Patrón 'Simple Merge' en APEL



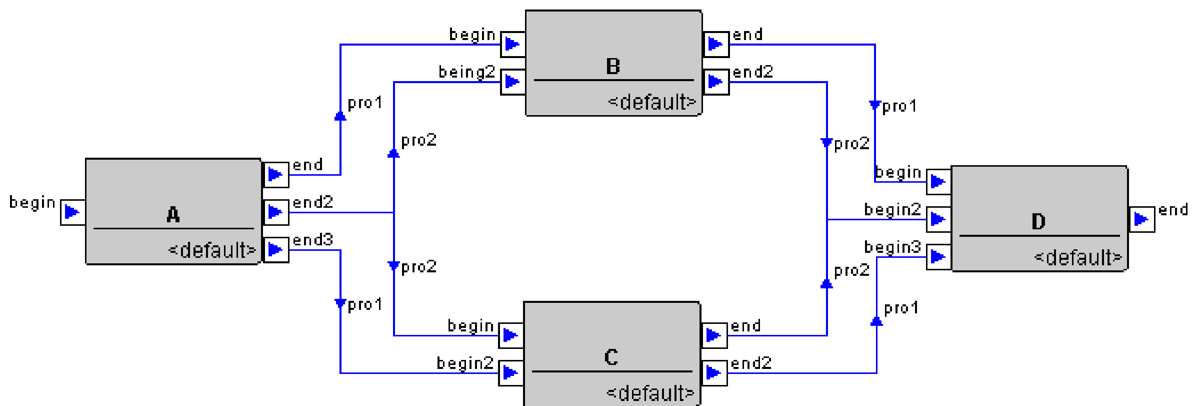
PF6 ***Multiple Choice.*** Para poder expresar lo que solicita el patrón, se definen 3 puertos de salida: uno para darle control a la actividad B; otro para darle control a la actividad C; y otro para darle control a ambas actividades. De igual forma, se crean dos puertos de entrada tanto para la actividad B como para la C. Como se puede observar, a medida que incrementen el número de caminos alternos a tomar, así aumentará el número de puertos de salida de la actividad A (en una relación de 2^n-1).

Figura 4.26. Diagrama del Patrón 'Multiple Choice' en APEL



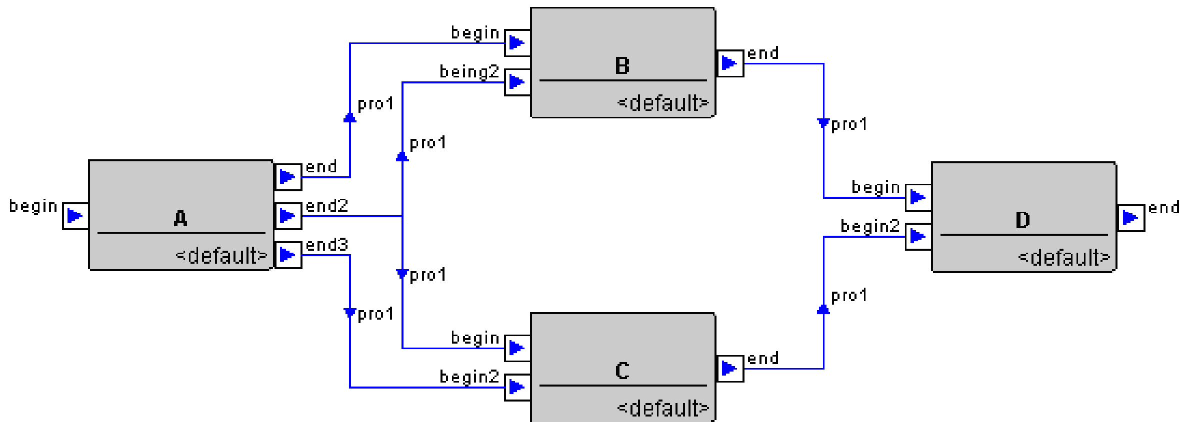
PF7 ***Synchronizing Merge.*** Para representar este patrón es necesario utilizar el patrón anterior (*Multi-Choice*). De manera similar, se definen tres puertos de entrada para la actividad D: uno que indica que sólo B fue activada; otro que indica que sólo C fue activada; y otro que indica que ambas (B y C) fueron activadas. En este punto es importante también el manejo de otro concepto: Producto. Se deben definir productos diferentes para cada uno de los flujos mencionados anteriormente, con el fin de que el patrón se pueda cumplir. Es por esto que se definió un producto adicional para cuando se tomen ambos caminos. De esta forma, B y C sabrán cual de sus dos puertos de salida tomar, llegando así al correspondiente puerto de entrada en D. Sin embargo, al igual que en el *Multi-Choice*, a medida que sean más los caminos a unir, el número de puertos de entrada de D se incrementará exponencialmente (2^n-1, donde 'n' es el número de caminos).

Figura 4.27. Diagrama del Patrón 'Synchronizing Merge' en APEL



PF8 ***Multiple Merge.*** Este diagrama es fácilmente soportado gracias al ciclo de vida que tienen las actividades en APEL. En dicho ciclo de vida, una actividad puede ser reactivada cuando alguno de sus puertos recibe el conjunto de productos que espera. De esta forma, como se aprecia en el diagrama, D será activada por cada *dataflow* que haya sido activado.

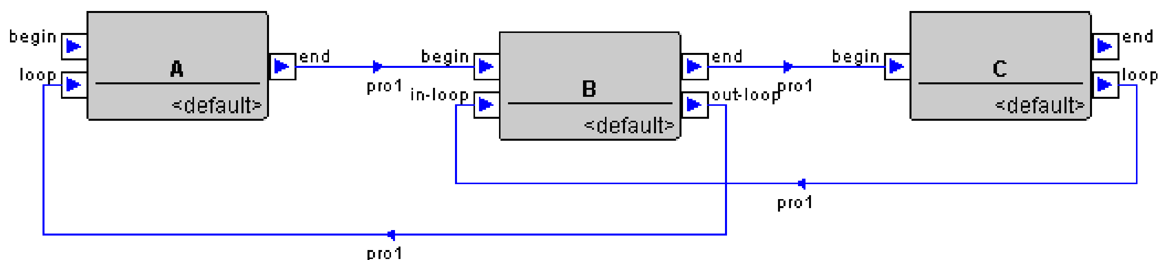
Figura 4.28. Diagrama del Patrón 'Multiple Merge' en APEL



PF9 *Discriminator.* Este patrón no es soportado por APEL, ya que cada vez que un puerto de entrada de una actividad tenga todos sus productos esperados, la actividad esta lista para ser ejecutada. De esta forma, cada vez que se active la actividad por uno de los caminos, ésta podrá ser ejecutada, en contraposición del concepto del patrón que dice que se ejecuta una sola vez.

PF10 *Arbitrary Cycles.* Para modelar el patrón *Arbitrary Cycles* en APEL considérese una actividad A que, una vez ejecutada, pasa el control a una actividad B. La actividad B tiene dos puertos de salida: uno dirigido hacia la actividad C y otro hacia la actividad A, es decir, su actividad predecesora. De forma análoga, la actividad C tiene un puerto de salida que se dirige hacia su predecesora (la actividad B). En este modelo, el control puede ser pasado de la actividad A hacia la actividad B, y en este punto el control podría volver hacia A, realizando un ciclo. La actividad B también podría tomar el camino hacia la actividad C, la actividad C podría volver hacia B y de esta forma, también realizar un ciclo. Note que el control puede ser dirigido desde la actividad C hasta la actividad A utilizando los dos ciclos definidos en el proceso. Esto quiere decir que en APEL no se utilizan ciclos estructurados.

Figura 4.29. Diagrama del Patrón 'Arbitrary Cycles' en APEL

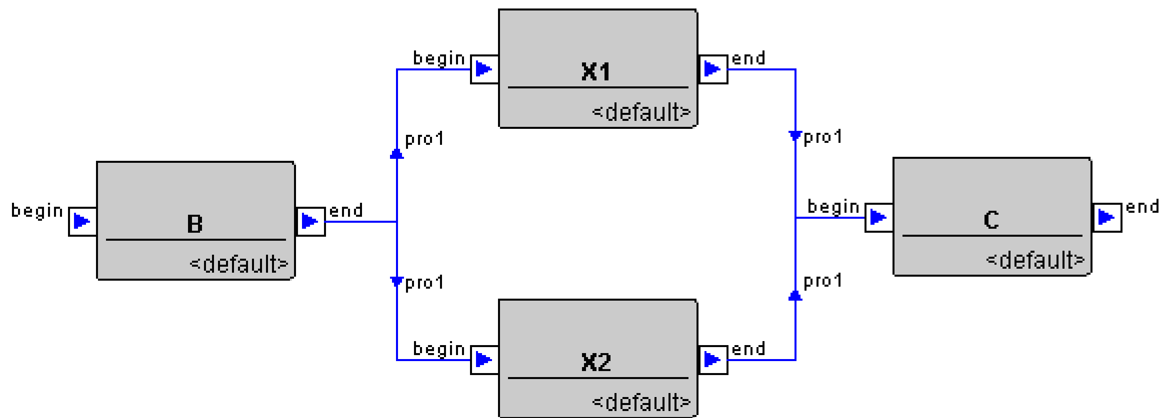


PF11 *Implicit Termination.* La terminación implícita de un proceso no es soportado por APEL. Cuando todas las sub-actividades que contiene una actividad terminan, la actividad por sí misma no lo hace. De esta forma, en APEL la terminación de un actividad debe ser indicada por un tercer elemento (ya sea un usuario o de forma automática a través de un aspecto).

PF12 *MI Without Synchronization.* APEL no soporta la creación de múltiples instancias de una actividad sin sincronización. Siempre la actividad contenedora de las múltiples instancias espera a éstas terminen para continuar con la ejecución del proceso.

PF13-PF15 *MI With Synchronization.* Para soportar el patrón 13, se crea una actividad del mismo tipo por cada vez que se desee ejecutar la actividad.

Figura 4.30. Diagrama del Patrón 13 en APEL. Las actividades X1 y X2 son iguales (instancias).

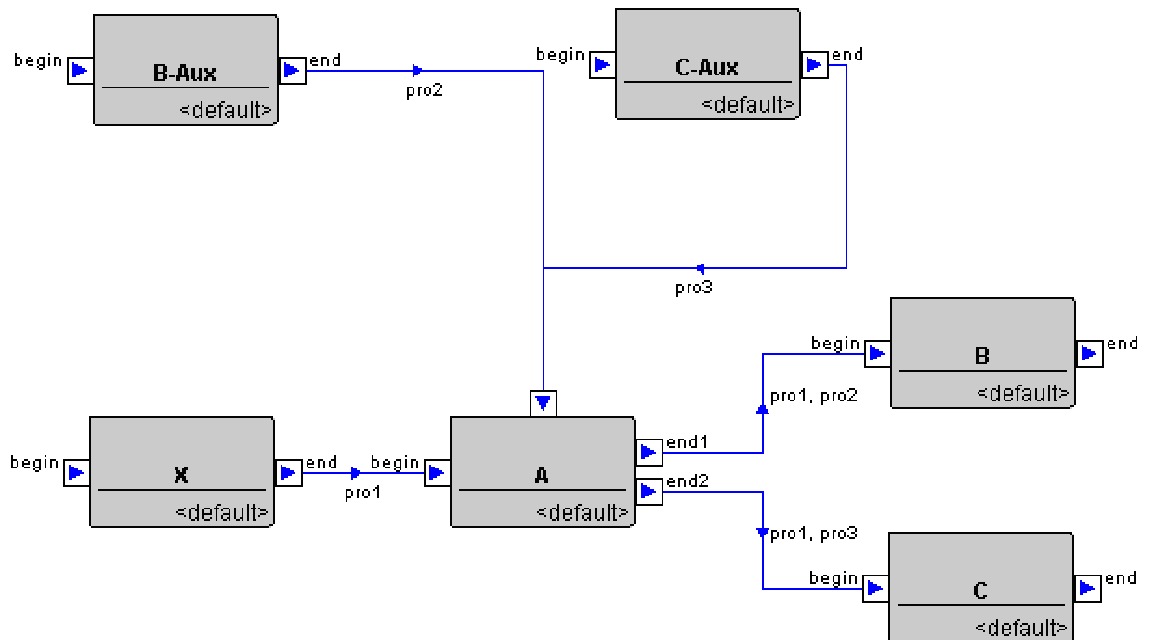


En APEL es permitida la creación de múltiples instancias de una actividad. APEL permite en tiempo de ejecución definir cuantas instancias de la actividad crear siempre y cuando no exceda el número máximo de instancias definidas. El problema para soportar los patrones PF14 y PF15 está en la parte de sincronización de la actividad, debido a que la actividad siguiente a la actividad de múltiples instancias se ejecuta cada vez que termina una de las instancias, y no cuando terminan todas las instancias creadas, lo cual es la idea de estos patrones.

PF16 *Deferred Choice* La primera opción para soportar el patrón está basada en el concepto de puerto manual en APEL. Este concepto permite que un usuario pueda tomar la decisión de por cuál de los puertos continuará la ejecución del proceso. En este caso, la decisión deberá ser tomada por un usuario (alternativa del ambiente) del proceso.

Otra alternativa propuesta para soportar el patrón es que la decisión no sea tomada por un usuario. En este caso, se podría colocar un puerto de entrada asíncrono en la actividad que debe tomar la decisión y por lo menos un producto esperado en cada puerto de salida deberá llegar por este puerto de entrada asíncrono. De esta forma, cuando el primer puerto de salida este listo para continuar, es decir, tenga todos sus productos (los que llegaron por el puerto de entrada y los que llegaron por el puerto asíncrono de este puerto) será el escogido.

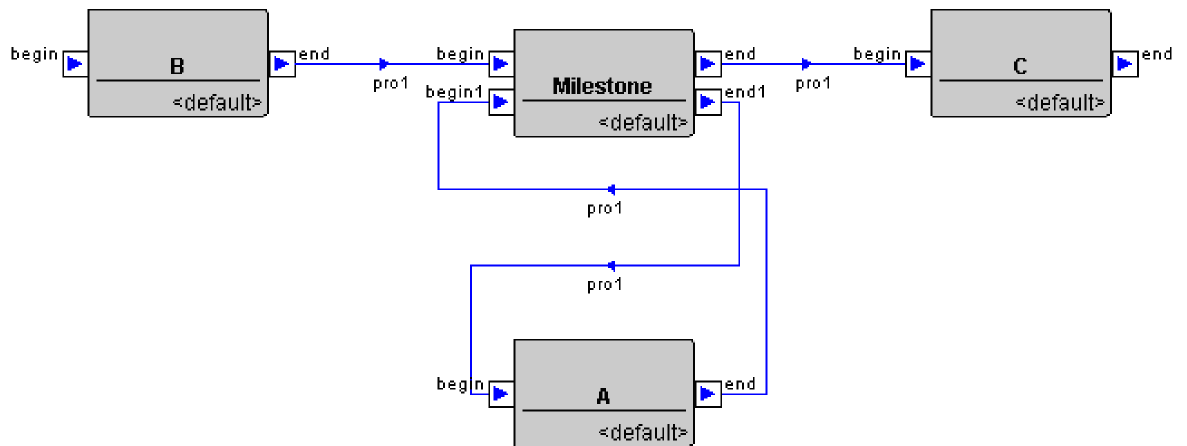
Figura 4.31. Diagrama del Patrón 'Deferred Choice' en APEL, usando puertos asíncronos



PF17 ***Interleaved Parallel Routing.*** APEL no soporta este patrón, ya que no es posible activar varios puertos de salida en una actividad que se encuentre en ejecución. Esto es, una vez la actividad escogió un puerto de salida, se termina su ejecución. Por otro lado, si se optara por utilizar un ciclo de tal manera que *Start Interleaving* tomara de nuevo el control, no habría forma de indicarle a ésta que no puede volver a utilizar un camino ya tomado.

PF18 ***Milestone.*** Considerando tres actividades A, B y C, donde la actividad B será modelada como el punto que hay que alcanzar para que la actividad A se habilite (milestone). La actividad B tiene dos puertos de salida uno hacia A y otro hacia C, donde sólo uno de estos puertos puede ser escogido como resultado de la ejecución de B. La actividad A, por su parte, sólo se puede ejecutar si ya se ejecutó B, y no se ha ejecutado C. Dicha actividad (A) tiene un puerto de salida que es dirigido hacia la actividad B (milestone). De esta forma, una vez ha sido ejecutado A, el control vuelve a la actividad B, la cual nuevamente tiene la opción de continuar por A o por C. Si la actividad B continúa su ejecución hacia la actividad C, no será posible ejecutar nuevamente la actividad A.

Figura 4.32. Diagrama del Patrón 'Milestone' en APEL



PF19 ***Cancel Activity & PF20 Cancel Case.*** En APEL, las actividades pueden pasar del estado de ejecución al estado abortado. De esta forma, se puede cancelar cualquier actividad en un momento dado en el proceso, soportando así el patrón PF19. Por otro lado, debido a que una actividad puede contener varias subactividades, y de forma recursiva dichas subactividades contener a otras, una vez se cancela una actividad se puede cancelar la instancia total de un proceso. Esto último explica cómo APEL soporta el patrón PF20.

5. MOTIVACIÓN

Los capítulos anteriores (3 y 4) sirven para tener una idea de las tendencias que actualmente existen en cuanto a modelos de Workflows. Sobre éstos, se crean motores, editores y todo tipo de herramientas que permiten tener un mejor control sobre la administración de los Procesos de Negocio.

Sin embargo, la Automatización de Procesos aún no cuenta con la atención necesaria. Es más, para algunos es un área que aún está en sus comienzos, en comparación con otras como las Bases de Datos relacionales [BAY2004]. Lo deseable en un futuro (ojalá no muy lejano) es que en una empresa se hable de automatizar un proceso como hoy en día se habla de crear una base de datos. Que sea algo natural, aceptado y completamente necesario.

Sin embargo, la falta de madurez mencionada en el párrafo anterior se debe principalmente a que no hay consenso en los principales interesados y desarrolladores de esta tecnología. Cada quien va por su camino, investigando y desarrollando sobre la línea que creen es la mejor opción. Es por eso que existen soluciones enfocadas a orquestar servicios, otras a integrar aplicaciones y otras a definir workflows. Pero ninguna se ha preocupado por integrar, en una sola propuesta, estos tres campos de acción.

