

UNIVERSIDAD DE LOS ANDES

**Motor de Aspectos que Resuelve
Interferencias Basado en Cumbia: Caso
BPEL**

por

Manuel Muñoz Lara

Tesis presentada como requisito de grado de
Maestría en Construcción de Software

para la
Facultad de Ingeniería
Departamento de Ingeniería y Computación

29 de enero de 2009

“If you really want something in life you have to work for it. Now quiet, they’re about to announce the lottery numbers.”

H. J. Simpson

Agradecimientos

Quiero agradecer a todos los integrantes del proyecto Cumbia por su colaboración en el desarrollo de éste trabajo. Especialmente quiero agradecer a mi asesor Jorge Villalobos, por darme la oportunidad de pertenecer al grupo y guiarme en la realización de este trabajo. Sin su habilidad para identificar cuando necesitaba un latigazo o una palabra de apoyo éste trabajo no hubiera sido terminado.

También quiero agradecer muy especialmente a todos mis amigos y personas que se interesaron y se preocuparon por mí durante ésta etapa llena de retos. Especialmente quiero agradecer a¹ Nadya, Juliana, Felipe, Camilo, Pablo, Mario y Mentor², ya que gracias a ellos me di cuenta que ésta era una etapa más y que para poder superarla necesitaba tanto de un gran esfuerzo personal como del gran apoyo de la gente que me rodea. Ellos fueron los encargados de que no me olvidara que “si se puede” porque todos han demostrado que estamos hechos para grandes cosas y que los verdaderos amigos son los que están para ofrecer su ayuda en los momentos más difíciles, pero sobre todo para despejar tu cabeza cuando las cosas se ponen peor.

Especialmente quiero agradecer a Viviana, ya que sin su apoyo y entendimiento durante todos estos meses de trabajo, no lo hubiera podido lograr. Gracias por esos pequeños mensajes que dejabas regados por todas partes para que yo los encontrara, no te alcanzas a imaginar lo mucho que significa para mí que estés a mi lado durante uno de los momentos más importantes en mi formación como profesional y como persona.

Para finalizar, pero no por eso menos importante, quisiera agradecer de todo corazón a mi familia. A mis padres por demostrar su apoyo incondicional y sobre todo su paciencia conmigo, cuando el trabajo me consumía y era realmente un reto aguantarme. Alguien dijo alguna vez “If I have seen further it is by standing on the shoulders of giants” y para mí esos gigantes son mis padres, no conozco mejor ejemplo a seguir de un par de personas trabajadoras, fieles a sus ideales, donde la vida les ha lanzado de todo y ellos han salido victoriosos no de todas, pero juntos siempre. A los tintos con mi hermano Jaime, eran la manera perfecta de poder salir a dar una vuelta, hablar de todo un poco y desconectarse lo suficiente para poder disfrutar de la charla, pero sin olvidar que había que regresar a continuar trabajando. A mi hermana Paola, ya que a pesar de todos los problemas, me mostró que es posible voltearlos y ponerles una cara amable, no es fácil ni rápido, pero se puede.

...

¹No están organizados en un orden particular, solamente las “damas primero”.

²No es su nombre real, pero si lo escribo como Carlos Lopez, nadie sabe quien es.

Tabla de Contenido

Agradecimientos	IV
Índice de Figuras	VII
Índice de Tablas	IX
1. Introducción	1
2. BPEL	5
2.1. Introducción	5
2.2. Conceptos Básicos	6
2.2.1. Elementos Básicos	6
2.2.1.1. Variables	6
2.2.1.2. Colaboradores	6
2.2.2. Actividades	6
2.2.2.1. Actividades Estructuradoras	7
2.2.2.2. Actividades de Interacción	7
2.2.2.3. Actividades Básicas	7
2.3. Conceptos Avanzados	7
2.3.1. Scope	8
2.3.2. Fault Handler	8
2.3.3. Compensation Handler	8
2.3.4. Correlation Sets	8
2.3.5. Event Handler	9
2.4. Implementaciones	9
2.4.1. ActiveVOS	9
2.4.2. Oracle BPEL Process Manager	12
2.4.3. Apache ODE	14
2.5. Extensiones	15
2.5.1. WS-HumanTask	15
2.5.1.1. Tarea Humana (<i>Human Task</i>)	15
2.5.1.2. Roles Humanos Genéricos	16
2.5.1.3. Notificaciones	17
2.5.2. BPEL4People	17
2.5.2.1. People Activity	18
3. Aspect-Oriented Programming	21

3.1.	Introducción	21
3.2.	AspectJ	21
3.2.1.	Modelo de <i>Join Points</i>	22
3.2.2.	Lenguaje de Puntos de Corte	22
3.2.3.	Lenguaje de <i>Advices</i>	22
3.2.4.	Tejido de Aspectos	23
3.3.	Interacción Entre Aspectos	24
3.3.1.	Interferencia Semántica Entre Aspectos	24
3.3.1.1.	Ejemplo	25
3.3.1.2.	Clasificación de Interferencias	26
3.4.	Propuestas para Resolución de Conflictos	27
3.4.1.	Especificar precedencia Aspectos	27
3.4.2.	Detección de Conflictos	29
3.4.2.1.	Etapa de Composición	30
3.4.2.2.	Etapa de Abstracción de Comportamiento de los Aspectos	30
3.4.2.3.	Etapa de Detección de Conflictos	31
4.	Aspect-Oriented Workflow Languages	33
4.1.	Introducción	33
4.2.	Problemas de Modularización de Lenguajes de Workflow	34
4.2.1.	Ejemplo	34
4.2.2.	Problemas de Modularización de Preocupaciones Transversales . .	35
4.2.2.1.	Recolección de Información	35
4.2.2.2.	Monitoreo de Tiempo de Ejecución de Actividades	35
4.2.3.	Problemas de Modularización de Cambios	36
4.3.	AOP en Contextos Workflow	37
4.3.1.	Modelo de <i>Join Points</i>	37
4.3.2.	Lenguaje de Puntos de Corte	37
4.3.3.	Lenguaje de <i>Advices</i>	38
4.3.4.	Tejido de Aspectos	38
4.3.4.1.	Transformación de Procesos	38
4.3.4.2.	Modificación del Motor para Verificación de Aspectos . .	39
4.4.	Lenguajes de BPEL Orientados Por Aspectos	39
4.4.1.	Padus	39
4.4.1.1.	Modelo de <i>Join Points</i>	39
4.4.1.2.	Lenguaje de Puntos de Corte	40
4.4.1.3.	Lenguaje de <i>Advices</i>	40
4.4.1.4.	Tejido de Aspectos	41
4.4.2.	AO4BPEL	41
4.4.2.1.	Modelo de <i>Join Points</i>	41
4.4.2.2.	Lenguaje de Puntos de Corte	41
4.4.2.3.	Lenguaje de <i>Advices</i>	42
4.4.2.4.	Tejido de Aspectos	42
4.5.	Interferencia de Aspectos en Contextos Workflow	44
4.5.1.	Cambios en las Políticas de Distribución	44
4.5.2.	Interferencia Entre los Aspectos	45

5. El Proyecto Cumbia	47
5.1. Aplicaciones Orientadas a Control	48
5.2. Metamodelos	48
5.3. Modelo de Objetos Abiertos (OO)	49
5.4. Modelos Ejecutables Cumbia	50
5.4.1. Estrategia de Composición de Modelos	50
6. Caffeine v2.0: Motor BPEL sobre Cumbia	51
6.1. Introducción	51
6.2. Elementos Presentes	51
6.2.1. Elementos de Inicio	52
6.2.1.1. Elemento <i>Process</i>	53
6.2.1.2. Elemento <i>StartingPick</i>	54
6.2.1.3. Elemento <i>StartingReceive</i>	55
6.2.2. Elementos de Interacción	56
6.2.2.1. Elemento <i>Pick</i>	56
6.2.2.2. Elemento <i>OnAlarm</i>	56
6.2.2.3. Elemento <i>OnMessage</i>	58
6.2.2.4. Elemento <i>Receive</i>	58
6.2.2.5. Elemento <i>Reply</i>	59
6.2.2.6. Elemento <i>Invoke</i>	60
6.2.3. Elementos Estructuradores	61
6.2.3.1. Elemento <i>Sequence</i>	61
6.2.3.2. Elemento <i>Flow</i>	61
6.2.3.3. Elemento <i>While</i>	61
6.2.3.4. Elemento <i>Condicional</i>	64
6.2.4. Instrucciones	64
6.2.4.1. Elemento <i>Exit</i>	64
6.2.4.2. Elemento <i>Empty</i>	67
6.2.4.3. Elemento <i>Wait</i>	67
6.2.4.4. Elemento <i>Assign</i>	68
6.3. Arquitectura	69
6.3.1. CumbiaOpenObjectsKernel	69
6.3.2. CaffeineEngine	70
6.3.3. CaffeineDeployer	73
6.4. Pruebas	74
6.4.1. Framework de Pruebas	74
6.5. Experimentación <i>Caffeine</i> Cliente Web	76
7. AspectCaffine: Extensión de Aspectos a Caffeine	79
7.1. Introducción	79
7.2. Modelo de <i>Join Points</i>	79
7.3. Lenguaje de Puntos de Corte	79
7.4. Lenguaje de <i>Advices</i>	82
7.4.1. Advice del Aspecto de Facturación	82
7.5. Tejido de Aspectos	83
7.6. Elementos	84

7.6.0.1. Elemento <i>Aspect</i>	85
7.6.0.2. Elemento <i>Transition Point</i>	85
7.6.0.3. Elemento <i>Advice</i>	86
7.6.0.4. Elemento <i>Instruction</i>	86
7.7. Manejo de Interferencias	88
7.7.1. Árbol de Recubrimiento	91
7.8. Pruebas	94
8. Conclusiones y Trabajo Futuro	97
8.1. Conclusiones	97
8.2. Trabajo Futuro	98
Bibliografía	99

Índice de Figuras

2.1. Constelaciones de BPEL4People	18
3.1. Sistema de reproducción de música	25
3.2. Diagrama de <i>deployment</i> mostrando los estereotipos para aspectos	28
3.3. Proceso de detección de conflictos	31
4.1. Ejemplo de un proceso workflow	34
4.2. Arquitectura de Padus	41
4.3. Estados de una actividad de AO4BPEL	43
4.4. Arquitectura del motor de AO4BPEL	43
4.5. Aspecto de facturación	45
4.6. Aspecto de verificación de ubicación	46
6.1. Elementos de inicio	53
6.2. Máquina de estados del elemento <i>Process</i>	53
6.3. Máquina de estados del elemento <i>StartingPick</i>	54
6.4. Máquina de estados del elemento <i>StartingReceive</i>	55
6.5. Elementos de interacción	56
6.6. Máquina de estados del elemento <i>Pick</i>	57
6.7. Máquina de estados del elemento <i>OnAlarm</i>	57
6.8. Máquina de estados del elemento <i>onMessage</i>	58
6.9. Máquina de estados del elemento <i>Receive</i>	59
6.10. Máquina de estados del elemento <i>Reply</i>	59
6.11. Máquina de estados del elemento <i>Invoke</i>	60
6.12. Elementos estructuradores	61
6.13. Máquina de estados del elemento <i>Sequence</i>	62
6.14. Máquina de estados del elemento <i>Flow</i>	63
6.15. Máquina de estados del elemento <i>While</i>	64
6.16. Máquina de estados del comportamiento <i>Condicional</i>	65
6.17. Instrucciones	66
6.18. Máquina de estados del elemento <i>Exit</i>	66
6.19. Máquina de estados del elemento <i>Empty</i>	67
6.20. Máquina de estados del elemento <i>Wait</i>	67
6.21. Máquina de estados del elemento <i>Assign</i>	68
6.22. Arquitectura de Caffeine	69
6.23. Arquitectura de CaffeineEngine	71
6.24. Arquitectura del CaffeineDeployer	73
6.25. Vista en forma de árbol de los elementos de un proceso	77

6.26. Detalles de una actividad del proceso	78
7.1. Elementos de <i>AspectCaffeine</i>	84
7.2. Máquina de estados del elemento <i>Aspect</i>	85
7.3. Máquina de estados del elemento <i>Transition Point</i>	86
7.4. Máquina de estados del elemento <i>Advice</i>	87
7.5. Máquina de estados del elemento <i>Instruction</i>	88
7.6. Grafo únicamente para el <i>advice1</i>	89
7.7. Grafo únicamente para el <i>advice11</i>	90
7.8. Grafo únicamente para el <i>advice5</i>	90
7.9. Grafo únicamente para el <i>advice7</i>	91
7.10. Grafo únicamente para el <i>advice4</i>	92
7.11. Grafo completo	92
7.12. Primer paso del algoritmo para obtener el árbol de recubrimiento	93
7.13. Segundo paso del algoritmo para obtener el árbol de recubrimiento	93
7.14. Continuar agregando los vértices sucesores como hijos	94
7.15. Árbol terminado resaltando una rama con todos los <i>advices</i>	95

Índice de Tablas

4.1. Modelo de join points en Padus	40
6.1. Elementos de la Capa 1	52
6.2. Estados del Elemento <i>Process</i>	54
6.3. Acciones del elemento <i>Process</i>	54
6.4. Estados del Elemento <i>StartingPick</i>	55
6.5. Acciones del elemento <i>StartingPick</i>	55
6.6. Estados del Elemento <i>StartingReceive</i>	55
6.7. Acciones del elemento <i>StartingReceive</i>	56
6.8. Estados del Elemento <i>Pick</i>	57
6.9. Acciones del elemento <i>Pick</i>	57
6.10. Estados del Elemento <i>onAlarm</i>	58
6.11. Acciones del elemento <i>onAlarm</i>	58
6.12. Estados del Elemento <i>onMessage</i>	58
6.13. Acciones del elemento <i>onMessage</i>	59
6.14. Estados del Elemento <i>Receive</i>	59
6.15. Acciones del elemento <i>Receive</i>	59
6.16. Estados del Elemento <i>Reply</i>	60
6.17. Acciones del elemento <i>Reply</i>	60
6.18. Estados del Elemento <i>Invoke</i>	60
6.19. Acciones del elemento <i>Invoke</i>	61
6.20. Estados del Elemento <i>Sequence</i>	62
6.21. Acciones del elemento <i>Sequence</i>	62
6.22. Estados del Elemento <i>Flow</i>	62
6.23. Acciones del elemento <i>Flow</i>	63
6.24. Estados del Elemento <i>While</i>	63
6.25. Acciones del elemento <i>While</i>	64
6.26. Estados del Elemento <i>Condicional</i>	65
6.27. Acciones del elemento <i>Condicional</i>	66
6.28. Estados del Elemento <i>Exit</i>	67
6.29. Acciones del elemento <i>Exit</i>	67
6.30. Estados del Elemento <i>Empty</i>	67
6.31. Estados del Elemento <i>Wait</i>	68
6.32. Acciones del elemento <i>Wait</i>	68
6.33. Estados del Elemento <i>Assign</i>	68
6.34. Acciones del elemento <i>Assign</i>	69
7.1. Estados del Elemento <i>Aspect</i>	85

7.2. Acciones del elemento <i>Aspect</i>	85
7.3. Estados del Elemento <i>TransitionPoint</i>	86
7.4. Acciones del elemento <i>Transition Point</i>	86
7.5. Estados del Elemento <i>Advice</i>	87
7.6. Acciones del elemento <i>Advice</i>	87
7.7. Estados del Elemento <i>Instruction</i>	87
7.8. Acciones del elemento <i>Instruction</i>	87

Capítulo 1

Introducción

Los inicios de los sistemas de *workflow* se remontan a los años setenta, donde las organizaciones se dieron cuenta de la necesidad de automatizar los procesos empresariales y la ventaja que tenía hacerlo usando herramientas informáticas. Con el transcurrir del tiempo, se han identificado otros contextos donde la necesidad de automatizar procesos se ha hecho evidente. Por ejemplo, contextos donde es necesario hacer orquestación de servicios distribuidos, integración de aplicaciones para procesos de negocio, eLearning y el desarrollo de software geográficamente distribuido, entre otros.

Otro factor que ha sido determinante para la evolución de los sistemas de *workflow* ha sido la introducción de nuevas tecnologías, uno de los ejemplos más representativos por su alto impacto es Internet. Al comienzo Internet fue utilizado principalmente como un ambiente donde publicar datos. Actualmente Internet ha evolucionado y ahora es el lugar donde no solo donde se puede acceder a la información de los sitios Web, sino también a un lugar donde se puede acceder a servicios.

Debido a que los contextos tienen necesidades diferentes entre sí, cada uno de los lenguajes de *workflow* ha evolucionado de manera diferente, para poder solucionar los problemas específicos que cada contexto presenta. Es por eso que, por ejemplo, en el contexto empresarial uno de los lenguajes existentes es BPMN (*Business Process Modeling Language*), éste es un lenguaje gráfico que permite definir varios conceptos que manejan las organizaciones. El principal objetivo de este lenguaje es que sea de fácil lectura, de tal manera que las personas involucradas, ya sea con el diseño o la ejecución del proceso, no tengan que invertir demasiado tiempo aprendiendo el lenguaje. Para el contexto educativo o eLearning existe IMS-LD, el cual es un metamodelo diseñado para poder expresar procesos de enseñanza de tal manera que sea posible representar todos los modelos pedagógicos. Para la orquestación de servicios distribuidos, existe BPEL (*Business Process Execution Language*), el cual permite usar, componer y coordinar Web services.

A pesar que se pueden encontrar múltiples lenguajes para múltiples contextos, todos tienen en común que los problemas se modelan ordenando y sincronizando la ejecución de un conjunto de recursos o elementos para lograr un objetivo en un tiempo específico. Esto se conoce como el componente de control de un sistema de *workflow*.

Los sistemas de *workflow* tienen asociado al lenguaje una serie de herramientas como editores, motores y clientes. Los editores permiten realizar el diseño o la especificación de cada proceso para un lenguaje determinado. Los motores son los encargados de interpretar el componente de control del lenguaje, es decir, son los encargados de materializar los procesos para su ejecución. Por último, los clientes que permiten a los usuarios de los procesos interactuar con los elementos o con la solución del problema.

Algunas de las características de los sistemas de *workflow* hacen que las especificaciones de los procesos sean rígidas y muy costosas de modificar. Por ejemplo, las especificaciones de un *workflow* permiten especificar de un proceso el flujo de control, el flujo de datos, aspectos organizacionales y de tecnología. Además, a veces se asocian los conceptos de procesos con los conceptos de líneas de producción, donde las instancias del proceso casi nunca varían. Es decir, se piensa que la definición del proceso es estática, lo cual prueba ser una desventaja en ambientes cambiantes donde se desenvuelven las organizaciones.

Dentro de las deficiencias de los sistemas de *workflow* se encuentran la falta de soporte a los comportamientos transversales (*crosscutting concerns*), es decir que los lenguajes no ofrecen elementos necesarios para implementar requerimientos que afectan transversalmente los procesos, tales como monitoreo de actividades, recolección de datos, métricas, etc. Actualmente, para poder implementar estos cambios, es necesario modificar la especificación del proceso, lo que causa que no exista una clara separación clara entre los elementos que componen el proceso y los elementos que soportan los comportamientos transversales. Los sistemas de *workflow* tampoco proveen la modularidad necesaria para que los elementos que soportan los comportamientos transversales puedan ser activados o desactivados durante la ejecución del proceso. Otra desventaja identificada, es que pierde el control de los cambios de las especificaciones de los procesos, lo que implica que no es posible tener una historia de los cambios realizados.

La falta de “modularización” de los comportamientos transversales, es un problema que ya ha habido sido identificado en los lenguajes de programación. Como solución a esto, se planteó el uso de AOP (*Aspect-Oriented Programming*). AOP soporta una descomposición entre los comportamientos transversales y la lógica de negocio, proveyendo nuevos elementos programáticos, llamados aspectos.

Sin embargo, con el tiempo han surgido limitaciones del uso de AOP. Una de éstas limitaciones es que los aspectos no son ortogonales, es decir, que es posible que al colocar varios aspectos correctamente implementados en un mismo punto, causen conflictos e interacciones inesperadas, esto se conoce como interferencia entre aspectos.

El objetivo de este trabajo de tesis es proponer una solución a los problemas de modularidad de los comportamientos transversales en sistemas de *workflow* junto con los problemas de interferencia de aspectos, utilizando a BPEL como lenguaje de *workflow* enmarcado en el proyecto Cumbia.

Éste documento se encuentra organizado en 9 capítulos, de la siguiente manera: En el segundo capítulo se hace una descripción del lenguaje BPEL. Se habla específicamente de los elementos que lo componen, implementaciones actuales y extensiones existentes. En el tercer capítulo se habla acerca de AOP, describiendo los elementos que lo componen, implementaciones, el problema de la interferencia entre aspectos y como ha sido abordado por implementaciones existentes. El cuarto capítulo hace una descripción de lenguajes de *workflow* orientados por aspectos, haciendo un paralelo con los lenguajes de programación orientados por aspectos, específicamente el caso de AspectJ. Se aborda el problema de interferencia entre aspectos en un contexto de *workflow* a través de un ejemplo. El quinto capítulo describe el proyecto Cumbia, los metamodelos ejecutables extensibles, el modelo de objetos abiertos y la estrategia de composición entre modelos. El sexto capítulo habla acerca de Caffeine, el nuevo motor de BPEL desarrollado sobre Cumbia. El séptimo capítulo trata acerca de AspectCaffeine, el motor de aspectos desarrollado para BPEL. El octavo capítulo presenta las conclusiones y posibles trabajos futuros.

Capítulo 2

BPEL

2.1. Introducción

BPEL es el acrónimo para *Business Process Execution Language*. Es un lenguaje de composición, orquestación y coordinación de Web services orientado a procesos *workflow*. El objetivo principal de BPEL es estandarizar la definición del flujo de los procesos de negocio, de tal manera que las compañías puedan entenderse en un ambiente de tecnologías heterogéneas. BPEL es un lenguaje recursivo, en otras palabras, la composición resultante de los Web services es un nuevo Web service.

La primera especificación de BPEL (originalmente llamada *Business Process Execution Language for Web Services* o *BPEL4WS*) fue publicada en julio de 2002, como resultado del trabajo conjunto entre Microsoft, IBM y BEA para combinar dos lenguajes de composición existentes *WSFL* (*Web Service Flow Language*) de IBM y *XLANG* de Microsoft. Luego, en mayo de 2003 se lanzó la versión 1.1 con contribuciones de otras empresas como SAP y Siebel Systems. Además de esto, la especificación fue presentada a un comité técnico de OASIS para que se convirtiera en un estándar oficial. En abril de 2007 se aprobó la siguiente versión, llamada *WS-BPEL 2.0*. Este comité contó con la colaboración de más de 37 representantes de diferentes organizaciones como Active Endpoints, Adobe Systems, BEA Systems, Booz Allen Hamilton, EDS, HP, Hitachi, IBM, IONA, Microsoft, NEC, Nortel, Oracle, Red Hat, Rogue Wave, SAP, Sun Microsystems, TIBCO, WebMethods[1].

Un proceso de *workflow* BPEL consiste de actividades que interactúan con los Web services que participan en la composición, junto con actividades donde se especifica el flujo de control, el flujo de datos a otros Web services y el manejo de estos datos. A continuación se presenta una explicación breve de los elementos que se manejan en

BPEL, no se espera que sea una explicación profunda de BPEL, para eso se puede consultar la especificación[2].

2.2. Conceptos Básicos

Los conceptos básicos de BPEL pueden ser divididos en elementos básicos y actividades.

2.2.1. Elementos Básicos

Los elementos básicos se dividen en variables y colaboradores.

2.2.1.1. Variables

En BPEL los datos del *workflow* son leídos y escritos en variables de tipo XML. En éstas variables se guardan los mensajes que han sido recibidos de un colaborador, las actividades que van a ser enviadas a algún colaborador y las variables que son utilizadas para mantener los datos necesitados para mantener el estado del proceso y nunca van a ser intercambiados con los colaboradores[2].

2.2.1.2. Colaboradores

Representan las partes con quien el proceso BPEL interactúa, como los clientes y los Web services que son llamados por el proceso. El proceso y los colaboradores se comunican a través de un *partner link*. Éste es simplemente una instancia de un conector tipificado, que va a conectar dos tipos de un puerto WSDL. Dentro de la especificación de un *partner link* se define que es lo que el proceso BPEL provee al colaborador y que es lo que el proceso espera del colaborador, es decir, un *partner link* puede ser considerado como un canal en una conversación *peer-to-peer* entre un proceso y el colaborador[3].

2.2.2. Actividades

La especificación del lenguaje hace una diferenciación entre las actividades básicas, las actividades estructuradoras y actividades de interacción.

2.2.2.1. Actividades Estructuradoras

Las actividades estructuradoras describen el orden en el que se van a ejecutar un conjunto de actividades. Describen cómo se expresa el control del proceso, manejo de eventos externos y la coordinación del intercambio de mensajes entre los participantes del proceso. Con las actividades estructuradoras se pueden expresar varios patrones de control:

- La secuencialidad se puede definir usando las actividades *sequence*, *flow*, *if*, *while*, *repeatUntil* y una variación de *forEach*.
- La concurrencia de actividades puede ser definida usando *flow* y *forEach*.
- Escogencia de camino en ejecución ya sea por eventos externos o internos está soportada por la actividad *pick*.

2.2.2.2. Actividades de Interacción

Las actividades de interacción definen cómo se va a realizar el intercambio de mensajes entre los participantes del proceso. La actividad *receive* está encargada de bloquear el proceso y esperar que se reciba el mensaje del colaborador definido. La actividad *reply* está encargada de enviar un mensaje a alguno de los colaboradores del proceso sin esperar respuesta. La actividad *invoke* es utilizada para llamar Web services del colaborador designado. Ésta actividad puede ser asíncrona o síncrona, es decir, si es definida como síncrona bloqueara el proceso hasta recibir una respuesta del colaborador.

2.2.2.3. Actividades Básicas

Éstas actividades tienen diferentes propósitos. Por ejemplo, es posible terminar inmediatamente un proceso (*exit*). También se puede frenar el proceso por un tiempo determinado o mientras se alcanza un límite de tiempo (*wait*). Igualmente para poder copiar datos de una variable a otra o asignar nuevos datos a las variables (*assign*). También existe una actividad que no hace nada (*empty*), que es utilizada cuando se quiere atrapar una falla y no hacer nada o si se necesita un punto de sincronización en un *flow*.

2.3. Conceptos Avanzados

BPEL define elementos con los cuales se puede hacer manejo de errores, elementos para compensar actividades terminadas, elementos para poder diferenciar instancias de

proceso para que los mensajes lleguen a la instancia correcta, elementos para poder definir un contexto para el manejo de error y elementos para reaccionar a eventos externos.

2.3.1. Scope

El *scope* es un elemento que provee un contexto que va a afectar la manera cómo van a ser ejecutadas las actividades que contiene. En éste contexto se incluyen variables, *partner links*, *message exchanges*, *correlation sets*, *event handlers*, *fault handlers*, un *compensation handler* y un *termination handler*.

2.3.2. Fault Handler

Durante la ejecución de un proceso pueden ocurrir errores que deben ser manejados dentro del proceso. El *fault handler* está diseñado para qué se pueda deshacer parte del trabajo realizado que causó la falla. Dentro de un *fault handler* se especifica una actividad que será ejecutada si se lanza un error durante la ejecución del *scope* donde está definido. En caso de que no se haya determinado un *fault handler* para la falla encontrada, está será lanzada al *scope* padre hasta que se encuentre un *fault handler* que pueda manejar la falla o hasta que el proceso termine.

2.3.3. Compensation Handler

Un *compensation handler* permite definir dentro de un *scope* un conjunto de actividades que pueden ser reversibles. Esto es útil sí, por ejemplo, se tiene un proceso BPEL de larga duración, donde todas las actividades no pueden ser terminadas de manera atómica. En el caso que ocurra una falla es posible deshacer ciertas actividades que fueron ejecutadas hasta ese punto.

2.3.4. Correlation Sets

Los *correlation sets* son un mecanismo que permite identificar a qué instancia de un proceso BPEL le corresponde un mensaje SOAP recibido, debido a que múltiples instancias de un proceso BPEL pueden estar en ejecución en un momento dado.

2.3.5. Event Handler

Los *event handlers* son elementos asociados a un *scope* que permiten ejecutar una actividad especificada cuando ocurre cierto evento, como recibir un mensaje o qué se dispare una alarma.

2.4. Implementaciones

En esta sección se van a tratar tres de los motores de *workflow* actuales: ActiveVOS, Oracle BPEL Process Manager y Apache ODE.

2.4.1. ActiveVOS

ActiveVOS es un producto de la compañía Active Endpoints. Éste producto es promocionado como un sistema visual de orquestación, donde una sola herramienta permite automatizar los procesos de negocio, la colaboración de diferentes tipos de usuarios organizacionales, controlar los procesos de la organización y adaptar procesos en ejecución. La herramienta está dividida en ActiveVOS Designer y ActiveVOS Server. Entre otras, ActiveVOS permite:

Diseño de Procesos

ActiveVOS Designer es un ambiente de desarrollo gráfico basado en Eclipse, que permite diseñar los procesos de negocio usando directamente WS-BPEL 2.0 o BPMN. Además, permite importar procesos existentes realizados en Visio o XPDL, para luego ser traducidos a WS-BPEL 2.0. Así mismo, permite integrar conceptos de BPEL4People de tal manera que es posible orquestar actividades asignadas a un ser humano dentro de un proceso[4].

Deployment de Procesos

ActiveVOS Designer también permite hacer *deploy* directamente a ActiveVOS Server. Una de las ventajas que tiene utilizar éste producto, es que es posible subir varias versiones de un mismo proceso al servidor. Cuando esto ocurre, el servidor permite mantener las versiones anteriores de los procesos. Con respecto a cómo se van a manejar las instancias ya existentes de un proceso que ha cambiado, el servidor tiene tres políticas. La primera de ellas es que las instancias existentes pueden mantener su versión hasta que

terminen. La segunda política es que las instancias pueden migrar hacia el nuevo proceso. Y la última política es que las instancias existentes deben terminar. La herramienta también permite que las definiciones tengan un ciclo de vida, es decir, es posible definir una fecha a partir de la cual el proceso estará disponible y definir una fecha hasta cuando el proceso estará disponible[5].

El servidor permite también definir quién prestará el servicio de los colaboradores. En el momento de hacer *deploy*, se puede asignar a cada *partner link* si el servicio es un Web service o servicios Java, JMS (*Java Messaging System*), REST (*Representational State Transfer*), Ejb Wrappers, Java Wrappers o correo electrónico. Esta configuración también puede ser definida como estática o dinámica, es decir, los servicios pueden ser seleccionados en tiempo de ejecución, dependiendo de diferentes políticas configurables[5].

Monitoreo

ActiveVOS Server permite hacer monitoreo a través de la consola de los procesos activos, de las tareas del servidor. Con respecto a los procesos, la consola lista todos los procesos que están activos y es posible seleccionar cada uno de ellos para obtener la información que se encuentra en la definición del proceso y además su representación gráfica, igual como si se estuviera viendo desde ActiveVOS Designer. La consola también tiene la funcionalidad de poder monitorear específicamente la lista de elementos *onAlarm* que se encuentran activos. A su vez, también permite monitorear la cola de elementos *receive*, *onMessage* y *onEvent* que se encuentran esperando recibir un mensaje.

Con respecto al monitoreo de tareas, el servidor tiene la funcionalidad de permitir monitorear las tareas asociadas con una actividad realizada por un humano, ya sea en ejecución o completada. Contiene los roles genéricos humanos, como los dueños potenciales y los administradores que permiten a los usuarios trabajar en ellas. También tiene la funcionalidad de monitorear la cola de trabajo, la cual es una lista de grupos asociados con tareas que no se han asignado a un ser humano.

El monitoreo que se le puede hacer al servidor es obtener estadísticas como el tiempo para validar mensajes, eficiencia del cache por proceso, tiempo de instanciación de un proceso, entre otras. Los registros que el servidor generó al momento de hacer *deploy* de cada proceso también pueden ser vistos a través de la consola.

Reportes

ActiveVOS permite generar reportes sobre la información de los procesos o de las tareas. Los reportes que se pueden obtener de los procesos son[6]:

- Lista de los procesos fallidos.
- Gráfica con la cantidad de elementos *receive* que están esperando un mensaje.
- Gráfica con los procesos más instanciados en las últimas 24 horas.
- Gráfica con los procesos que más se encuentran en ejecución en las últimas 24 horas.

Los reportes que se pueden obtener de las tareas son:

- Lista con el resumen de la cantidad de tareas cerradas.
- Lista con el resumen de la cantidad de tareas abiertas.
- Lista con el resumen de la cantidad de tareas que han tenido algún problema.
- Gráfica con las tareas más instanciados en las últimas 24 horas.

ActiveVOS Designer también permite crear reportes a la medida para poder ser generados desde el servidor. Para hacer esto es necesario conocer tanto el modelo de datos del proceso y el modelo de datos de las tareas humanas definido por ActiveVOS[7]. Una vez se conocen los modelos de datos, ActiveVOS Designer permite seleccionar de varias plantillas la manera como se mostrará el reporte y el usuario es responsable de definir las consultas para poder obtener los datos para cada elemento que se va a reportar.

Simulación

Una de las características importantes de ActiveVOS Designer es que es posible evaluar el impacto que va a tener un nuevo proceso o los cambios sobre un proceso existente, a través de escenarios de simulación. En estos escenarios se definen archivos ejemplo que contienen la información de entrada de un proceso, permitiendo así analizar gráficamente, paso a paso, cómo se comporta del proceso de acuerdo a los datos proporcionados[8].

Pruebas

La herramienta también permite guardar las simulaciones realizadas para poder crear lo que ellos denominan *BPEL Unit Testing*. Éste es un procedimiento que puede ser utilizado para validar que la ejecución de las actividades *invoke*, *receive* y *reply* está funcionando correctamente[9], de esta manera es posible verificar que todos los componentes de un proceso estén funcionando.

Depuración

ActiveVOS Designer tiene la capacidad de conectarse directamente con el servidor que está corriendo la instancia de proceso y hacer depuración remota. Es posible colocar puntos de quiebre en las actividades que componen el proceso. Estos puntos de quiebre se colocan dentro de la parte gráfica del editor y al ejecutar una instancia, examinar toda la información que pasa a través de la actividad donde se colocó el punto de quiebre[8].

2.4.2. Oracle BPEL Process Manager

Ésta suite de herramientas permite el diseño y la ejecución de procesos definidos en BPEL 1.1. También tiene algunas herramientas adicionales para el despliegue, monitoreo y la administración de los procesos. OBPM corre sobre cualquier servidor J2EE. Entre otras, OBPM permite:

Diseño de Procesos

Dentro de la suite de herramientas contenidas dentro de OBPM se encuentra BPEL Designer. Ésta permite el desarrollo visual de procesos. Existen dos versiones: la de JDeveloper y un plugin para Eclipse. BPEL Designer no tiene que ser utilizado obligatoriamente con BPEL Server. La aplicación permite crear de manera gráfica las definiciones de los procesos BPEL, pero solamente para la versión 1.1 de BPEL. BPEL Designer permite definir los procesos y hacer *deploy* directamente al servidor[10].

Deployment de Procesos

El ambiente de ejecución donde se hace *deploy* de los procesos y donde estos son ejecutados se llama BPEL Server. Proporciona soporte para tecnologías de orquestación, particularmente *WS-Addressing* y *BPEL Compensation Transaction Model*. Permite el manejo de versiones de los procesos. También ofrece la hidratación/deshidratación de procesos y soporte para clustering. BPEL Server tiene un framework que es responsable de la comunicación de los procesos con los clientes y también es responsable de la comunicación de los procesos con los *partnerLinks*. Aunque la especificación de BPEL sólo habla de Web services como colaboradores, el servidor permite la conectividad usando otros protocolos además de SOAP, gracias al Web Services Invocation Framework (WSIF) de apache, el servidor puede comunicarse con colaboradores que están definidos usando otras tecnologías, como EJB, RMI, JMS, JCA y también correo electrónico.

Monitoreo

La información que se encuentra en el servidor puede ser accedida a través de BPEL Console. Ésta permite hacer *deploy*, depuración, manejo y administración de los procesos. Permite visualizar el flujo de los procesos y realizar auditoría sobre los mismos. A través de la consola también es posible realizar depuración de las instancias (tanto de las que ya terminaron como las que se encuentran en ejecución) ver el historial de los procesos[11].

Reportes

BPEL Server permite generar reportes sobre la información de los procesos o sobre sensores colocados en las tareas. Los reportes que se pueden obtener de los procesos son[12]:

- Gráfica con la cantidad de instancias creadas por proceso.
- Gráfica con la cantidad de instancias cerradas por proceso debido a un error.
- Gráfica con la cantidad de instancias completadas satisfactoriamente por proceso.
- Gráfica con la cantidad de instancias en ejecución por proceso.
- Gráfica con el tiempo mínimo de ejecución de una instancia por proceso.
- Gráfica con el tiempo máximo de ejecución de una instancia por proceso.
- Gráfica con el tiempo promedio de ejecución de una instancia por proceso.

Los reportes que se pueden obtener de los sensores son:

- Gráfica con el tiempo mínimo de ejecución de una tarea donde se colocó un sensor.
- Gráfica con el tiempo máximo de ejecución de una tarea donde se colocó un sensor.
- Gráfica con el tiempo promedio de ejecución de una tarea donde se colocó un sensor.

Pruebas

BPEL Server provee un mecanismo que permite que los usuarios automaticen las pruebas de los procesos. La herramienta permite emular servicios y colaboradores. De igual manera es posible definir *asserts* para verificar las acciones del proceso. Los resultados están integrados dentro de la consola, junto con reportes de Ant-JUnit[13].

BPEL Server permite crear instancias que van a ser utilizadas para hacer pruebas de rendimiento y de carga de los procesos. Con las estadísticas obtenidas de éstas pruebas, es posible identificar los cuellos de botella y mejorar el rendimiento de un proceso[14].

Depuración

La consola tiene una vista que permite ver gráficamente el proceso. En ésta vista, es posible ver el código de la definición de un proceso o acceder al contenido de las variables de una instancia en ejecución o terminada[14].

2.4.3. Apache ODE

Apache ODE (*Orchestration Director Engine*) es un motor de ejecución de procesos BPEL 2.0. Desarrollado por Apache, es de distribución gratuita y de código abierto bajo la licencia Apache 2.0.

Diseño de Procesos

Eclipse BPEL Designer es un plugin de Eclipse, que permite diseñar gráficamente los procesos de negocio usando la especificación 2.0 de BPEL[15].

Deployment de Procesos

El *deploy* de los procesos puede realizarse directamente desde Eclipse. Una vez se tiene el proceso diseñado, se define donde está corriendo el motor y la herramienta genera un descriptor donde es necesario definir donde se encuentran los servicios que va a utilizar el proceso[15]. Para el caso donde se quiere modificar un proceso existente, Apache ODE provee un sistema de versionamiento, sin embargo, no es posible migrar de las instancias existentes de la versión antigua a la nueva. La política de versionamiento permitirá la ejecución de las instancias creadas bajo el proceso viejo, mientras que las nuevas instancias serán creadas con la definición del proceso nuevo.

Pruebas

La manera como Apache ODE aborda las pruebas es que proveen un conjunto de escenarios de prueba que esperan que verifiquen todas las posibles configuraciones e

interacciones que se presentan en la ejecución de un proceso. El framework no está destinado a probar cada uno de los procesos desarrollados por los usuarios del motor, sino a demostrar que el motor funciona correctamente[15].

2.5. Extensiones

BPEL especifica el comportamiento de procesos de negocio mientras las actividades del proceso sean Web services. Sin embargo en muchos procesos existen actividades donde debe intervenir un humano, y estas interacciones humanas no son parte del dominio de BPEL. A pesar de la gran aceptación de Web services en aplicaciones de negocio distribuidas, la ausencia de interacciones humanas es un gran vacío para los procesos del mundo real. Para llenar este vacío, se crearon dos extensiones de BPEL llamadas WS-HumanTask[16] y BPEL4People[17]. Estas extensiones permiten la orquestación de Web services junto con la orquestación de actividades humanas.

2.5.1. WS-HumanTask

La especificación define nuevos elementos para que la interacción humana pueda ser integrada dentro de un proceso BPEL. A continuación se presenta un ejemplo[16] de un proceso con una tarea donde un humano debe interactuar con el proceso.

En un proceso de un préstamo bancario es imaginable que exista una tarea de aprobación del monto solicitado. Después que la información acerca del préstamo es recolectada y si el valor excede cierta cantidad, una persona debe manualmente evaluar si el préstamo es concedido o no. La invocación del servicio de aprobación por el proceso de negocio crea una instancia de la tarea. De esta manera la tarea aparecerá en la lista de tareas de los aprobadores. Uno de los aprobadores se hará cargo de la tarea, evaluando la información de la solicitud y completará la tarea aprobando o no el préstamo. El mensaje de salida de la tarea contendrá la evaluación del aprobador.

Para lograr esto, la especificación define tres conceptos básicos[16].

2.5.1.1. Tarea Humana (*Human Task*)

Estas tareas humanas permiten la integración de seres humanos en aplicaciones orientadas a servicios. Las tareas tienen dos interfaces. Una interfaz expone el servicio ofrecido por la tarea, por ejemplo, un servicio de aprobación en un proceso de un préstamo. La segunda interfaz permite que las personas interactúen con las tareas, por ejemplo, obtener las tareas que han sido asignadas a una persona para poder trabajar en ellas.

2.5.1.2. Roles Humanos Genéricos

Estos roles definen lo que una persona puede hacer con las tareas y las notificaciones. Los roles están clasificados en:

Iniciadores de Tareas (*Task Initiator*)

Es la persona que crea una instancia de una tarea. Dependiendo de cómo se crea una tarea el iniciador es opcional.

Interesados en la Tarea (*Task Stakeholder*)

Son las personas responsables de la instancia de una tarea y su resultado. Un *task stakeholder* puede influenciar el progreso de una tarea, por ejemplo, reenviando una tarea o siendo un observador de su estado. También puede realizar acciones administrativas sobre la instancia de una tarea y sus notificaciones, como resolver plazos vencidos.

Dueños Potenciales (*Potential Owners*)

Son las personas quienes reciben la tarea para que la puedan reclamar y completar. Un dueño potencial de una tarea se convierte en el dueño actual cuando reclama la tarea. Los dueños potenciales de una tarea pueden influenciar el progreso de la tarea antes de reclamarla, por ejemplo, cambiando la prioridad de la tarea.

Dueños Excluidos (*Excluded Owners*)

Un dueño excluido no puede ser ni dueño potencial ni el dueño actual de una tarea.

Dueño Real (*Actual Owner*)

Es la persona que realmente va a realizar la tarea, puede realizar acciones sobre ella como negar una reclamación, reenviar la tarea a revisión o negar un préstamo. Una tarea solo puede tener un dueño real.

Administradores de Negocio (*Business Administrators*)

Los administradores de tareas cumplen el mismo rol que los *task stakeholders* pero a nivel de tipo de tarea. Los administradores de proceso también pueden observar el progreso de las notificaciones.

Receptores de Notificaciones (*Notification Recipients*)

Personas que reciben una notificación, por ejemplo, cuando un plazo se vence o cuando se llega a un hito.

2.5.1.3. Notificaciones

Las notificaciones son mensajes simples, utilizados para informar a una persona o a un grupo de personas que algún evento de negocio ha ocurrido. También son utilizadas cuando es necesario escalar eventos, como una tarea que no ha empezado o una que se ha vencido. Las notificaciones son interacciones simples que no bloquean el proceso. Los receptores de una notificación pueden ser receptores y de manera opcional uno o más administradores de negocio.

2.5.2. BPEL4People

BPEL4People es otra extensión de BPEL, que también extiende a WS-HumanTask, donde se hace una definición más amplia de los roles y de la manera como los humanos pueden interactuar con los procesos[17]. BPEL4People utiliza muchos de los conceptos de WS-HumanTask y define conceptos nuevos[17].

Roles Humanos Genéricos

Estos roles complementan los roles definidos en WS-HumanTask y definen que es lo que una persona o grupo de personas pueden hacer con una instancia de proceso.

Iniciador de Proceso (*Process Initiator*)

EL iniciador de proceso es una persona asociada con ejecutar la instancia de proceso en el momento de su creación. El iniciador generalmente es seleccionado automáticamente por el sistema, pero también puede ser definido realizando la asignación a una persona específicamente.

Interesados en el Proceso (*Process Stakeholders*)

Los interesados en el proceso son las personas que pueden influir el progreso de una instancia de proceso, por ejemplo, reenviando una tarea o siendo un observador del progreso de una instancia de proceso. Cada instancia de proceso está relacionada con un interesado en el proceso. En caso que no se haya definido un interesado, el iniciador de proceso será el interesado.

Administradores de Negocio (*Business Administrators*)

Los administradores de negocio son las personas que tienen permitido tomar acciones administrativas de un proceso, como resolver incumplimiento de plazos. La diferencia con el *process stakeholder* es que los administradores de negocios están interesados en todas las instancias del proceso. En caso que no se hayan especificado administradores de negocio, los interesados del proceso serán los administradores de negocio.

2.5.2.1. People Activity

Para poder modelar las interacciones humanas dentro del proceso se define el elemento *people activity*. Cada *people activity* está relacionada con un elemento *inline task* de tipo WS-HumanTask.

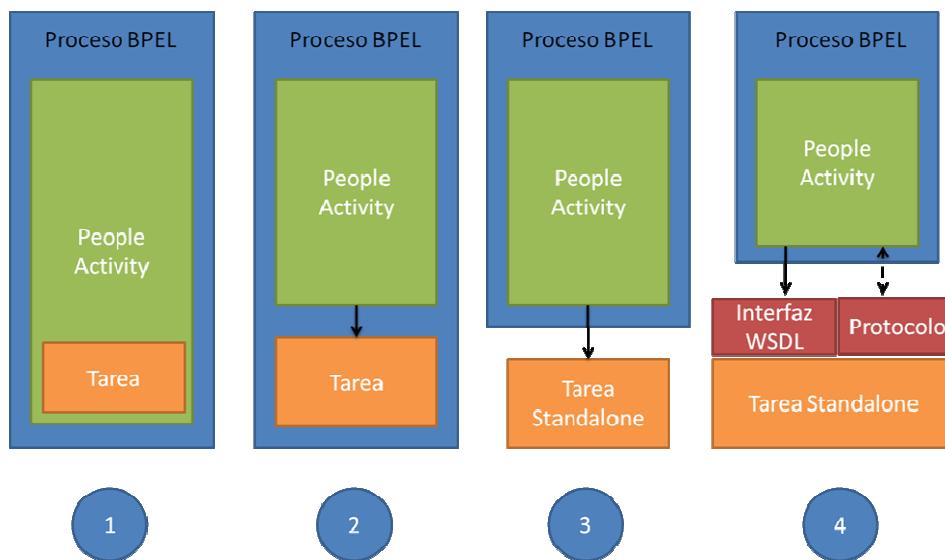


FIGURA 2.1: Constelaciones de BPEL4People.

La manera de cómo se relacionan la *people activity* con la *inline task* está categorizada de cuatro formas diferentes, llamadas constelaciones (ver figura 2.1). La primera constelación indica que la *inline task* realizada por un humano está definida dentro de una *people activity* y sólo puede ser utilizada por esa *people activity*, es decir, que el alcance de esa *inline task* es la *people activity*. Para la segunda constelación, la *inline task* a ser completada por un humano se encuentra fuera de la *people activity*, pero dentro del alcance de un *scope* o del proceso, lo que implica que una sola *inline task* puede ser llamada desde diferentes *people activity*. La tercera constelación es muy similar a la segunda, pero con la diferencia que la *inline task* se encuentra definida fuera del proceso (la tarea es *standalone*) pero dentro del mismo ambiente, es por esto que no tiene acceso al contexto del proceso, pero puede ser llamada desde cualquier *people activity*. La última constelación es muy similar a la tercera, pero con la diferencia que la *inline task* a realizar se encuentra fuera del ambiente del proceso. Para que el proceso pueda llamar está *inline task* es necesario que la invocación sea a través de Web services, utilizando un protocolo predefinido para que el proceso pueda controlar el ciclo de vida de la *inline task*.

Capítulo 3

Aspect-Oriented Programming

3.1. Introducción

Hay unidades funcionales de los sistemas que no pueden ser aisladas usando programación orientada por objetos, porque su funcionalidad es transversal a múltiples componentes. Es por eso que se desarrolló AOP o *Aspect-Oriented Programming*[18]. Con éste paradigma es posible resolver los problemas de modularización de las preocupaciones transversales, lo que resulta en código más fácil de desarrollar, mantener, aumentar su potencial de reutilización y de reducir la cantidad de código enredado y repetido, además de reducir los costos de introducir nuevo comportamiento en la aplicación base[19].

Existen varias aproximaciones de cómo se deben modelar los sistemas de programación orientada por aspectos, cinco¹ de ellos fueron descritos en [22] basándose en herramientas existentes, el más utilizado de ellos es el modelo de AspectJ.

3.2. AspectJ

El modelo de AspectJ define cuatro conceptos básicos que son necesarios para definir un aspecto[19].

¹Modelo de Pointcuts y Advices basado en AspectJ. Modelo de Recorridos basado en la ley de Demeter o el principio del menor conocimiento[20]. Modelo de Composición de Clases basado en la programación orientada por temas (*Subject-Oriented Programming*[21]). Modelo de Clases Abiertas. Modelo de Navegador Basado en Consultas basado en QJBrower.

3.2.1. Modelo de *Join Points*

El modelo de *join points* define los lugares donde los *advices* van a ser ubicados en la ejecución de la aplicación base. Son puntos bien definidos, que proveen un marco de referencia común que hace posible que la ejecución del código del programa y la ejecución del código del *advice* sea coordinada. Debido a que AspectJ es una extensión de Java, el modelo de *join points* define puntos dentro de la ejecución de un programa, como llamados a métodos, llamados a constructores, escritura y lectura de atributos, etc.

3.2.2. Lenguaje de Puntos de Corte

El lenguaje de puntos de corte es utilizado para seleccionar un conjunto específico de *join points*. Los puntos de corte dentro de AspectJ se seleccionan utilizando designadores de puntos de corte, los cuales son predicados sobre los *join points*. Estos designadores son un conjunto de predicados predefinidos, por ejemplo *call* que selecciona los llamados de métodos como *join points*. Junto con estos designadores, el lenguaje también selecciona los *join points* gracias a las características sus características. Por ejemplo, es posible seleccionar puntos específicos dentro de la ejecución gracias a los tipos de parámetros que tenga un método o tipos de retorno. De acuerdo a [23], definir el lenguaje de ésta manera tiene tres limitaciones. La primera es que no provee un mecanismo de propósito general para relacionar diferentes *join points*. La segunda es que el usuario no puede definir sus propios designadores, es decir el lenguaje no es un lenguaje extensible. La tercera es que no soportan los puntos de corte semánticos, es decir, especifica cómo están implementados los *join points* más no lo que representan.

3.2.3. Lenguaje de *Advices*

El lenguaje de *advices* define la funcionalidad que debe ser ejecutada en los *join points* específicos. El *advice* es un fragmento de código que es ejecutado cuando se alcanza un *join point* identificado por el respectivo punto de corte. El lenguaje de *advices* generalmente en el mismo lenguaje que la aplicación base. El *advice* puede ser ejecutado antes, después, o en vez de los *join points* que son seleccionados por el punto de corte. En AspectJ corresponde a los tipos de *advices before, after* o *around*.

AspectJ también provee instrucciones que permiten al código definido en el *advice* a obtener información con respecto al contexto donde se está ejecutando. Por ejemplo, puede obtener quien es el que hace el llamado de un método, los parámetros que tiene ese llamado, etc.

```

1 public aspect Logging
{
3 //Donde?
4 pointcut loggableMethods(Object o): call(_ bar(..)) && this(o);
5 //Cuando?
6 before(Object o): loggableMethods(o)
7 {
8     //Qué?
9     System.out.println("Llamado el método bar desde el objeto " + o.
10        toString());
11 }

```

CÓDIGO 3.1: Ejemplo de AspectJ.

En el código mostrado 3.1 se ejemplifica un aspecto de monitoreo usando AspectJ. Este aspecto define un punto de corte llamado *loggableMethods*, el cuál especifica donde se debe agregar la funcionalidad de ésta preocupación transversal a la aplicación base. En éste ejemplo los *join points* son los métodos llamados *bar*, sin importar en que clase están definidos, su tipo de retorno o su lista de parámetros. El *advice* también define cuándo y cuál debe ser el comportamiento que debe ser ejecutado en los *join points* escogidos. La lógica del *advice* es imprimir un mensaje antes de que se ejecuten los *join points* asociados. Dentro de la lógica del *advice* también se está haciendo uso de las instrucciones para poder obtener información del contexto donde se está ejecutando el *advice*, específicamente, poder llamar el método *toString()* sobre el objeto que va a llamar el método *bar*.

3.2.4. Tejido de Aspectos

El tejido de aspectos es la parte de la implementación que debe asegurar que el código de los *advices* y el de la aplicación base se ejecuten de manera coordinada, en los *join points* que fueron definidos para cada aspecto. En AspectJ[24] el proceso de tejido de aspectos comienza en el compilador de AspectJ, una extensión al compilador de Java. El compilador de AspectJ está dividido en dos partes, el *front-end* y el *back-end*. El *front-end* recibe como entrada el código fuente del aspecto y el código fuente de la aplicación base para ser compilados. El código del *advice* es compilado como un método de Java normal, con los mismos parámetros con los que fue implementado, más uno extra que indica qué es la declaración de un *advice*, para guardar la información del punto de corte que es referenciado por el *advice* y transmitir información al *back-end* del compilador. El *back-end* del compilador instrumenta el código de la aplicación base con el código de los *advices*. El *back-end* primero evalúa en el *bytecode* todos los posibles lugares donde se puede instrumentar un *advice*. Estos puntos se conocen como la sombra estática de los

join points. Luego, el compilador compara si el punto de corte de cada *advice* corresponde a esa sombra estática, en caso de hacerlo, inserta una llamada al método del *advice*.

3.3. Interacción Entre Aspectos

Casi todos los lenguajes de AOP permiten componer aspectos independientes en un mismo *join point*. Esto fue denominado *shared join point* en [25]. Ésta característica puede causar que se genere comportamiento imprevisto, causando interacciones semánticas inesperadas.

Las interacciones entre aspectos pueden ser clasificadas de acuerdo al comportamiento que se genera entre los aspectos y la aplicación base. Estas interacciones pueden ser clasificadas en cuatro grupos:

Exclusión Mutua - Si existen dos aspectos que implementen funcionalidades o algoritmos similares, puede darse el caso que solo uno de esos aspectos pueda ser utilizado. No existe la posibilidad de relacionar los dos aspectos porque no se complementan, solo uno de ellos puede ser utilizado.

Dependencia - La dependencia ocurre cuando un aspecto específicamente necesita otro aspecto y por eso depende de él. Una dependencia no resulta en comportamiento errado o inesperado, mientras se garantice que el aspecto con él que se tiene dependencia existe sin cambios.

Refuerzo - El refuerzo se presenta cuando un aspecto influye positivamente en el correcto funcionamiento de otro aspecto. Cuando existe el refuerzo entre dos aspectos, funcionalidades adicionales son ofrecidas.

Conflicto - Representan la interferencia semántica. Un aspecto correctamente implementado no funciona de manera esperada cuando es compuesto con otros aspectos en un *shared join point* o afecta el correcto funcionamiento de los demás aspectos tejidos en el mismo *shared join point*.

El trabajo de ésta tesis se concentra más en las interacciones de conflicto.

3.3.1. Interferencia Semántica Entre Aspectos

Para ilustrar los conflictos de interferencia semántica, se presenta un ejemplo en AspectJ de dos preocupaciones transversales, tomado de [26].

3.3.1.1. Ejemplo

La figura 3.1 muestra un sistema de reproductor de música. Si se selecciona una canción, a través de la interfaz del reproductor (*JukeBoxUI*), se llama el método *play(Song)* de la clase *JukeBox*, pasándole como parámetro la canción que se quiere escuchar. Éste método a su vez llama a *play(String)* en la clase *Player*, quien es la interfaz con el sistema de audio.

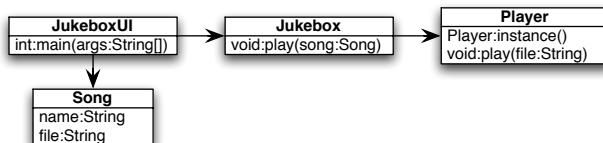


FIGURA 3.1: Sistema de reproducción de música.

Si se agregan dos aspectos, uno de ellos dice que se debe revisar si el usuario tiene suficiente dinero, de ser así, se tiene que retirar cierta cantidad cada vez que se llame el método *play*.

```

1 aspect CreditsAspect
{
3   void around () : call ( public void Jukebox . play ( Song ) )
{
5     if( Credits . instance () . enoughCredits () )
{
7       Credits . instance () . withdraw ();
9       proceed ();
}
11   else
{
13     throw new NotEnoughCreditsException ();
}
15 }
}

```

CÓDIGO 3.2: Aspecto de Cobrar.

El segundo aspecto pone en cola las canciones y inmediatamente retorna el control a quien lo llamó. En el *advice* se llama el método *enqueue(Song)* en la instancia del objeto *Playlist* que es un *singleton*. Éste método pone el objeto *Song* en la cola y comienza a tocar las canciones hasta que esté desocupada.

```

1 aspect PlaylistAspect
{
3   void around ( Song song ): call ( public void Jukebox . play ( Song ) ) &&
args ( song )
}

```

```

5   {
6     Playlist . instance () . enqueue ( song );
7     return ;
8   }

```

CÓDIGO 3.3: Aspecto de colocar en cola las canciones.