Las principales falencias encontradas en los modelos de hoy en día son:

- **Lenguajes Incompletos.** Estudios realizados ([WVA2003], [VDA2003], [PWO2002] y [PWO2004]) a varios modelos existentes basados en los Patrones de Control de Flujo mencionados en el capítulo 4 y definidos en [VDA2003], muestran lo incapaces que son los lenguajes basados en estos modelos para expresar (de manera directa) ciertas situaciones particulares. Esto da la noción de que ninguno de los modelos existentes posee el poder expresivo suficiente para expresar completamente cualquier escenario deseado.

- **Poco Extensibles.** Como se pudo observar en el análisis comparativo del capítulo 3, sólo jBPM presenta un modelo de extensibilidad interesante, pero más que todo, que realmente permite extender el modelo según lo necesite el desarrollador. Los demás Modelos poseen puntos de extensión (Datos definidos en el usuario –XPDL–, aspectos –APEL–) pero que o son muy complicados de usar o no son lo suficientemente flexibles.

- **Dependencia Tecnológica.** Esto se evidencia en la dependencia que existe con Java (en el caso de jBPM) o los Web Services (BPEL). El caso de jBPM es inclusive peor, puesto que no existe una clara separación entre los conceptos de su modelo y la implementación del mismo. Es fundamental crear una especificación abierta que permita su implementación de la manera que cualquier desarrollador lo desee.

Si los conceptos mencionados anteriormente son analizados detenidamente, se puede observar que, en el fondo, la solución para las dos primeras falencias es la misma, y que inclusive, está más relacionada con la segunda.

La primera falencia trata sobre el pobre o insuficiente poder expresivo que tienen los modelos de hoy. Cómo solucionarlo? Sería muy complicado hacer un modelo que incluya todos los conceptos necesarios para modelar cualquier escenario deseado. La experiencia en el desarrollo de software (y prácticamente cualquier disciplina) nos ha enseñado que en el momento menos pensado, surge un nuevo concepto, y por ende, la necesidad de expresarlo de la mejor forma. Ni siquiera los ya mencionados patrones de control de flujo deben considerarse como el objetivo final de un modelo, ya que es posible que más adelante surja uno nuevo.

Es por esta razón que es fundamental no definir un super conjunto de conceptos con lo que se busque tapar el sol con un dedo. Lo ideal es desarrollar un modelo que proporcione los mecanismos con los cuales, a partir de un conjunto minimal de elementos, se pueda extender prácticamente a lo que cualquier usuario del mismo desee. Al obtener un modelo extensible, no importa que los conceptos elementales del mismo no soporten un patrón de control de flujo (por ejemplo), ya que sus mecanismos de extensibilidad permitirían lograr dicho objetivo.

6. EL MODELO CUMBIA-XPM

Cumbia Extensible Process Modeling (C-XPM de ahora en adelante) es un modelo de Workflow para la definición de procesos ejecutables [PED2005]. Dicho modelo está basado en un conjunto minimal de conceptos y mecanismos de extensibilidad, con el fin de permitir la modelación de cualquier dominio potencial. C-XPM es un modelo altamente extensible (tal como su nombre lo indica), lo cual establece una clara diferencia con la gran mayoría de los modelos actuales.

Los principales elementos de C-XPM son: Proceso (*Process*), Actividad (*Activity*), MultiActividad (*MultiActivity*), Puerto (*Port*), *Dataflow, Workspace,* Dato (*Data*) y Autómata (*StateMachine*). Un proceso contiene otros procesos o actividades, las cuales se interconectan por medio de Dataflows. Un Puerto es un punto de entrada o salida para acceder un Proceso, Actividad o MultiActividad, por lo que, a su vez, los puertos son puntos de interconexión entre Dataflows. Una Actividad contiene un Workspace, el cual es el encargado de ejecutar la tarea representada por la Actividad. Una MultiActividad es un conjunto de Actividades idénticas que se ejecutan de manera concurrente. La información fluye a través del Proceso como Data.

Todos los elementos mencionados en el párrafo anterior, tienen un Autómata (*StateMachine*) que representa la semántica de cada uno.

La extensibilidad del modelo está basada en dos aspectos: Definición de Semántica y Reflexión. Como se mencionó anteriormente, un Autómata representa la semántica de un elemento. Al separar estos dos conceptos (la semántica del elemento mismo), C-XPM presenta el comportamiento de un elemento de manera más entendible, permitiendo así extender un elemento fácilmente: modificando su Autómata añadiendo Acciones (*Actions*). Además, es posible cambiar el comportamiento de un elemento (creando así un nuevo elemento) al añadir, modificar o borrar estados de su Autómata.

La Reflexión se consigue al establecer una clara separación entre la Definición del Proceso (Modelo Estático) y sus instancias (Modelo Dinámico). Por lo tanto, cada uno de los seis elementos descritos anteriormente (Proceso, Actividad, MultiActividad, Puerto, Dataflow y Workspace) tiene una instancia correspondiente (Proceso-i, Actividad-i, MultiActividad-i, Puerto-i, Dataflow-i y Workspace-i). Haciendo esto, un usuario puede extender el modelo no sólo en tiempo de diseño, sino también en tiempo de ejecución.

Todas las ideas mencionadas anteriormente son presentadas en el resto del capítulo de una manera más detallada.

6.1 ELEMENTOS DEL MODELO ESTÁTICO

Como se mencionó en la introducción del capítulo, los elementos del Modelo Estático son aquellos relacionados con la definición del proceso. Son elementos que ayudan a representar un proceso en tiempo de diseño.

Pero antes de describir dichos elementos, es necesario presentar algunos conceptos que faciliten el entendimiento de próximas definiciones.

- **Literales.** Se presentan dos conceptos utilizados a lo largo del capítulo.
  - **<Nombre>:** es una secuencia no vacía de caracteres alfanuméricos, que comienza con una letra.
  - **Elemento:** Nombre usado para hacer referencia a cualquiera de los siguientes: Puerto, Dataflow, Workspace, Actividad, MultiActividad y Proceso.

6.1.1 Variable. Una Variable es un <Nombre> usado para identificar información, específicamente, un Dato.

6.1.2 Actividad. Una actividad representa un paso atómico en la definición de un Proceso. Es una de las tareas que pueden conformar un proceso.

Una Actividad está compuesta por dos conjuntos de Puertos y un Workspace. El primer conjunto de Puertos (llamados Puertos de Entrada) proporcionan los Datos a usar por la Actividad. Estos datos son pasados al Workspace para su ejecución. Después de la ejecución, los datos resultantes son entregados al segundo conjunto de Puertos (llamados Puertos de Salida). Como todos los Elementos, posee un Autómata el cual especifica su semántica.

La representación gráfica usada en el resto del documento es la siguiente:

Figura 6.1. Representación gráfica de una Actividad.

6.1.3 Recurso. Un Recurso es un nombre que representa un recurso capaz de ejecutar un Workspace.

6.1.4 Workspace. Es un elemento que permite la comunicación entre la Actividad que lo contiene y el mundo exterior del Proceso. Es el corazón de la Actividad, ya que esta es la que en sí ejecuta la tarea que la actividad representa. Puede producir, modificar o consumir conjuntos de Datos.

6.1.5 Puerto. Un Puerto es un elemento que define el punto de entrada o salida de un Proceso, Actividad o MultiActividad. Posee un grupo de Datos esperados (representado por un grupo de Variables), los cuales son proporcionados por un conjunto de Dataflows (Dataflows de Entrada). Una vez el puerto recibe todos los datos esperados (se 'llena'), los entrega a otro conjunto de Dataflows (Dataflows de Salida). Además, al igual que la Actividad, posee un autómata que especifica su semántica.

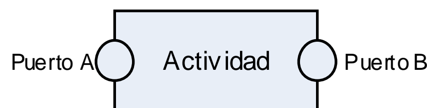
Existe una variable que todo puerto espera: el token. Esta variable está oculta al usuario, y debe ser manipulada internamente por el sistema. Un Puerto que sólo espera el token se le llama Puerto Delgado.

A pesar de que existe una definición general de Puerto, éste se puede comportar de cuatro maneras distintas:

- **Puerto de Entrada de un Proceso o MultiActividad:** Representa un punto de entrada a la ejecución de un Proceso o MultiActivity, usado para comunicar los elementos externos con los elementos internos.
- **Puerto de Salida de un Proceso o MultiActividad:** Representa un punto de salida de la ejecución de un Proceso o MultiActivity, usado para comunicar los elementos internos con los elementos externos.
- **Puerto de Entrada de una Actividad:** Representa un punto de entrada a una Actividad. No tiene Dataflows de salida, porque la Actividad misma es la responsable de tomar la información del Puerto, una vez éste se llena.
- **Puerto de Salida de una Actividad:** Representa un punto de salida de una Actividad. No tiene Dataflows de entrada, porque la Actividad misma es la responsable de darle la información al Puerto.

La representación gráfica usada en el resto del documento es la siguiente:

Figura 6.2. Representación gráfica de un Puerto.



6.1.6 Map. Es una pareja de Variables que representa una conversión o transformación entre ellas. La primera variable representa la variable a convertir y la segunda es el resultado de dicha conversión.

6.1.7 Dataflow. Es un elemento que comunica Actividades, MultiActividades y Procesos al interconectar sus Puertos de Salida con sus Puertos de Entrada correspondientes. En otras palabras, convierte la información proporcionada por un puerto (Puerto de Entrada del dataflow), en información esperada por otro puerto (Puerto de Salida del dataflow), de acuerdo a un conjunto establecido de Maps. Dicha pareja de puertos no pueden ser iguales. Además, al igual que el Puerto, posee un autómata que especifica su semántica.

Al igual que un Puerto, un Dataflow siempre espera la Variable token, en este caso, puede convertirla ('Mapearla'). Un Dataflow que solo es capaz de convertir tokens se le llama Dataflow Delgado. La variable token se usa para asegurar el flujo de control a través de Dataflows Delgados.

La conversión de un token produce el mismo token.

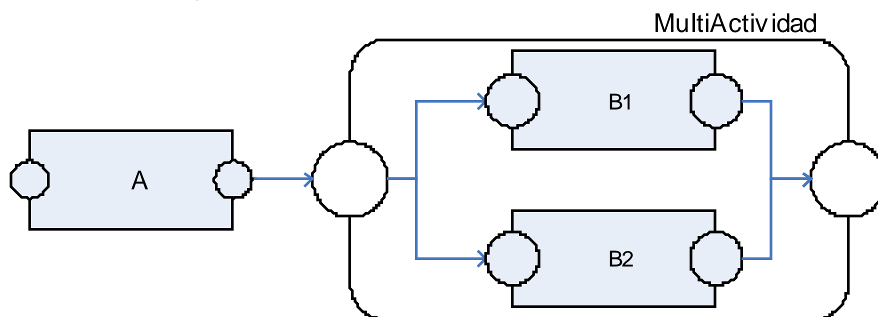
La representación gráfica de un dataflow, usada en el resto del documento es la siguiente:

Figura 6.3. Representación gráfica de un Dataflow.

6.1.8 MultiActividad. Representa una actividad que será ejecutada un determinado número de veces, de manera concurrente. Al igual que una actividad, posee un conjunto de Puertos de Entrada y otro de Puertos de Salida, los cuales le permiten comunicarse actividades, procesos u otras multiActividades. También posee un workspace, el cual le sirve de modelo para las instancias que va a crear. Además, como la mayoría de los elementos mencionados, posee un autómata que especifica su semántica.

La representación gráfica de un multiActividad, usada en el resto del documento es la siguiente:

Figura 6.4. Representación gráfica de una MultiActividad.

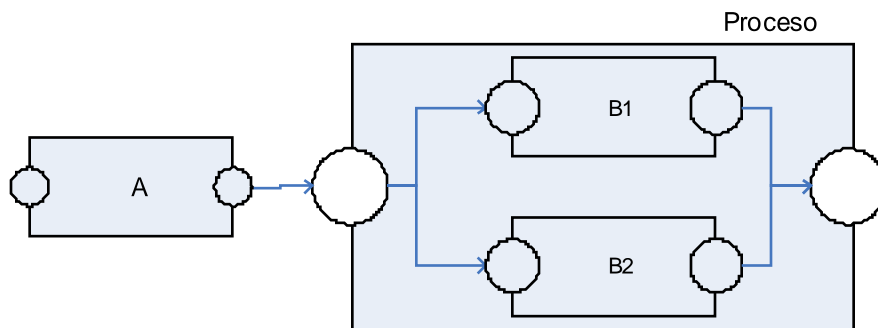


6.1.9 Proceso. Es un elemento que intenta realizar un objetivo dado, agrupando Actividades, MultiActividades u otros Procesos. Adicionalmente, contiene los dataflows que interconectan dichos elementos.

El Proceso que contiene todos los elementos de una definición dada, se le llama Proceso-Raíz (Process-Root). Al igual que una actividad o multiActividad, posee un conjunto de Puertos de Entrada y otro de Puertos de Salida, los cuales le permiten comunicarse actividades, multiActividades u otros procesos. Además, como la mayoría de los elementos mencionados, posee un autómata que especifica su semántica.

La representación gráfica de un Proceso, usada en el resto del documento es la siguiente:

Figura 6.5. Representación gráfica de un Proceso.



6.1.10 Autómata. Conjunto de etapas o Estados que definen el comportamiento o semántica de un elemento dado. Un elemento puede ir de una etapa a otra a través de una Transición. Una transición es generada por un evento, a la vez que genera otro evento.

6.1.11 Estado. Es una etapa en el Autómata de un elemento. Tiene un conjunto de enlaces (Transiciones) que representan las posibles transiciones a otros estados. Cada una de estas transiciones es generada por un evento distinto.

6.1.12 Transición. Representa un cambio de Estado de un elemento, generado por un Evento. A su vez, también genera un Evento y puede generar una Acción.

6.1.13 Evento. Notificación recibida por un Elemento. Puede ser generado por una transición (indicando que hubo un cambio de estado) o por una operación del elemento. Cada evento va identificado por el elemento que lo generó.

6.1.14 ColaEventos (*eventQueue*). Cola que guarda los Eventos que un Elemento no puede procesar en el momento en que los recibe. Estructuralmente, es un conjunto de Eventos que se comporta como una cola.

6.1.15 Acción. Acción que un Elemento ejecuta, por orden de su Autómata. Las acciones están relacionadas a las Transiciones de los Autómatas. Una transición puede ejecutar una o varias acciones.

Toda acción posee una clase que la implementa. Esta clase debe tener un método llamado 'Execute', el cual recibe un contexto como parámetro de entrada. Dicho Contexto no es más que una referencia a la Memoria del Elemento que invocó la Acción.

6.2 ELEMENTOS DEL MODELO DINÁMICO

Este grupo de elementos ayuda a representar una instancia de la definición de un Proceso dado. Por lo tanto, cada uno de estos elementos se encuentra relacionado con aquellos mencionados en la sección anterior, es decir, poseen una referencia del elemento al que instancian.

Una característica especial de algunos de estos elementos, son las *operaciones*. Éstas son funciones o métodos particulares de un elemento.

Así como las Transiciones, las operaciones pueden generar Eventos. La diferencia radica en que dichos Eventos sólo son escuchados o recibidos por el elemento cuya operación los genera.

6.2.1 Valor (*Value*). Es una instancia de un tipo simple de dato (enteros, cadenas de caracteres, reales, fecha, hora, Propiedad) o una Secuencia de los mismos. Una Propiedad es una pareja Nombre-Valor. Una Secuencia es un grupo ordenado de Valores de tipos simples.

Utilizando gramática para la definición de lenguajes, podemos observar la estructura de un Valor:

- <Valor> ::= ValorSimple | Secuencia

- <ValorSimple> ::= int | cadena | real | fecha | Propiedad
- <Propiedad> ::= [nombre, valor]
- <Secuencia> ::= <Vacío> | <Valor><Secuencia>
- <Vacío> ::= ∅

Tabla 6.1. Elementos de la estructura de un Valor

Nombre	Tipo	Descripción
int	Entero	Usado para representar números enteros
cadena	Cadena de Caracteres	Usado para representar cadenas alfanuméricas
real	Real	Usado para representar números reales
fecha	Fecha	Usado para expresar fecha y hora
nombre	<Nombre>	Nombre que identifica una Propiedad
value	<Value>	Valor contenido por una Propiedad

6.2.2 Dato (*Data*). Es un pedazo de información, representado por medio de una pareja Nombre-Valor. Esta es la información que fluye entre Actividades, MultiActividades y Procesos, a través de sus Puertos y Dataflows. Es importante anotar que no existen tipos de Datos, permitiendo que los Puertos, Dataflows y Workspaces sean más flexibles.

6.2.3 Recurso-i. Es una instancia de <Recurso>.

6.2.4 Memoria (*Memory*). Conjunto de Datos que un elemento tiene en un momento dado. Estructuralmente, es (como se acaba de mencionar) un conjunto de Datos.

Tiene dos alcances: *Local* y *Global*. El alcance *Local* representa Datos contenidos por el un Elemento o por alguno de los éste conozca estructuralmente; el alcance *Global* representa Datos contenidos por el Proceso-Raíz o por alguno de los elementos que éste conozca estructuralmente.

El esquema de navegación para obtener un dato es mediante la ruta absoluta (alcance Global) o relativa (alcance Local) del Elemento que posee dicho Dato. Por ejemplo, la siguiente ruta obtiene el Dato 'ejemplo' del Puerto-i 'pe' de la Actividad-i 'A1': *A1.pe.ejemplo*; si suponemos que A1 está en el Proceso-i P1, la siguiente ruta obtiene el Dato 'ejemplo2' de P1: *Proceso-Raiz.P1.ejemplo2*.

6.2.5 Puerto-i. Instancia de un Puerto. Su estructura es similar a la de un Puerto, sólo que en vez de tener Dataflows, posee Dataflows-i. Adicionalmente, tiene: un estado que representa la etapa en la que se encuentra según su autómata; una cola de los eventos recibidos sin procesar, y una memoria.