Los dos aspectos se van a tejer en el mismo punto de corte, la llamada del método *play(Song)* de la clase *Jukebox*. Al no declarar de ninguna manera cuál de ellos es primero, solo se pueden ordenar de dos maneras. De la primera manera, el aspecto que cobra se aplica primero y luego el aspecto que pone en cola las canciones. De la otra manera, primero se aplica el aspecto que pone en cola las canciones y luego el aspecto que cobra. Sin embargo, al ejecutar primero el aspecto que pone en cola las canciones, el aspecto que cobra nunca es ejecutado por el *return* que se encuentra en el segundo aspecto, como resultado las canciones sonarán sin que sean cobradas al cliente.

3.3.1.2. Clasificación de Interferencias

De acuerdo a [25], éstas interferencias semánticas pueden ser clasificadas, de acuerdo al orden de ejecución de los aspectos:

No hay diferencias en el comportamiento observable - Al tener dos aspectos independientes en un *shared join point*, para cualquier orden de ejecución no se verá ninguna diferencia después de la ejecución de los *advice* de los aspectos.

Diferente orden exhibe comportamiento diferente - Distribuido en tres categorías

- El cambio en el orden de la ejecución de los aspectos presenta cambios observables en el comportamiento, pero no hay un requerimiento específico de cómo debería ser ese comportamiento.
- El orden de los aspectos importa, debido a que hay un requerimiento explícito que indica el orden de ejecución de los *advices* de los aspectos.
- No hay ningún requerimiento de orden de la ejecución de los aspectos, pero hay órdenes de ejecución que pueden violar la semántica de los aspectos. Por ejemplo, cuando múltiples *advices* bloquean ciertos recursos pueden ocurrir *deadlocks*, lo que quiere decir que debido a la semántica de los aspectos hay orden de ejecución implícito.

3.4. Propuestas para Resolución de Conflictos

Existen varias propuestas para reducir los problemas de interferencia de aspectos. Desde los enfoques sencillos, como declarar relaciones de precedencia entre los aspectos en el código como lo hace AspectJ[27], o aproximaciones desde la etapa de modelamiento[28]. Otras aproximaciones van un poco más allá e introducen dependencias más complejas y relaciones de orden entre aspectos[25]. Otras aproximaciones detectan los conflictos entre los aspectos gracias a una especificación más completa del comportamiento de los aspectos, realizada por quien los programa[26, 29].

A continuación se describen algunas de las propuestas para resolución de conflictos semánticos entre aspectos.

3.4.1. Especificar precedencia Aspectos

La primera propuesta es que el lenguaje de aspectos permita poder definir un orden o especificar precedencia entre aspectos, de ésta manera reduciendo los posibles conflictos semánticos entre ellos. AspectJ tiene los elementos del lenguaje necesarios para declarar precedencia entre aspectos a través de la instrucción *declare precedence*[27]. La desventaja que tiene ésta aproximación es que el programador es responsable de identificar donde se pueden presentar estos problemas, lo que causa que en sistemas grandes, se convierta en un proceso largo, propenso a errores e implica que se conozcan todos los aspectos y cuales aspectos pueden llegar a interferir con otros. Además, cuando se quiera introducir un nuevo aspecto, éste debe ser tenido en cuenta en las declaraciones de precedencia de los demás aspectos existentes.

Para el caso del ejemplo del sistema de música, la declaración de independencia usando AspectJ es la mostrada en 3.4.

Otra propuesta para realizar la definición de precedencia de aspectos, es la implementada para Motorola WEAVER, un plugin para manipular modelos ejecutables UML en Telelogic TAU G2[30]. Motorola WEAVER es una herramienta diseñada para poder tejer aspectos en modelos ejecutables UML. La premisa principal es que al tejer los aspectos de esa manera, los modelos para plataformas específicas y el código fuente puede ser generado automáticamente.

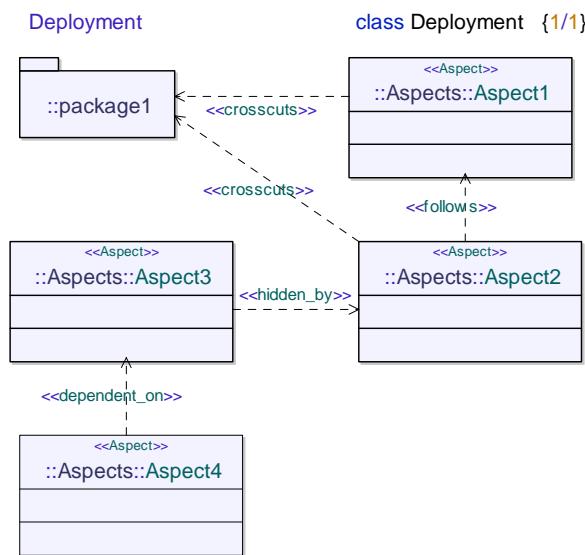
En Motorola WEAVER, la composición de aspectos es alcanzada a través de un diagrama de *deployment*, como se puede ver en la figura 3.2, tomada de [30]. En éste diagrama, se definen relaciones estereotipadas entre aspectos, que dictan la política de cómo van a ser tejidos al modelo base. Los estereotipos definidos son «*follows*», «*hidden_by*» y

```

1 aspect CreditsAspect
2 {
3
4     declare precedence : CreditsAspect , PlaylistAspect ;
5
6     void around () : call ( public void Jukebox . play ( Song ) )
7     {
8         if( Credits . instance () . enoughCredits () )
9         {
10            Credits . instance () . withdraw () ;
11            proceed () ;
12        }
13        else
14        {
15            throw new NotEnoughCreditsException () ;
16        }
17    }
18 }
```

CÓDIGO 3.4: Declaración de precedencia entre aspectos en AspectJ.

«*dependent_on*», los cuales corresponden a tres maneras diferentes de manejar la interferencia.

FIGURA 3.2: Diagrama de *deployment* mostrando los estereotipos para aspectos.

Follows - En un *shared join point*, el *Aspect1* tiene mayor precedencia que el *Aspect2*, lo que quiere decir que todas las instancias de *Aspect2* serán ejecutadas después de las instancias de *Aspect1*.

Hidden By - Cuando se encuentren el *Aspect2* y el *Aspect3* en un *shared join point*, el *Aspect3* será desactivado y no se ejecutara.

Dependant On - El *Aspect4* solo podrá ser ejecutado si en el *shared join point* se encuentran tejidos el *Aspect3* y el *Aspect4*.

3.4.2. Detección de Conflictos

La idea principal de la propuesta[31] se basa en que para que un conflicto ocurra, debe existir una interacción con consecuencias indeseables entre los aspectos. Ésta interacción puede ser modelada por las operaciones que se hacen sobre uno o más recursos compartidos. Un conflicto es modelado como las ocurrencias de ciertos patrones de operaciones sobre un recurso compartido.

Para poder detectar los conflictos entre aspectos, es necesario tener más información sobre las operaciones (comportamiento) de los aspectos. Con éste fin, la propuesta introduce una formalización para poder expresar el comportamiento del aspecto y además, poder modelar reglas de detección de conflictos sobre dicho comportamiento. Para abstraer el comportamiento relevando del aspecto, es necesario definir cómo se va a abstraer el comportamiento del aspecto. La abstracción propuesta consiste en un modelo de recursos_operaciones, el cual permite representar comportamiento de bajo nivel y de alto nivel.

Un **recurso** puede representar una propiedad concreta, como un atributo o los parámetros de un método, una propiedad abstracta o un concepto específico de la aplicación que encapsula el área problema. Un recurso está compuesto por un nombre y un conjunto de operaciones permitidas para éste recurso. En el ejemplo del sistema de reproducción de música, el recurso pude ser modelado como *Jukebox* y es de tipo abstracto. Como las operaciones, los recursos pueden hacer referencia a los elementos concretos de la aplicación.

Las **operaciones** representan el efecto que tiene un *advice* sobre cierto recurso. Las operaciones más primitivas sobre datos compartidos que se pueden modelar, son las operaciones de lectura y escritura, pero el modelo también permite modelar acciones de más alto nivel para poder ingresar información más específica sobre el comportamiento. En el ejemplo del sistema de música, una operación sobre el recurso *JukeBox* es el método *checkCredits*.

Gracias a el modelo, es posible hacer una especificación del comportamiento por cada aspecto. Ésta especificación consiste en un conjunto de recursos con una secuencia de operaciones que se les otorga. Para el ejemplo del sistema de música, en el aspecto de la lista de reproducción descrito en 3.3 la especificación del comportamiento es:

```
JukeBox: enqueue ; end
```

CÓDIGO 3.5: Especificación del comportamiento del aspecto PlaylistAspect

Para el aspecto de cobrar por escuchar una canción:

```
1 JukeBox: checkCredits ; withdrawCredits
```

CÓDIGO 3.6: Especificación del comportamiento del aspecto CreditsAspect

Además de la especificación del comportamiento de los aspectos, es necesario definir unas reglas para la detección de conflictos. Estas reglas describen cuales son las operaciones permitidas para un conjunto de recursos. Las reglas están compuestas por un recurso, una expresión que describe el patrón de conflicto.

Para el ejemplo del sistema de música, la regla de detección de aspectos es la siguiente:

```
1 Conflict (Jukebox) : .* (end) .*
```

CÓDIGO 3.7: Regla de detección de conflictos para el aspecto CredisAspect

La propuesta también describe un proceso para poder detectar los posibles conflictos. A continuación se presentan sus tres etapas.

3.4.2.1. Etapa de Composición

En ésta primera etapa, se evalúan todos los puntos de corte sobre la aplicación base, en caso que se encuentre un punto donde deba ir un aspecto, se teje el *advice* sobre ese *join point*. Luego, se determina el orden de ejecución de los aspectos, en caso que se haya hecho alguna definición de ello con anterioridad.

Como resultado se obtiene un conjunto de *join points* con una secuencia de *advices* tejidos a ellos. Como lo define la propuesta, el análisis de conflictos solo debe hacerse sobre un *join point* cuando la cantidad de *advices* es superior a uno.

3.4.2.2. Etapa de Abstracción de Comportamiento de los Aspectos

Ésta segunda etapa toma el producto de la etapa anterior junto con la abstracción del comportamiento de los aspectos y transforma la secuencia de los *advices* de un *join point* en una secuencia de operaciones por recurso por *shared join point*.

Para el ejemplo del sistema de sonido, la secuencia sería:

1 JukeBox: enqueue ; end ; checkCredits ; withdrawCredits

CÓDIGO 3.8: Secuencia de operaciones para el recurso *JukeBox*

3.4.2.3. Etapa de Detección de Conflictos

Ésta etapa toma las reglas de detección de conflictos y las transforma en un autómata a partir de la expresión que define el conflicto. Luego, para cada una de las secuencias de operaciones obtenidas en la etapa anterior, se determina si el autómata que representa el conflicto acepta dicha secuencia, indicando si existe un conflicto o no. En caso que algún autómata no acepté alguna secuencia, al usuario se le muestra algún tipo de error o se registra en un sistema de monitoreo.

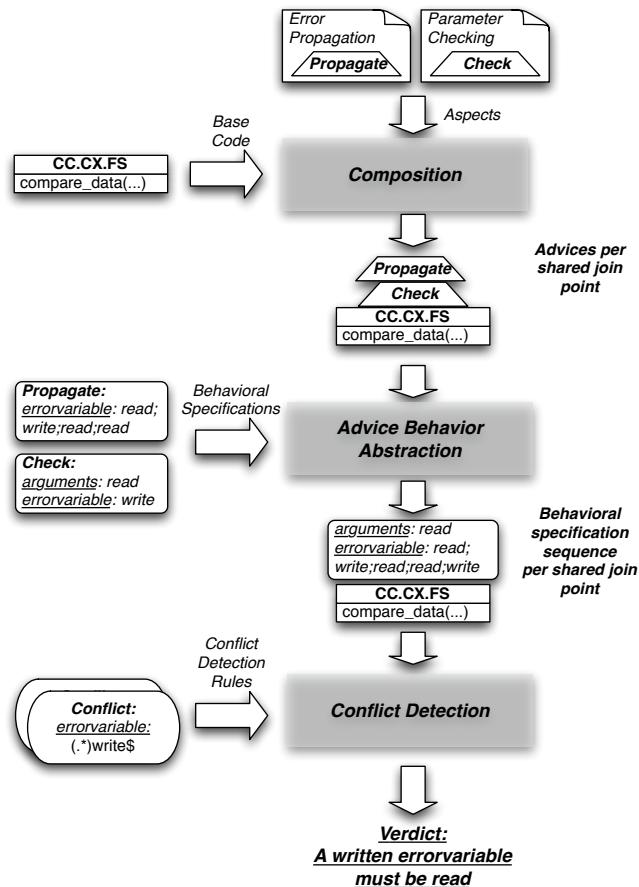


FIGURA 3.3: Proceso de detección de conflictos.

Capítulo 4

Aspect-Oriented Workflow Languages

4.1. Introducción

Éste capítulo presenta las limitaciones de los lenguajes de *workflow* con respecto a la modularidad de las preocupaciones transversales y la modularidad de los cambios. Dentro de las limitaciones de los sistemas de *workflow* se encuentran la falta de soporte a los comportamientos transversales (*crosscutting concerns*), es decir que los lenguajes no ofrecen elementos necesarios para implementar modularmente requerimientos que afectan transversalmente los procesos, tales como monitoreo de actividades, recolección de datos, métricas, etc. Actualmente, para poder implementar estos cambios, es necesario modificar la especificación del proceso, lo que tiene varias implicaciones negativas:

- Las preocupaciones transversales pueden afectar más de un proceso, en más de un punto. Si el programador tiene que modificar la definición de los procesos para agregar éste comportamiento, debe conocer todos los procesos y todos los lugares dentro de los procesos donde debe realizar la modificación. Éste es un procedimiento largo, donde la probabilidad de injectar errores es muy alta.
- Modificar la especificación del proceso para satisfacer las preocupaciones transversales causa que no exista una clara separación entre los elementos que componen el proceso y los elementos que soportan los comportamientos transversales.
- Debido a que el comportamiento que satisface las preocupaciones transversales se encuentra dispersado a través de los procesos, no hay manera de que los elementos que soportan los comportamientos transversales puedan ser activados o desactivados durante la ejecución del proceso.

- No poder expresar los cambios sobre una definición de procesos como entidades de primera clase implica que la única manera de poder conocer los cambios que ha sufrido un proceso, es comparando el proceso inicial con el actual para luego deducir los cambios.

4.2. Problemas de Modularización de Lenguajes de Workflow

Para poder ilustrar los problemas de modularización de los lenguajes de *workflow*, se establece el siguiente ejemplo de un proceso.

4.2.1. Ejemplo

En el mercado existen aplicaciones para dispositivos móviles, mediante las cuales es posible identificar una canción, registrando a través de un micrófono un fragmento corto que esté sonando en la radio o en televisión. Una vez identificada la canción, es ofrecida al usuario para que la compre de diferentes tiendas de música en línea. Ejemplos de éstas aplicaciones son Shazam[32] y Midomi[33].

En la figura 4.1 se muestra como puede ser el proceso de una de éstas aplicaciones. El proceso comienza cuando es recibida la información capturada a través del micrófono por la aplicación. Una vez la solicitud es recibida, la siguiente actividad es encargada de comunicarse con el servicio que analiza la canción y como resultado provee la información completa de la canción. Al tener la información de la canción, dos actividades de búsqueda interactúan con dos tiendas de música en línea, para buscar la información de compra para la canción. Luego la información es consolidada para posteriormente ser retornada al usuario.

4.2.2. Problemas de Modularización de Preocupaciones Transversales

Para poder ilustrar como los lenguajes de *workflow* no tienen los mecanismos necesarios de modularización para las preocupaciones transversales, se presentaran algunos ejemplos de recolección de información y monitoreo de tiempo de ejecución de actividades, basados en [23].

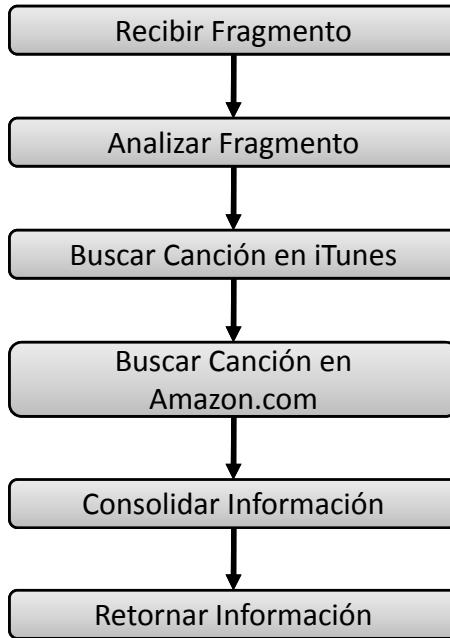


FIGURA 4.1: Ejemplo de un proceso workflow.

4.2.2.1. Recolección de Información

Pueden existir varios modelos de precios por usar los Web services de iTunes y de Amazon.com. Las políticas de recaudación pueden ser cobrar por cada llamado que se haga al Web service, cobrar a la empresa que haga más de cierto número de consultas o cobrar por consultas que no resulten en una compra. En caso de existir dichas políticas de cobro, el sistema debe poder llevar las cuentas de cuantos accesos a los Web services ha realizado el usuario, de ésta manera es posible corroborar que los cobros realizados por las tiendas de música en línea es correcto. Para verificar la veracidad de una cuenta, el sistema debe contar cuantas veces el proceso ha ejecutado la actividad que se comunica con las tiendas de música.

Para poder implementar la funcionalidad de recolección de información, se debe modificar todos los procesos de tal manera que cuando alguno de ellos se comunique con alguna de las tiendas de música en línea se lleve la cuenta, cómo lo muestra la figura ???. Para poder implementar éste cambio en BPEL, es necesario tener un Web service cuya funcionalidad es llevar la cuenta de los contadores. Se debe modificar la definición del proceso para agregar tanto las variables donde se va a llevar la cuenta, como los *partner links* para poderse comunicar con el Web service anteriormente mencionado. Asimismo es necesario modificar la estructura del proceso para agregar un nuevo *assign* antes de hacer el llamado a el Web Service que comunica el proceso con cada tienda de música en



FIGURA 4.2: Ejemplo de recolección de información.

línea, para poder establecer el valor del contador en la variable, para luego ser enviada a través de un nuevo *invoke* que llama el Web service para incrementar el contador.

La recolección de datos es transversal, porque puede ocurrir en diferentes puntos del proceso, en diferentes procesos. La adición de la definición de las variables y de los *partner links* tienen que repetirse en todos los procesos y también tiene que agregarse el *assign* y el *invoke* por cada ocurrencia de una actividad que se comunica con las tiendas de música en línea, dispersando y repitiendo los mismos elementos muchas veces. Además, no se va a tener una separación clara entre cuales son los elementos del proceso y cuales son los elementos usados para satisfacer las preocupaciones transversales.

4.2.2.2. Monitoreo de Tiempo de Ejecución de Actividades

Las organizaciones que utilizan *workflows* usualmente están interesadas en medir los tiempos de ejecución de ciertas actividades de los procesos[23]. Si el sistema de *workflow* que se utilice no provee las herramientas necesarias para poder realizar el

monitoreo del tiempo de ejecución de las actividades, una de las opciones es agregar ésta funcionalidad directamente sobre el proceso. Si se quiere agregar ésta funcionalidad a las dos actividades de búsqueda en el proceso de la figura 4.1, obliga modificar el proceso para agregar una actividad cuando se quiere comenzar a monitorear el tiempo de ejecución antes de la actividad a monitorear y agregar otra actividad después, para detener el monitoreo, como lo muestra la figura ???. Una posible implementación en



FIGURA 4.3: Ejemplo monitoreo de tiempo de ejecución.

BPEL, es crear un Web service de auditoría e invocar operaciones para iniciar o parar un temporizador. Se debe modificar la definición del proceso para agregar tanto las variables donde se va a llevar la información de la actividad que está siendo monitoreada, como

los *partner links* para poderse comunicar con el Web service anteriormente mencionado. Asimismo es necesario modificar la estructura del proceso para agregar un nuevo *invoke* antes de la actividad que se quiere monitorear y un *invoke* después de la misma.

El monitoreo del tiempo de ejecución de una actividad también es transversal, porque puede ocurrir en diferentes puntos del proceso, en diferentes procesos. La cantidad de elementos que se debe agregar es aún mayor que en el ejemplo anterior.

4.2.3. Problemas de Modularización de Cambios

Para ilustrar las deficiencias que tienen los lenguajes de *workflow* con respecto a la modularización de los cambios, se presentarán algunos ejemplos de recolección de información y monitoreo de tiempo de ejecución de actividades, basados en [23].

De acuerdo a [23] los cambios que puede sufrir una definición de un *workflow* son los siguientes:

- **Cambios Evolutivos** - Los contextos donde se utilizan los sistemas de *workflow* son altamente cambiantes. Multiples elementos pueden afectar en cualquier momento una definición de procesos, por ejemplo, nuevas estrategias de negocio, colaboraciones, nuevas condiciones externas, avance tecnológicos y cambios organizacionales. Estos cambios tienen que ser soportados por los lenguajes de *workflow*, ya que un cambio evolutivo va a afectar a todos los procesos junto con sus instancias.
- **Cambios Ad-hoc** - Los cambios ad-hoc generalmente ocurren porque es imposible tener en cuenta todas las situaciones excepcionales al momento de diseñar un proceso. Pueden ocurrir comportamientos inesperados debido a la interacción con usuarios, eventos impredecibles o situaciones erróneas. Los sistemas de *workflow* deberían proveer soporte para la adaptación dinámica de las instancias de *workflow*, para poder corregir dichas situaciones excepcionales.

Para poder ilustrar como los lenguajes de *workflow* no tienen los mecanismos necesarios de modularización de cambios, se presentaran algunos ejemplos de incorporación de un cambio evolutivo y un cambio ad-hoc, a partir del proceso mostrado en la figura 4.1. Estos ejemplos son basados en [23].

4.2.3.1. Agregar una Actividad

Se quiere modificar el proceso para que después de buscar la canción en las tiendas de música en línea, tenga una actividad extra que busque si el usuario es elegible para un código de promoción, cómo se muestra en la figura ??.



FIGURA 4.4: Ejemplo agregar una actividad.

Para poder realizar éste cambio, el programador tiene que bajar el proceso, modificarlo y luego volverlo a subir al servidor. En BPEL, se debe agregar un nuevo *partner link* hacia el servicio que retorna un código de promoción. Además, debe agregar dos nuevas variables donde mantendrá la información de entrada y de salida para la actividad de búsqueda. En cuanto a la modificación del control del proceso, es necesario agregar tres nuevas actividades. Primero, un *assign* donde se establecerá el valor de la variable de entrada para la búsqueda del código de promoción. Segundo, un *invoke* que es quién llama al servicio de búsqueda. Tercero, otro *assign* que copia la información de la búsqueda a la respuesta del proceso.

4.3. AOP en Contextos Workflow

Gracias a que la orientación por aspectos es una descomposición de uso general y paradigma de modularización, puede ser utilizado en otros contextos[23]. De la misma manera que AOP permite reducir la cantidad de código enredado y repetido, y agregar nuevo comportamiento de manera modular en los lenguajes de programación[19], se ha propuesto aplicar esta técnica dentro de los contextos de programación.

Los lenguajes orientados por aspectos definen nuevos elementos al lenguaje que serán utilizados junto con los elementos del lenguaje existentes para proveer soporte a la modularidad, encapsulando los comportamientos transversales y los nuevos comportamientos. Estos elementos son:

4.3.1. Modelo de *Join Points*

El modelo de *join points* define los lugares donde los *advices* van a ser ubicados en la ejecución de la aplicación base. Son puntos bien definidos, que proveen un marco de referencia común que hace posible que la ejecución del código del programa y la ejecución del código del *advice* sea coordinada.

De acuerdo a [23], el modelo de *join points* más intuitivo es basado en las actividades. La idea del modelo es que los *join points* corresponden a las ejecuciones de las actividades y pueden ser diferenciados en dos: ***Join points de Actividades*** - Son *join points* de grano grueso, es decir, estos puntos capturan el inicio o la terminación de la ejecución de una actividad. ***Join points Internos*** - Son *join points* de grano fino, capturan puntos internos en la ejecución de una actividad. Estos son necesarios cuando los *join points* de actividades no son lo suficientemente granulares para poder implementar algún comportamiento transversal.

4.3.2. Lenguaje de Puntos de Corte

El lenguaje de puntos de corte es utilizado para seleccionar un conjunto específico de *join points*.

El lenguaje de puntos de corte, en contextos de *workflows* puede ser pensado de dos maneras. La primera aproximación es desarrollando un lenguaje de texto, como XML, donde mediante instrucciones específicas o utilizando expresiones, se defina donde se quiere componer el nuevo comportamiento. La segunda es poder seleccionar sobre una representación gráfica del proceso, donde se quiere componer algún comportamiento nuevo. Ésta aproximación tiene la ventaja que la composición del nuevo comportamiento la

puede hacer cualquier persona que esté familiarizada con el proceso, ya que se requeriría una herramienta donde gráficamente se pueda seleccionar donde se quiere hacer la composición, para que después la herramienta genere un archivo de texto o se comunique directamente con el servidor.

4.3.3. Lenguaje de *Advices*

El lenguaje de *advices* define la funcionalidad que debe ser ejecutada en los *join points* específicos. El *advice* es un fragmento de código que es ejecutado cuando se alcanza un *join point* identificado por el respectivo punto de corte. El lenguaje de *advices* generalmente es el mismo lenguaje que la aplicación base. En lenguajes de *workflow* orientados por aspectos el lenguaje de *advices* debería ser el mismo que el lenguaje de *workflow* base, para evitar equivocaciones de quien está programando los *advices*[34]. De acuerdo al modelo discutido en 3.2 los *advices* pueden ser ejecutados antes, después, o en vez de los *join points* que son seleccionados por el punto de corte.

4.3.4. Tejido de Aspectos

El tejido de aspectos es la parte de la implementación que debe asegurar que el código de los *advices* y él de la aplicación base se ejecuten de manera coordinada, en los *join points* que fueron definidos para cada aspecto.

Existen dos maneras de hacer el tejido entre un proceso y sus aspectos[23]. De la primera forma, se conoce como tejido estático. Usando ésta manera de tejido, el proceso y los aspectos son tejidos antes de que se haga *deploy* del proceso al motor. La otra forma se conoce como tejido dinámico y ocurre en ejecución. Estás dos aproximaciones implican dos maneras diferentes de implementar los motores donde se van a ejecutar tanto los procesos como los aspectos[23].

4.3.4.1. Transformación de Procesos

De ésta manera debe existir una herramienta de transformación, que a partir de la definición del proceso y la definición de los aspectos, genere una nueva definición de proceso. Ésta aproximación soporta la composición estática, muy similar a como funciona AspectJ (sección 3.2).

Una de las ventajas que tiene ésta aproximación es que cualquier motor de BPEL puede tomar la definición de proceso producida por la herramienta de transformación y hacer *deploy* del proceso sin modificar el motor. En cambio, la desventaja más clara,

es que la composición no puede ser realizada en tiempo de ejecución y por tanto, los aspectos no pueden tener puntos de corte que estén relacionados con información que solo se tiene en ejecución, a menos que se tengan en cuenta todas las posibilidades en diseño. Además, con ésta aproximación, los aspectos no son definidos como entidades de primera clase, lo que implica que no se les puede hacer *deploy* o *undeploy* en tiempo de ejecución.

4.3.4.2. Modificación del Motor para Verificación de Aspectos

En ésta aproximación, el motor tiene que ser modificado para verificar si debe realizar la ejecución de un aspecto antes o después de la ejecución de cada actividad.

Ésta aproximación soporta la composición dinámica entre los aspectos y procesos. A diferencia de la aproximación anterior, permite hacer *deploy* y *undeploy* de los aspectos, sin necesidad de crear nuevas instancias de procesos, lo cual es importante en caso de tener procesos que tardan mucho tiempo en ejecutar, ya que sería necesario detener la instancia, modificarla y tener políticas para poder retornar la instancia al estado en el que se encontraba, como también políticas para manejar las posibles inconsistencias. Ésta aproximación trata a los aspectos como entidades de primera clase, permitiendo que se puedan implementar funcionalidades de administración en el motor. La desventaja de ésta aproximación es que los archivos que componen a los aspectos están ligados a un solo motor.

4.4. Lenguajes de BPEL Orientados Por Aspectos

A continuación se discuten dos de los lenguajes de BPEL orientados por aspectos más conocidos, utilizando los elementos descritos en la sección 4.3.

4.4.1. Padus

4.4.1.1. Modelo de *Join Points*

En Padus[34] el modelo de *join points* está relacionado con las actividades. Existen once tipos de *join points* definidos para Padus, cada uno relacionado con una actividad BPEL específica (ver tabla 4.1).

CUADRO 4.1: Modelo de join points en Padus

Join Point
invoking
receiving
throwing
compensating
replying
assigning
terminating
doingNothing ("empty")
sequencing
looping ("while")
flowing
switching
picking
scoping

```
1 invoking(Joinpoint , 'smsService' , 'smsServicePT' , Operation) , startsWith(
    Operation , 'send')
```

CÓDIGO 4.1: Ejemplo de un punto de corte que identifica *join points* que llaman una operación cuyo nombre comienza con “send”.

4.4.1.2. Lenguaje de Puntos de Corte

El lenguaje de puntos de corte de Padus[34] está compuesto por una serie de predicados que restringen los tipos de *join points* (ver tabla 4.1) a partir de sus propiedades. Los predicados definidos pueden ser combinados con predicados de Prolog, para así poder restringir también sobre los tipos de datos, buscar en listas, etc. Como en los *join points*, los predicados del lenguaje de puntos de corte, también permite definir o exponer propiedades de los *join points*, para ser utilizadas en ejecución.

CUADRO 4.2: Modelo de puntos de corte en Padus

Predicado	Descripción
invoking(Joinpoint, Name, PartnerLink, PortType, Operation, InputVariable, OutputVariable)	Todos los atributos posibles
invoking(Joinpoint, Name, PartnerLink, PortType, Operation)	No importan los nombres de las variables de salida
invoking(Joinpoint, PartnerLink, PortType, Operation)	Solo importan el partner link, el portType y operation

En el código 4.2 se muestra un ejemplo tomado de [34], donde un punto de corte que identifica *join points* que llaman una operación cuyo nombre comienza con “send”.

```
1 invoking(Joinpoint , 'smsService' , 'smsServicePT' , Operation) , startsWith(
    Operation , 'send')
```

CÓDIGO 4.2: Ejemplo de un punto de corte

4.4.1.3. Lenguaje de *Advices*

El lenguaje de los *advices* en Padus[32] se define dentro de un elemento XML, utilizando los elementos definidos en BPEL. Éste lenguaje también permite definir *advices* que pueden ser ejecutados antes, después y en vez de la actividad que se haya seleccionado. Padus también introduce un concepto nuevo que es el *in advice*. Éste tipo de *in advice* permite agregar comportamiento extra dentro de una actividad, por ejemplo, agregando una nueva actividad dentro de un *flow*.

4.4.1.4. Tejido de Aspectos

De los dos tipos de tejido, discutidos en la sección 4.3.4, Padus hace el tejido de manera estática, es decir, hace una transformación del proceso base y de los aspectos (figura 4.2 tomada de [34]), para generar un proceso nuevo. La motivación detrás de usar esta aproximación, es que Padus es utilizado en un contexto donde se espera que el motor se desempeñe con una alta eficiencia.

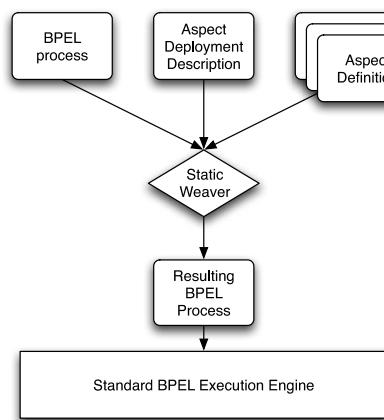


FIGURA 4.5: Arquitectura de Padus.

4.4.2. AO4BPEL

4.4.2.1. Modelo de *Join Points*

AO4BPEL soporta *join points* de actividad y *join points* internos[23] (sección 4.3.1). Los *join points* internos pueden capturar puntos definidos dentro de las actividades, por ejemplo, justo antes de enviar un mensaje en un *invoke*.

4.4.2.2. Lenguaje de Puntos de Corte

El lenguaje de puntos de corte en AO4BPEL[23] es XPath, debido a que es un lenguaje basado en consultas, que provee mecanismos para que los puntos de corte no sean estáticos y sean definibles por el usuario, además de utilizar los atributos de los elementos BPEL para restringir mejor los *join points* deseados. Gracias a que BPEL es un lenguaje basado en XML, el uso de XPath es una decisión natural, porque permite seleccionar fácilmente las actividades BPEL como puntos de corte.

Un ejemplo tomado de [23], de un punto de corte que selecciona todas las actividades *invoke*, cuyo nombre sea *travelPackage* en cualquier proceso.

```

1 <pointcut>
2   //invoke[@operation="findAFlight"]
3 </pointcut>
```

CÓDIGO 4.3: Ejemplo de un punto de corte en AO4BPEL.

Utilizando las ventajas de XPath, se pueden restringir los *join points* de acuerdo a los atributos de los elementos, como lo muestra el siguiente ejemplo, donde se restringe que el *join point* solo se aplicará sobre la operación *findAFlight* del proceso llamado *travelPackage*.

```

1 <pointcut>
2   /process[@name="travelPackage"]//invoke[@operation="findAFlight"]
3 </pointcut>
```

CÓDIGO 4.4: Ejemplo de un punto de corte en AO4BPEL.

4.4.2.3. Lenguaje de *Advices*

El lenguaje de *advices* que AO4BPEL utiliza es una versión modificada de BPEL[23]. El lenguaje provee elementos para que sea posible acceder al contexto de la instancia donde se está ejecutando, es decir, ofrece elementos para acceder a valores de variables

del *join point* donde se encuentra, entre otras. El lenguaje de *advices* también permite definir si el *advice* se va a ejecutar antes, después o en vez de una actividad o del envío o recepción de un mensaje.

4.4.2.4. Tejido de Aspectos

Para el caso de AO4BPEL, el tejido se hace de manera dinámica, es decir, la implementación del motor incluye verificaciones dentro del ciclo de vida de la actividad, para saber luego decidir si se debe ejecutar algún aspecto en el punto de corte que se está ejecutando.

¿Dónde se Hacen las Verificaciones?

Para implementar AO4BPEL fue extendido BPWS4J, un motor de orquestación producido por IBM. AO4BPEL modifica los estados del ciclo de vida de una actividad para incluir las verificaciones de si se debe ejecutar algún aspecto.

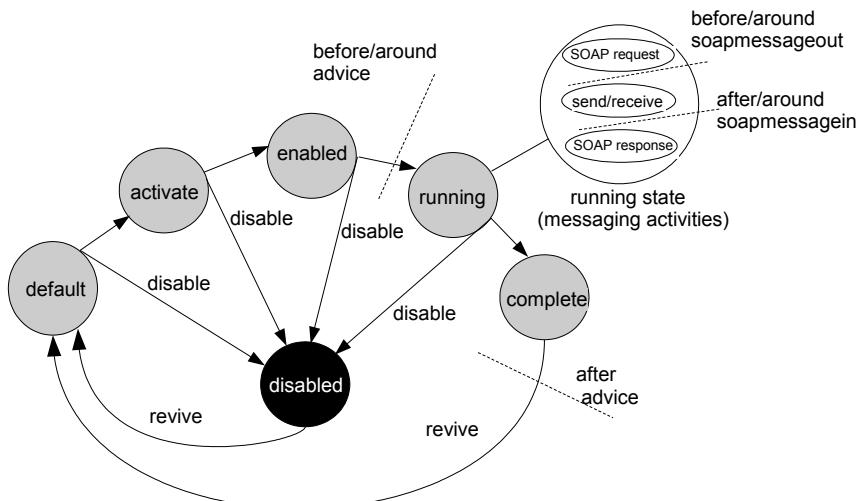


FIGURA 4.6: Estados de una actividad de AO4BPEL.

El ciclo de vida de una actividad se muestra en la figura 4.3[23]. Las verificaciones de los *advices* son hechas dependiendo si el *advice* es de tipo *before*, *after* o *around* o si los *advices* son de tipo interno. La verificación para los *advices* de tipo *before* se hace cuando la actividad pasa de estado *enabled* a estado *running*. La verificación para los *advices* de tipo *after* se hace cuando la actividad sale del estado *complete*. Para los *advices* internos, las verificaciones solo se hacen si la actividad es una actividad de interacción (sección 2.2.2.2) ya que el comportamiento se puede tejer antes, después o en vez de recibir o

mandar un mensaje. En éste caso, las verificaciones se encuentran dentro del estado *running*.

¿Qué se Verifica?

Para cada uno de los posibles *join points* es necesario verificar si existe un punto de corte asociado, a ese lugar en particular para la actividad que se está ejecutando. En AO4BPEL el proceso comienza desde el momento que se hace *deploy* de un aspecto. Al hacer *deploy* del aspecto se evalúan las expresiones de los puntos de corte sobre los documentos XML de los aspectos, para obtener así una serie de nodos XML para cada expresión de los puntos de corte, para luego poder extraer metainformación, como nombres, tipos de actividad, etc, para ser guardada. Luego, al ejecutar una actividad, se compara la metainformación de la actividad con la metainformación que fue previamente obtenida y en caso de concordar, quiere decir que ese punto de corte corresponde a esa actividad.

Ejecución de los *Advice*

Para hablar acerca de cómo AO4BPEL ejecuta los *advices*, es necesario hablar acerca de su arquitectura. Al motor BPWS4J se le agregaron dos componentes, el *aspect*

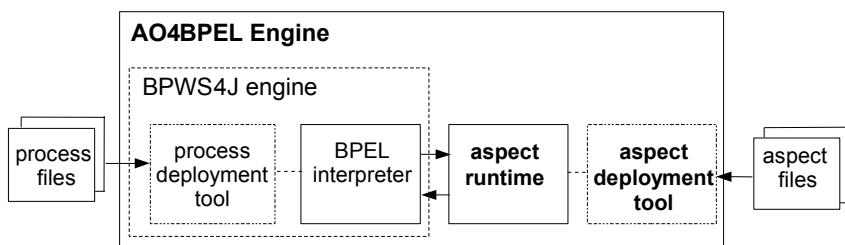


FIGURA 4.7: Arquitectura del motor de AO4BPEL.

deployment tool y el *aspect runtime*. **Aspect deployment tool** es una aplicación Web que permite hacer una administración sencilla de los aspectos. A través de éste componente se puede hacer *deploy* y *undeploy* de los aspectos, junto con conocer cuáles son los aspectos que se encuentran en el servidor. El **Aspect runtime** es responsable de manejar y ejecutar los aspectos. Éste componente es quién es responsable de realizar las verificaciones de los puntos de corte. Debido a que el comportamiento está definido en el *advice* usando BPEL, el *aspect runtime* delega la ejecución de esa actividad al interpretador BPEL y coordina la ejecución así[23]:

- Si el tipo del *advice* es *before* o *after* se suspende la ejecución del proceso hasta que se complete la actividad del *advice*, luego continua la ejecución del proceso con la actividad suspendida.
- Si el tipo del *advice* es *around* la ejecución del proceso se suspende, luego, al terminar de ejecutar la actividad del *advice*, la actividad suspendida salta el estado *running*, terminando su ejecución.

4.5. Interferencia de Aspectos en Contextos Workflow

A continuación se presenta un ejemplo para ilustrar como los problemas de interferencia en aspectos, se presentan también en contextos de *workflows*. Se agregarán dos aspectos al ejemplo utilizado en la sección 4.2.1.

4.5.1. Facturación por Busqueda

Se quiere agregar la funcionalidad que se le cobre al usuario de la aplicación por cada búsqueda que se realiza en las tiendas de música en línea. Para implementar éste cambio en las reglas de negocio se implementa usando un aspecto.

Una posible implementación es tejer una actividad que envíe la información del usuario a un servicio de recaudo para que la consulta sea agregada a su cuenta como se muestra en la figura 4.5.

4.5.2. Cambios en las Políticas de Distribución

Debido a que en Estados Unidos existe una ley llamada *Digital Millennium Copyright Act (DMCA)*[35] el contenido que es distribuido fuera de Estados Unidos debe ser restringido. Es por éste cambio en las reglas de negocio que se debe hacer una verificación de la ubicación donde se originó la solicitud. Para implementar éste cambio también se usa un aspecto.

Una posible implementación es colocar un aspecto en vez (*around*) de las actividades que buscan las canciones en las tiendas de música en línea, de tal manera que se verifique la localización del usuario antes de buscar la canción y de ser una ubicación no permitida, no hacer nada, como se muestra en la figura 4.6

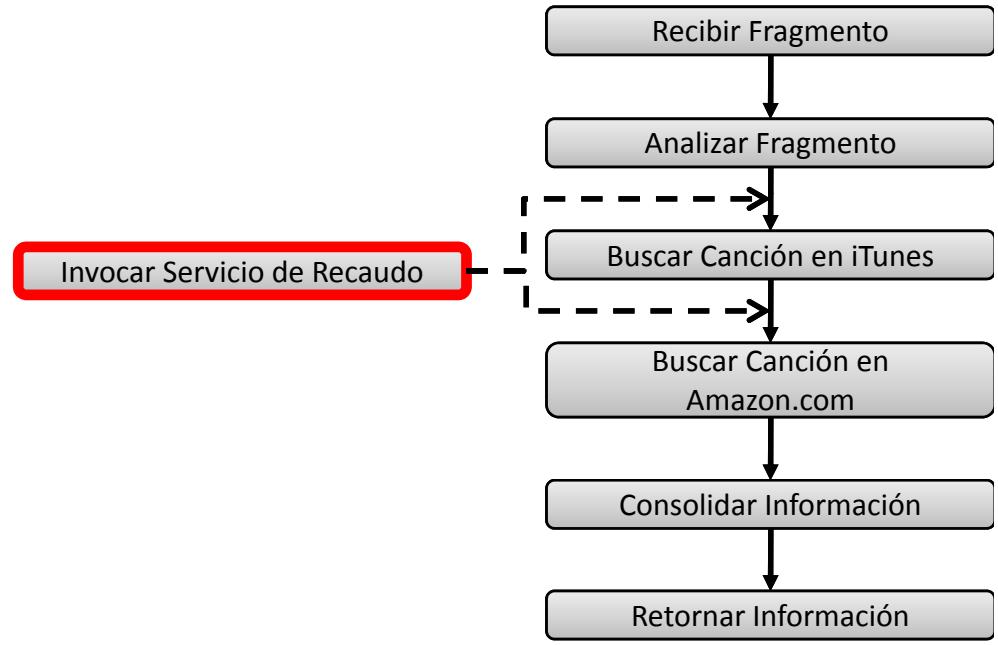


FIGURA 4.8: Aspecto de facturación.

4.5.3. Interferencia Entre los Aspectos

Para el caso de la composición de los dos aspectos anteriores en un *join point* compartido, la interferencia se da porque es posible cobrarle a un usuario por una consulta que no se realiza. Primero el proceso recibe la información del fragmento, junto con la ubicación del usuario. Luego se llama la actividad que procesa el fragmento para obtener la información acerca de la canción. El siguiente paso es ejecutar el primer aspecto, invocando el servicio de recaudo para registrar el cobro del acceso al usuario. Luego, se procede a ejecutar el siguiente aspecto, el cual verifica la ubicación del usuario y en el caso que no se encuentre en una ubicación válida, no realiza la búsqueda en la tienda de música en línea y no se tiene en cuenta el hecho que ya se le cobró por esa consulta. Si el primer aspecto no fuera un aspecto de tipo antes sino de tipo después, ocurriría lo mismo porque no se haría el acceso hacia el servicio de búsqueda, pero se le terminaría cobrando al usuario.

Las políticas de ordenamiento definidas por *Padus* y por *AO4BPEL* no funcionarían, porque no es posible definir un orden donde se puedan ejecutar los aspectos sin que ocurra el problema de interferencia entre ellos.

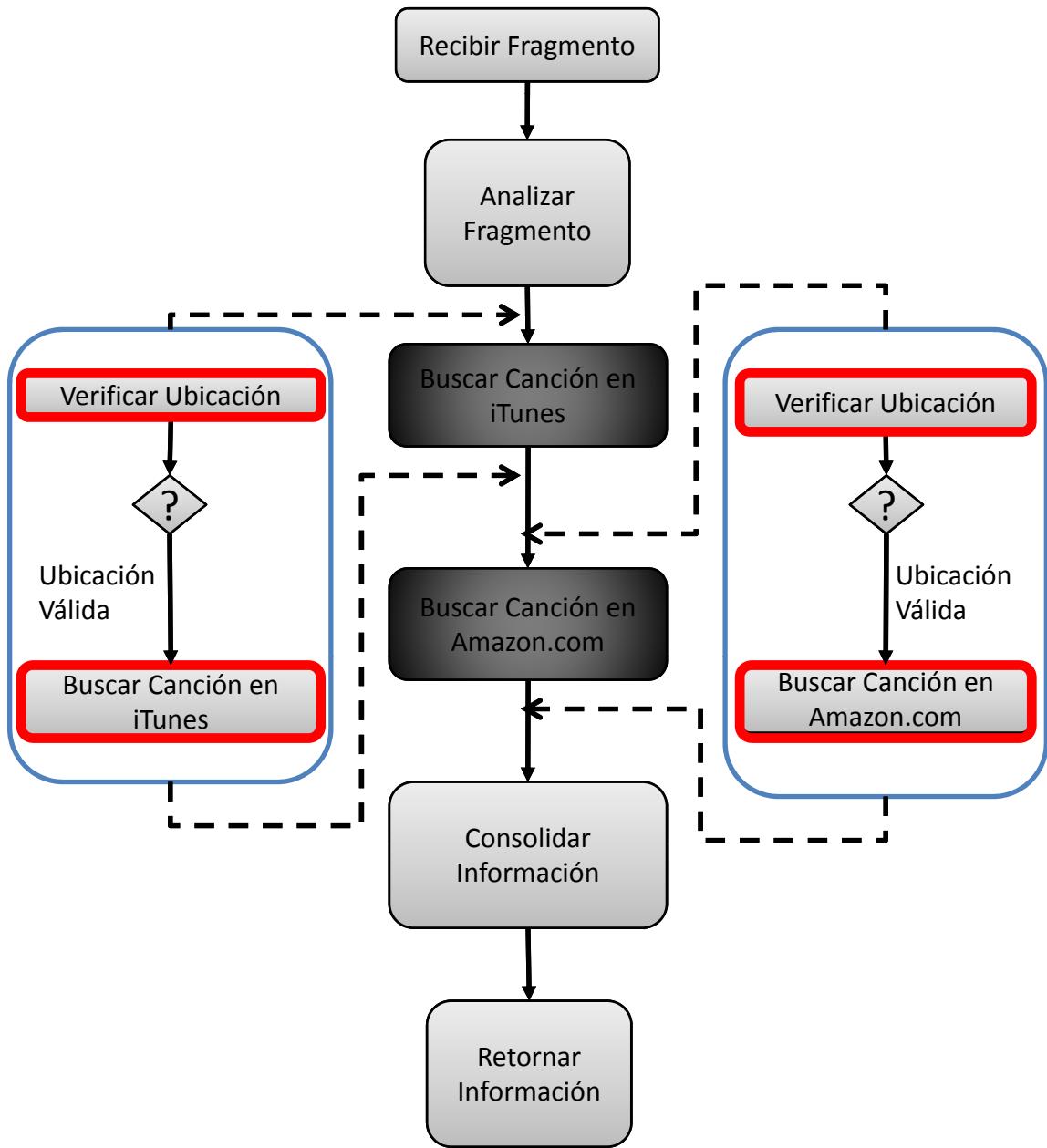


FIGURA 4.9: Aspecto de verificación de ubicación.

Capítulo 5

El Proyecto Cumbia

Los sistemas de *workflow* se desenvuelven en diferentes contextos (salud, educación, negocios), debido a que cada contexto presenta necesidades diferentes, los sistemas de *workflow* satisfacen de manera diferente esas necesidades. Sin embargo, todos tienen en común que los problemas se modelan ordenando y sincronizando la ejecución de un conjunto de recursos o elementos para lograr un objetivo en un tiempo específico. Éstos se conocen como aplicaciones basadas en control.

Múltiples factores influencian tanto los contextos como las aplicaciones que las soporan y no es común que las arquitecturas de dichas aplicaciones no son lo suficientemente flexibles para poder adaptarse a estos cambios[36].

El proyecto Cumbia¹ del grupo de construcción de software de la Universidad de los Andes, es un proyecto de investigación que propone la construcción de fábricas de software para la familia de aplicaciones basadas en control, donde predomine la evolución y adaptación de dichas aplicaciones. Una aplicación Cumbia, es un conjunto de componentes que se comunican entre sí y uno de los componentes tiene como responsabilidad manejar el control de la aplicación[37].

Dentro de Cumbia, estos componentes se denominan activos. Estos activos están construidos a partir de modelos ejecutables. Los modelos ejecutables a su vez, están definidos por componentes modulares llamados objetos abiertos. Ésta arquitectura permite tener un alto nivel de modularización, lo cual ayuda a construir aplicaciones que cuentan con una arquitectura totalmente flexible, además de exponer un modelo natural de composición de activos, no sólo para definir nuevas aplicaciones sino también para generar activos reutilizables en la generación de aplicaciones de la familia de control[38].

¹Proyecto Cumbia: <http://cumbia.uniandes.edu.co>

5.1. Aplicaciones Orientadas a Control

El componente de control de una aplicación está compuesto por tres elementos principales. Primero, el conjunto de actividades que se deben ejecutar, segundo por el modelo de asignación de responsables para estas actividades y tercero, por el orden en el cual se debe desarrollar su ejecución. El componente de control es responsable de ordenar, administrar y sincronizar un conjunto de tareas de manera automática para lograr un objetivo dado[38].

Sin embargo, el componente de control no es el único componente que constituye sistemas complejos. Existen diferentes perspectivas pertenecientes a diferentes dominios, que también deben ser tenidas en cuenta y coordinadas para poder resolver el problema propuesto. Por ejemplo, el conjunto de responsables que se asignarán a las tareas, ciertas restricciones de tiempo que puedan estar asociadas a la ejecución de cada actividad y el uso de datos o recursos de contenido necesarios para el desarrollo de cada actividad son ejemplos de las preocupaciones que conforman un problema en donde el control es una necesidad principal.

El proyecto Cumbia propone una abstracción de metamodelos para poder modelar todas las perspectivas que hacen parte de un problema. Mediante el uso de los metamodelos, se describen todos los elementos de cada dominio particular y la construcción consecuente de modelos ejecutables conformes, que permitan la composición de las diferentes perspectivas para poder solucionar el problema.

5.2. Metamodelos

De acuerdo a [39] un modelo es la abstracción de un sistema construido para un propósito específico. La especificación de dicha abstracción se conoce como metamodelo. En un metamodelo se identifican los elementos relevantes y la relación entre ellos. Gracias a los metamodelos es posible hacer una clara separación de todas las perspectivas que componen un sistema complejo. Ésta separación permite la representación de dominios que pueden ser comunes para diferentes aplicaciones y además permite crear nuevas soluciones componiendo diferentes dominios.

En Cumbia, cada una de las perspectivas que hacen parte del problema se describen en un metamodelo. La definición de cada metamodelo está compuesta por elementos de un dominio específico que modifica o complementa el componente de control. Por ejemplo, asignación de tiempo, manejo de recursos, manipulación de datos, etc. A cada uno de los

metamodelos se agrega semántica de ejecución materializando modelos conformes cuyos elementos se encuentran representados como objetos abiertos.

Producto de la construcción de modelos conformes a cada uno de estos metamodelos, se obtiene un modelo ejecutable y extensible que cumple con dos características. Primero, ofrece elementos de composición en un nivel de granularidad muy fino, asegurando la flexibilidad del modelo. Segundo, garantiza la extensibilidad y adaptación de los modelos a requerimientos cambiantes para los dominios que representan. La capacidad de composición y extensibilidad de estos modelos, ofrece ventajas relacionadas con modularidad y reutilización en la construcción de soluciones de manera flexible[40].

5.3. Modelo de Objetos Abiertos (OO)

Cumbia propone que los elementos expresados en los metamodelos, sean implementados de tal manera que puedan exponer fácilmente su semántica de ejecución. La propuesta es que se extienda el modelo tradicional de objetos, de tal manera que sea más fácil componer y coordinar los elementos.

Cada objeto abierto (ver figura ??) está compuesto por un objeto tradicional llamado entidad, una máquina de estados y un conjunto de acciones asociadas a las transiciones de los estados.

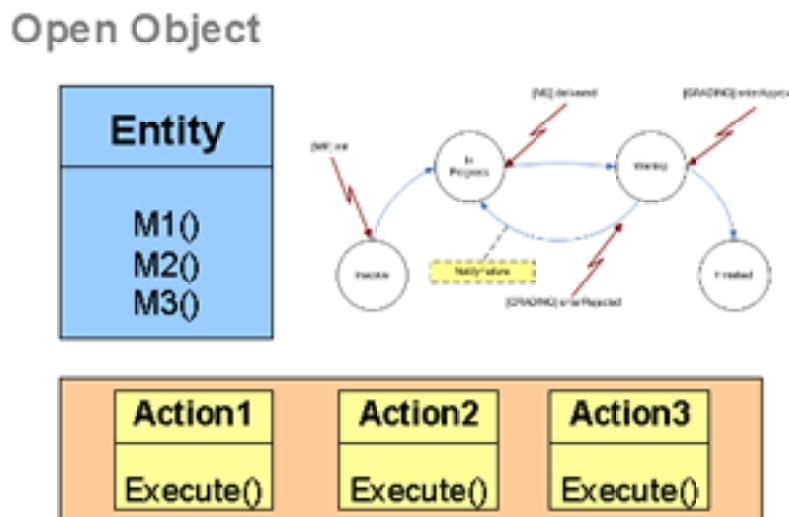


FIGURA 5.1: Estructura de un objeto abierto.

Los objetos pueden tener muchos estados que son dependientes de los posibles valores que puedan tomar sus atributos en un momento dado, pero no todos los posibles estados

son representativos o puedan interesarle a los demás objetos con los que interactúa. La máquina de estados es utilizada para poder exponer los estados relevantes de la entidad a otros objetos. Ya que la máquina de estados es la exposición del estado de la entidad, ésta siempre debe estar sincronizada con el estado interno de la entidad. El objeto interno sincroniza la máquina de estados, moviéndola de un estado a otro, de acuerdo a las acciones que se ejerzan sobre él.

Al utilizar una máquina de estados para exponer el estado de una entidad, es posible escuchar y/o generar eventos para mover otras máquinas de estados. Cuando una maquina de estado escucha un evento, éste es procesado y una transición es tomada para cambiar el estado actual, además de sincronizar su estado interno de ser necesario. En las transiciones de los estados es posible definir acciones, que serán ejecutadas secuencialmente una vez la máquina de estados cambie de un estado a otro a través de esa transición. El uso de acciones permite generar nuevos eventos y de ésta manera obtener coordinación sincrónica con otros objetos abiertos o componentes externos que pueda integrarse al sistema de eventos[41].

5.4. Modelos Ejecutables Cumbia

Un modelo ejecutable representa la instancia de un metamodelo de dominio específico, que gracias al uso de objetos abiertos, tiene semántica de ejecución y también permiten entrelazarse a nivel de entidades o de máquinas de estado con otros elementos definidos en el metamodelo.

5.4.1. Estrategia de Composición de Modelos

Una vez definidos tanto el modelo de control, como los demás modelos necesarios para obtener una aplicación, se necesita un mecanismo que sea capaz de coordinar la interacción de los elementos de modelos heterogéneos. El mecanismo de entrelazado propuesto, utiliza la misma estrategia de composición y coordinación que se usa dentro de cada modelo: los objetos abiertos se coordinan a través del paso de eventos, aún si están definidos en dominios diferentes. Ésta composición se realiza cuando los modelos se ejecutan, no existen tareas intermedias de compilación[40].

Las ventajas de este tipo de construcciones son enumeradas en [40] como:

- El nivel de granularidad de los puntos de unión disponibles para la composición y coordinación, es más alto que con otras aproximaciones. Estos puntos se encuentran

más relacionados con el estado de los elementos y no con el flujo de control o las interfaces, por lo tanto pueden ser modificados de acuerdo con la aplicación específica que se esté construyendo.

- Ésta estrategia se puede aplicar en niveles complejos para entrelazar preocupaciones sobre aplicaciones que ya contienen otros modelos entrelazados.
- Cada preocupación puede ser expresada de manera independiente usando lenguajes o metamodelos diferentes. Gracias a esto, se puede utilizar el metamodelo o mecanismo de extensión más adecuado para cada una de estas preocupaciones.

En la figura ?? se muestra cómo es posible crear aplicaciones separando los problemas de acuerdo a sus diferentes perspectivas y luego componiéndolas gracias al uso de la materialización de los modelos de cada dominio. En el ejemplo se muestra un modelo de control que especifica el orden y la sincronización del flujo de trabajo, un modelo de tiempo que permite definir reglas con diferentes patrones de tiempo, el dominio de roles que hace referencia a la estructura de usuarios en roles y los lugares donde ocurre la composición de los 3 modelos, en donde diferentes tareas tienen reglas de tiempo asociadas y son ejecutadas por un rol específico.