La siguiente tabla muestra las operaciones de un Puerto-i:

Tabla 6.2. Operaciones de un Puerto-i

Firma	Tipo	Eventos Generados	Descripción
checkODFState(state:cadena):boolean	**Pública**	• allDF Initialized	Verifica si todos los Dataflows de Salida se encuentran en el estado dado. De ser así, lanza el Evento *allDF Initialized*.
checkDFState(df:Dataflow, state:Nombre):boolean	**Privada**		Verifica que el Dataflow dado se encuentre en el estado correspondiente.
deliverData( )	**Pública**	• dataDelivered • allDataDelivered	Coloca un Dato en los Dataflows correspondientes y genera el Evento *dataDelivered*. Si el Puerto-i está vacío, genera el Evento *allDataDelivered.*
receiveData(data:Dato)	**Pública**	• dataReceived • allDataReceived	Coloca un Dato en el grupo de Datos esperados, y genera el Evento *dataReceived*. Si ese era el ultimo Dato esperado por el Puerto-i, genera el Evento *allDataReceived.*
getLocalData(variable:Variable):Dato	**Pública**		Toma un Dato que concuerde con la Variable dada, de la Memoria Local.
getGlobalData(variable:Variable):Dato	**Pública**		Toma un Dato que concuerde con la Variable dada, de la Memoria Global.
setLocalData(data:Dato)	**Pública**		Coloca un Dato en la Memoria Local
setGlobalData(data:Dato)	**Pública**		Coloca un Dato en la Memoria Global
changeState(estado:Nombre)	**Privada**	arrivedTo<Estado>	Establece el Estado del Puerto con el nombre del Estado dado. Después de cambiar el Estado, genera el Evento *'arrivedTo<estado>'*

6.2.6 Dataflow-i. Instancia de un <Dataflow>. Su estructura es similar a la de un Dataflow, sólo que en vez de tener Puertos, posee Puertos-i. Adicionalmente, posee un conjunto de datos a traducir (datos recibidos sin traducir) y un conjunto de datos traducidos (a entregar).

Tal como el Puerto-i, un Dataflow-i tiene: un estado que representa la etapa en la que se encuentra según su autómata; una cola de los eventos recibidos sin procesar, y una memoria.

La siguiente tabla muestra las operaciones de un Dataflow-i:

Tabla 6.3. Operaciones de un Dataflow-i

Firma	Tipo	Eventos Generados	Descripción
deliverData( )	**Pública**	• dataDelivered • allDataDelivered	Toma un dato de 'datosTraducidos', lo coloca en el Puerto de Salida correspondiente (sólo si el Puerto espera dicho Dato) y genera el Evento *dataDelivered*. Si 'datosTraducidos' está vacío, genera el Evento *allDataDelivered.*
receiveData(data:Dato)	**Pública**	• dataReceived • allDataReceived	Coloca un Dato en 'datosTraducir', y genera el Evento *dataReceived*.
mapData( )	**Pública**	• dataMapped • allDataMapped	Toma un Dato de 'datosTraducir', lo traduce, lo coloca en 'datosTraducidos', y genera el Evento *dataMapped*. Si ese era el ultimo Dato a traducer, genera el Evento *allDataMapped*.
checkInputPort(estado:Nombre)	**Pública**	• inputPortWaiting	Verifica que el Puerto de Entrada esté en el Estado dado (en este caso, 'Waiting Notifications'). De ser así, genera el Evento *inputPortWaiting*.
getLocalData(variable:Variable): Data	**Pública**		Toma un Dato que concuerde con la Variable dada, de la Memoria Local.
getGlobalData(variable:Variable) :Data	**Pública**		Toma un Dato que concuerde con la Variable dada, de la Memoria Global.
setLocalData(data:Dato)	**Pública**		Coloca un Dato en la Memoria Local
setGlobalData(data:Dato)	**Pública**		Coloca un Dato en la Memoria Global
changeState(estado:Nombre)	**Privada**	arrivedTo<Estado>	Establece el Estado del Dataflow con el nombre del Estado dado. Después de cambiar el Estado, genera el Evento *'arrivedTo<estado>'*

6.2.7 Workspace-i. Instancia de un <Workspace>. Posee un conjunto de datos por procesar (parámetros de entrada) y otro de datos procesados (datos de salida). Adicionalmente, posee un estado que representa la etapa en la que se encuentra según su autómata, y una cola de los eventos recibidos sin procesar. A diferencia del Dataflow-i o del Puerto-i, no posee memoria.

A continuación se presentan las operaciones de un Workspace-i:

Tabla 6.4. Operaciones de un Workspace-i

Firma	Tipo	Eventos Generados	Descripción
execute( )	**Pública**	• executionEnded	Esta operación ejecuta las tareas que el Workspace representa. Su implementación es lo que determina un 'Tipo' de Workspace
putData(data:Dato)	**Pública**	• dataReceived	Coloca un Dato en 'datosProcesar', y genera el Evento *dataReceived.*
getData( ):Dato	**Pública**	• dataReturned	Toma un Dato de 'datosProcesados', y genera el Evento *dataReturned.*
checkGeneratedData( )	**Pública**	• allDataReturned	Verifica que 'datosProcesados' esté vacío. De ser así, genera el Evento *allDataReturned.*
changeState(estado:Nombre)	**Privada**	arrivedTo<Estado>	Establece el Estado del Workspace con el nombre del Estado dado. Después de cambiar el Estado, genera el Evento *'arrivedTo<estado>'*

6.2.8 Actividad-i. Instancia de un Actividad. Su estructura es similar a la de una Actividad, sólo que en vez de tener Puertos, posee Puertos-i, y en vez de workspace, posee workspace-i.

Así como el Puerto-i, una Actividad-i tiene: un estado que representa la etapa en la que se encuentra según su autómata; una cola de los eventos recibidos sin procesar, y una memoria.

A continuación se presentan las operaciones de una Actividad-i:

Tabla 6.5. Operaciones de una Actividad-i

Firma	Tipo	Eventos Generados	Descripción
putDataToWS(puerto: Nombre)	**Pública**	• dataPut • entryPortIsEmpty	Toma un dato del Puerto dado, lo coloca en 'datosProcesar' del Workspace, y genera el Evento *dataPut.* Si el Puerto está vacío, genera el Evento *entryPortIsEmpty*.
putDataToExitPort( )	**Pública**	• putDataExitPort • workspaceEmpty	Toma un Dato de 'datosProcesados' del workspace, lo coloca en los Puertos de Salida correspondientes (los que

			esperan dicho Dato), y genera el Evento *putDataExitPort.* Si 'datosProcesados' está vacío, genera el Evento *workspaceEmpty*.
checkEntryPorts(state:Name)	**Pública**	• entryPortFull	Verifica si hay algún Puerto de Entrada en el Estado dado (en este caso, 'Full'). De ser así, genera el Evento *entryPortFull*.
checkExitPorts(state:Name)	**Pública**	• portsFlushed	Verifica que todos los Puertos de Salida estén en el estado dado (en este caso, 'Inactive'). De ser así, genera el Evento *portsFlushed*.
getLocalData(variable:Variable): Data	**Pública**		Toma un Dato que concuerde con la Variable dada, de la Memoria Local.
getGlobalData(variable:Variable) :Data	**Pública**		Toma un Dato que concuerde con la Variable dada, de la Memoria Global.
setLocalData(data:Data)	**Pública**		Coloca un Dato en la Memoria Local
setGlobalData(data:Data)	**Pública**		Coloca un Dato en la Memoria Global
changeState(estado:Nombre)	**Privada**	arrivedTo<Estado>	Establece el Estado de la Actividad con el nombre del Estado dado. Después de cambiar el Estado, genera el Evento *'arrivedTo<estado>'*

6.2.9 MultiActividad-i. Instancia de un <MultiActividad>. Cuando una MultiActividad-i (MA-i, de ahora en adelante) es activada (*activated*), crea un número determinado de Actividades-i, según su atributo 'nInstancias'. Cada una de estas instancias tendrán Workspaces-i idénticos y Puertos-i de Entrada y salida también idénticos. Por lo tanto, el único propósito del Workspace-i definido en MA-i es de servir de modelo para las instancias a crear.

Para garantizar el flujo de control interno, la MA-i conecta cada instancia internamente. Cada instancia, estará conectada con los Puertos-i de Entrada de la MA-i mediante los Dataflows necesarios; de igual forma, dichas instancias estarán conectadas con los Puertos-i de salida de la MA-i. Si no se desean crear instancias (nInstancias igual a cero), se crean Dataflows-i de tal forma que los datos fluyan directamente de los Puertos de Entrada a los Puertos de Salida.

De manera similar a los elementos ya vistos, una MultiActividad-i tiene: un estado que representa la etapa en la que se encuentra según su autómata; una cola de los eventos recibidos sin procesar, y una memoria.

A continuación se presentan las operaciones de una MultiActividad-i:

Tabla 6.6. Operaciones de una MultiActividad-i

Firma	Tipo	Eventos Generados	Descripción
checkNumberOfInstances( )	**Pública**	• instancesValid • zeroInstances	Verfica el atributo 'nInstancias'. Si el valor es un entero mayor que cero, genera el Evento *instancesValid*; si el valor es igual a cero, genera el Evento *zeroInstances*.
createInstance( )	**Pública**	• instanceCreated • allInstancesCreated	Si todas las instancias han sido creadas, genera el Evento a*llInstancesCreated*. De no ser así, crea una instancia con los Puertos-i y Workspace-i definidos en la MA-i, con sus respectivos Dataflows-i, y genera el Evento *instanceCreated*.
createDataflows( )	**Pública**		Crea los Dataflows necesarios para interconectar directamente 'puertosEntrada' con 'puertosSalida'.
checkEntryPorts(state:Name)	**Pública**	• entryPortFull	Verifica si hay algún Puerto-i de 'puertosEntrada' en el Estado dado (en este caso, 'Full'). De ser así, genera el Evento *entryPortFull*.
checkExitPorts(state:Name)	**Pública**	• exitPortsFlushed	Verifica si todos los Puertos-I de 'puertosSalida' se encuentran en el Estado dado (en este caso, 'Inactive'). De ser así, genera el Evento *exitPortsFlushed*.
deleteInstances( )	**Pública**		Borra todas las instancias y dataflows.
getLocalData(variable:Variable):Data	**Pública**		Toma un Dato que concuerde con la Variable dada, de la Memoria Local.

getGlobalData(variable:Variable) :Data	**Pública**		Toma un Dato que concuerde con la Variable dada, de la Memoria Global.
setLocalData(data:Data)	**Pública**		Coloca un Dato en la Memoria Local
setGlobalData(data:Data)	**Pública**		Coloca un Dato en la Memoria Global
changeState(estado:Nombre)	**Privada**	arrivedTo<Estado>	Establece el Estado de la MultiActividad con el nombre del Estado dado. Después de cambiar el Estado, genera el Evento *'arrivedTo<estado>'*

6.2.10 Proceso-i. Instancia de un <Proceso>. Su estructura es similar a la de un Proceso, pero con referencias a instancias y no a los elementos estáticos.

Así como el Puerto-i, un proceso-i tiene: un estado que representa la etapa en la que se encuentra según su autómata; una cola de los eventos recibidos sin procesar, y una memoria.

La siguiente tabla muestra la estructura de un Proceso-i:

A continuación se presentan las operaciones de un Proceso-i:

Tabla 6.7. Operaciones de un Proceso-i

Firma	Tipo	Eventos Generados	Descripción
checkSubElementsState() :boolean	**Pública**	• allSubElementsEnded	Verifica si todos los sub-elementos del Proceso-i se encuentran inicializados. De ser así, genera el Evento *allSubElementsEnded*
checkEntryPorts(state:Name)	**Pública**	• entryPortFull	Verifica si hay algún Puerto-i de 'puertosEntrada' en el Estado dado (en este caso, 'Full'). De ser así, genera el Evento *entryPortFull*.
checkExitPorts(state:Name)	**Pública**	• exitPortsFlushed	Verifica si todos los Puertos-i de 'puertosSalida' se encuentran en el Estado dado (en este caso, 'Inactive'). De ser así, genera el Evento *exitPortsFlushed*.
getLocalData(variable:Variable): Data	**Pública**		Toma un Dato que concuerde con la Variable dada, de la Memoria Local.

getGlobalData(variable:Variable):Data	**Pública**		Toma un Dato que concuerde con la Variable dada, de la Memoria Global.
setLocalData(data:Data)	**Pública**		Coloca un Dato en la Memoria Local
setGlobalData(data:Data)	**Pública**		Coloca un Dato en la Memoria Global
changeState(estado:Nombre)	**Privada**	arrivedTo<Estado>	Establece el Estado del Proceso con el nombre del Estado dado. Después de cambiar el Estado, genera el Evento *'arrivedTo<estado>'*

6.3 SEMÁNTICA DE LOS ELEMENTOS

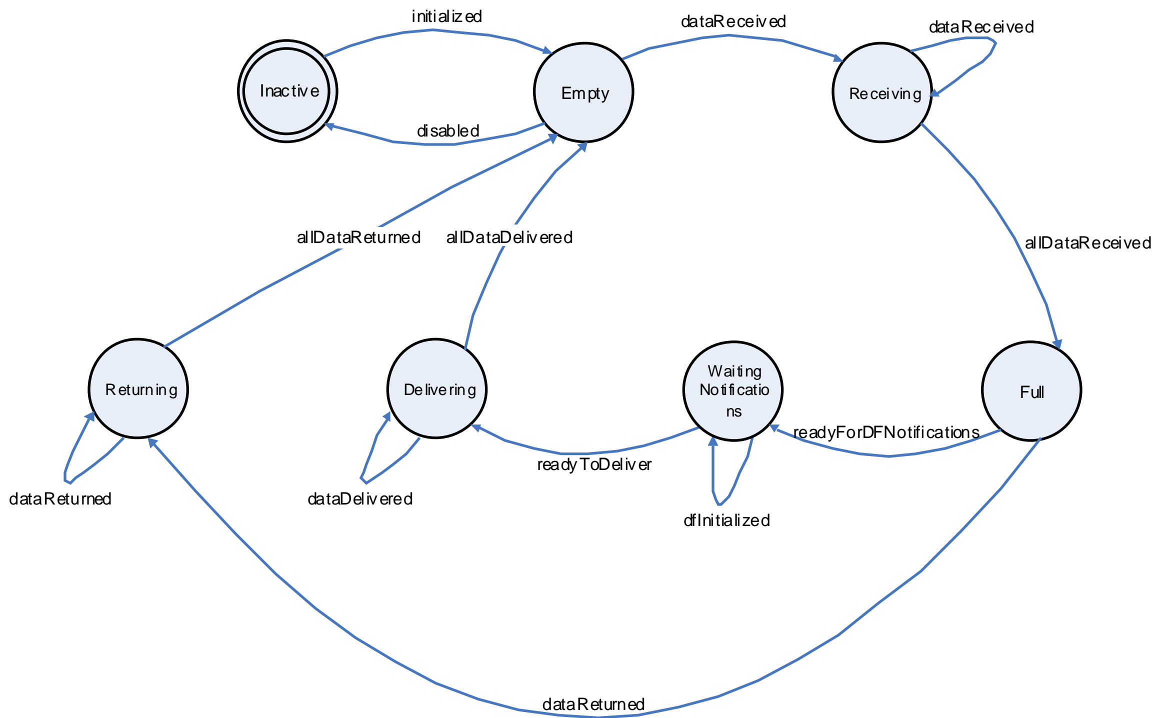
Uno de los aspectos más importantes (si no el más importante) de C-XPM es la manera en que se define la semántica. El comportamiento de cada elemento se define mediante un Autómata, el cual representa su semántica. Esta separación Elemento-Semántica (tal como se mencionó al comienzo del capítulo) permite la extensión en tiempo de diseño del modelo de C-XPM, al poder crear elementos con nuevos Autómatas, o simplemente modificando los ya existentes agregando Acciones a sus respectivos Autómatas.

A continuación se presentan los Autómatas Básicos de los elementos de C-XPM; éstos representan la semántica básica de cada elemento del modelo.

6.3.1 Autómatas de los Elementos Básicos de C-XPM

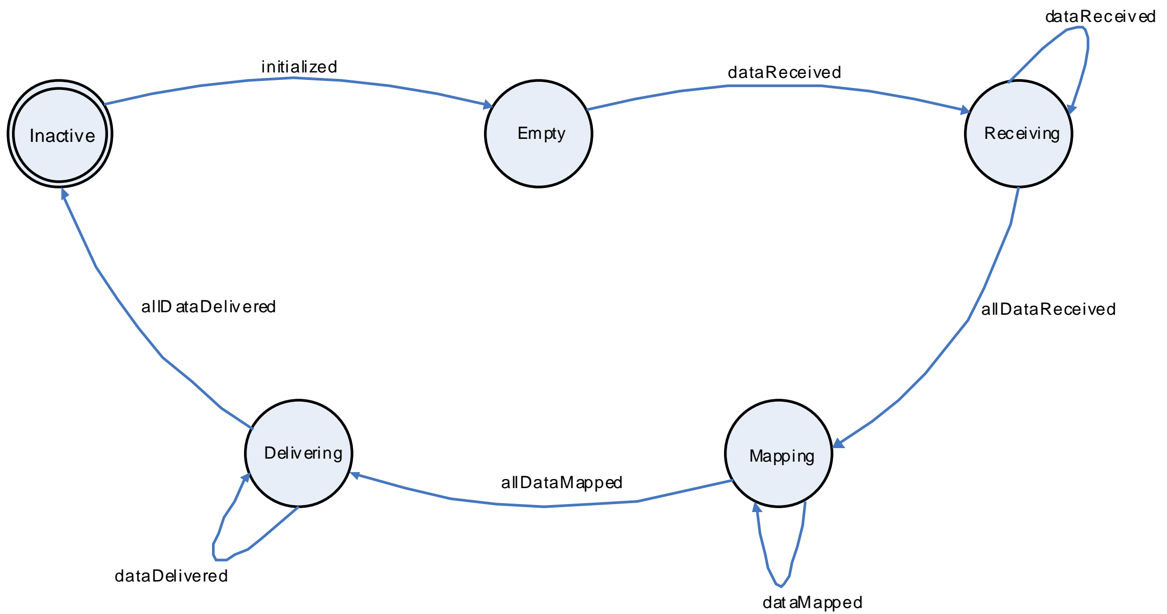
- **Puerto**

Figura 6.6. Autómata Básico de un Puerto



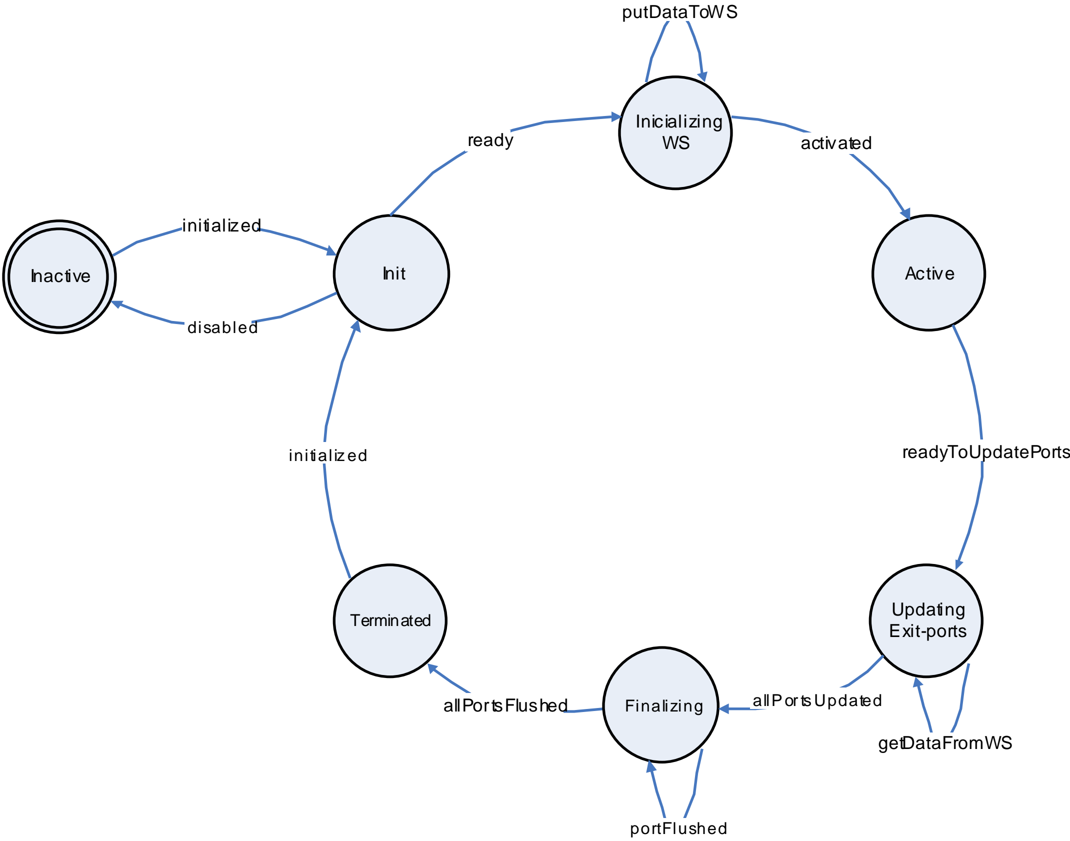
- **Dataflow**

Figura 6.7. Autómata Básico de un Dataflow



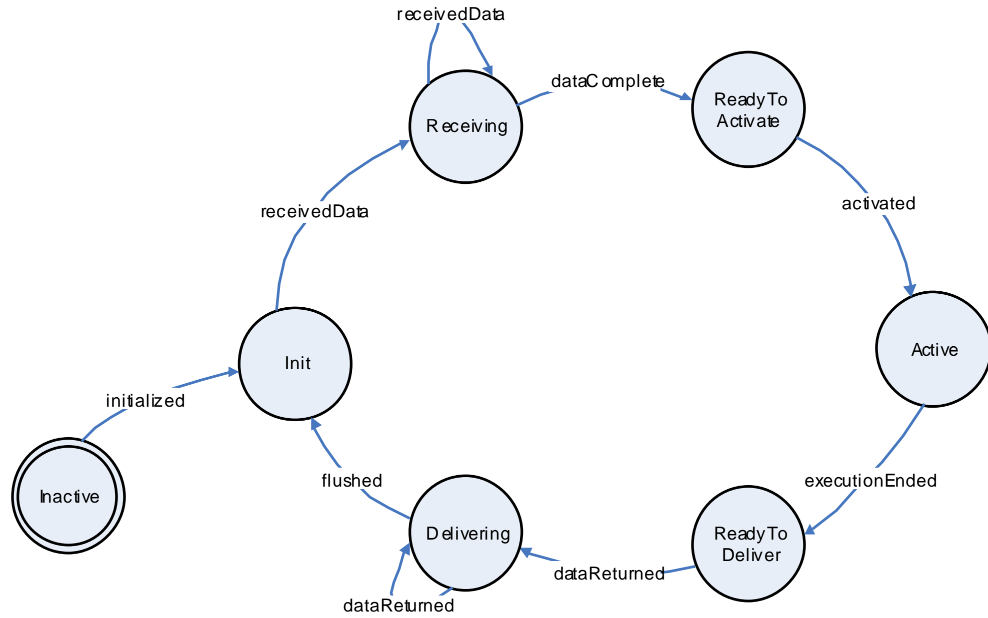
- **Actividad**

Figura 6.8. Autómata Básico de una Actividad



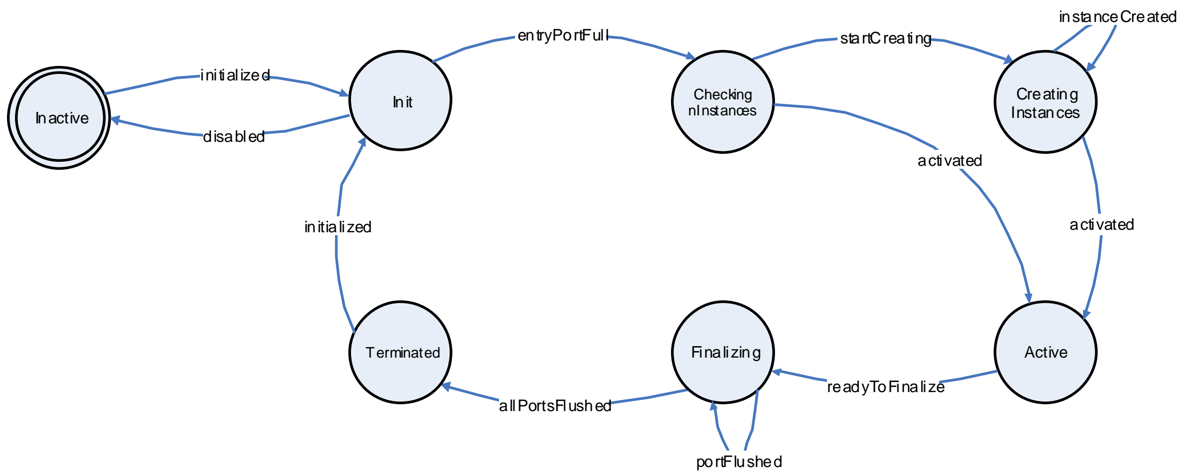
- <u>**Workspace**</u>

Figura 6.9. Autómata Básico de un Workspace



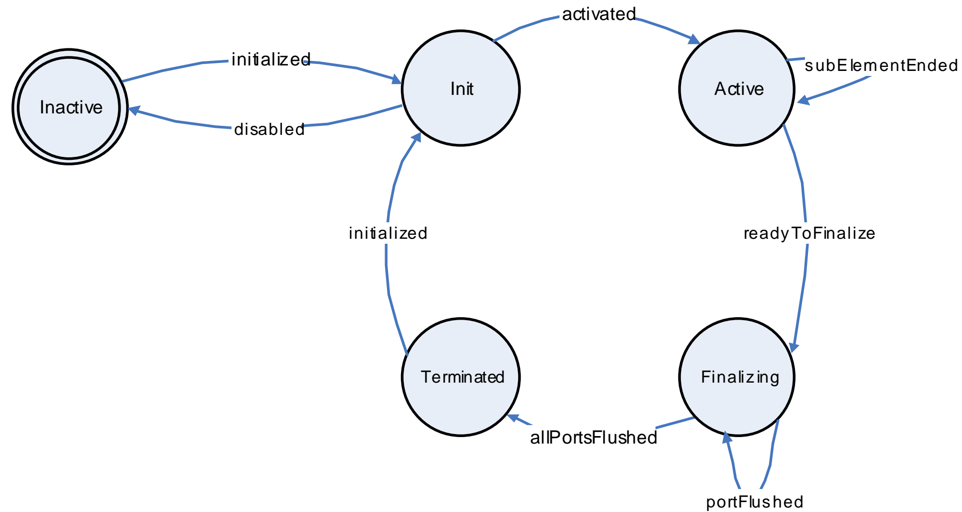
- **MultiActividad**

Figura 6.10. Autómata Básico de una MultiActividad



- **Proceso**

Figura 6.11. Autómata Básico de un Proceso



6.4 AUTÓMATAS SINCRONIZADOS

Como se mencionó en la sección anterior, el Autómata de cada elemento del modelo describe su semántica. Sin embargo, para asegurar que la semántica del modelo como tal sea correcta y coherente, los Autómatas de todos sus elementos deben estar sincronizados. Así, desarrolladores en C-XPM podrán construir elementos con un comportamiento particular. Para lograr esto, un nuevo Autómata para el nuevo elemento debe definirse. Dicho Autómata también debe estar sincronizado con los demás elementos del modelo.

Como se vio en la sección 6.1, un Autómata está compuesto de Estados, Transiciones, Eventos y Acciones. Los Eventos son generados y recibidos por Elementos del modelo. Cuando un Elemento recibe un Evento, ejecuta la Transición (y por ende, generación de Eventos y Acciones) correspondiente, según su Autómata.

Un Elemento puede recibir varios Eventos al mismo tiempo. Por esta razón, cada Elemento posee una cola de Eventos (ColaEvento). Los Eventos que no puedan ser procesados (el Elemento está ejecutando una Transición) son colocados en dicha cola, de tal forma que no se pierdan y el Elemento los revise cuando finalice la Transición.

Un Autómata debe verificar que un Evento sea válido para el Estado actual de un elemento; es decir, que dicho Evento origina una Transición en el Estado en que se encuentra el Elemento. De no ser así, el Evento es ignorado.

Cuando una Transición ha sido activada, el Elemento sigue los siguientes pasos:

1. Cambia su Estado.

2. Ejecuta las Acciones correspondientes (si hay).

3. Genera un Evento notificando que la Transición se ejecutó.

4. El Elemento toma un Evento de la Cola y se lo pasa a su Autómata, el cual verifica que el Evento sea válido. De ser así, volver al paso 1; si no, se repite este paso 4 hasta que no queden Eventos en la cola.

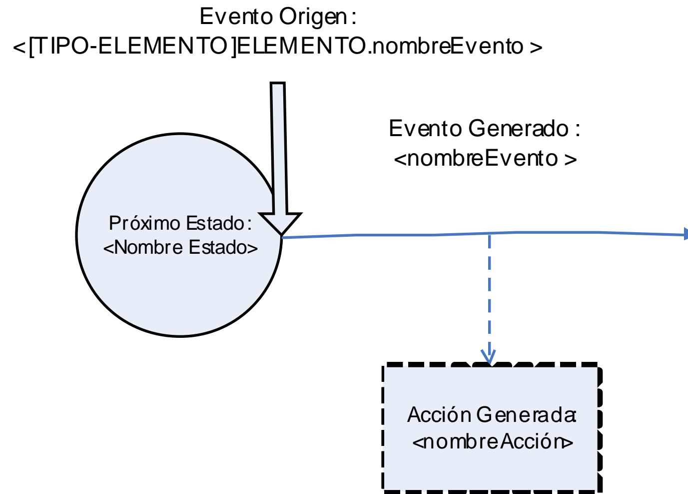
En la sección anterior se mostró, mediante autómatas de estado finito, el comportamiento de cada elemento del modelo C-XPM. A continuación, se presenta la forma en que cada elemento interactúa con los demás. De esta forma, se logrará apreciar mejor la semántica básica de C-XPM.

Primero es necesario definir varios conceptos que permitirán facilitar la comprensión del resto de la sección.

- **Evento Origen:** Evento que genera una Transición.
- **Evento Generado:** Evento generado por una Transición.
- **Acción Generada:** Acción generada por una Transición.
- **Tipo de Elemento:** Es el rol que un elemento juega en otro elemento. Por ejemplo, un Puerto puede ser de Entrada o de salida para una Actividad dada; un Dataflow puede ser de Entrada o Salida para un Puerto dado. Los tipos de elementos definidos son:
    - **PORT-ENTRY:** Puerto de Entrada de una Actividad, MultiActividad o Proceso dado.
    - **PORT-EXIT:** Puerto de Salida de una Actividad, MultiActividad o Proceso dado.
    - **PROC-CONT:** Proceso que contiene un Elemento dado.
    - **ACT-CONT:** Actividad que contiene un Elemento dado.
    - **MACT-CONT:** MultiActividad que contiene un Elemento dado.
    - **PROC-SUB:** Proceso contenido por otro Proceso.
    - **ACT-SUB:** Actividad contenida por un Proceso dado.
    - **MACT-SUB:** MultiActividad contenida por un Proceso dado.
    - **DATAFLOW-INPUT:** Dataflow de Entrada de un Puerto dado.
    - **DATAFLOW-OUTPUT:** Dataflow de Salida de un Puerto dado.
    - **WORKSPACE:** Workspace de una Actividad dada.

Para una mejor comprensión de la sincronización de los elementos, se usará una representación: gráfica. La sintaxis gráfica usada es la siguiente:

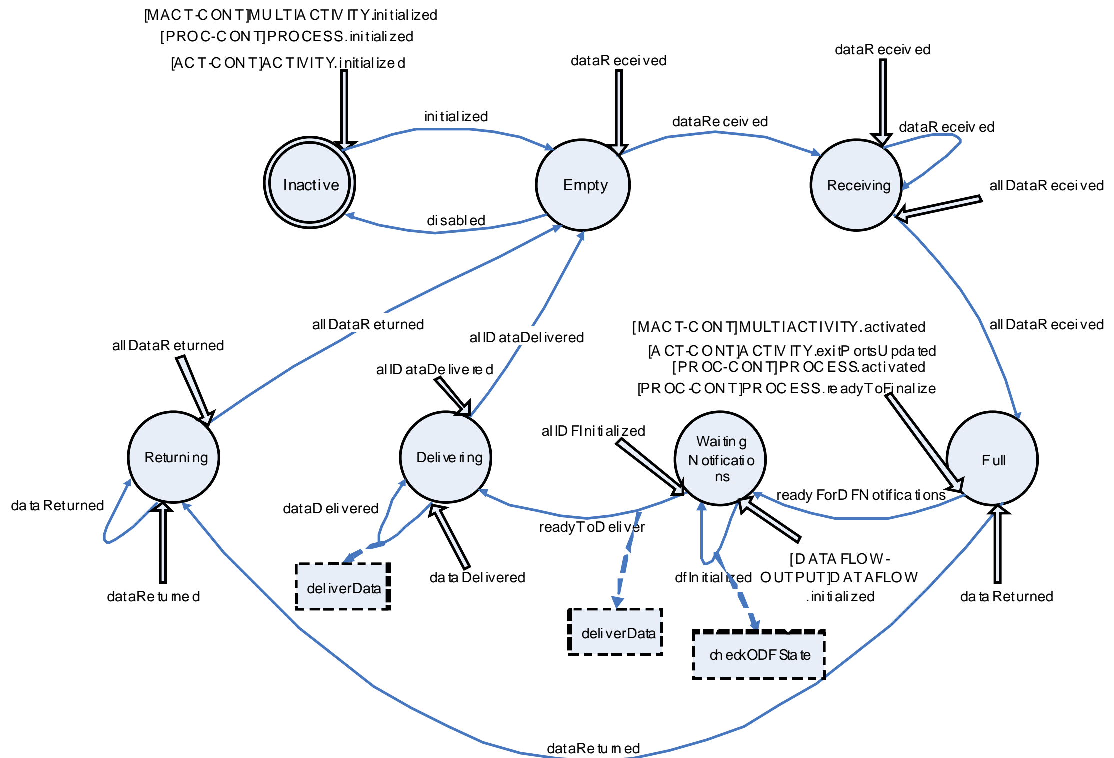
Figura 6.12. Representación Gráfica de Sincronización de Autómatas