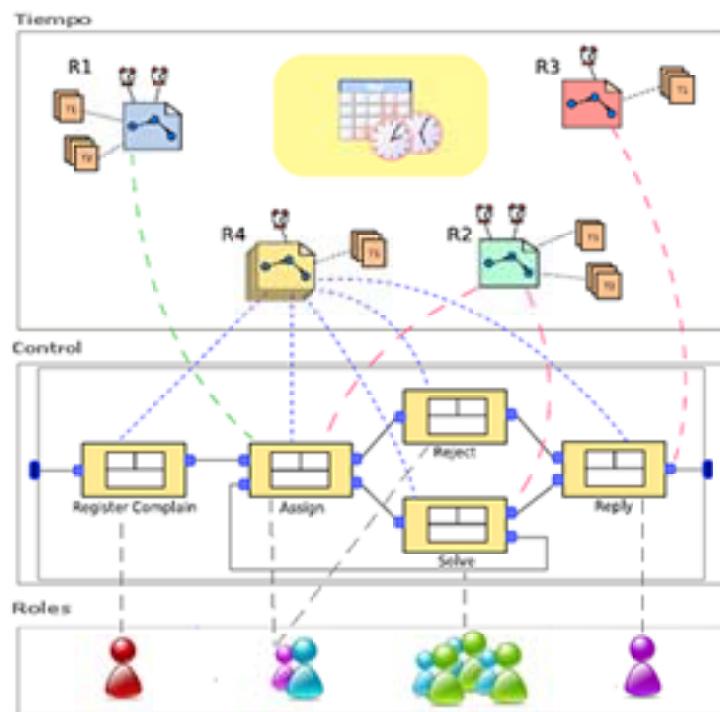


FIGURA 5.2: Composición de Modelos de Diferentes Dominios.

Capítulo 6

Caffeine v2.0: Motor BPEL sobre Cumbia

6.1. Introducción

Caffeine es un proyecto desarrollado dentro del marco de investigación del proyecto Cumbia. La primera versión de este motor fue desarrollada por Daniel Romero como trabajo de tesis[42]. Con el objetivo de poder extender la funcionalidad del motor para hacerlo orientado por aspectos, fue necesario rediseñarlo y reimplementarlo, es por eso que se desarrollo la versión 2.0.

6.2. Elementos Presentes

En el trabajo desarrollado en [42], se propone una metodología para construir motores de *workflow* usando Cumbia. Dentro de la metodología se propone que los elementos del lenguaje deben ser agrupados por capas, de tal manera que sea posible hacer un desarrollo incremental del motor de *workflow* que se busca construir[42]. Allí mismo se hizo una división de los elementos BPEL en tres capas. En la primera capa se colocó todos los elementos de BPEL que se consideran necesarios para poder definir un proceso de forma completa (sin manejo de fallas ni funcionalidad adicional). En la segunda capa, se clasificaron los elementos asociados con el manejo de fallas y transacciones. En la tercera capa se encuentran los elementos que permiten extender el lenguaje, definiendo elementos personalizables. Los elementos desarrollados en la segunda versión de Caffeine son los elementos que se muestran en la tabla 6.1.

CUADRO 6.1: Elementos de la Capa 1

Elemento	Descripción
process	Elemento raíz.
partnerLink	Describe un servicio colaborador, incluyendo roles (de cada parte involucrada) y la operación a ser invocada (WSDL portType).
variables	Elemento para declarar un conjunto de variables.
variable	Dato usado en el proceso.
receive	Bloqueo para esperar de un mensaje proveniente de uno de los partnerLink del proceso.
reply	Respuesta enviada a uno de los partnerLink.
invoke	Invoca uno de los servicios colaboradores, bien sea de forma asíncrona o sincrónica.
assign (copy, query, from, to, literal, expres- sion)	Permite copiar datos en las variables.
wait (until, for)	Suspende la operación por un período dado de tiempo.
empty	No hacer nada.
sequence	Ejecuta una serie de actividades en orden secuencial.
flow	Ejecución de actividades de forma paralela.
if, elseIf, else (condition)	Instrucción condicional.
while	Instrucción repetitiva.
pick	Bloquearse y esperar por un mensaje o alarma.
onMessage	Elemento definido dentro de pick o eventHandler. Bloquearse y esperar por un mensaje.
onAlarm	Elemento definido dentro de pick y eventHandler. Bloquearse y esperar a que un periodo de tiempo se cumpla.
exit	Termina y destruye los procesos de forma inmediata.

Debido a la cantidad de elementos, se decidió dividirlos en categorías para mejor ilustración.

6.2.1. Elementos de Inicio

Para poder iniciar un proceso BPEL se deben definir elementos para que al recibir un mensaje se instancie el proceso[2], esos elementos se encuentran agrupados en ésta categoría.

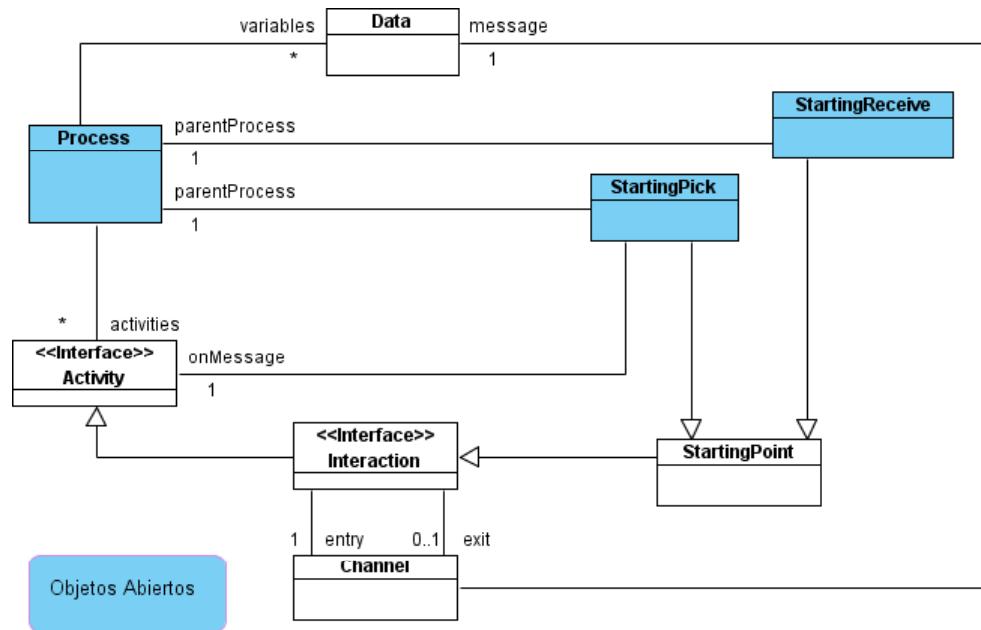
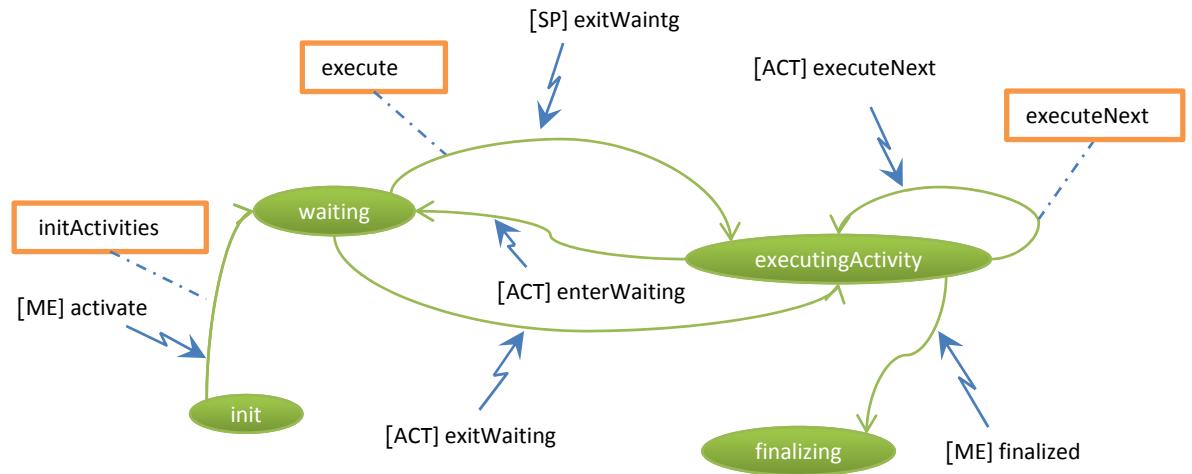


FIGURA 6.1: Elementos de inicio.

6.2.1.1. Elemento *Process*

Este elemento es el encargado de agrupar todas las actividades para alcanzar un objetivo.

FIGURA 6.2: Máquina de estados del elemento *Process*.

CUADRO 6.2: Estados del Elemento *Process***Descripción de Estados**

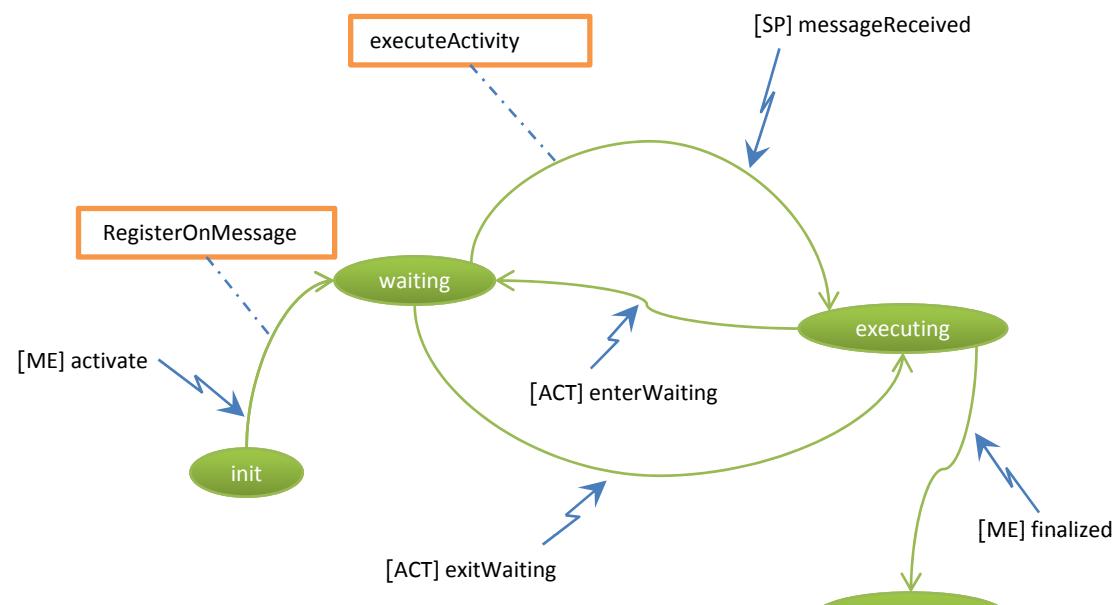
Init	Es el estado inicial del elemento.
Waiting	Una vez el elemento es inicializado, queda en espera hasta que se reciba el mensaje que indica que va a comenzar su ejecución. Éste estado también es utilizado para indicar que el proceso está detenido, debido a que alguna actividad interna bloquea el proceso.
ExecutingActivity	Éste estado indica que se encuentra ejecutando una actividad interna.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.3: Acciones del elemento *Process***Acciones**

InitActivities	Realiza todas las actividades necesarias para inicializar las actividades internas del proceso.
Execute	Ejecuta la primera actividad definida para el proceso.
ExecuteNext	Ejecuta la siguiente actividad en el proceso.

6.2.1.2. Elemento *StartingPick*

El elemento *StartingPick* representa cuando un *Pick* es configurado para que cuando llegue un mensaje inicie la ejecución del proceso.

FIGURA 6.3: Máquina de estados del elemento *StartingPick*.

CUADRO 6.4: Estados del Elemento *StartingPick***Descripción de Estados**

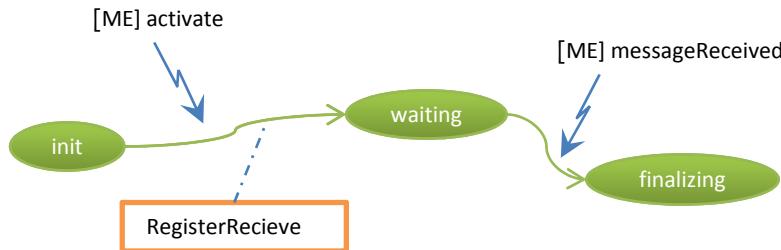
Init	Es el estado inicial del proceso.
Waiting	Una vez el elemento es inicializado, queda en espera hasta que se reciba el mensaje que indica que va a comenzar su ejecución. Este estado también es utilizado para indicar que el proceso está detenido, debido a que alguna actividad interna bloquea el proceso.
Executing	Éste estado indica que se encuentra ejecutando la actividad asociada al elemento.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.5: Acciones del elemento *StartingPick***Acciones**

RegisterReceive	Registra al elemento como un elemento que está esperando un mensaje.
RegisterOnMessage	Ejecuta la actividad asociada al elemento.

6.2.1.3. Elemento *StartingReceive*

El elemento *StartingReceive* representa cuando un *Receive* es configurado para que cuando llegue un mensaje inicie la ejecución del proceso.

FIGURA 6.4: Máquina de estados del elemento *StartingReceive*.CUADRO 6.6: Estados del Elemento *StartingReceive***Descripción de Estados**

Init	Es el estado inicial de la actividad.
Waiting	Una vez el elemento es inicializado, queda en espera hasta que se reciba el mensaje que indica que va a comenzar su ejecución.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.7: Acciones del elemento *StartingReceive*

Acciones

RegisterReceive	Registra al elemento como un elemento que está esperando un mensaje.
-----------------	--

6.2.2. Elementos de Interacción

Los elementos de interacción son aquellos que pueden enviar o recibir mensajes de los colaboradores y realizar una acción conforme.

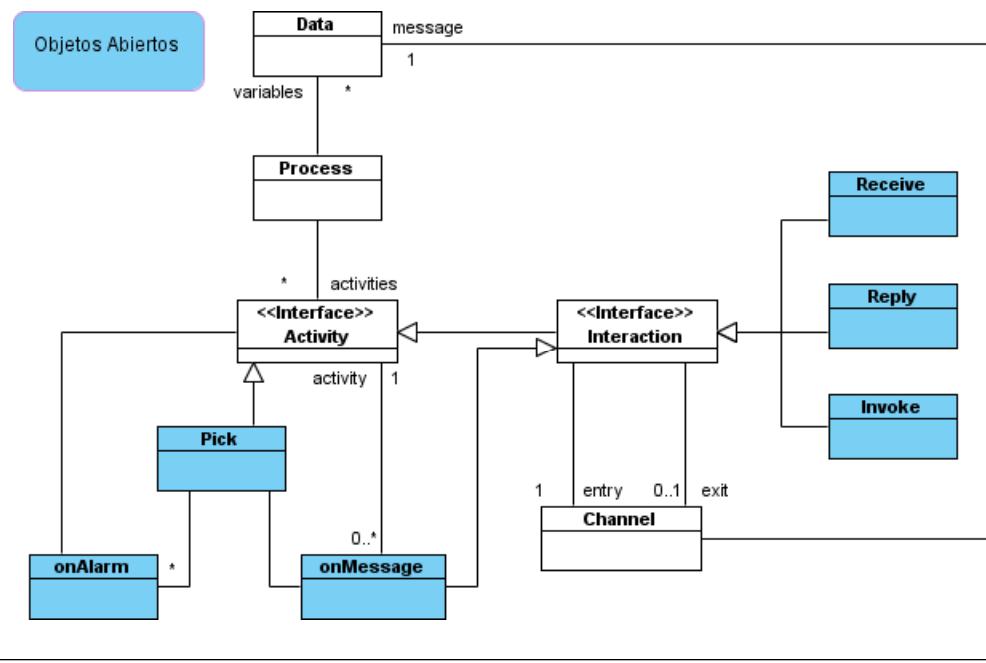


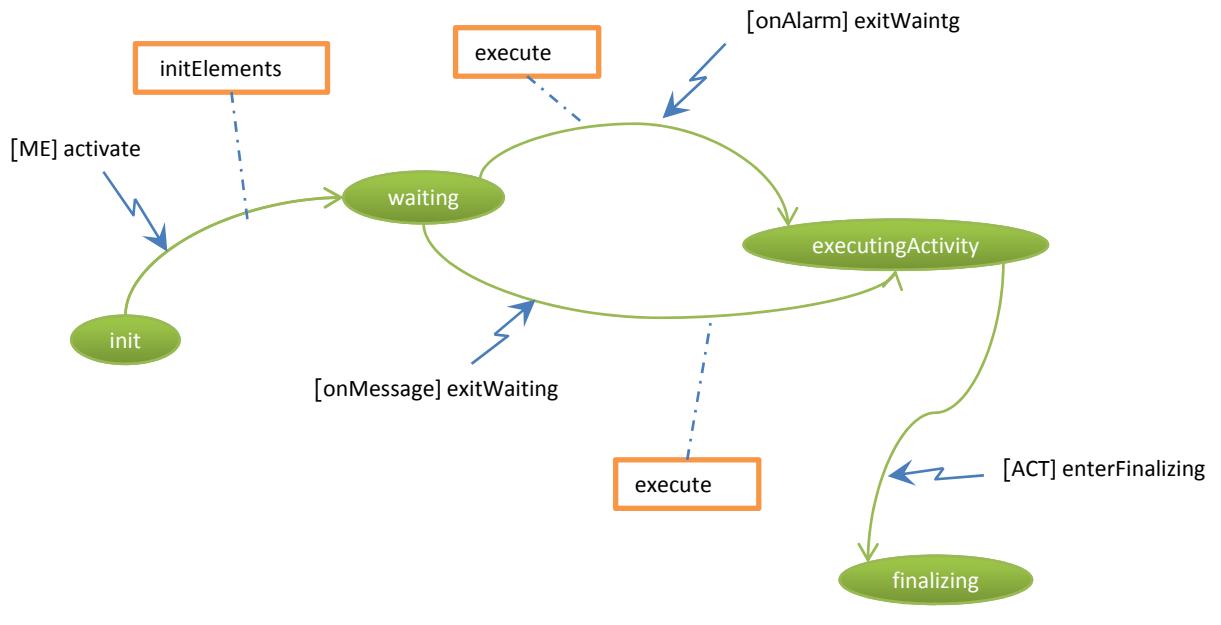
FIGURA 6.5: Elementos de interacción.

6.2.2.1. Elemento *Pick*

El elemento *Pick* espera la ocurrencia de exactamente un evento de un conjunto de eventos, luego ejecuta la actividad asociada con ese evento. Después que se ha seleccionado un evento, los demás eventos no son aceptados[2].

6.2.2.2. Elemento *OnAlarm*

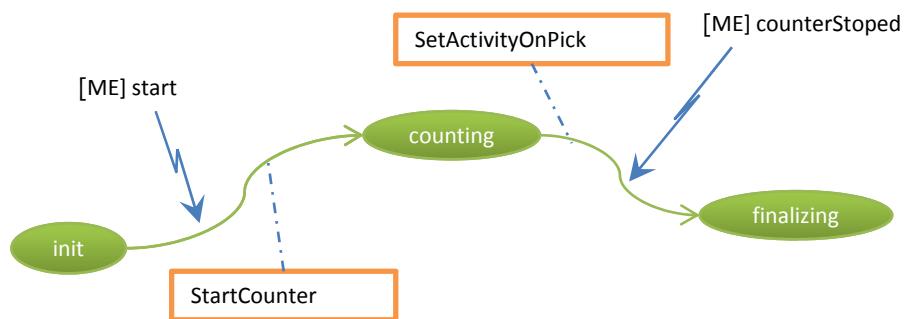
El elemento *onAlarm* corresponde a un temporizador, que tiene una actividad asociada. Cuando el temporizador se termina, se lo informa al *Pick* al que pertenece, para que este tome la decisión si se debe ejecutar o no la actividad.

FIGURA 6.6: Máquina de estados del elemento *Pick*.CUADRO 6.8: Estados del Elemento *Pick***Descripción de Estados**

Init	Es el estado inicial del elemento.
Waiting	Una vez el elemento es inicializado, queda en espera hasta que se dispare el temporizador de alguno de sus <i>onAlarm</i> o alguno de sus <i>onMessage</i> reciba un mensaje.
Executing	Una vez seleccionada la actividad, pasa a ejecutarla. Éste estado indica que se encuentra ejecutando dicha actividad.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.9: Acciones del elemento *Pick***Acciones**

InitElements	Realiza todas las actividades necesarias para inicializar los <i>onAlarm</i> y <i>onMessage</i> asociados al elemento.
Execute	Ejecuta la actividad que fue seleccionada.

FIGURA 6.7: Máquina de estados del elemento *onAlarm*.

CUADRO 6.10: Estados del Elemento *onAlarm***Descripción de Estados**

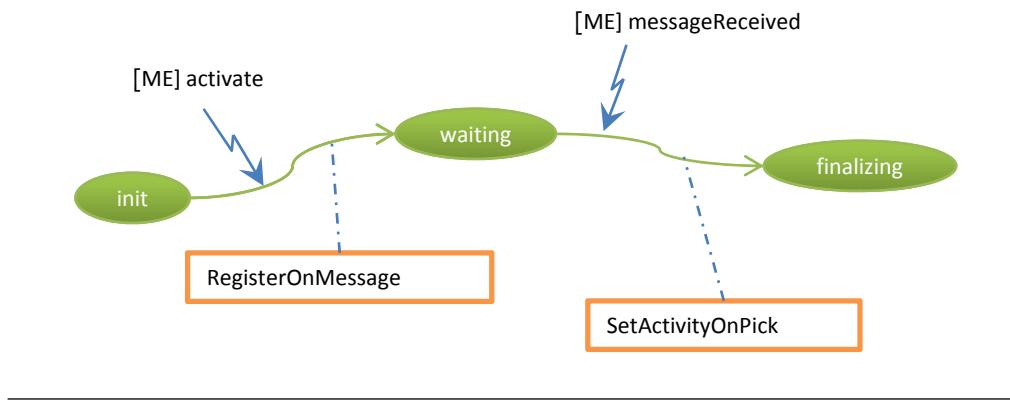
Init	Es el estado inicial del elemento.
Counting	Indica que el temporizador del elemento se encuentra activo.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó, debido a que el temporizador terminó.

CUADRO 6.11: Acciones del elemento *onAlarm***Acciones**

StartCounter	Inicia el temporizador del elemento.
SetActivityOnPick	Le indica al <i>Pick</i> al que pertenece, que terminó el temporizador y le pasa la actividad. Es responsabilidad del <i>Pick</i> decidir si esa es la actividad que se debe ejecutar.

6.2.2.3. Elemento *OnMessage*

El elemento *onMessage* espera hasta recibir un mensaje de un colaborador.

FIGURA 6.8: Máquina de estados del elemento *onMessage*.CUADRO 6.12: Estados del Elemento *onMessage***Descripción de Estados**

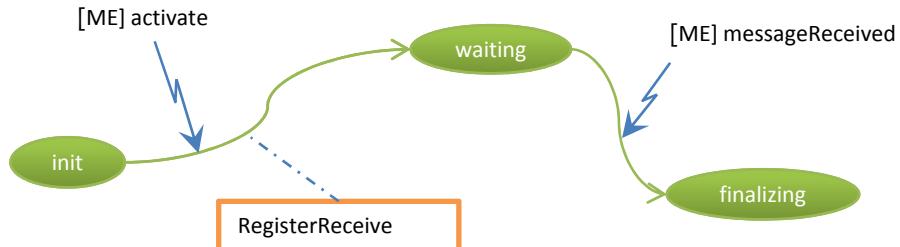
Init	Es el estado inicial del elemento.
Waiting	Una vez el elemento es inicializado, queda en espera hasta que se reciba un mensaje.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

6.2.2.4. Elemento *Receive*

El elemento *Receive* es el encargado de recibir mensajes de los colaboradores.

CUADRO 6.13: Acciones del elemento *onMessage***Acciones**

RegisterOnMessage	Ejecuta la actividad asociada al elemento.
SetActivityOnPick	Le indica al <i>Pick</i> al que pertenece, que recibo un mensaje y le pasa la actividad. Es responsabilidad del <i>Pick</i> decidir si esa es la actividad que se debe ejecutar.

FIGURA 6.9: Máquina de estados del elemento *Receive*.CUADRO 6.14: Estados del Elemento *Receive***Descripción de Estados**

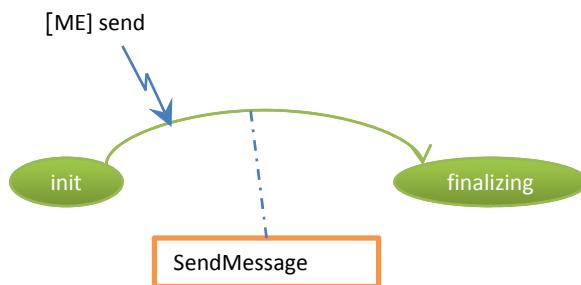
Init	Es el estado inicial del elemento.
Waiting	Una vez el elemento es inicializado, queda en espera hasta que se reciba un mensaje.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.15: Acciones del elemento *Receive***Acciones**

RegisterOnMessage	Ejecuta la actividad asociada al elemento.
-------------------	--

6.2.2.5. Elemento *Reply*

El elemento *Reply* es utilizado para enviar una respuesta a una solicitud previamente aceptada, a través de un elemento de interacción.

FIGURA 6.10: Máquina de estados del elemento *Reply*.

CUADRO 6.16: Estados del Elemento *Reply***Descripción de Estados**

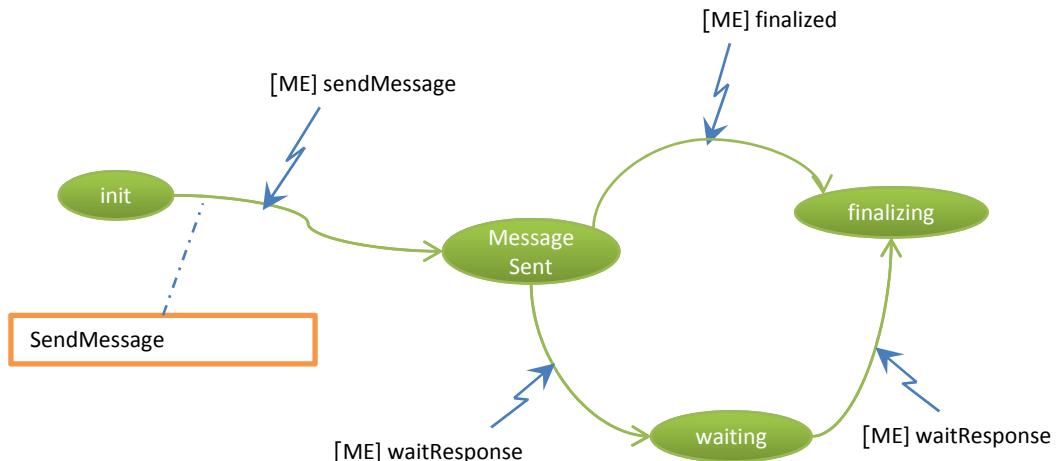
Init	Es el estado inicial del elemento.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.17: Acciones del elemento *Reply***Acciones**

SendMessage	Envía el mensaje.
-------------	-------------------

6.2.2.6. Elemento *Invoke*

El elemento *Invoke* es utilizado para enviar mensajes a los colaboradores. Éste elemento puede ser configurado para que envíe el mensaje y termine o para que envíe el mensaje y quede esperando una respuesta.

FIGURA 6.11: Máquina de estados del elemento *Invoke*.CUADRO 6.18: Estados del Elemento *Invoke***Descripción de Estados**

Init	Es el estado inicial del elemento.
MessageSent	Indica que envío el mensaje al colaborador.
Waiting	Este estado solo es alcanzado si el elemento tiene que esperar una respuesta del colaborador.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.19: Acciones del elemento *Invoke*

Acciones

SendMessage	Envía el mensaje al colaborador.
RegisterReceive	Registra que es una actividad que está esperando la llegada de un mensaje.

6.2.3. Elementos Estructuradores

Los elementos estructuradores son los que forman el proceso, influyendo en qué orden se van a ejecutar los elementos que se encuentran dentro de ellos.

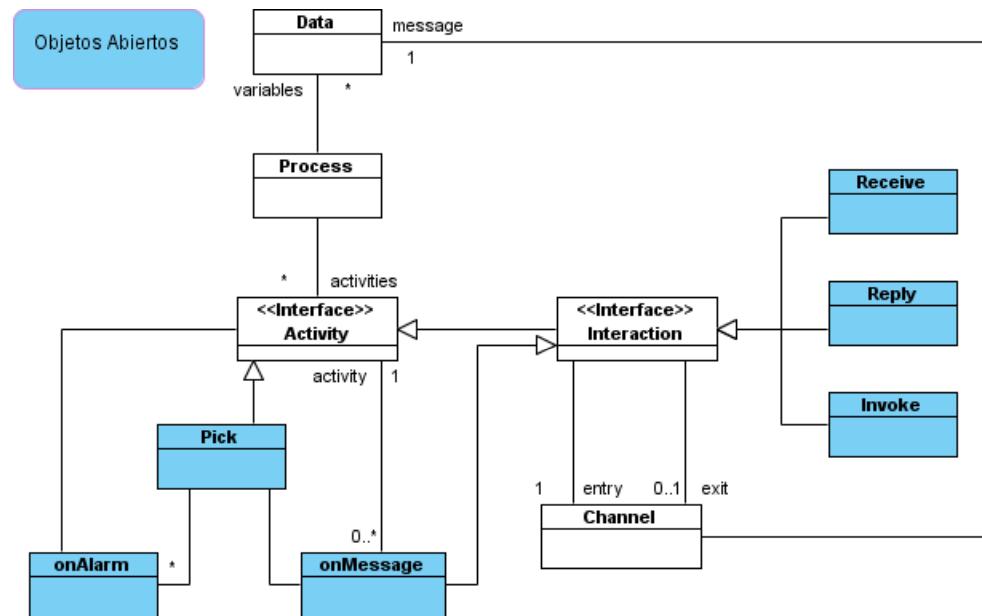


FIGURA 6.12: Elementos estructuradores.