6.4.1 Representación Gráfica.

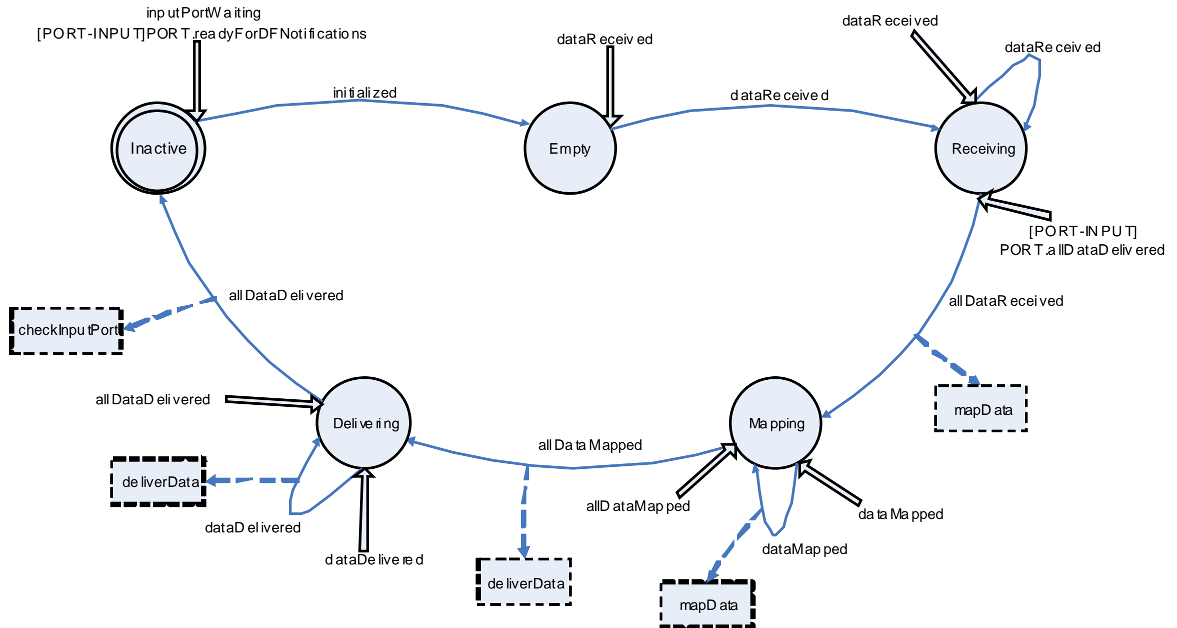
- **Puerto**

Figura 6.13. Autómata Sincronizado de un Puerto



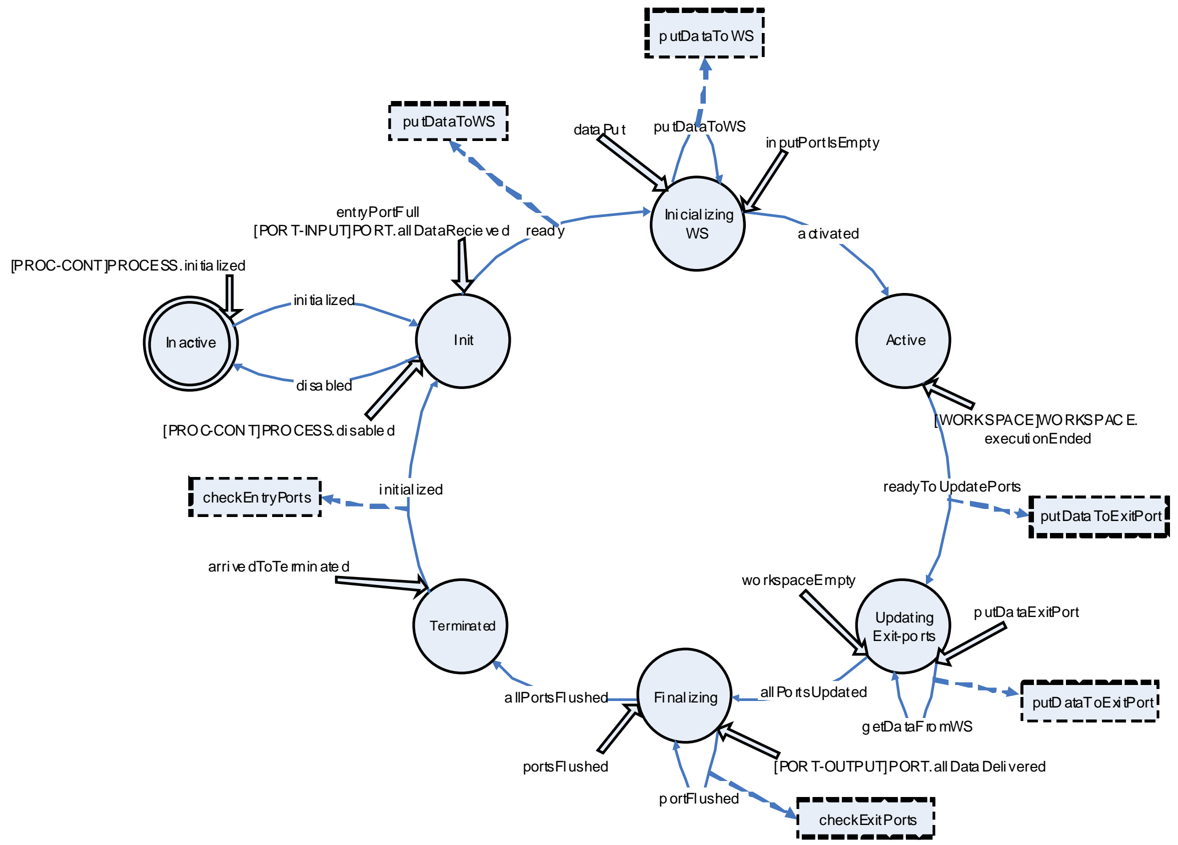
- **<u>Dataflow</u>**

Figura 6.14. Autómata Sincronizado de un Dataflow



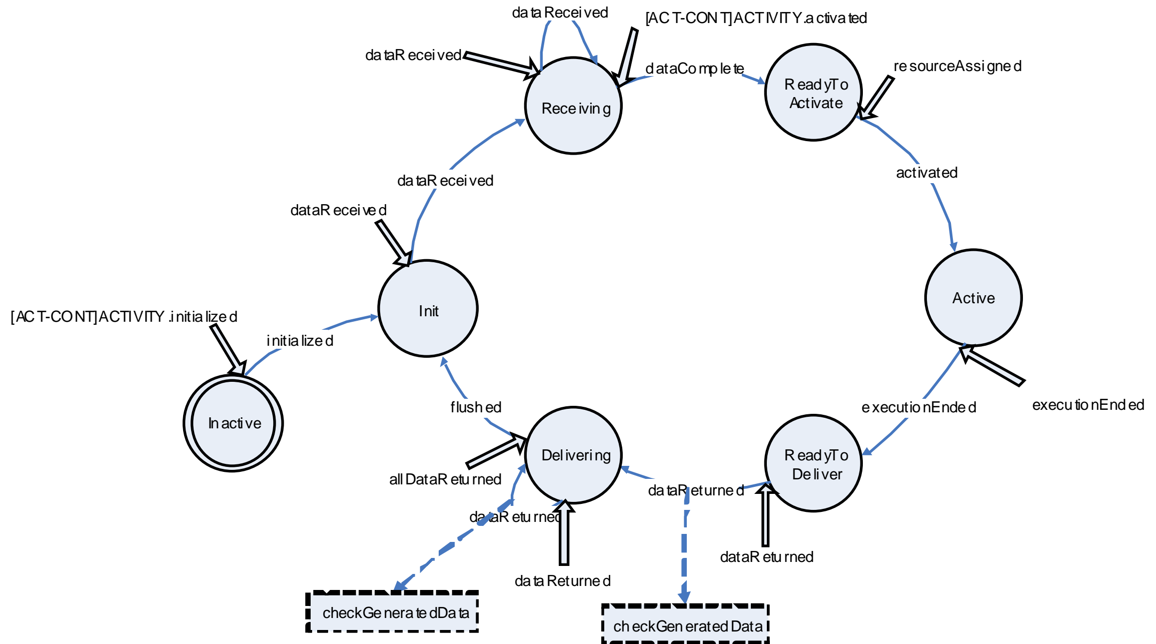
- **<u>Actividad</u>**

Figura 6.15. Autómata Sincronizado de una Actividad



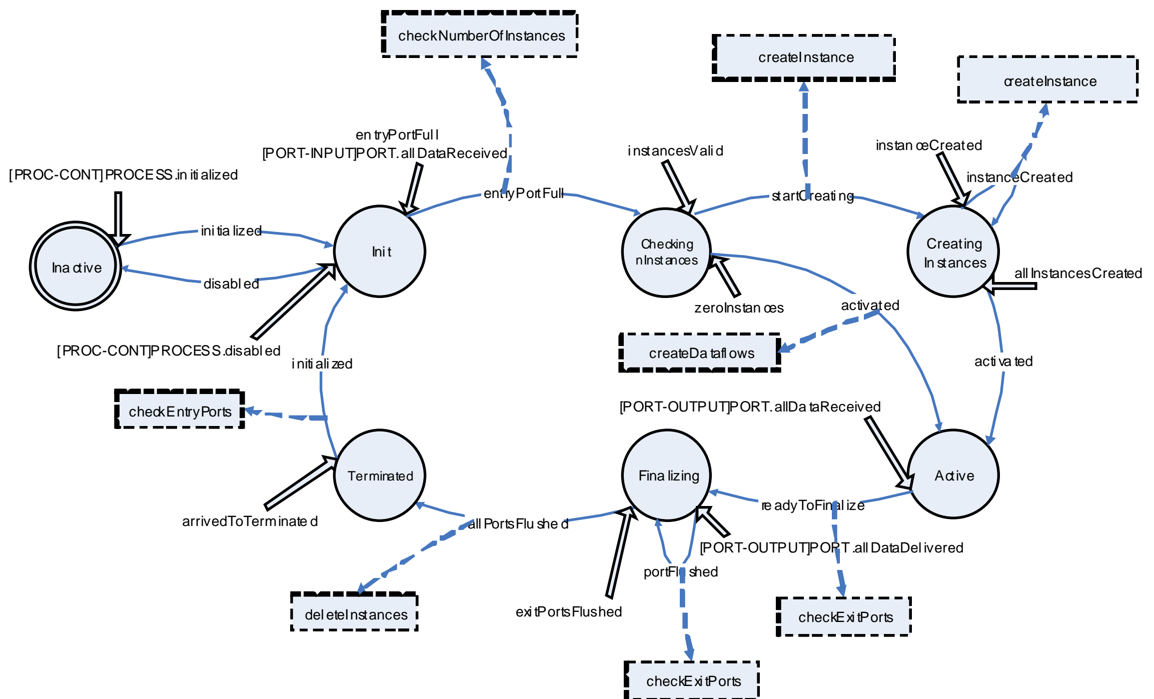
- **Workspace**

Figura 6.16. Autómata Sincronizado de un Workspace



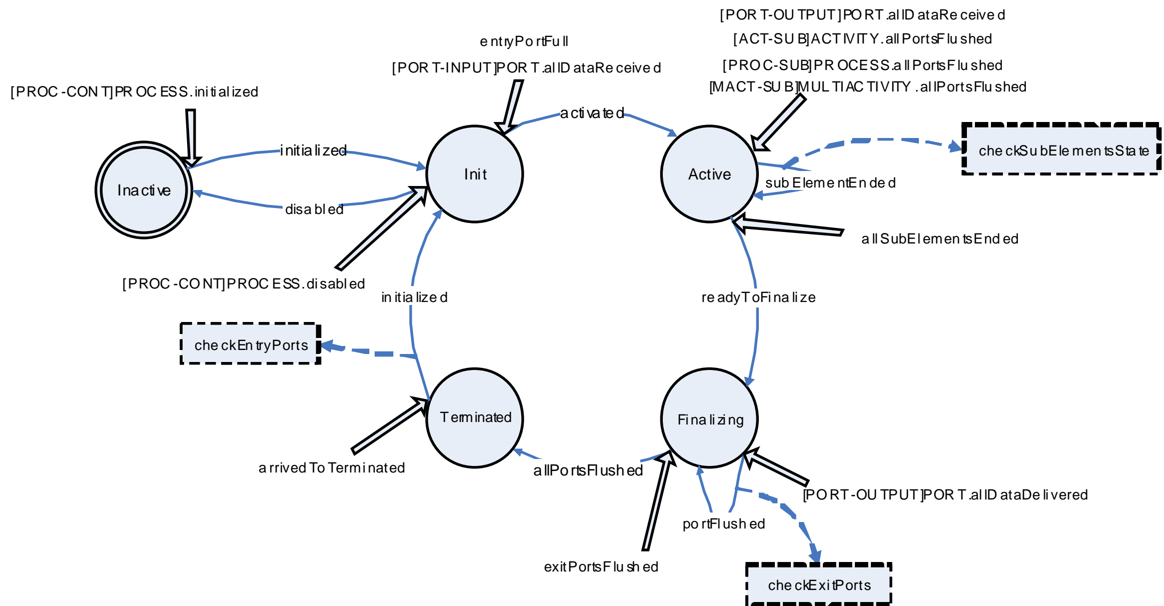
- **<u>MultiActividad</u>**

Figura 6.17. Autómata Sincronizado de una MultiActividad



- **Proceso**

Figura 6.18. Autómata Sincronizado de un Proceso



6.7 MECANISMOS DE EXTENSIBILIDAD

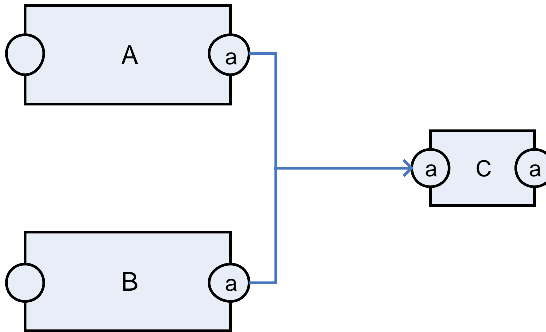
La flexibilidad del modelo Cumbia-XPM esta basada en cuatro conceptos: Reflexión, Extensión Simple, Adaptación y Especialización. La extensión simple, la adaptación y la especialización toman provecho de la separación explícita que existe entre el elemento y su semántica, para así cambiar el comportamiento del primero. Por su parte, la Reflexión se basa en la relación que existe entre los elementos del modelo dinámico con los respectivos elementos del modelo estático.

6.7.1 Reflexión. La reflexión brinda a C-XPM la capacidad de realizar Introspección e Reificación. La Introspección permite en tiempo de ejecución de conocer información acerca de los elementos involucrados en la definición de un proceso. La reificación brinda a C-XPM la capacidad de modificar un Proceso que se encuentra en ejecución. De esta forma, en una definición de Procesos se pueden agregar elementos como Actividades, Puertos, Dataflows y otros, según sea necesario.

A través de la reificación se puede cambiar la composición de un elemento cualquiera de una instancia de un proceso, o agregar nuevos elementos mientras se encuentra en ejecución. Dicha modificación se realiza dependiendo de diferentes situaciones relacionadas con la ejecución del proceso. Antes de seguir, es importante aclarar que dichos cambios se efectúan sobre la instancia en ejecución y no sobre el modelo del cual se instancia.

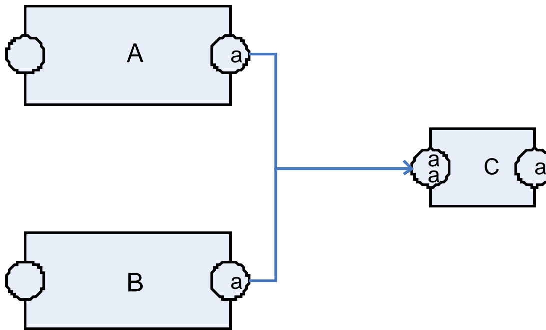
Por ejemplo, en el siguiente fragmento de un proceso, la actividad C se ejecutará cuando reciba el dato 'a', ya sea de A o de B:

Figura 6.19. Ejemplo de Reflexión: Vista en tiempo de diseño



Ahora, dicha definición se podría cambiar en tiempo de ejecución (dependiendo de alguna decisión, por ejemplo) para obtener lo siguiente:

Figura 6.20. Ejemplo de Reflexión: Vista en tiempo de ejecución



En este caso, la actividad C sólo se ejecutará cuando le llegue el dato 'a', tanto de A como de B.

Por otro lado, la reflexión también es utilizada por elementos del modelo en su semántica básica. Es el caso de la MultiActividad, la cual se basa en la reificación para crear el número indicado de instancias de una Actividad cuando llega a su estado *Creating Instances*.

Para poder soportar la Reflexión, es la razón por la que el modelo C-XPM está divido en dos secciones: la estática y la dinámica. En la parte estática del modelo se realizan las definiciones básicas de cada uno de los elementos que van a ser parte de la definición del proceso. En la parte dinámica del modelo se crean las instancias de los Procesos y las instancias de cada uno de los elementos que hacen parte de la definición del Proceso. Cada instancia de un elemento tiene una referencia al elemento estático al que está asociado. Es esta referencia la que permite conocer la información del elemento, a la vez que proporciona el acceso a las operaciones que permiten la modificación del proceso, en tiempo de ejecución.

6.7.2 Extensión Simple. La característica de extensión simple busca aumentar la funcionalidad existente en el comportamiento básico de los elementos. La nueva funcionalidad de un elemento se consigue agregando Acciones a su Autómata, el cual mantiene sus Estados y Transiciones originales.

Las Acciones en C-XPM representan funcionalidad que se ejecuta cuando un elemento cambia de estado. Hacen parte de la definición de los Autómatas, permitiendo así poder agregarlas y/o quitarlas sin afectar la implementación del elemento. Las Acciones son implementadas a través de Clases externas al modelo.

La Extensión se puede considerar un mecanismo similar a los Aspectos [KIC1997], ya que la nueva funcionalidad del elemento puede agregarse en cualquier punto de ejecución del mismo. La diferencia entre el mecanismo de extensión de C-XPM y los Aspectos radica en el primero no intercepta la ejecución de métodos específicos. Simplemente, se agrega esta nueva funcionalidad en el autómata del elemento. Como el autómata que define el comportamiento esta separado de la implementación del elemento evita tener que conocer la implementación del concepto para agregar la funcionalidad requerida. Además, su trazabilidad es mucho más sencilla, puesto que toda la definición de la semántica se encuentra en un solo plano (los autómatas).

Como escenario considere que se desea cargar un documento de un repositorio CVS para ser utilizado dentro de la ejecución de una Actividad. Para realizar esta extensión agregamos una acción cuya implementación se conecta al repositorio CVS y copia el documento necesitado en un directorio del sistema local.

En el Autómata de la Actividad indicada se agrega la Acción *cargarDocumento* en la Transición que va del Estado *Initializing WS* al Estado *Active*. Cuando la Actividad toma la Transición y alcanza el Estado Active, busca una implementación para la Acción definida como *cargarDocumento* y la ejecuta. La representación gráfica del Autómata es la siguiente.

Figura 6.21. Ejemplo de Extensión: Acción 'cargarDocumento'

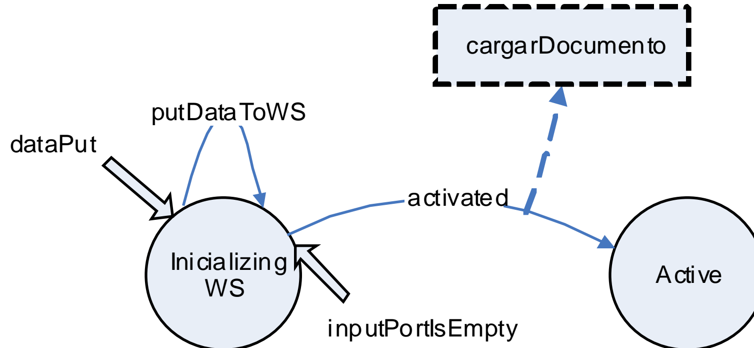


Tabla 6.8. Ejemplo de Implementación de la Acción 'cargarDocumento'

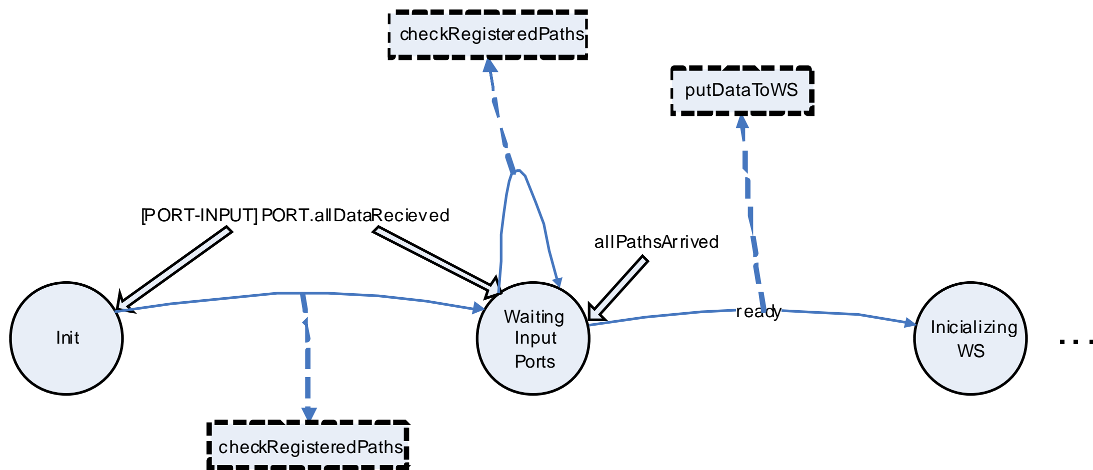
```
execute(Context ctx){

   //Conectarse al repositorio CVS
   connectCVS()

   //Autenticarse en el repositorio
   login(user, password)

   //Copia el documento
   copy(doc, dir)
}
```

6.7.3 Adaptación. La adaptación se refiere a la capacidad de modificar el comportamiento básico de un elemento del modelo. En C-XPM no se pueden crear nuevos conceptos dentro del modelo, pero si se pueden adaptar los conceptos existentes para que se comporten de forma diferente a su definición básica. De esta forma, se logra tener nuevos elementos.