6.2.3.1. Elemento *Sequence*

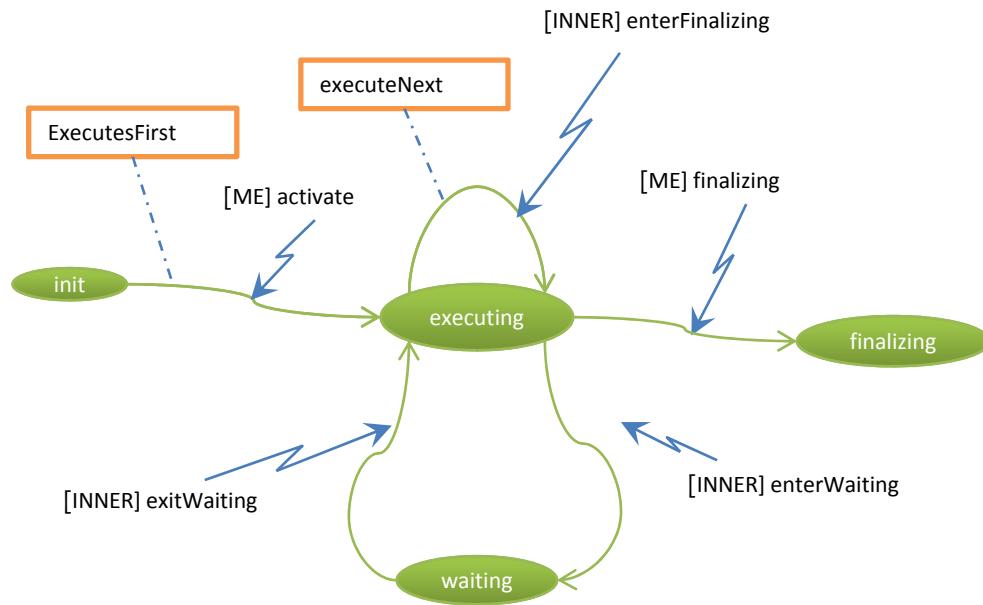
Este elemento contiene una o más actividades que son ejecutadas en serie.

6.2.3.2. Elemento *Flow*

El elemento *Flow* contiene una o más actividades que son ejecutadas en paralelo.

6.2.3.3. Elemento *While*

Este elemento provee una ejecución repetida para el elemento que contiene.

FIGURA 6.13: Máquina de estados del elemento *Sequence*.CUADRO 6.20: Estados del Elemento *Sequence***Descripción de Estados**

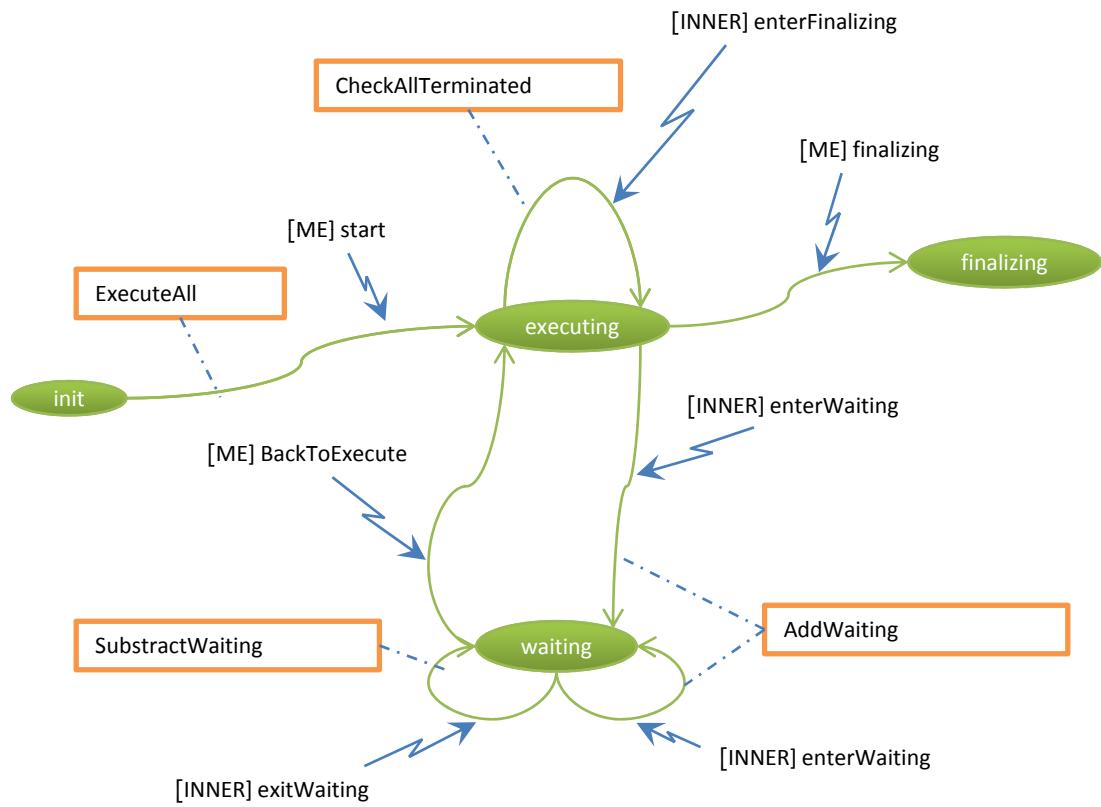
Init	Es el estado inicial del elemento.
Executing	Este estado indica que se encuentra ejecutando una actividad interna.
Waiting	Debido a que la ejecución de un elemento puede bloquear el proceso, el elemento <i>Sequence</i> se bloquea hasta que la actividad en ejecución también se desbloquee.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.21: Acciones del elemento *Sequence***Acciones**

ExecuteFirst	Ejecuta la primera actividad de la secuencia.
ExecuteNext	Ejecuta la siguiente actividad, de no haber siguiente le indica al elemento que terminó.

CUADRO 6.22: Estados del Elemento *Flow***Descripción de Estados**

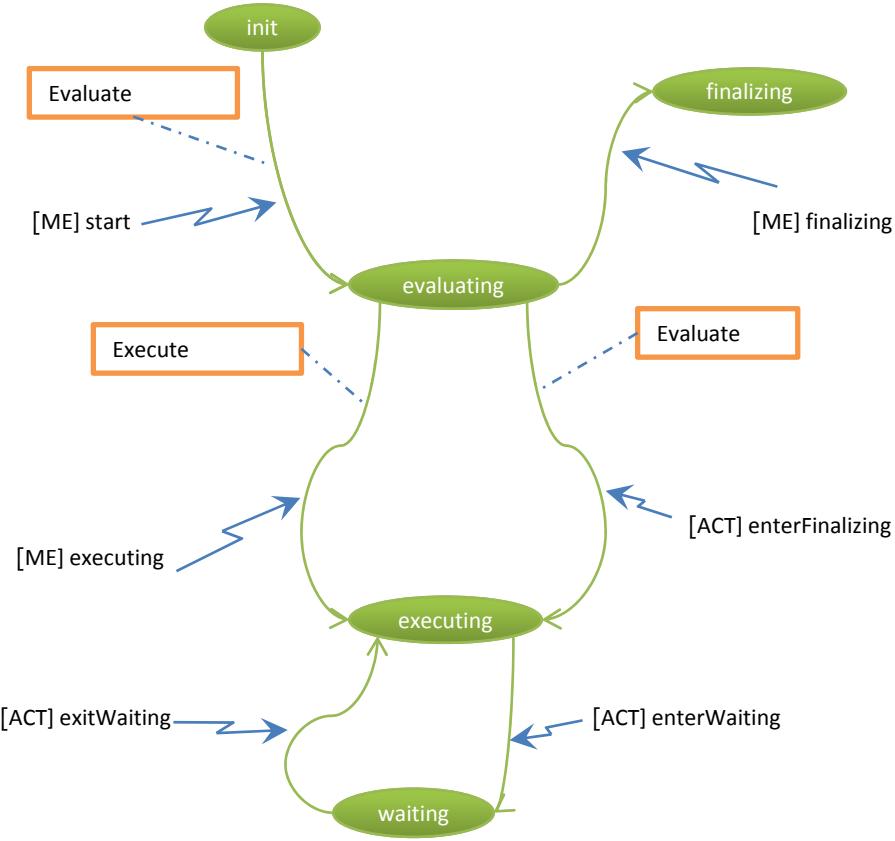
Init	Es el estado inicial del elemento.
Executing	Este estado indica que se encuentra ejecutando alguna de las actividades internas.
Waiting	Debido a que la ejecución de un elemento puede bloquear el proceso, el elemento <i>Flow</i> debe esperar hasta que todas las actividades internas se desbloqueen.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

FIGURA 6.14: Máquina de estados del elemento *Flow*.CUADRO 6.23: Acciones del elemento *Flow***Acciones**

ExecuteAll	Ejecuta todas las actividades que contiene.
CheckAllTerminated	Verifica que todas las actividades hayan terminado su ejecución.
AddWaiting	Agrega a un contador que una actividad está esperando.
SubtractWaiting	Resta del contador una actividad que dejó de esperar.

CUADRO 6.24: Estados del Elemento *While***Descripción de Estados**

Init	Es el estado inicial del elemento.
Evaluating	Indica que se encuentra evaluando la condición de repetición del <i>While</i> .
Executing	Una vez evaluada la condición como cierta, debe pasar a ejecutar el elemento. Este estado indica que se encuentra ejecutando la actividad interna.
Waiting	El elemento interno puede pasar a un estado de espera, lo que bloquea el elemento <i>While</i> hasta que la actividad interna se desbloquee.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

FIGURA 6.15: Máquina de estados del elemento *While*.CUADRO 6.25: Acciones del elemento *While*

Acciones

Evaluate	Evalúa la condición de repetición del elemento.
Execute	Ejecuta el elemento interno.

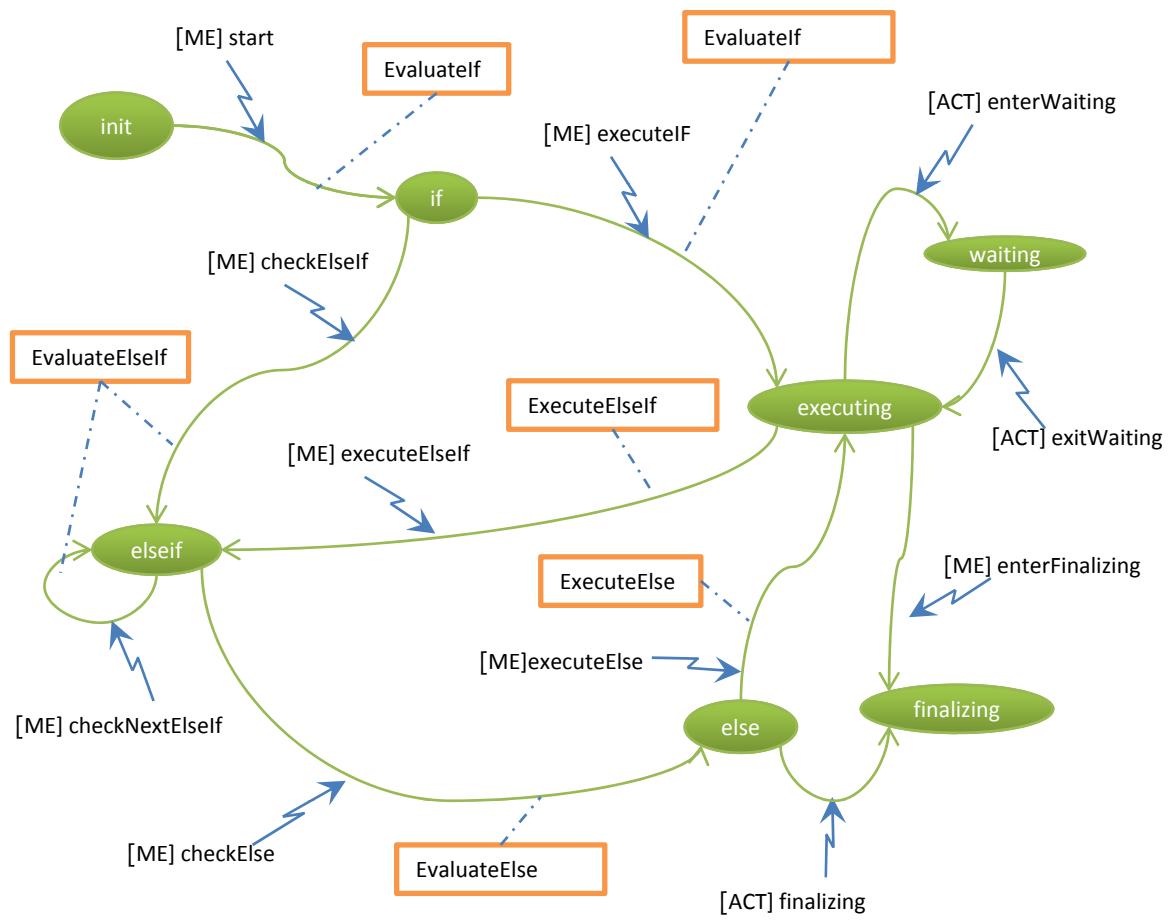
6.2.3.4. Elemento *Condicional*

BPEL no tiene un elemento llamado “condicional”, pero provee elementos como *If*, *ElseIf*, *Else* para proveer este comportamiento. Se decidió encapsular a los tres elementos dentro de una actividad *Condicional* para poder exponer el estado de la ejecución de los tres elementos, en uno solo.

6.2.4. Instrucciones

6.2.4.1. Elemento *Exit*

Éste elemento es utilizado para terminar de ejecutar la instancia del proceso.

FIGURA 6.16: Máquina de estados del elemento *Condicional*.CUADRO 6.26: Estados del Elemento *Condicional***Descripción de Estados**

Init	Es el estado inicial del elemento.
If	Indica que el elemento está realizando la evaluación de la condición que se encuentra definida para el <i>If</i> .
ElseIf	Indica que la evaluación del elemento <i>If</i> fue negativa, y que está realizando la evaluación de la condición que se encuentra definida para cada uno de los <i>ElseIf</i> .
Else	Indica que la evaluación de todas las condiciones de los elementos <i>ElseIf</i> fue negativa, resultando en la ejecución de la condición por defecto o en la ejecución de nada.
Executing	Indica que encontró una condición verdadera y que está ejecutando la actividad del elemento correspondiente.
Waiting	Si la actividad en ejecución debe bloquearse, el elemento tiene que esperar hasta que la actividad continúe con su ejecución.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.27: Acciones del elemento *Condicional*

Acciones

EvaluateIf	Evalua la condición del elemento <i>If</i> , en caso de ser verdadera ejecuta la actividad allí definida, de lo contrario comienza a evaluar los <i>ElseIf</i> .
ExecuteIf	Ejecuta la actividad definida para el elemento <i>If</i> .
EvaluateElseIf	Evalua la condición de los elementos <i>ElseIf</i> , en caso de encontrar uno que sea verdadera, ejecuta su actividad asociada.
ExecuteElseIf	Ejecuta la actividad asociada con el elemento <i>ElseIf</i> .
ExecuteElse	Ejecuta la actividad asociada al elemento <i>Else</i> .

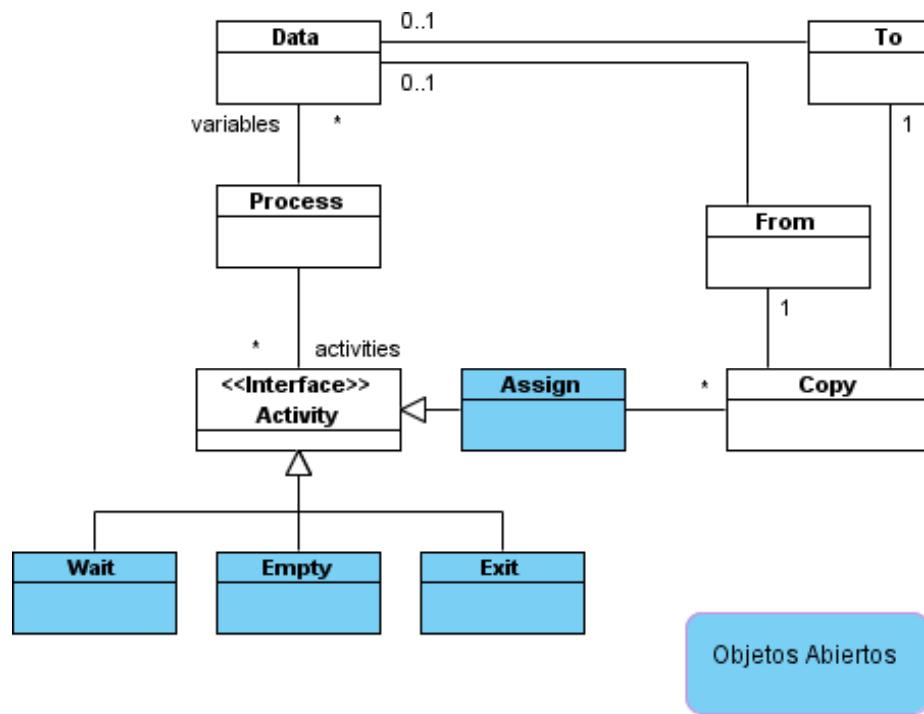


FIGURA 6.17: Instrucciones.

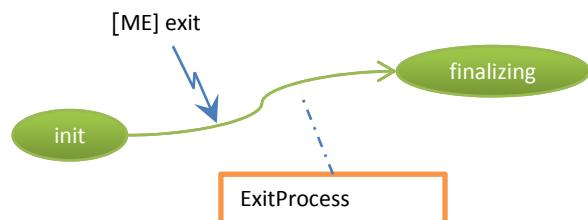


FIGURA 6.18: Máquina de estados del elemento *Exit*.

CUADRO 6.28: Estados del Elemento *Exit***Descripción de Estados**

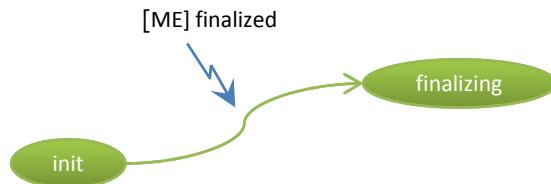
Init	Es el estado inicial del elemento.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.29: Acciones del elemento *Exit***Acciones**

ExitProcess	Realiza todas las actividades necesarias para terminar la ejecución de la instancia del proceso.
-------------	--

6.2.4.2. Elemento *Empty*

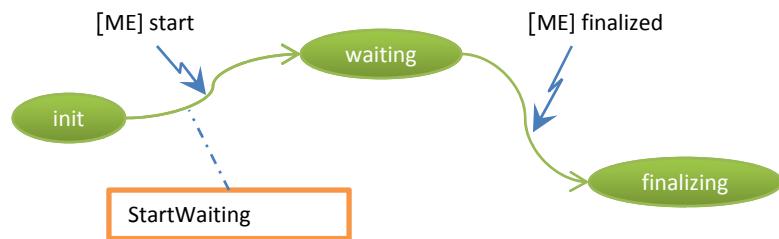
Este es un elemento que representa una actividad que no hace nada.

FIGURA 6.19: Máquina de estados del elemento *Empty*.CUADRO 6.30: Estados del Elemento *Empty***Descripción de Estados**

Init	Es el estado inicial del elemento.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

6.2.4.3. Elemento *Wait*

Éste elemento especifica una espera de un tiempo determinado o hasta que determinado plazo sea alcanzado.

FIGURA 6.20: Máquina de estados del elemento *Wait*.

CUADRO 6.31: Estados del Elemento *Wait***Descripción de Estados**

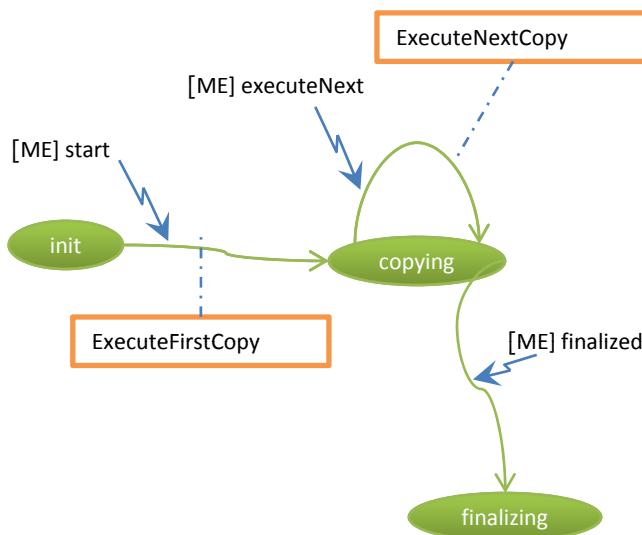
Init	Es el estado inicial del elemento.
Waiting	Indica que la actividad se encuentra esperando que se venza el plazo determinado se termine un temporizador.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.32: Acciones del elemento *Wait***Acciones**

StartingWait	Inicia el temporizador.
--------------	-------------------------

6.2.4.4. Elemento *Assign*

Éste elemento es utilizado para copiar datos de una variable a otra o para insertar nuevos valores.

FIGURA 6.21: Máquina de estados del elemento *Assign*.CUADRO 6.33: Estados del Elemento *Assign***Descripción de Estados**

Init	Es el estado inicial del elemento.
Copying	Indica que se está copiando los valores de una variable a otra.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 6.34: Acciones del elemento *Assign***Acciones**

ExecuteFirstCopy	Realiza la copia del primer elemento interno.
ExecuteNext	Realiza la copia del siguiente elemento si existe.

6.3. Arquitectura

En ésta sección se hablará de la arquitectura de Caffeine.

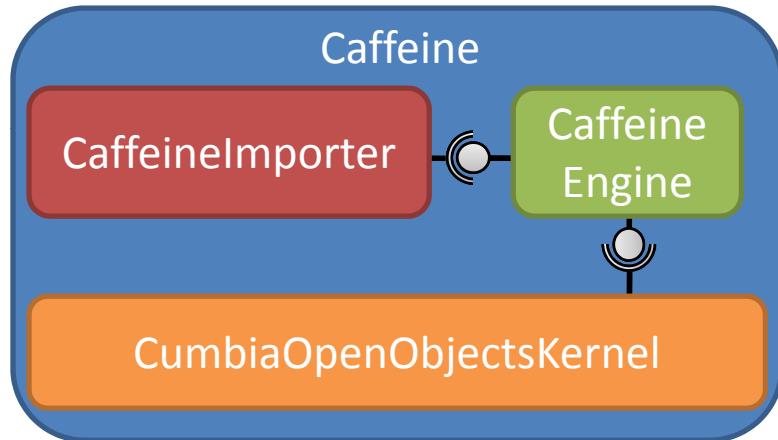


FIGURA 6.22: Arquitectura de Caffeine.

6.3.1. CumbiaOpenObjectsKernel

El trabajo con metamodelos ejecutables extensibles ha sido un trabajo constante en el proyecto Cumbia. A partir de la tesis de maestría de Pablo Barvo y Mario Sánchez[43] se desarrolló un *framework* para la fácil generación de otras aplicaciones y fácil definición de nuevos metamodelos ejecutables usando objetos abiertos. A continuación se describe a grandes rasgos cuales son las responsabilidades de éste *framework* y a describir algunos de sus elementos. Éste componente fue desarrollado por Mario Sánchez, uno de los integrantes del Proyecto Cumbia como parte de su trabajo doctoral.

La responsabilidad principal de éste *Kernel*, es la de a partir de una serie de archivos que definen el metamodelo, crear e instanciar sus modelos, siendo sus elementos objetos abiertos o elementos que puedan generar o esperar eventos. Además, es el responsable de realizar la coordinación de todas las máquinas de estado de los objetos.

El primer archivo que debe crearse para poder utilizar el *framework* es un XML (código 6.1) donde se definen cuales son los elementos que componen el metamodelo. De

```

1 <metamodel name="BPEL" version="0.1">
3   <!-- State machines used by the elements of the metamodel -->
5     <state-machine-reference name="assign" file="assign.xml" />
6     ...
7     <state-machine-reference name="while" file="while.xml" />
8
9     <!-- Elements of the metamodel -->
10    <type name="Assign" class="uniandes.cumbia.bpel.elements.assign.Assign"
11      statemachine="assign"/>
12    ...
13    <type name="Copy" class="uniandes.cumbia.bpel.elements.assign.Copy" />
14    ...
15    <type name="From" class="uniandes.cumbia.bpel.elements.assign.from.From" />
16
17    ...
18 </metamodel>

```

CÓDIGO 6.1: Archivo de declaración de los elementos del metamodelo.

igual manera en éste archivo se indica cuál es el descriptor que define la máquina de estados para cada elemento.

Luego de tener la definición de los elementos que componen el metamodelo, se debe implementar para cada metamodelo definido una serie de activos que permitan construir e instanciar el modelo con la estructura específica, de tal manera el *Kernel* sabe como cargar de un archivo la estructura del modelo. Además de saber cómo cargar la estructura del modelo, debe saber cuál es el modelo que debe crear, éste se define en un archivo archivo que contiene específicamente cuales son los elementos que componen un modelo en particular (código 6.2). Por último, se debe crear la implementación de todos los elementos que hacen parte del modelo y asociarlos al *Kernel*.

6.3.2. CaffeineEngine

Éste es el componente principal de Caffeine. Éste es el motor encargado de hacer *deploy* de las definiciones de los procesos, crear las nuevas instancias de los procesos y el encargado de hacer la coordinación del intercambio de mensajes. Ver figura 6.23.

Process Space

El *Process Space* es encargado de mantener las definiciones de proceso y crear las instancias de los procesos cuando sea necesario. Por cada proceso existente en el motor hay un correspondiente *Process Space*.

```

1<?xml version="1.0" encoding="UTF-8"?>
2<definition metamodel="BPEL" modelName="HelloWorld" version="0.1">
3    <metamodel-extensions/>
4    <elements>
5        <open-object name="HelloWorld" typeName="Process"/>
6        ...
7        <open-object name="replyOutput" typeName="Reply"/>
8    </elements>
9    <runtime/>
10   <model>
11       <process ...>
12
13           <!-- List of services participating in this BPEL process -->
14           <partnerLinks>
15               ...
16           </partnerLinks>
17
18           <!-- List of messages and XML documents used as part of this BPEL
19               process -->
20           <variables>
21               ...
22           </variables>
23
24           <!-- Orchestration Logic -->
25           <sequence name="sequence">
26               ...
27           </sequence>
28       </process>
29   </model>
30</definition>

```

CÓDIGO 6.2: Archivo de definición de los elementos del metamodelo.

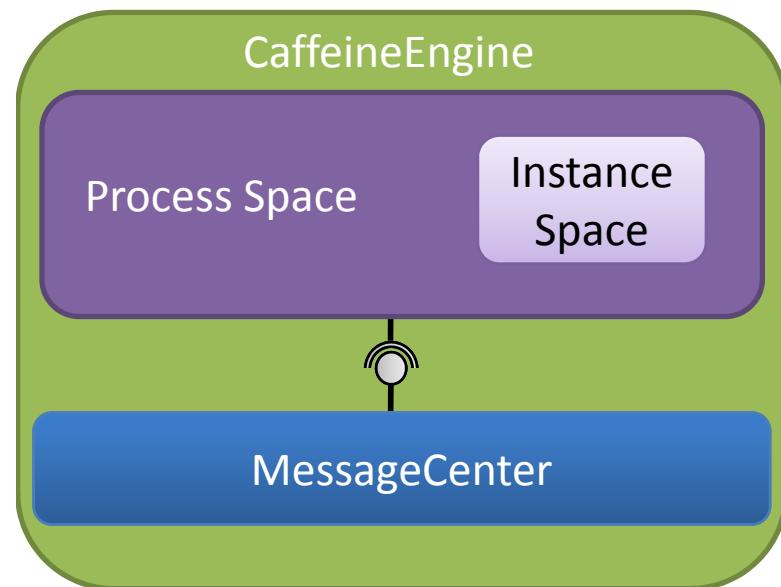


FIGURA 6.23: Arquitectura de CaffeineEngine.

Instance Space

El *Instance Space* es quien contiene al elemento raíz del proceso (*Process*). De igual manera contiene una serie de elementos que monitorean la ejecución de elementos para propósitos administrativos.

Message Center

Este componente es el encargado de la administración de los mensajes recibidos o enviados por los colaboradores. Cuando recibe un mensaje, se encarga de localizar la instancia a la cual le pertenece y entregársela al elemento que le corresponde. De igual manera mantiene una lista de todos los colaboradores que hacen parte de los procesos, para que los procesos puedan enviar los mensajes a los colaboradores.

Protocolo de Instanciación de un Proceso

Éste protocolo describe la manera cómo se realiza la creación de una nueva instancia de proceso cuando se recibe un mensaje que no está dirigido a una instancia existente.

1 - El Web service que representa el proceso recibe un mensaje dirigido a alguno de los elementos de inicio del proceso.

2 - El *MessageCenter* es encargado de revisar que el mensaje no esté dirigido a una instancia existente del proceso.

3 - Al no existir una instancia a la cual pertenezca el mensaje, el *MessageCenter* crea una instancia nueva del proceso.

4 - Luego el mensaje es transmitido a la actividad de inicio a la cual es dirigido.

5 - La actividad de inicio procesa el mensaje y le indica al proceso que debe comenzar su ejecución.

Protocolo de Envío de un Mensaje a un Colaborador

A continuación se presenta los pasos para realizar la invocación de un Web service.

1 - Durante la ejecución de una instancia, una actividad *Invoke* le indica al motor que va a invocar un servicio (proporcionando la información de éste) y el mensaje a ser enviado.

2 - El motor, utilizando el Web service que representa el proceso, solicita la invocación del *partnerLink* especificado al *MessageCenter*.

Protocolo de Recepción de Mensaje por una Instancia Existente

Este protocolo describe la manera como se realiza la creación de una nueva instancia de proceso cuando se recibe un mensaje que está dirigido a una instancia existente.

1 - Durante la ejecución de una instancia de proceso, una actividad *receive* pide al motor que la registre como una actividad que está esperando un mensaje.

2 - Cuando un mensaje llega, el *MessageCenter* verifica entre los elementos que se registraron para esperar un mensaje.

3 - Al encontrar el elemento al cual le pertenece el mensaje recibido, se lo notifica

4 - Al entregar el mensaje, la actividad se elimina de la lista de actividades que están esperando mensaje.

6.3.3. CaffeineDeployer

La responsabilidad de este componente es la de traducir el archivo BPEL de entrada al archivo que recibe el *framework* de objetos abiertos y de generar los archivos requeridos para publicar el proceso como un Web service.

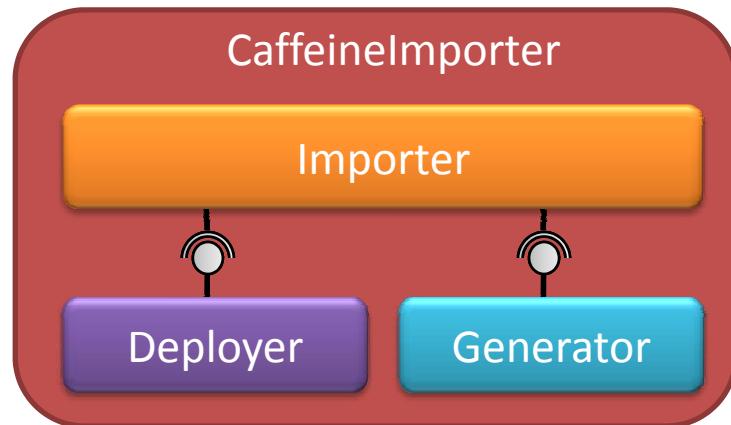


FIGURA 6.24: Arquitectura de CaffeineDeployer.

Importer

El *Importer* es el componente principal del *CaffeineDeployer*. Éste es el encargado de leer un archivo con un proceso BPEL y crear una serie de objetos que representan el proceso, pero estos objetos no tienen ningún tipo de lógica, son creados para mantener la información que está contenida en los nodos XML.

Generator

Parte del *Generator* son una serie de *visitors* que recorren la estructura de los elementos que representan el proceso y generan el archivo del modelo discutido en la sección 6.3.1. El *Generator* permite la generación de las clases para definir un Web service a partir del archivo WSDL del proceso BPEL.

Deployer

El Deployer también es el encargado de publicar los procesos BPEL, una vez finalice la importación, para que puedan ser utilizados como Web services.

6.4. Pruebas

Para probar la correcta implementación de *Caffeine* se utilizó otro de los activos del proyecto Cumbia, el *CumbiaTestFramework*, el cual fue desarrollado como tesis de grado[41] por Sergio Moreno.

6.4.1. Framework de Pruebas

El *framework* de pruebas provee un mecanismo para poder hacer pruebas sobre aplicaciones basadas en objetos abiertos. La idea principal del *framework* de pruebas es poder monitorear el comportamiento de un motor sin interferir con su ejecución, lo que presenta una serie de retos debido a que es necesario monitorear todos los elementos que pertenecen a un proceso, sus eventos, maquinas de estado, etc.

Para hacer esto, el *framework* está dividido en un modelo de capas, que representan las diferentes etapas por las cuales puede pasar un proceso. La primera capa es la capa que tiene el metamodelo que se quiere probar. La segunda capa es la encargada de materializar los modelos que se van a probar. La tercera capa es la encargada de

crear la instancia del proceso que se quiere probar. En la cuarta capa están definidos dos elementos básicos: sensores y trazas. Los sensores son elementos que se colocan sobre los elementos que se quiere monitorear, de ésta manera es posible conocer, por ejemplo, los eventos que han sido generados desde cierto elemento o los estados de una máquina que se activaron. Cuando un sensor es activado por alguno de los elementos que está monitoreando, esa información es guardada en una traza, de tal manera teniendo un registro de que sensores fueron activados por cuales elementos y porque razón. Por último existe la capa de prueba, ésta capa es la encarga de realizar aserciones sobre las trazas que fueron creadas durante la ejecución del proceso, así es posible conocer si la traza que era esperada fue la traza que el proceso generó.

Una prueba en el *CumbiaTestFramework* está definida como una serie de escenarios de prueba. Para cada escenario de prueba es necesario definir la estructura de los elementos que se va a probar, para el caso de *Caffeine*, que proceso BPEL es el que se quiere probar. Se debe especificar un lenguaje, llamado el lenguaje de animación, donde se enumere mediante instrucciones, cual es el comportamiento que se le quiere dar al proceso, es decir, se definen instrucciones que mueven el proceso. En el caso de *Caffeine*, por ejemplo, se debe especificar en ese lenguaje como mandarle un mensaje a un proceso BPEL. En la prueba también se debe definir a que elementos se les va a colocar un sensor y qué tipo de sensor. Para finalizar se definen las aserciones para cada escenario de prueba.

Escenarios de prueba

Para *Caffeine* se definieron 14 escenarios de prueba, cada uno de ellos probando diferentes elementos y su interacción. Uno de los escenarios definidos es un proceso simple que comienza cuando un elemento *receive* recibe un mensaje, luego ese mensaje es modificado por una actividad *assign* para luego ser retornado a quien envió el mensaje original.

Lenguaje de Animación

El lenguaje de animación para BPEL es relativamente sencillo. El comportamiento que se quiere simular es el intercambio de mensajes a una instancia de proceso existente.

Ejecutar una Llamada Síncronica - Ésta instrucción permite decirle al *MessageCenter* que envíe el mensaje que se definido a la instancia de proceso definida y espere a recibir una respuesta del proceso.

```
1 <execute-synchronous-call processName="HelloWorld" processID="0" instanceID="0" message="initial message" />
```

CÓDIGO 6.3: Instrucción sincrónica de envío de mensaje

Ejecutar una Llamada Asincrónica - Ésta instrucción permite decirle al *MessageCenter* que envié el mensaje que se definido a la instancia de proceso definida pero sin necesidad de esperar un mensaje de respuesta del proceso.

```
1 <execute-asynchronous-call processName="HelloWorld" processID="0"
    instanceID="0" message="initial message" />
```

CÓDIGO 6.4: Instrucción sincrónica de envío de mensaje

Crear un Servicio - Ésta instrucción permite decirle crear servicios colaboradores que van a esperar mensajes del proceso, por ejemplo, si dentro del proceso se tiene una actividad *invoke* que debe enviar un mensaje, el servicio que se crea con ésta instrucción es el encargado de recibir ese mensaje.

```
1 <createTestWsdlService processName="HelloWorld" serviceName="testWSDL" type
    ="dummy" />
```

CÓDIGO 6.5: Instrucción de creación de un servicio colaborador

Sensores

Se crearon dos tipos de sensores para los elementos. El primero de ellos es el encargado de monitorear las máquinas de estado de los elementos, de ésta manera es posible conocer cuáles fueron los elementos que se activaron y en qué orden, así es posible realizar aserciones acerca del orden de ejecución de los elementos. El segundo tipo de sensor es el encargado de monitorear si los valores de las variables cambiaron durante la ejecución del proceso.

6.5. Experimentación *Caffeine* Cliente Web

Para poder interactuar con *Caffeine* se construyó una interfaz Web para poder visualizar desde el punto de vista del usuario final. Al ingresar, la interfaz permite que los usuarios vean los procesos que se encuentran en el motor, además de ver las instancias de cada uno de los procesos. El usuario puede hacer *deploy* de nuevos procesos e instanciarlos, suministrando el mensaje de entrada para el proceso. Una vez el proceso ha completado se muestra su resultado. La interfaz provee dos vistas para ver la información de los procesos que se encuentran en el motor. La primera es una vista en forma

de árbol (figura 6.25), donde se listan todas las actividades de los procesos y es posible ingresar a cada actividad y ver la información de cada una, por ejemplo, si se selecciona un *reply*, es posible ver su nombre, la operación asociada, la variable y la información del colaborador (figura 6.26). La segunda vista permite a los usuarios ver cuando se instanció y de haber terminado, cuando terminó la ejecución de una instancia específica del proceso.

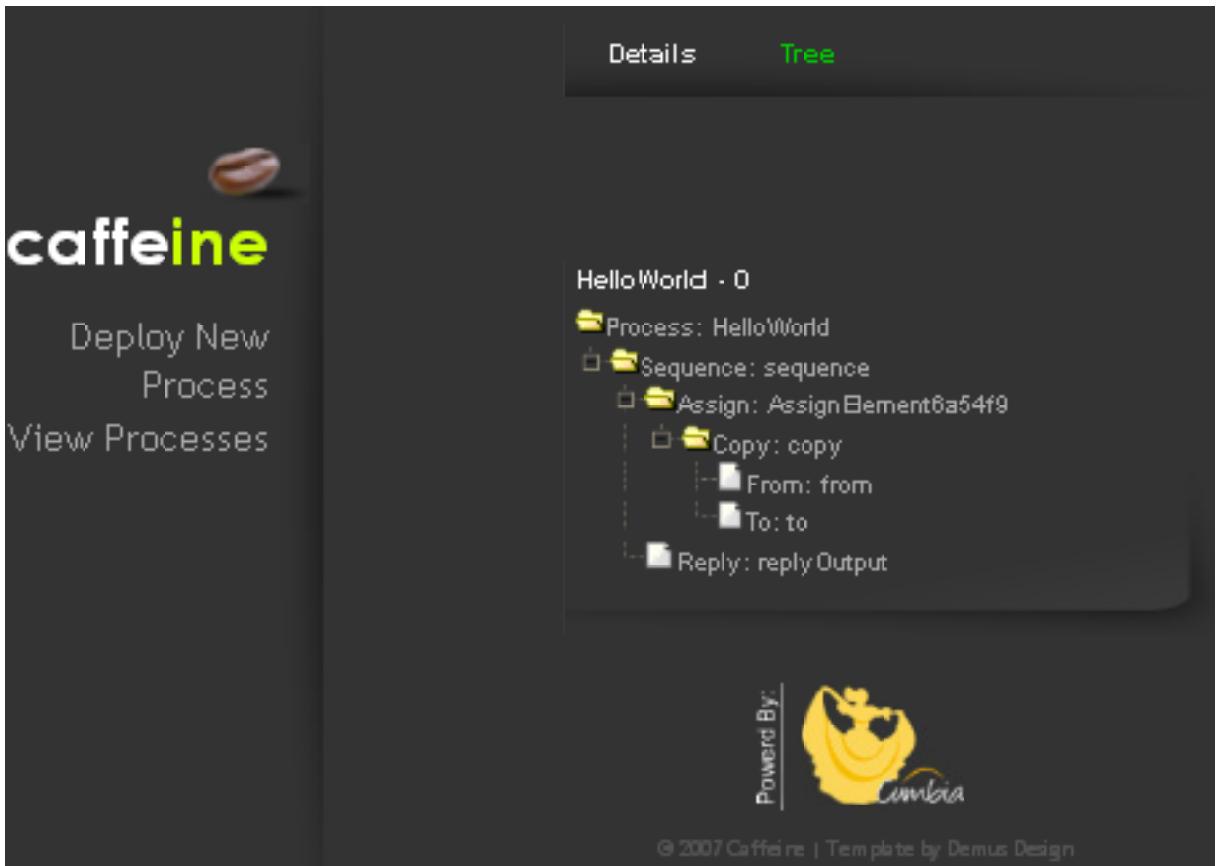


FIGURA 6.25: Vista en forma de árbol de los elementos de un proceso.

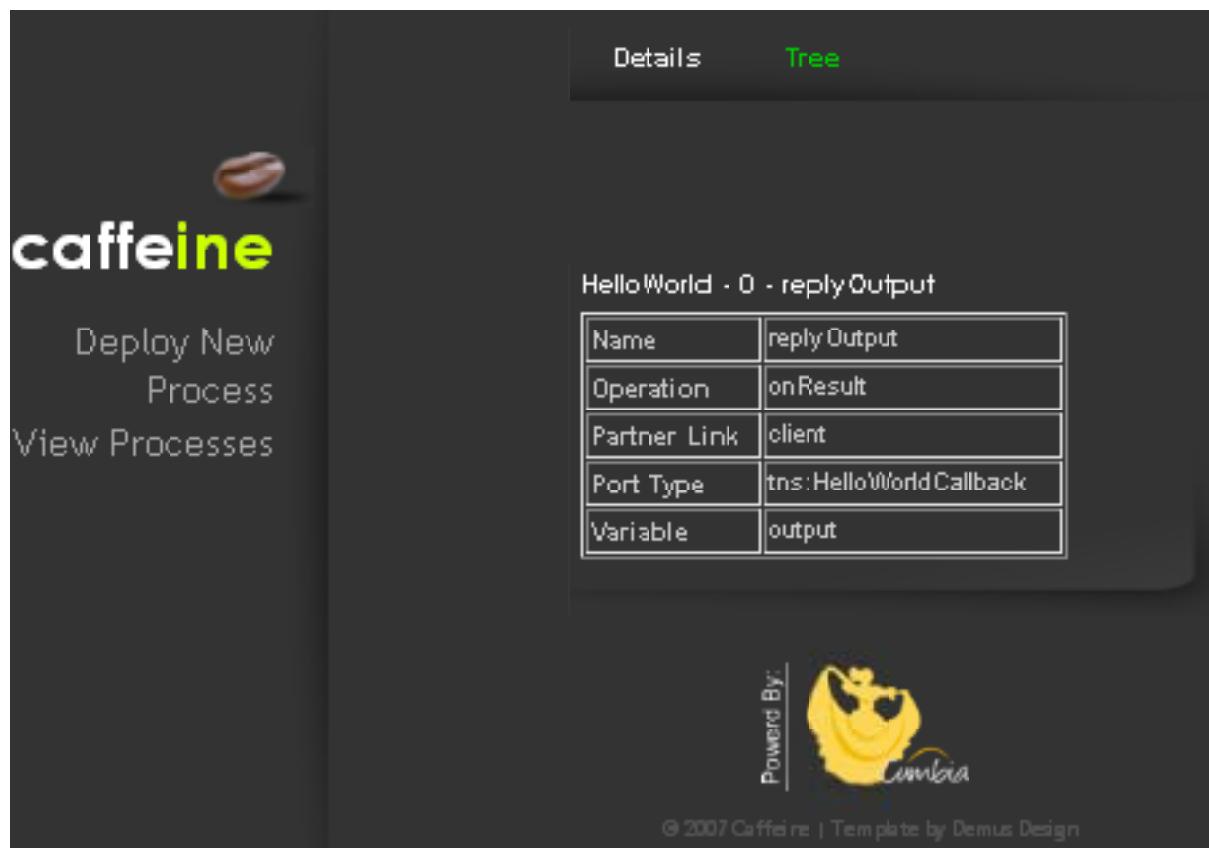


FIGURA 6.26: Detalles de una actividad del proceso.

Capítulo 7

AspectCaffeine: Extensión de Aspectos a Caffeine

7.1. Introducción

AspectCaffeine es una extensión desarrollada sobre *Caffeine* para poder integrar aspectos para resolver las problemáticas expuestas en el capítulo 4. Además se hace una propuesta para manejar la interferencia de aspectos sobre *shared join points*.

Se va a utilizar la categorización descrita en la sección 4.3 para exponer como está construido *AspectCaffeine*.

7.2. Modelo de *Join Points*

AspectCaffeine soporta dos tipos de *join points*: los *join points* de actividades que corresponden a la ejecución de las actividades y los *join points internos*, que corresponden a puntos internos en la ejecución de las actividades. Los *join points* se han denominado *transition points*, porque es posible colocarlos en cualquier transición de cualquier actividad BPEL, no solamente dentro de las actividades de interacción.

7.3. Lenguaje de Puntos de Corte

De acuerdo con las limitaciones del lenguaje de puntos de corte de *AspectJ* descritas en la sección 3.2.2, como una solución a éste problema, se ha propuesto que los lenguajes

de puntos de corte identifiquen los elementos de acuerdo a sus características, por ejemplo seleccionar cierto elemento que se encuentra en cierta instancia de proceso.

El lenguaje de puntos de corte para *AspectCaffeine* utiliza un lenguaje similar a los lenguajes de consulta como *XPath* o *XQuery*, la razón para utilizar un lenguaje propio es que el lenguaje también tiene que tener en cuenta las transiciones de los estados de los elementos.

El lenguaje de puntos de corte permite:

- Seleccionar todos los elementos dado un tipo, para todos los procesos. El ejemplo muestra las instrucciones del lenguaje de puntos de corte para seleccionar todos los elementos de tipo *invoke*, para todos los procesos.

```
1 *Invoke
```

CÓDIGO 7.1: Seleccionar todos los elementos dado un tipo para todos los procesos.

- Existe una variación al anterior, la cual provee la posibilidad de seleccionar todos los elementos dado un tipo y un nombre, para todos los procesos. Se muestra como se quiere seleccionar todos los elementos de tipo *invoke* que se llaman *InvocarServicioFacturacion*, para todos los procesos.

```
1 *Invoke [name=InvocarServicioFacturacion ]
```

CÓDIGO 7.2: Seleccionar todos los elementos dado un tipo y dado un nombre para todos los procesos.

- También es posible definir que se quiere seleccionar una transición de un elemento dado su tipo para todos los procesos. Para éste caso se seleccionará la transición llamada *ToCalculatingNextAdvice*, de todos los elementos de tipo *invoke*, para todos los procesos.

```
1 *Invoke->ToCalculatingNextAdvice
```

CÓDIGO 7.3: Seleccionar una transición en todos los elementos dado un tipo para todos los procesos.

- También es posible seleccionar una transición de un elemento dado su tipo y su nombre para todos los procesos. Para éste caso se seleccionará la transición llamada *ToCalculatingNextAdvice*, de todos los elementos de tipo *invoke* llamados *InvocarServicioFacturacion*, para todos los procesos.

```
1 *Invoke [name=InvocarServicioFacturacion]->ToCalculatingNextAdvice
```

CÓDIGO 7.4: Seleccionar para todos los procesos una transición dado su nombre en todos los elementos dado el tipo y el nombre.

- Seleccionar todos los elementos dado un tipo, para todas las instancias de los procesos dado su nombre. Para éste caso se seleccionará todos los elementos de tipo *invoke* para el proceso llamado Shazam.

```
1 *Invoke | Shazam
```

CÓDIGO 7.5: Seleccionar para un proceso dado su nombre los elementos dado su tipo

- Seleccionar todos los elementos dado un tipo y su nombre, para todas las instancias de los procesos dado su nombre. Para éste caso se seleccionará todos los elementos de tipo *invoke* con nombre InvocarServicioFacturacion, para el proceso llamado Shazam.

```
1 *Invoke [name=InvocarServicioFacturacion] | Shazam
```

CÓDIGO 7.6: Seleccionar para un proceso dado su nombre todos los elementos dado el tipo y el nombre.

- Seleccionar una transición dado su nombre, para todos los elementos dado un tipo y su nombre, para todas las instancias de los procesos dado su nombre. Para éste caso se seleccionará la transición llamada ToCalculatingNextAdvice, para todos los elementos de tipo *invoke* con nombre InvocarServicioFacturacion, para el proceso llamado Shazam.

```
1 *Invoke [name=InvocarServicioFacturacion] | Shazam->
    ToCalculatingNextAdvice
```

CÓDIGO 7.7: Seleccionar para un proceso dado su nombre una transición dado su nombre en todos los elementos dado el tipo y el nombre.

- Seleccionar un elemento específico dado su tipo, su nombre y la ubicación exacta en la estructura de elementos que componen un proceso dado su nombre. Para este ejemplo se seleccionará el elemento llamado InvocarServicioFacturacion de tipo *invoke*, que se encuentra dentro de una elemento *sequence* llamado secuencia, dentro de un proceso llamado Shazam.

```
1 *Invoke | Shazam:secuencia:InvocarServicioFacturacion
```

CÓDIGO 7.8: Seleccionar para un proceso dado su nombre un elemento dando su localización y su nombre.

- Seleccionar una transición de un elemento específico dado su tipo, su nombre y la ubicación exacta en la estructura de elementos que componen un proceso dado su nombre. Para este ejemplo se seleccionará la transición llamada ToCalculatingNextAdvice, del elemento llamado InvocarServicioFacturacion de tipo *invoke*, que se encuentra dentro de una elemento *sequence* llamado secuencia, dentro de un proceso llamado Shazam.

```

1 *Invoke | Shazam:secuencia:InvocarServicioFacturacion ->
    ToCalculatingNextAdvice

```

CÓDIGO 7.9: Seleccionar para un proceso dado su nombre una transición dado su nombre en todos los elementos dado el tipo el nombre y la ubicación dentro de la estructura.

7.4. Lenguaje de *Advices*

Como se discutió en la sección 4.3.3, el lenguaje de los *advices* generalmente es el mismo lenguaje de la aplicación base. Para el caso de *AspectCaffeine* se decidió que el lenguaje de los *advices* sería BPEL. El *advice* tiene un atributo tipo, el cual especifica si el *advice* va a ser ejecutado antes, después o en vez del elemento que identifica el punto de corte.

7.4.1. Advice del Aspecto de Facturación

En la sección ?? se mostró un aspecto de facturación, en *AspectCaffeine* ese aspecto se definiría como se muestra en el código 7.10.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2   <aspect name="facturacion">
3     <transitionPoint name="TP1" pointcut="*Invoke | Shazam">
4       <advice name="invokeFacturacion" type="before">
5         <partnerLinks>
6           <partnerLink name="recaudo"
7             partnerLinkType="tns:FacturacionShazam"
8             myRole="RecaudoRequester"
9             partnerRole="RecaudoProvider"
10            />
11        </partnerLinks>
12        <variables>
13          <variable name="facturacionInfo" messageType="
14            tns:FacturacionMessage" />
15        </variables>
16        <assign>
17          <copy name="copy">
18            <from name="from">bpel:getVariableData( 'input' , 'payload'
19              , '/tns:userName')</from>
20            <to name="to" variable="facturacionInfo" part="payload"
21              query="/tns:result" />
22          </copy>
23        </assign>

```