Para realizar la modificación del comportamiento del elemento se define un nuevo autómata para el mismo, realizando cambios ya sea en los estados y/o en las transiciones. Los autómatas de los elementos interactúan entre sí, por lo que es necesario que cuando se realice una adaptación, el desarrollador sincronice el comportamiento del nuevo elemento con el comportamiento de los elementos ya existentes dentro del modelo.

El siguiente escenario permitirá aclarar el concepto de Adaptación. Considere que se desea crear un nuevo tipo de Actividad A′, la cual, a diferencia de la Actividad básica, se activa una vez todos sus Puertos de Entrada reciben los Datos esperados. Para lograr este objetivo, el Autómata de la Actividad nueva debe tener un Estado adicional que represente la espera de la Actividad por cada uno de sus Puertos de Entrada. Además, se deben agregar las Transiciones y Acciones correspondientes para lograr el correcto comportamiento de A′.

Figura 6.22. Ejemplo de Adaptación: Autómata del nuevo tipo de Actividad



- **Estado 'Waiting Input Ports':** Representa la espera por la activación de los Puertos de Entrada. La Actividad se quedará en este Estado hasta que todos los Puertos de Entrada se llenen.
- **Acción 'CheckRegisteredPaths':** Invoca la operación '*checkRegisteredPaths*'. Esta operación verifica que todos los Puertos de Entrada estén llenos. De ser así, lanza el evento *allPathsArrived*.

6.7.4 Especialización. La especialización se refiere a la posibilidad de crear nuevos elementos mediante la extensión de la funcionalidad (operaciones, atributos) de elementos ya existentes. Sin embargo, a diferencia de ésta, al crear un nuevo elemento mediante la especialización, su semántica es la misma que la del elemento que sirvió de modelo.

Como se puede observar, por medio de ninguno de los mecanismos mencionados, es posible crear un nuevo elemento de la nada. En el caso de la especialización, se toma un elemento ya definido en el modelo y se extiende su funcionalidad, creando así uno nuevo.

El Workspace es el elemento que posiblemente más utilice este mecanismo de extensibilidad, puesto que, como se mencionó anteriormente, es el que ejecuta la tarea que representa una actividad. Al no poseer una implementación básica definida, es mediante la especialización que se logra darle una funcionalidad particular a dicho workspace.

Un ejemplo sencillo de especialización sería un workspace que invoque un Web Service.

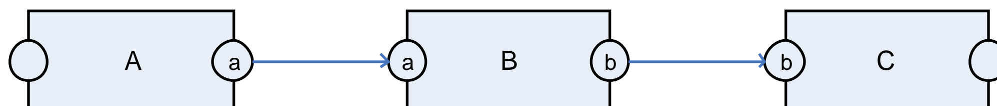
7. VALIDACIÓN DE CUMBIA-XPM BASADA EN LOS PATRONES DE CONTROL DE FLUJO

Tal como se mencionó en el capítulo 4, los Patrones de Control de Flujo es uno de los marcos más importantes actualmente, a la hora de evaluar un modelo de Workflow. En este capítulo, más que verificar si C-XPM soporta directamente todos los Patrones, lo que se quiere es demostrar su gran poder expresivo, al soportar dichos Patrones mediante alguno de los mecanismos de extensibilidad mencionados en el capítulo anterior.

7.1 PATRONES DE CONTROL BÁSICO

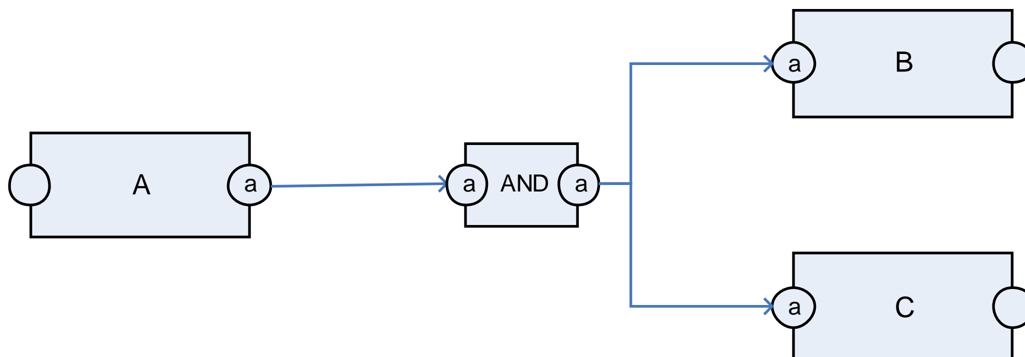
PF1 *Sequence.* Es el patrón más sencillo. En C-XPM, consiste en unir el único Puerto de Salida de una Actividad con el único Puerto de Entrada de otra Actividad, mediante un Dataflow. Dicho Dataflow debe ser capaz de traducir los Datos que recibe. Así mismo, el Puerto de Entrada debe poderse llenar con los Datos entregados por el Dataflow.

Figura 7.1. Diagrama del Patrón 'Sequence' en C-XPM



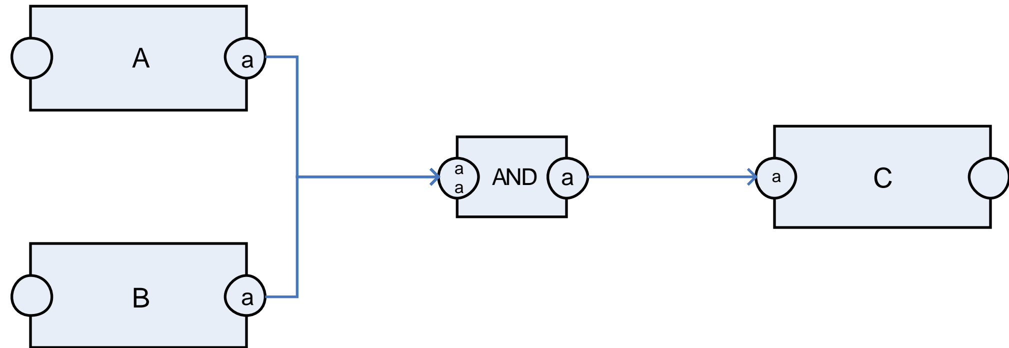
PF2 *Parallel Split.* C-XPM soporta el patrón utilizando una Actividad con un Puerto de Salida, del cual salen varios Dataflows dirigidos a cada uno de los Puertos de Entrada de las Actividades que van a tomar el control de forma paralela. Una vez dicho Puerto de Salida entrega todos sus datos a los Dataflows, éstos dirigen dichos datos (y por ende, el control) a las actividades siguientes, las cuales pueden iniciar su ejecución.

Figura 7.2. Diagrama del Patrón 'Parallel Split' en C-XPM



PF3 ***Synchronization.*** En C-XPM se soporta este patrón utilizando una Actividad con un Puerto de Entrada, donde convergen los diferentes caminos a sincronizar. En dicho puerto de entrada están conectados todos los Dataflows que provienen de los Puertos de Salida de aquellas Actividades a sincronizar, y están definidos cada uno de los conjuntos de variables que dichos Dataflows dan como resultado a su traducción. Sólo es posible ejecutar la Actividad que sincroniza cuando a su Puerto de Entrada han llegado todas las variables que dicho Puerto espera.

Figura 7.3. Diagrama del Patrón 'Synchronization' en C-XPM



PF4 ***Exclusive Choice.*** C-XPM soporta este patrón mediante una actividad con 'n' Puertos de Salida (un Puerto por cada posible camino a elegir), donde el grupo de variables esperadas en cada Puerto es diferente. El Workspace de la Actividad será el responsable, mediante condiciones establecidas, de hacer la elección del camino a tomar, generando **sólo** los datos cuyas variables espera el Puerto elegido.

Figura 7.4. Diagrama del Patrón 'Exclusive Choice' en C-XPM

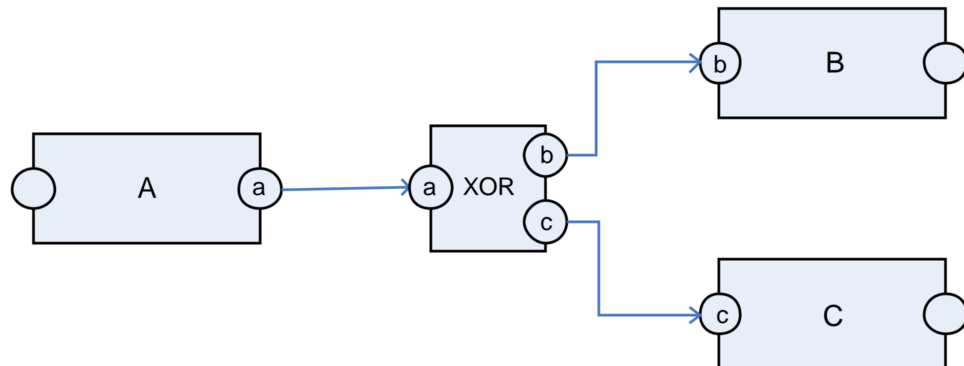
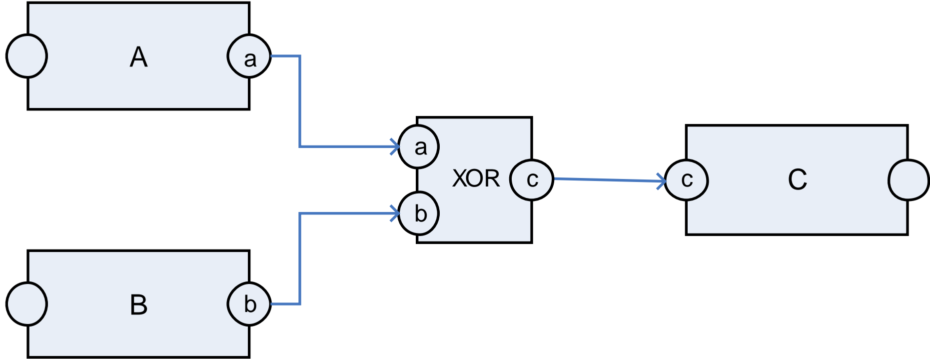


Tabla 7.1. Ejemplo de implementación del Workspace 'Exclusive-Choice'

```
execute(){

   if (condition = true)                                //Si  la  condición  se  cumple,
                                                        //se debe tomar el camino 'B'…

      //generar datos para salir por el camino 'B'.

   else

      //generar datos para salir por el camino 'C'.

   end_if

   notifyElement("executionEnded")

}
```

PF5 ***Simple Merge.*** La semántica básica de una Actividad en C-XPM es lo que permite soportar este patrón: una Actividad se activa cuando uno de sus Puertos de Entrada ha recibido todos los datos esperados. De esta forma, basta con definir una Actividad con un número de Puertos de Entrada igual al número de caminos a unir (*merge*).

Figura 7.5. Diagrama del Patrón 'Simple Merge' en C-XPM



7.2 PATRONES DE RAMIFICACIÓN Y SINCRONIZACIÓN AVANZADA

PF6 ***Multiple Choice.*** C-XPM soporta este patrón mediante una actividad con 'n' puertos de salida (un puerto por cada posible camino a elegir), donde el grupo de variables esperadas en cada puerto es diferente. El Workspace de la actividad será el responsable de hacer la elección de los caminos a tomar, generando **sólo** los datos cuyas variables esperan los puertos elegidos.

Figura 7.6. Diagrama del Patrón 'Multiple Choice' en C-XPM

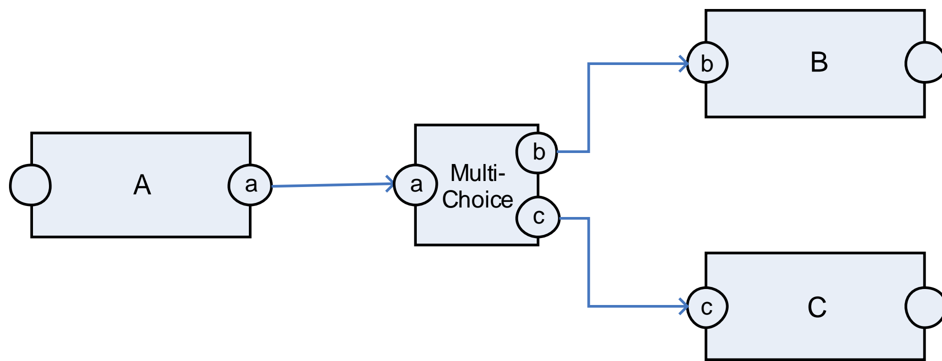


Tabla 7.2. Ejemplo de implementación del Workspace 'Multi-Choice'

```
execute(){
   if (condition1 = true)
      //generar datos para salir por el camino 'A'.
   end_if
   if (condition2 = true)
      //generar datos para salir por el camino 'B'.
   end_if
   if (condition3 = true)
      //generar datos para salir por el camino 'C'.
   end_if
   notifyElement("executionEnded")
}
```

PF7 *Synchronizing Merge.* Para representar este patrón se debe crear una actividad que sincronice sólo los caminos que un *Multi-Choice* previo activó (ver patrón anterior). Es necesario entonces crear un nuevo tipo de actividad, cuyo Autómata será una variante al Autómata de una Actividad Básica.

Figura 7.7. Diagrama del Patrón 'Synchronizing Merge' en C-XPM

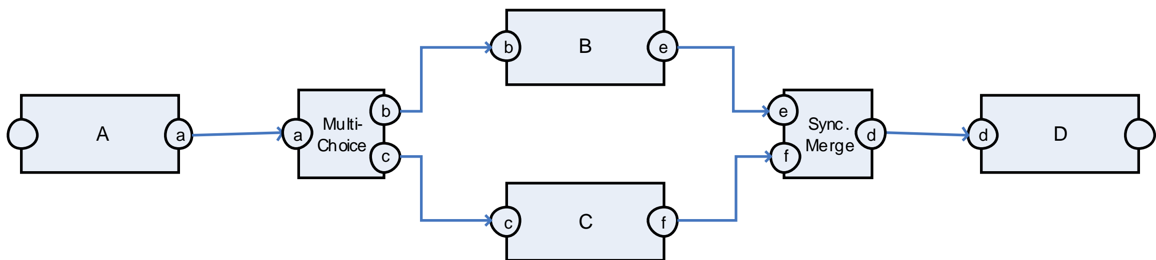
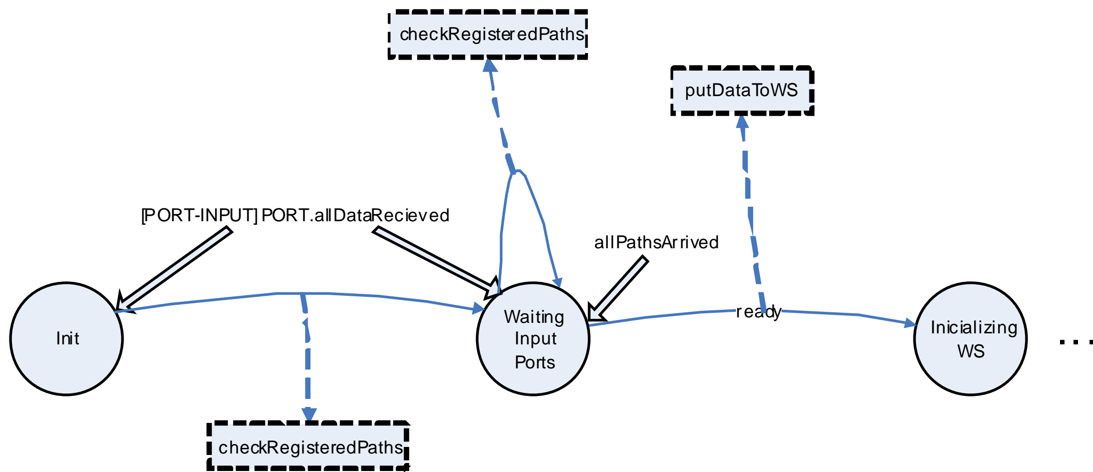


Figura 7.8. Modificación al Autómata de una Actividad, para crear el 'Sync. Merge'



- **Estado 'Waiting Input Ports':** Representa la sincronización de los caminos activados. La Actividad se quedará en este Estado hasta que todos los caminos esperados lleguen. En otras palabras, hasta que el número de Puertos de Entrada llenos sea igual al número de caminos activados por la actividad *Multi-Choice*.
- **Acción 'CheckRegisteredPaths':** Invoca la operación *'checkRegisteredPaths'*. Esta operación verifica si el Puerto de Entrada activado era el último camino a esperar. De ser así, lanza el evento *allPathsArrived*.

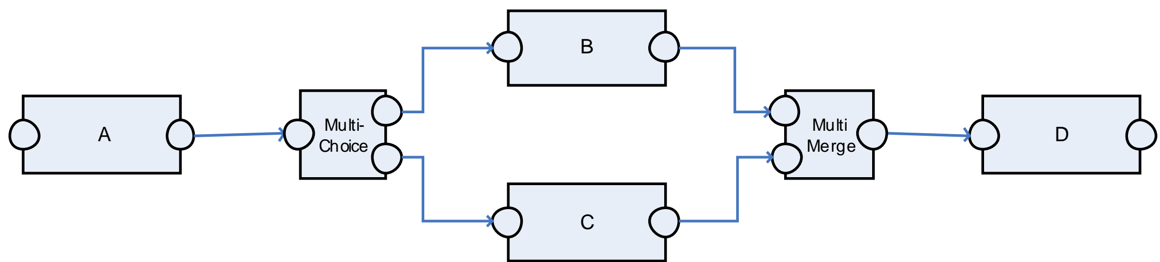
Tabla 7.3. Ejemplo de la operación 'checkRegisteredPaths'

```
checkRegisteredPaths(){

   activePaths ← getMemory
                 ("ROOT.MultiChoiceA.paths")        //Obtiene de la memoria del
                                                    //multichoice, el dato cuya
                                                    //variable es "paths".

   receivedPaths ← getMemory("paths") + 1          //Obtiene de la memoria
                                                    //local, el dato cuya
                                                    //variable es "paths".


   if (activePaths = receivedPaths)                 //Si todos los caminos
                                                    //esperados ya llegaron…

        notifyElement("allPathsArrived")
        receivedPaths ← 0

   end_if

   setMemory("paths",receivedPaths)                 //Actualiza en la memoria
                                                    //local, el dato cuya
                                                    //variable es "paths".

}
```

Como se puede observar, esta operación necesita que el Multi-Choice que activó los caminos establezca un Dato '*paths*' en el cual guarde el número de caminos activados. Para esto, es necesario agregar una Acción (en el Autómata del Multi-Choice) a la Transición que lanza el Evento *portFlushed*. Dicha Acción incrementará en uno el valor de la variable 'paths'.

PF8 *Multiple Merge*. Este patrón es soportado directamente por C-XPM, gracias a la semántica básica de un Puerto. En este caso, los Puertos de entrada de una Actividad siguen recibiendo datos, a pesar de que ésta se encuentre activa. Una vez la Actividad termine su ejecución, revisa si hay un Puerto de Entrada lleno (operación 'checkEntryPorts'). De ser así, vuelve a activarse con los Datos de dicho Puerto.

Figura 7.9. Diagrama del Patrón 'Multiple Merge' en C-XPM



PF9 *Discriminator*. Este patrón es soportado creando un nuevo tipo de actividad, cuyo Autómata será una variante al Autómata de una Actividad Básica.

Figura 7.10. Diagrama del Patrón 'Discriminator' en C-XPM

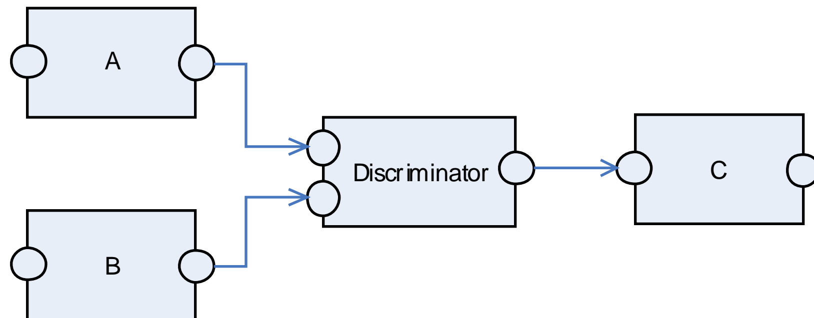
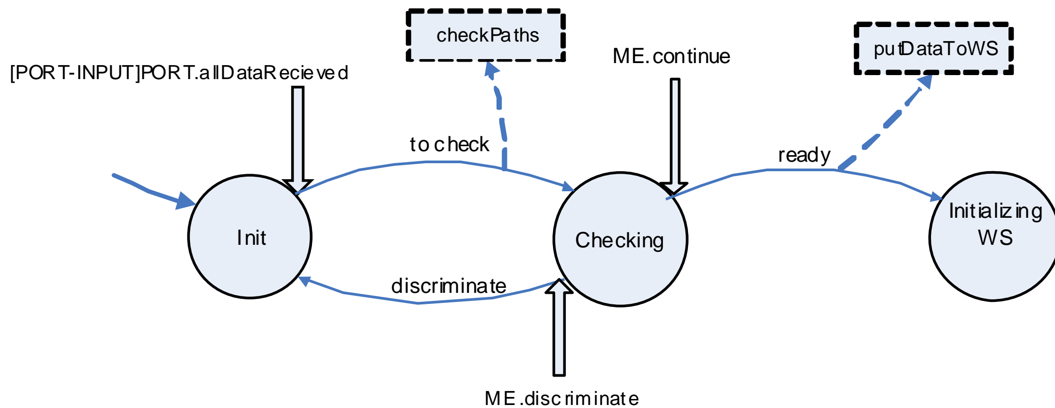


Figura 7.11. Modificación al Autómata de una Actividad, para crear el 'Discriminator'



- **Estado 'Checking':** Verifica si el camino recibido (representado por un Puerto de Entrada lleno) debe ser discriminado o no. Si recibe el Evento *continue* sigue al próximo Estado descrito en la figura (camino no discriminado); si recibe el Evento *discriminatePath* pasa al estado Init (camino discriminado).
- **Acción 'checkPaths':** invoca la operación 'checkPaths'. Esta operación verifica si el Puerto de Entrada activado, es el primer camino en llegar. Si es así, lanza el evento *continue*; si no, lanza el evento *discriminatePath*. Si era el último camino en llegar, el registro de caminos se reinicia (valor cero).

Tabla 7.4. Ejemplo de la operación 'checkPaths'

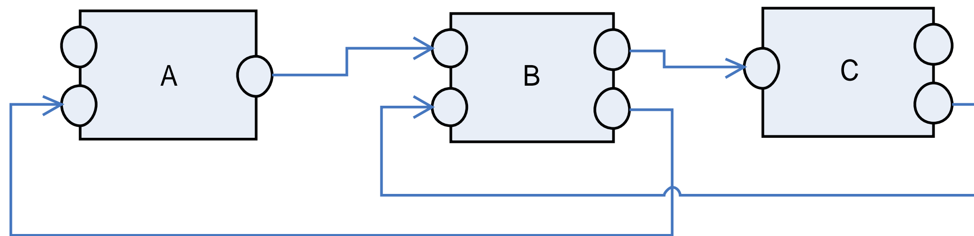
```
checkPaths(){

   activePaths ← getMemory
                ("ROOT.MultichoiceA.paths")          //Obtiene de la memoria del
                                                     //multichoice, el dato cuya
                                                     //variable es "paths".

   receivedPaths ← getMemory("paths") + 1            //Obtiene  de  la  memoria
                                                     //local,   el   dato   cuya
                                                     //variable es "paths".

   if (receivedPaths = 1)                            //Si es el primer camino en
                                                     //llegar…

        notifyElement("continue")

   else

                notifyElement("discriminate")
                //Discriminar camino

        if (activedPaths = receivedPaths)
        //Último camino en llegar

          receivedPaths ← 0

      end_if

   end_if

   setMemory("paths",receivedPaths)                  //Actualiza  en  la  memoria
                                                     //local,   el   dato   cuya
                                                     //variable es "paths".

}
```

Como se puede observar, esta operación necesita que el Multi-Choice que activó los caminos establezca un Dato '*paths*' en el cual guarde el número de caminos activados (tal como en el patrón anterior). Para esto, es necesario agregar una Acción (en el Autómata del Multi-Choice) a la Transición que lanza el Evento *portFlushed*. Dicha Acción incrementará en uno el valor de la variable 'paths'.

7.3 PATRONES ESTRUCTURALES

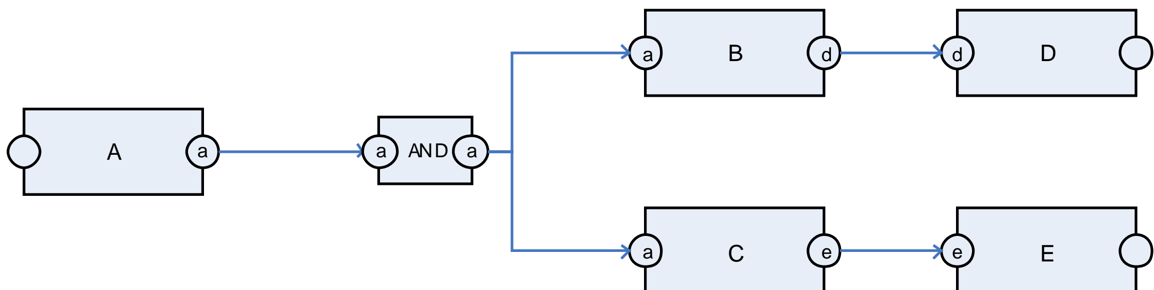
PF10 *Arbitrary Cycles.* Este patrón es soportado directamente por C-XPM. Para entender más fácilmente como es soportado el patrón, considere el siguiente escenario: existe una Actividad A que, una vez ejecutada, pasa el control a una Actividad B. B tiene dos Puertos de salida: uno cuyo Dataflow va hacia la Actividad C y otro cuyo Dataflow va hacia A. De forma análoga, C tiene un Puerto de Salida cuyo Dataflow se dirige hacia B. En este modelo, el control puede ser pasado de A hacia B, y en este punto el control podría volver hacia A realizando un ciclo. La Actividad B también podría tomar el camino hacia C, la cual podría volver hacia B, realizando así un ciclo también. Note que el control puede ser dirigido desde C hasta A utilizando los dos ciclos definidos en el proceso. Esto muestra claramente que en C-XPM se pueden definir ciclos no estructurados.

Figura 7.12. Diagrama del Patrón 'Arbitrary Cycles' en C-XPM



PF11 *Implicit Termination.* La terminación implícita de un proceso es soportado de forma directa por C-XPM, gracias a la semántica básica de los Procesos: cada vez que una Actividad termina su ejecución, el Proceso que la contiene verifica que no haya más nada por hacer, para así terminar su propia ejecución.

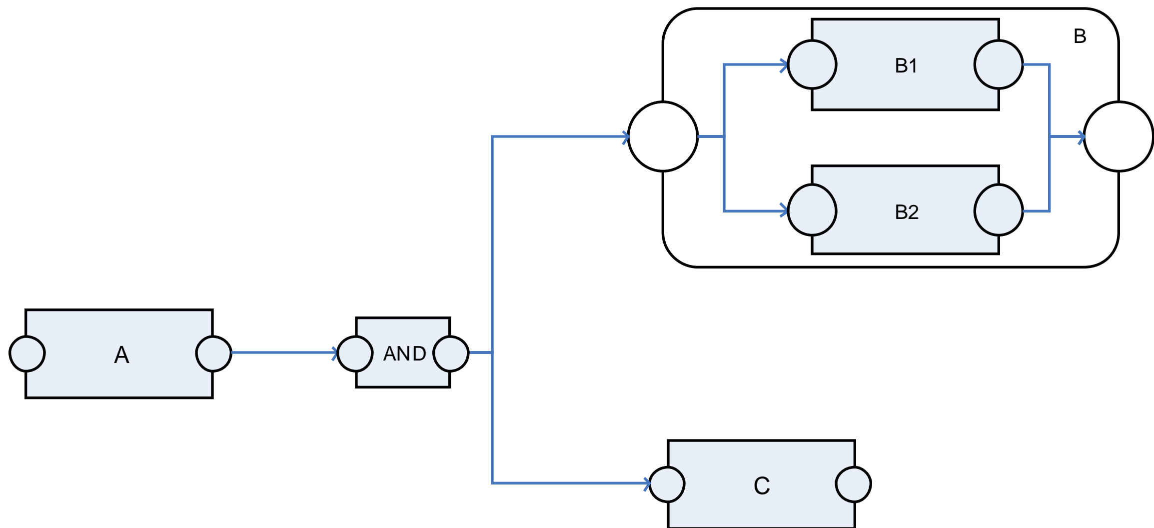
Figura 7.13. Diagrama del Patrón 'Implicit Termination' en C-XPM



7.4 PATRONES DE INSTANCIAS MÚLTIPLES

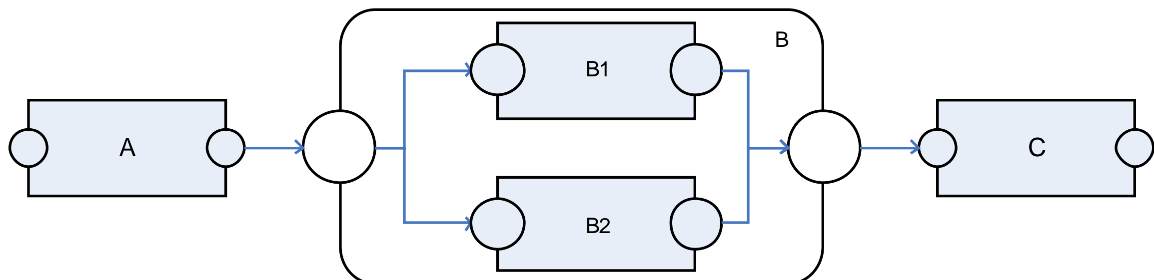
PF12 MI *without Synchronization.* Para soportar los patrones de múltiples instancias en C-XPM, es necesario utilizar el elemento MultiActivity. Dicho elemento crea y contiene las instancias necesarias, de acuerdo a su atributo *nInstances* (número de instancias). Con el fin de lograr la no-sincronización, es necesario apoyarse en el patrón *Parallel-Split*, descrito anteriormente (Ver diagrama anterior). Además, del Puerto (o Puertos, en caso de así serlo) de Salida de la MultiActivity B no salen *Dataflows*.

Figura 7.14. Diagrama del Patrón 'MI without Synchronization' en C-XPM



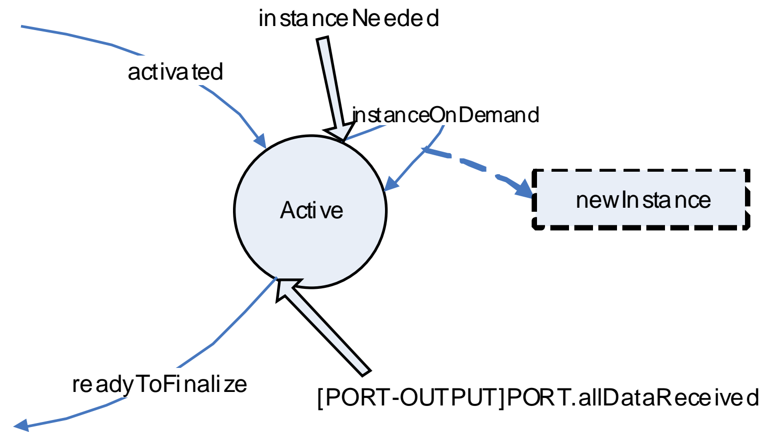
PF13-PF15 *MI with Synchronization.* Los patrones 13 y 14 se modelan de manera similar en C-XPM (Ver figura 7.15 y 7.16). Esto se debe a que la MultiActivity crea sus instancias en el momento en que es inicializada, basándose en su atributo nInstances. Dicho atributo puede ser definido en tiempo de diseño (patrón 13) o editado en tiempo de ejecución (patrón 14).

Figura 7.15. Diagrama de los Patrones 'MI with Synchronization 13 y 14' en C-XPM



El patrón 15 es modelado en C-XPM de manera similar a los anteriores. Sin embargo, de debe crear un nuevo tipo de MultiActividad, cuyo Autómata será una variante al Autómata de una MultiActividad Básica.

Figura 7.16. Modificación al Autómata de una MultiActividad, para soportar el Patrón 15



- **Acción 'newInstance':** invoca la operación 'newInstance'. Esta operación crea una nueva instancia para la MultiActividad.

7.5 PATRONES BASADOS EN ESTADO

PF16 ***Deferred Choice.*** Para soportar este patrón en C-XPM, se debe definir un Workspace bloqueante, es decir, que espera por una señal para continuar (o finalizar, en este caso) su ejecución. Dicha señal será la encargada además de indicarle al Workspace qué camino tomar (Ver patrón *Exclusive-Choice*)

Figura 7.17. Diagrama del Patrón 'Deferred Choice' en C-XPM

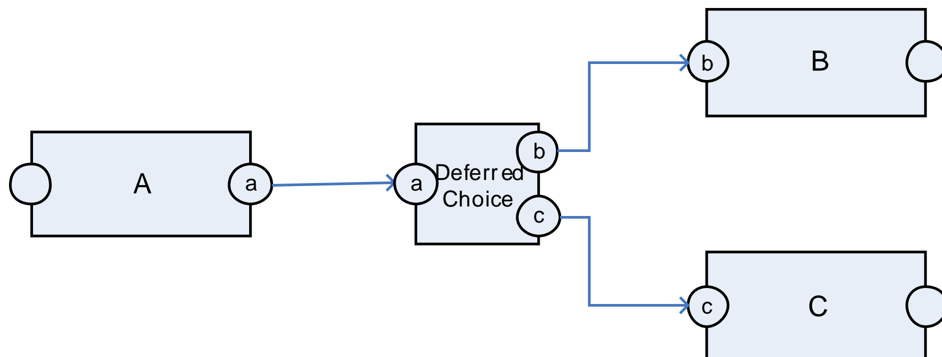


Tabla 7.5. Ejemplo de la implementación del Workspace 'Deferred-Choice'

```
execute(){

   waitingForSignal(signal)                         //Esperando la señal "signal"

   if (signal.value = "A")                          //Si la señal indica que se
                                                    //debe tomar el camino "A"…

      //generar datos para salir por el camino A.

   else

      //generar datos para salir por el camino B.

   end_if

   notifyElement("executionEnded")

}
```

PF17 ***Interleaved Parallel Routing.*** Para soportar este patrón, basta con que las Actividades que actúen como 'Start Interleaving' y 'End Interleaving' tengan Workspaces especiales. Adicionalmente, el manejo de la Memoria en el Workspace del 'Start-Interleave' es muy importante. Debe existir un dato que contenga la información que representa el orden de los caminos a escoger, otro que contenga el número de caminos tomados y otro que contenga el número total de caminos.

Figura 7.18. Diagrama del Patrón 'Interleaved Parallel Routing' en C-XPM

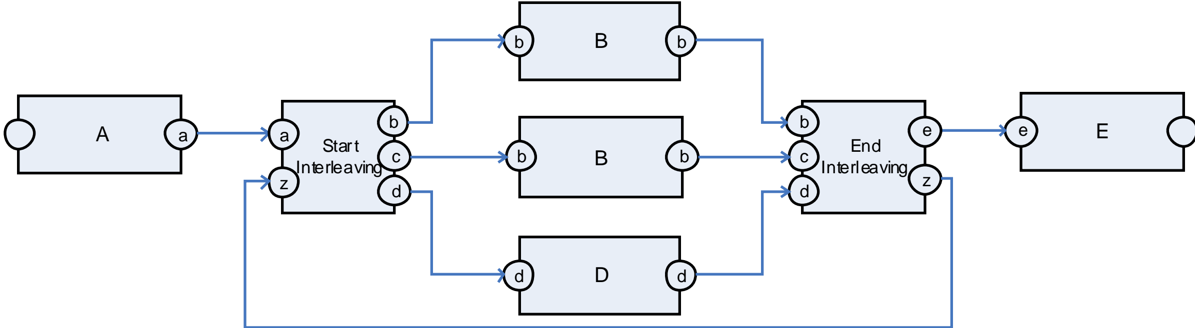


Tabla 7.6. Ejemplo de implementación del Workspace 'Start-Interleaving'

```
execute(){

   pathsTaken ← getMemory("pathsTaken")          //Obtiene el número de
                                                 caminos //tomados…

   order ← getMemory("order")                    //Obtiene el orden de
                                                 //ejecución de los caminos

   generateData(order.value[pathsTaken + 1])     //Genera los datos para salir
                                                 //por el camino respectivo

   pathsTaken ← pathsTaken + 1

   paths ← getMemory("paths")                    //Obtiene el total de caminos
                                                 //a tomar…

   if (pathsTaken = paths)                       //Si ya se tomaron todos los
                                                 //caminos…

      pathsTaken ← 0

   end_if

   setMemory("pathsTaken", pathsTaken)

   notifyElement("executionEnded")

}
```

El número total de caminos a tomar es un dato colocado mediante una acción, en la Transición *initialized* de la Actividad que contiene el Workspace 'Start Interleaving':

Tabla 7.7. Ejemplo de la Acción 'paths'

```
paths(Context ctx){

   ctx.setMemory("paths", ctx.numberOfExitPorts())

}
```

Por otro lado, el workspace 'End Interleaving' sólo debe tener en cuenta el número de veces que ha sido ejecutado para escoger el camino de salida (volver al 'Start Interleaving' o seguir con el proceso).

Tabla 7.8. Ejemplo de implementación del Workspace 'End-Interleaving'

```
execute(){

   totExecutions ← getMemory("totExecutions")        //Obtiene el número total
                                                     //de ejecuciones

   executions ← getMemory("executions")              //Obtiene el número de
                                                     //ejecuciones realizadas

   executions ← executions + 1

   if (executions = totExecutions)                   //Si ya se ejecutó el
                                                     //número de veces necesario

      executions ← 0

      //Generar los datos para seguir con
      //la ejecución del proceso
      .
      .
   else

      //Generar los datos para tomar el camino
      //de regreso a 'Start Interleaving'
      .
      .

   end_if

   setMemory("executions", executions)

   notifyElement("executionEnded")

}
```

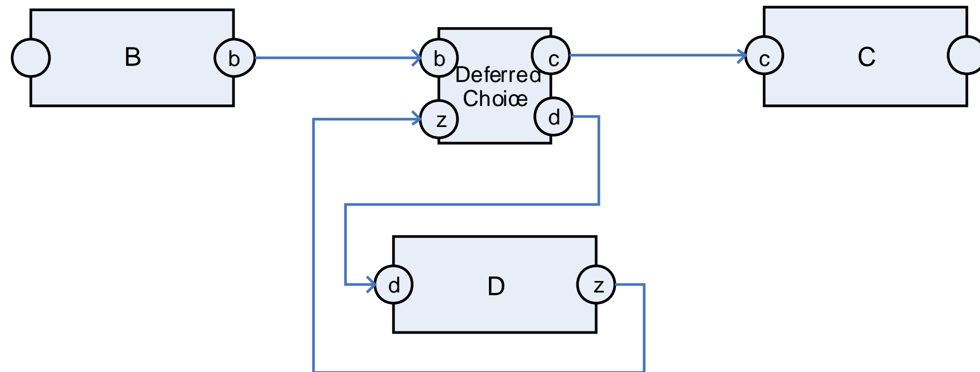
De igual forma que en el workspace 'Start Interleaving', el número total de ejecuciones a realizar es un dato colocado mediante una acción, en la transición *initialized* de la actividad que contiene el workspace 'End Interleaving':

Tabla 7.9. Ejemplo de la Acción 'executions'

```
executions(Context ctx){

   ctx.setMemory("totExecutions", ctx.numberOfEntryPorts())

}
```

PF18 *Milestone.* La forma más fácil de mostrar cómo se soporta este patrón en C-XPM es mediante un ejemplo. Considerando tres actividades A, B y C, la actividad B será modelada como el punto que hay que alcanzar para que la actividad A se habilite (milestone). La actividad B tiene dos puertos de salida: uno hacia A y otro hacia C, donde solo uno de dichos puertos puede ser escogido como resultado de la ejecución de B (*Deferred-Choice*). La actividad A, por su parte, solo se puede ejecutar si ya se ejecutó B y no se ha ejecutado C. Además, posee un puerto de salida que es dirigido hacia la actividad B (milestone). De esta forma, una vez la actividad A ha sido ejecutada el control vuelve a la actividad B, la cual nuevamente tiene la opción de continuar por A o por C. Si la actividad B continúa su ejecución hacia la actividad C, no será posible volver a ejecutar la actividad A.

Figura 7.19. Diagrama del Patrón 'Milestone' en C-XPM



7.6 INTERPRETACIÓN DE LOS RESULTADOS OBTENIDOS

Como principal resultado se tiene que todos los patrones fueron soportados. Ya sea de manera directa o mediante alguno de los mecanismos de extensibilidad, fue posible expresar cada uno de los 18 patrones escogidos, en Cumbia-XPM.

De manera general, los resultados son los siguientes:

- Se lograron representar de manera directa diez (10) de los patrones de control de flujo propuestos.
- Los ocho (8) patrones restantes se representaron mediante mecanismos de extensibilidad, así:
  - Cuatro por medio de especialización.
  - Dos mediante una combinación de Adaptación y Extensión simple.
  - Uno mediante una combinación de Extensión simple y Especialización.
  - Uno por medio de Adaptación.

A continuación se presenta una tabla con los resultados del análisis anterior, así como estudios realizados a otros modelos.

Tabla 7.10. Comparación de C-XPM, APEL, XPDL y BPEL utilizando los Patrones de Control de Flujo

Patrón	C-XPM	APEL	XPDL	BPEL
Sequence	+	+	+	+
Parallel Split	+	+	+	+
Synchronization	+	+	+	+
Exclusive Choice	Esp	+	+	+
Simple Merge	+	+	+	+
Multi Choice	Esp	+	+	+
Synchronizing Merge	Ada-Esp	+	+	+
Multi Merge	+	+	-	-
Discriminator	Ada-Esp	-	-	-

Arbitrary Cycles	+	+	+	-
Implicit Termination	+	-	+	+
MI without Synchronization	+	-	+	+
MI with a Priori Design Time Knowledge	+	+	+	+
MI with a Priori Runtime Knowledge	+	-	-	-
MI without a Priori Runtime Knowledge	Ada	-	-	-
Deferred Choice	Esp	+	-	+
Interleaved Parallel Routing	Esp-Ext	-	-	+/-
Milestone	Esp	+	-	-

Resultados de XPDL y BPEL tomados de http://www.workflowpatterns.com

En la tabla se presenta un listado de los patrones y una columna por cada uno de los modelos comparados. Para cada celda de intersección se presenta un "+" si existe una construcción que soporte directamente el patrón, un "–" si el patrón no es soportado por el modelo y un "+/-" si existe alguna alternativa que imponga restricciones en la estructura del proceso para soportar el patrón. Para el caso de C-XPM, se espcifica el mecanismo de extensión que permitió soportar dicho patrón (Esp: Especialización, Ada: Adaptación, Ext: Extensión Simple).

Se puede observar que el mecanismo más usado es el de la Especialización, el cual está muy relacionado con la toma de decisiones en un proceso. Adicionalmente, en varios casos fue necesario utilizar no sólo uno, sino dos mecanismos de extensibilidad para soportar un patrón. Esto demuestra una vez más el gran poder expresivo que se puede conseguir mediante los mecanismos de extensión propuestos.

8. CONCLUSIONES

Cumbia-XPM es un Modelo de Workflows Extensible, que utiliza un conjunto minimal de conceptos. Sin embargo, ese conjunto mínimo de conceptos se basa en un poderoso esquema de extensibilidad, fundamentado en la definición semántica de los elementos que componen el modelo.

Como se pudo observar a lo largo del documento, C-XPM reúne lo mejor de cada modelo presentado en el capítulo 3, pero superando a la mayoría por su generalización, ya que no depende de ninguna tecnología. Dicha abstracción es la que permitirá que su especificación pueda ser implementada por la tecnología deseada, lo cual permite que sus campos de aplicación sean aún mayores (aplicaciones *standalone*, web services, etc).

Por otro lado, su esquema de extensión es el más flexible. A pesar de que jBPM facilita conceptos de extensibilidad similares a C-XPM, limita al usuario al no permitir la interacción con la semántica de los elementos que presenta. C-XPM permite no sólo agregar acciones en puntos específicos, sino que dichos puntos puedan ser definidos según los desee y necesite un escenario dado. Esto es fundamentalmente lo que hace pensar que cualquier situación puede ser modelada por medio de C-XPM.

Los conceptos propuestos por este modelo, al ser traducidos e implementados correctamente, permitirán la construcción de aplicaciones altamente evolutivas. Aplicaciones que podrán ser servicios orquestados, tareas humanas controladas, integración de aplicaciones más pequeñas o inclusive la combinación de estos tres enfoques.

Si bien aún falta mucho por desarrollar, se puede observar mediante el estudio realizado en el capítulo 7 que la dirección es la correcta. A pesar de que es necesario formalizar varios conceptos como la sincronización de los autómatas, fue posible tener la noción esperada, la cual no es otra que el gran poder expresivo que puede alcanzar C-XPM por medio de su extensibilidad. La principal enseñanza que este trabajo deja es la de mostrar que para que un modelo tenga gran poder de expresión no es necesario tener una gran cantidad de elementos. Lo fundamental es proporcionar al desarrollador o cualquier persona que utilice dicho modelo, de las herramientas suficientes que le permitan definir cualquier dominio deseado.

Por otro lado, un gran aporte que este trabajo da como resultado es la noción de crear modelos extensibles. El esquema de extensibilidad propuesto para C-XPM, basado en autómatas merece ser estudiado más a fondo, con el fin de identificar si es posible aplicarlo a cualquier modelo. Los alcances que se podrían lograr con esto serían enormes, puesto que se podría aumentar el poder expresivo de cualquier modelo, al volverlo extensible.

9. TRABAJO FUTURO

El Modelo C-XPM es sólo un primer paso en la búsqueda de un objetivo mayor: Un Sistema Manejador de Workflows. Para llevarlo a cabo, son muchos los trabajos y estudios que se deben realizar en torno a dicha meta.

A nivel de modelo, estos conceptos deben ser estudiados a fondo, con el fin de determinar si deben ser incorporados en el mismo:

- **Validación de la Sincronización de los Autómatas.** La definición de la semántica de los elementos del modelo por medio de autómatas, es el aspecto más llamativo de la propuesta presentada. Sin embargo, es necesario validar de manera formal que realmente dichos autómatas se encuentren sincronizados, con el fin de garantizar completamente la coherencia del modelo base. Además, esto servirá de gran ayuda porque al crear nuevos elementos y añadirlos al modelo, estos puedan encajar correctamente con los ya existentes.

- **Sistema de recuperación de Errores.** Es la capacidad que tiene un modelo para realizar una tarea de recuperación cuando ocurre un problema en la ejecución de un proceso determinado. Como se puede ver por la definición dada, está estrechamente relacionada con la ejecución del proceso. Es por esto que se deben analizar los conceptos a incluir en el modelo con el fin de permitir al diseñador agregar puntos de recuperación ante alguna falla.

- **Soporte de Transacciones.** Debe estudiarse la posibilidad de que los conceptos relacionados con transacciones estén incluidos directamente con el modelo. Esto permitirá que la tolerancia a fallos sea más fácil de implementar, a nivel del motor. Además, se debe analizar qué tipo de transacciones se ajusta más al ambiente Workflow.

- **Patrones de Control de flujo 19 y 20.** Dichos Patrones no fueron contemplados por el modelo actual, pero deben ser tenidos en cuenta. Estos patrones representan interacciones a un nivel más alto.

- **Análisis basado en Patrones de Recursos.** Este nuevo estudio [RUS2004] proporciona un nuevo framework para evaluar el Modelo Organizacional que poseen los modelos de Workflow. Dichos patrones pretenden capturar las diferentes formas en que los recursos son representados y usados. Este aspecto es muy importante, puesto que hace parte innata de los Procesos de Negocio de cualquier empresa.

- **Análisis basado en Patrones de Datos.** Este estudio [VDA2004], similar a los mencionados anteriormente (Patrones de Control y Patrones de Recursos), busca presentar las diferentes formas en que la información es representada y usada en los modelos de Workflow. En resumen, estos tres conjuntos de patrones representan los grandes dominios que debe poseer un modelo de Workflow: Control de Flujo, Datos y Recursos.

- **Formalización del lenguaje de Definición.** Con el fin de poder modelar un Proceso es necesario un lenguaje de definición, el cual debe representar fielmente los conceptos definidos por el modelo. A pesar de que se han realizado aproximaciones sobre posibles representaciones, es

necesario obtener un lenguaje formal que permita expresar todo aquello que el Modelo planteado puede expresar.

Además de estos estudios, es necesario desarrollar desde ya prototipos del motor y de herramientas de soporte que permitan tener una mejor perspectiva del funcionamiento del Modelo. En estos campos es necesario:

- **Diseño y Desarrollo de un Motor C-XPM.** Lo fundamental de este aspecto es diseñar e implementar los principales elementos de la arquitectura deseada, con el fin de obtener un núcleo base que sirva para realizar pruebas de ejecución de Procesos.

- **Desarrollo de Herramientas, como Editores, Monitores, administradores, etc.** Estas herramientas deben ser independientes del motor a utilizar, por lo que su desarrollo puede hacerse paralelamente con el de dicho motor. El desarrollo de estas aplicaciones es muy importante, puesto que se constituyen en la cara comercial del Modelo.

- **Prototipos de Procesos reales.** Deben modelarse y ejecutarse Procesos, en lo posible reales basados en situaciones reales, que permitan evaluar a C-XPM. Dichos prototipos deben evaluar la capacidad expresiva del Modelo, así como su extensibilidad.

REFERENCIAS

[BAY2004] Tom Baeyens. The State Of Workflow. JBoss jBPM, 2004. Disponible en: http://www.jbpm.org/state.of.workflow.html

[BAY2005] Tom Bayens. Java Business Process Management, Version 3.0. Disponible en: http://www.jbpm.org/3/.

[BOJ2004] Bojanic. S, Milakovic. Z, Puskas. V & Stefanovic. N. JaWE documentation. Disponible en http://jawe.objectweb.org/doc/1.4/index.html

[CUR2002] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.0. Standards propsal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2002.

[DEN2000] St-Denis. G, Shauer. R & Keller. R. Setting a Model Interchange Format – The SPOOL Case Study. Universidad de Montreal, Canada. 2000. Disponible en: http://csdl.computer.org/comp/proceedings/hicss/2000/0493/08/04938055.pdf.

[EST2003] Estublier. J, Villalobos. J, LE. T, Salanville. S & Vega, G. An Approach and Framework for Extensible Process Support System. 2003.

[GAM1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of reusable Object-Oriented Software. Addison-Wesley, 1995.

[JBP2005] JBoss jBPM Graphical Process Designer, version 3.0. JBoss jBPM, 2005. Disponible en: http://jbpm.org/gpd/

[KIC1997] Kiczales. G, Lamping. J, Mendhekar. A, Maeda. C, Videira. C, Loingtier J & Irwin. J. Aspect-Oriented Programming. Conferencia Europea de Programación Orientada a Objetos. 1997. Disponible en: http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-ECOOP97/for-web.pdf

[LEY2001] F. Leymann. Web Service Flow Language (WSFL 1.0). IBM Software Group. 2001. Disponible en: http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[PEL2003] C. Peltz. Web Services Orchestration. Hewlett-Packard Company. 2003. Disponible en: http://devresource.hp.com/drc/technical_white_papers/WSOrch/WSOrchestration.pdf.

[PED2005] Gabriel Pedraza, Jaime Solano, Jorge Villalobos. Cumbia-XPM: Cumbia Extensible Process Modeling, versión 0. Model Specification Draft. 2005.

[PWO2002] P. Wohed, W.M.P. van der Aalst, M. Dumas, y A.H.M. ter Hofstede. Pattern-Based Analysis of BPEL4WS. QUT Technical report, FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002.

[PWO2004] P. Wohed, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, y N. Russell. Pattern-based Analysis of UML Activity Diagrams. BETA Working Paper Series, WP 129, Eindhoven University of Technology, Eindhoven, 2004.

[RUS2004] N. Russell, A.H.M. ter Hofstede, D. Edmond, y W.M.P. van der Aalst. Workflow Resource Patterns. BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, 2004.

[THA2001] S. Thatte. XLANG Web Services for Business Process Design. Microsoft Corporation. 2001. Disponible en: http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[VDA2002] Will M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, y P. Wohed. Pattern-Based Analysis of BPML (and WSCI). QUT Technical report, FIT-TR-2002-05, Queensland University of Technology, Brisbane, 2002.

[VDA2003] Will M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, y A.P. Barros. Workflow Patterns. Distributed and Parallel Databases, 14(1):5–51, 2003.

[VDA2004] N. Russell, A.H.M. ter Hofstede, D. Edmond, y W.M.P. van der Aalst. Workflow Data Patterns. QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[VIL2002] J. Villalobos, APEL Light Specification. LSR-IMAG, Francia, 2002.

[WMC1995] Workflow Management Coalition. The Workflow Reference Model. (WFMCTC-1003). 1995.

[WMC2002] Workflow Management Coalition. Workflow Process Definition Interface – XML Process Definition Language (XPDL) (WFMCTC-1025). 2002.

[WVA2003] Will M.P. van der Aalst. Patterns and XPDL: A Critical Evaluation of the XML Process Definition Language. QUT Technical report, FIT-TR-2003-06, Queensland University of Technology, Brisbane, 2003.