```

21   <invoke name="invoke" operation="initiate" partnerLink="recaudo"
22     portType="tns:FacturacionShazam" inputVariable="facturacionInfo" /
23   >
</advice>
23 </transitionPoint>
</aspect>
```

CÓDIGO 7.10: Aspecto de facturación definido en *AspectCaffeine*

En la línea 4 se puede observar que al *advice* se le da un nombre para que pueda ser identificado. Allí mismo se le da el tipo que para éste caso es un *advice* de tipo *before*. Luego le siguen las líneas que corresponden al código de la solución de la preocupación transversal. Primero se declara un nuevo *partner link* que define la comunicación con el colaborador, luego la variable que tendrá la información que se quiere enviar al colaborador. En seguida, se agregan las nuevas actividades, la primera asigna el valor de la variable que será enviada al servicio colaborador y la segunda la actividad que envía la información.

7.5. Tejido de Aspectos

El tejido entre aspectos y los procesos BPEL puede ocurrir en dos momentos. El primero de ellos es cuando se hace *deploy* de un nuevo aspecto. El segundo es cuando se crea una nueva instancia de un proceso BPEL.

Para el caso cuando se hace *deploy* de una nueva definición de un aspecto, lo primero que se hace es ubicar todos los elementos que son afectados por ese aspecto, de acuerdo a lo que se definió para el punto de corte. En el caso cuando se crea una nueva instancia de un proceso BPEL, se revisan todos los elementos de la instancia creada, buscando posibles puntos de corte.

Una vez se tienen los puntos donde se debe colocar el aspecto, por cada uno de ellos se crea una nueva instancia del aspecto, para luego asignarla a cada uno de los puntos. En éste momento tanto el aspecto como el elemento que es afectado por el aspecto se presentan.

El siguiente paso depende del tipo de *advice* que tenga el aspecto. En caso que sea un *advice* de tipo *before* se modifica la máquina de estados del elemento que es afectado por el aspecto, para que en la transición del estado inicial al siguiente estado se coloque una primera acción que indica al aspecto que debe ejecutarse, de esta manera es posible pasar el control de la ejecución del elemento BPEL al aspecto, cuando el elemento BPEL es inicializado. En caso que el tipo del *advice* sea *after* se procede de manera similar, se

debe ubicar el último estado, donde se coloca una última acción que ejecuta el aspecto allí localizado. Para el caso de un *advice* de tipo *around*, la modificación de la máquina de estados va más allá de agregar una nueva acción. Para éste caso es necesario agregar una nueva transición del estado inicial del elemento al estado final con una única acción que ejecuta el aspecto. De ésta manera, cuando es momento de ejecutar el aspecto, se toma la nueva transición agregada para que el aspecto sea ejecutado, sin necesidad de ejecutar el elemento. Esto es posible gracias a que las máquinas de estado que están escuchando los eventos de otros elementos, escuchan son los eventos que se lanzan cuando una transición de una máquina de estado ingresa a un estado, no los eventos que son lanzados por la entidad a la cual corresponde la máquina de estados. Por ejemplo, la máquina del elemento *sequence* que depende de que los elementos internos terminen, escucha cuando la máquina de estado de esos elementos internos ingresa a un nuevo estado, sin importar cuales son el tipo de sus elementos internos.

7.6. Elementos

Para representar el motor de aspectos de *Caffeine* se decidió crear un nuevo domino que materialice los conceptos de sistemas basados en aspectos. Los elementos que lo componen este nuevo metamodelo son *Aspect*, *TransitionPoint*, *Advice*, *Instruction*.

7.6.0.1. Elemento *Aspect*

Como su nombre lo indica, éste elemento representa un aspecto, es decir, un conjunto de *advices*, que se ubican en cierto punto indicado por el *transition point*.

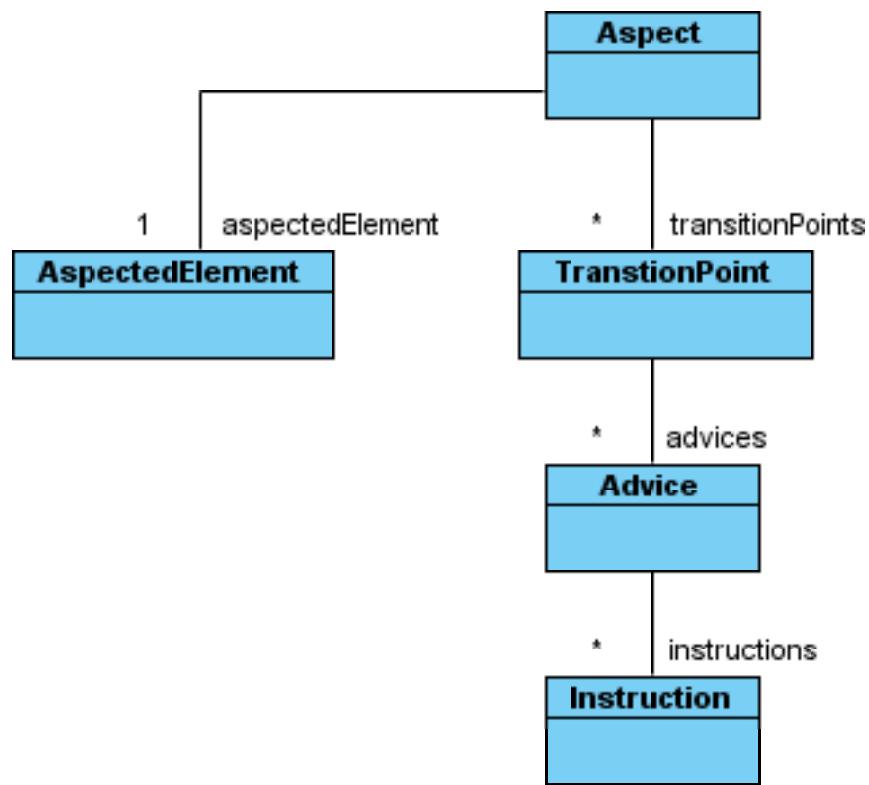
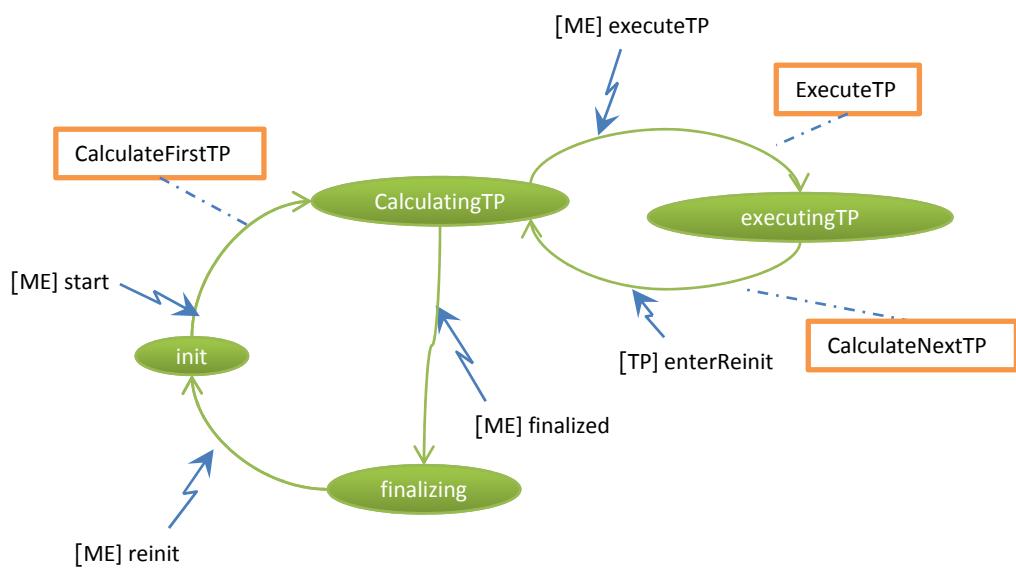
CUADRO 7.1: Estados del Elemento *Aspect*

Descripción de Estados

Init	Es el estado inicial del elemento.
CalculatingTP	Una vez el elemento es inicializado, pasa a calcular cual es el primer <i>transition point</i> donde estará ubicado el aspecto.
ExecutingTP	Indica que se ha encontrado el <i>Transition Point</i> y está ejecutando los <i>advices</i> designados para ese punto.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

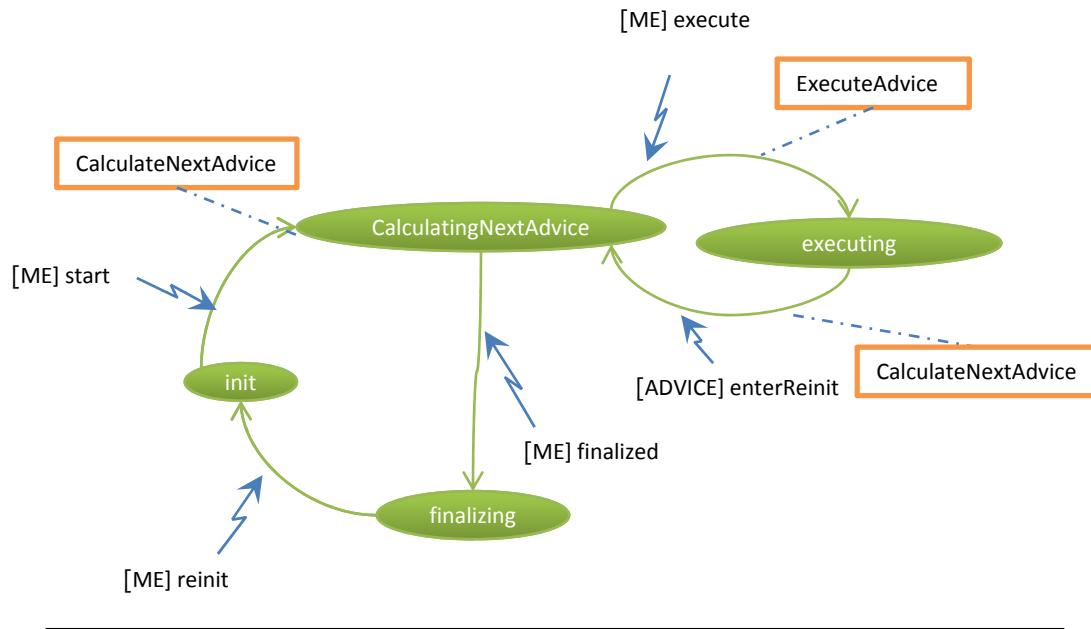
7.6.0.2. Elemento *Transition Point*

Éste elemento indica un lugar donde debe ejecutarse la lógica contenida en los *advices*

FIGURA 7.1: Elementos de *AspectCaffeine*.FIGURA 7.2: Máquina de estados del elemento *Aspect*.

CUADRO 7.2: Acciones del elemento *Aspect***Acciones**

CalculateFirstTP	Calcula cual es el primer <i>transition point</i> que se va a ejecutar.
ExecuteTP	Indica a los <i>Advices</i> que allí se encuentran que deben ejecutarse.
CalculateNextTP	Calcula el siguiente <i>transition point</i> .

FIGURA 7.3: Máquina de estados del elemento *Transition Point*.CUADRO 7.3: Estados del Elemento *TransitionPoint***Descripción de Estados**

Init	Es el estado inicial del elemento.
CalculatingNextAdvice	Una vez el elemento es inicializado, pasa a calcular cual es el primer <i>Advice</i> que debe ser ejecutado.
Executing	Indica que se ha encontrado el <i>Advice</i> y está ejecutando las instrucciones.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 7.4: Acciones del elemento *Transition Point***Acciones**

CalculateNextAdvice	Calcula cual es el siguiente <i>Advice</i> que se va a ejecutar.
ExecuteAdvice	Indica al <i>Advice</i> que allí se encuentra que se debe ejecutar.

7.6.0.3. Elemento Advice

Éste elemento contiene el conjunto de instrucciones que deben ser ejecutadas en ese punto.

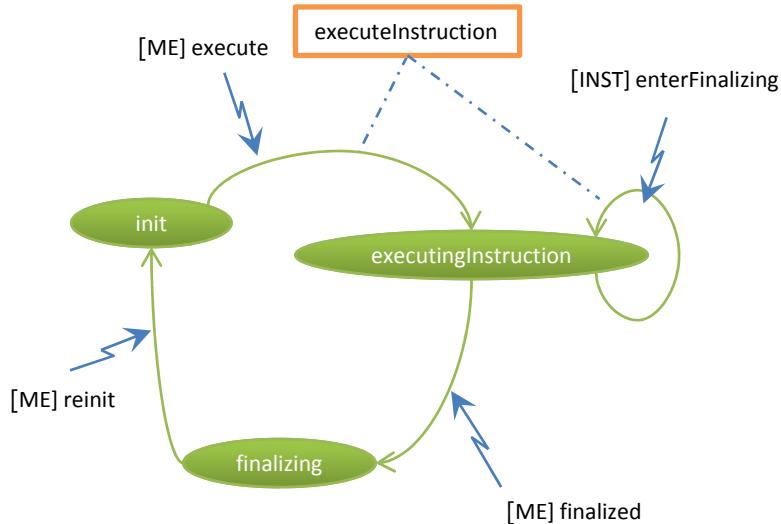


FIGURA 7.4: Máquina de estados del elemento *Advice*.

CUADRO 7.5: Estados del Elemento *Advice*

Descripción de Estados

Init	Es el estado inicial del elemento.
ExecutingInstruction	Indica que está ejecutando las instrucciones.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 7.6: Acciones del elemento *Advice*

Acciones

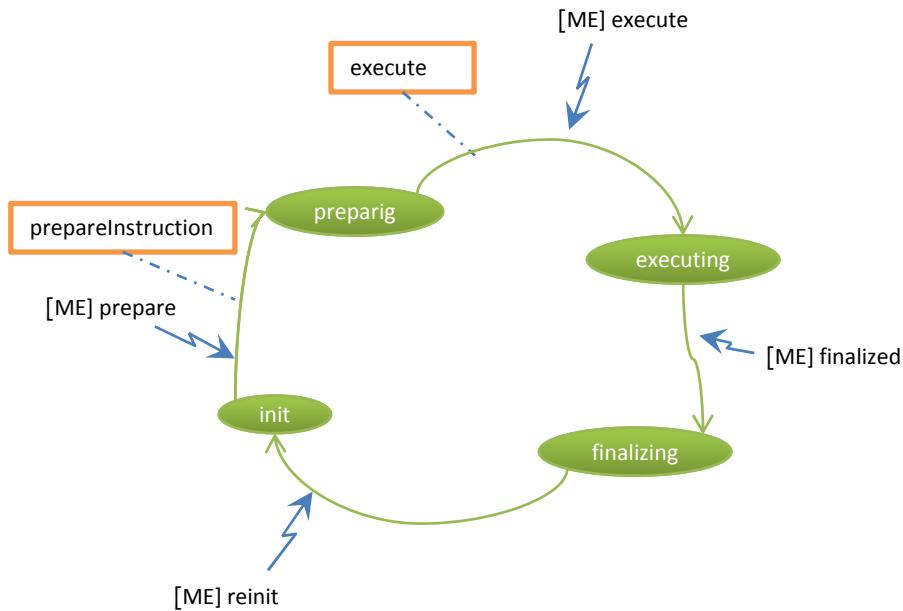
ExecuteInstruction	Ejecuta la siguiente instrucción en el orden que fue definida.
--------------------	--

7.6.0.4. Elemento *Instruction*

Este elemento representa una instrucción.

7.7. Manejo de Interferencias

Para realizar el manejo de interferencias en *AspectCaffeine* se hace uso de un archivo que describe cuales son las instrucciones o los *advice*s que potencialmente pueden entrar

FIGURA 7.5: Máquina de estados del elemento *Instruction*.CUADRO 7.7: Estados del Elemento *Instruction***Descripción de Estados**

Init	Es el estado inicial del elemento.
Preparing	Indica que la instrucción está siendo preparada para su ejecución.
Executing	Indica que está ejecutando la instrucción.
Finalizing	En éste estado se indica que la ejecución del elemento ya terminó.

CUADRO 7.8: Acciones del elemento *Instruction***Acciones**

PrepareInstruction	Prepara los posibles argumentos que pueda tener la instrucción.
ExecuteInstruction	Ejecuta la instrucción.

en conflicto con otros. A partir de éste archivo, junto con los *advices* que se encuentran en el motor, se arma un grafo dirigido de *advices* no conflictivos, donde los vértices representan los *advices* y los arcos entre los vértices representan cuales *advices* pueden ser ejecutados después de cada *advice*.

A continuación se muestra un ejemplo de un posible archivo de *advices* conflictivos, para un aspecto que en total tiene cinco *advices*, llamados *advice1*, *advice11*, *advice5*, *advice7* y *advice4*.

```

<?xml version="1.0" encoding="UTF-8"?>
2<conflicts>
    <advice name="advice1">
        <conflictsWith name="advice4"/>
4

```

```

5   </advice>
6   <advice name="advice4">
7     <conflictsWith name="advice7"/>
8     <conflictsWith name="advice11"/>
9   </advice>
10
11   <instruction name="inst1">
12     <conflictsWith name="inst8"/>
13   </instruction>
14   <instruction name="inst3">
15     <conflictsWith name="inst1"/>
16     <conflictsWith name="inst2"/>
17   </instruction>
18 </conflicts>

```

CÓDIGO 7.11: Ejemplo de archivo que define los conflictos entre las instrucciones o *advices*

A continuación se mostrará gráficamente como se armaría el grafo de *advices* no conflictivos para cada uno de los *advices*.

Advice1

De acuerdo al descriptor no es posible ejecutar el *advice4* después del *advice1*, así que no habrá un arco entre esos dos vértices. De igual manera, se definió que no se puede ejecutar la *inst8* después de ejecutar la *inst1*, así que tampoco existirá un arco entre el *advice1* y el *advice7*. Ver figura 7.6.

Advice11

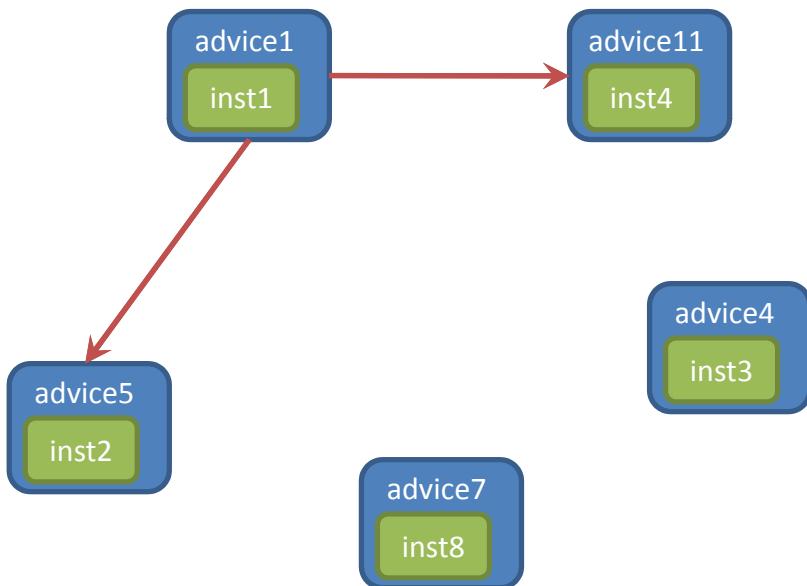
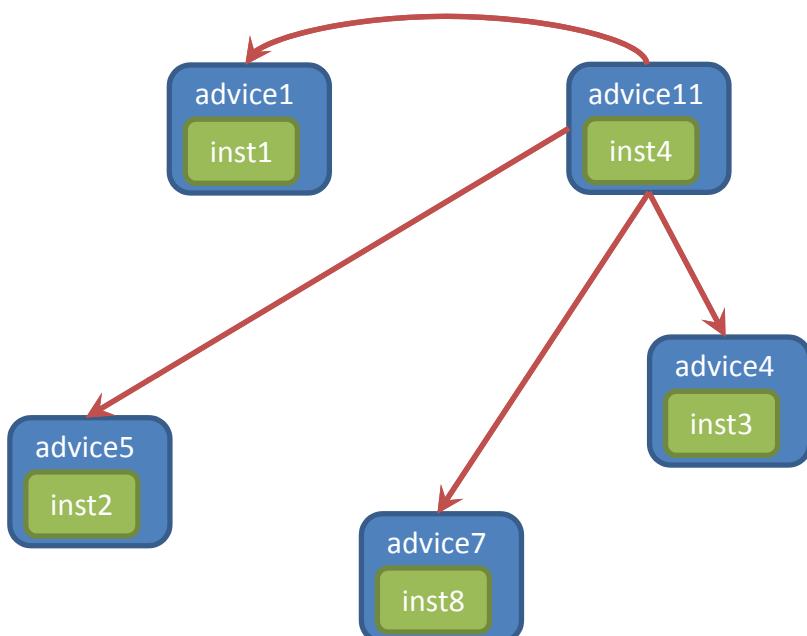
En el descriptor no existe ningún tipo de restricción para el *advice11*. Ver figura 7.7.

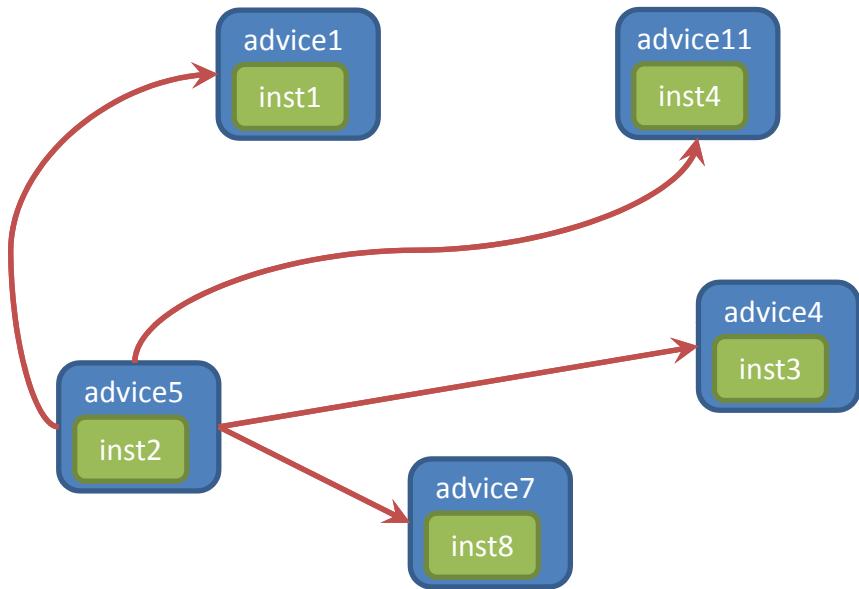
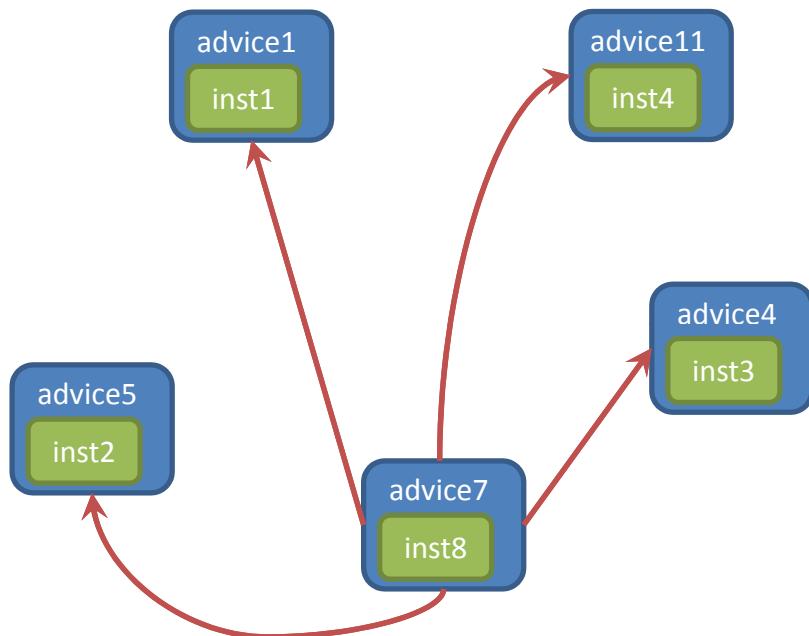
Advice5

En el descriptor no existe ningún tipo de restricción para el *advice5*. Ver figura 7.8.

Advice7

En el descriptor no existe ningún tipo de restricción para el *advice7*. Ver figura 7.9.

FIGURA 7.6: Grafo únicamente para el *advice1*.FIGURA 7.7: Grafo únicamente para el *advice11*.

FIGURA 7.8: Grafo únicamente para el *advice5*.FIGURA 7.9: Grafo únicamente para el *advice7*.

Advice4

De acuerdo al descriptor, ningún *advice* puede ser ejecutado después de ejecutar el *advice4*. Ver figura 7.10.

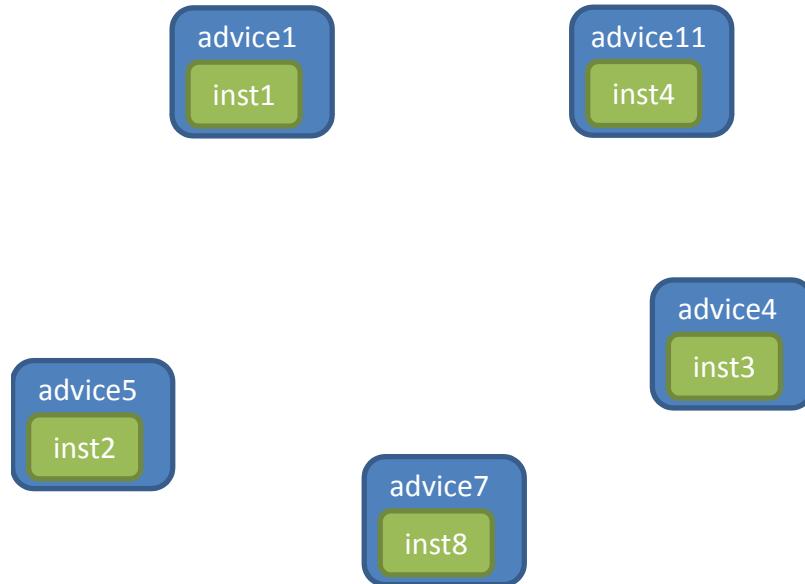


FIGURA 7.10: Grafo únicamente para el *advice4*.

Al finalizar el grafo de conflictos se vería como se muestra en la figura 7.11. Ver figura 7.11.

El siguiente paso después de obtener el grafo de *advices* no conflictivos, es encontrar una manera de poder ejecutarlos, tratando de que se ejecuten todos. Para esto, a partir del grafo, se obtiene el árbol de recubrimiento correspondiente.

7.7.1. Árbol de Recubrimiento

El árbol de recubrimiento se construye utilizando una variación del algoritmo de Wilson[44], con el cual es posible encontrar un árbol de recubrimiento con probabilidad uniforme. El algoritmo comienza seleccionando un vértice inicial aleatorio. Luego se deben agregar como hijos al árbol los vértices sucesores que no se encuentran en el camino hacia la raíz. El proceso debe continuar, por cada uno de los sucesores que tenga el vértice.

- 1. Escoger un Vértice Aleatorio** - De acuerdo al grafo resultante, mostrado en la figura 7.11, se va a escoger un vértice aleatorio. De escoger el vértice *advice4* al

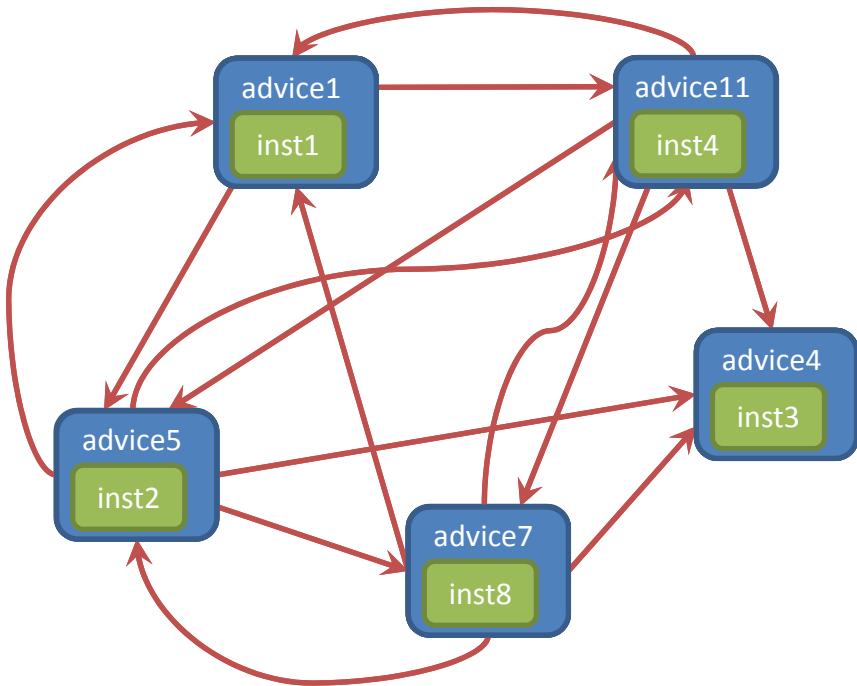


FIGURA 7.11: Grafo completo.

no tener sucesores, es descartado inmediatamente porque no es posible continuar armado el árbol.

Como raíz será escogido el *advice7*. Ver figura 7.12.



FIGURA 7.12: Primer paso del algoritmo para obtener el árbol de recubrimiento.

2. **Agregar los vértices sucesores como hijos** - El siguiente paso es agregar los vértices sucesores como hijos en el árbol, que no se encuentren en el camino hacia la raíz. Para éste caso, los vértices sucesores son *advice1*, *advice5*, *advice11* y *advice4*. Ver figura 7.13.
3. **Agregar los siguientes vértices sucesores como hijos** - El siguiente paso es agregar los vértices sucesores como hijos en el árbol, que no se encuentren en el camino hacia la raíz. Por ejemplo, el *advice7* es un sucesor del *advice5*, pero no se coloca nuevamente en el árbol porque ya se encuentra en el camino hacia la raíz. Se puede observar en la gráfica que en todas las ramas el *advice4* son hojas, porque no tiene sucesores. Ver figura 7.14.

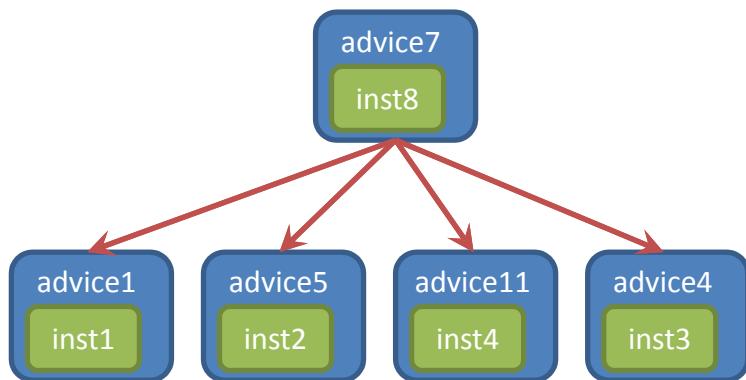


FIGURA 7.13: Segundo paso del algoritmo para obtener el árbol de recubrimiento.

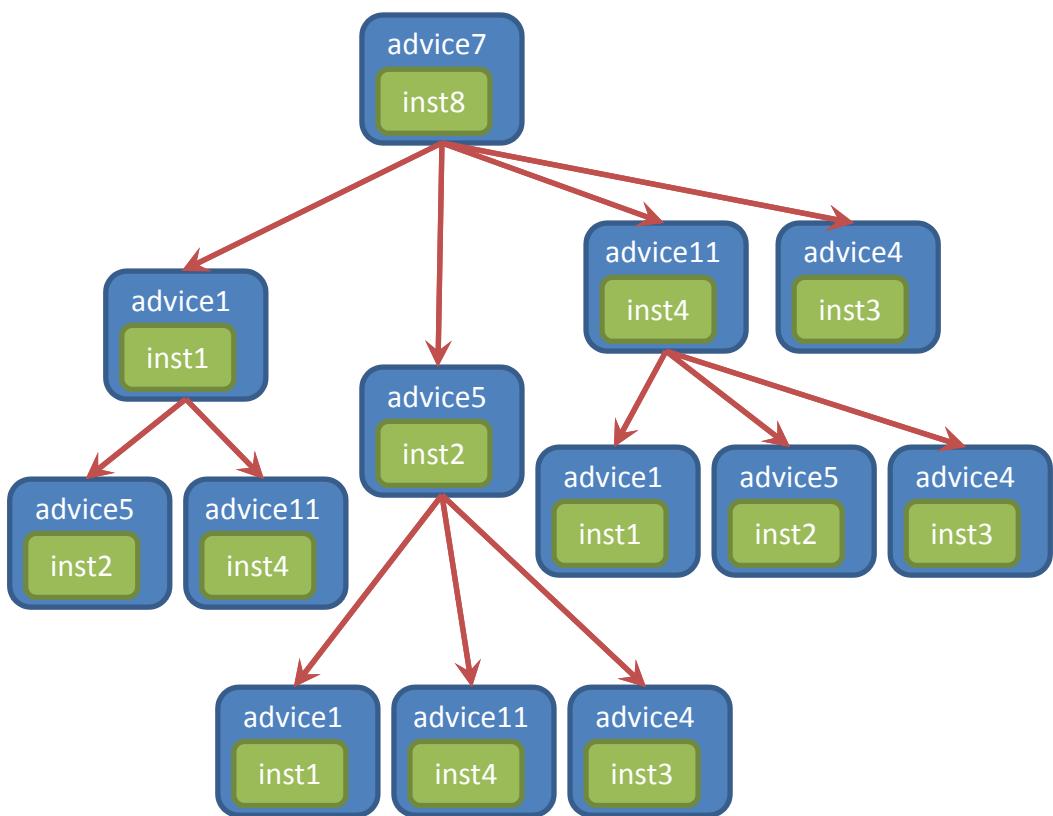


FIGURA 7.14: Continuar agregando los vértices sucesores como hijos.

4. **Árbol Terminado** - El algoritmo continúa hasta llegar a que los vértices no tienen más sucesores. En la figura 7.15 se puede ver el árbol terminado. El orden de ejecución escogido de los *advices* es el orden que proporciona la rama que contiene todos los *advices*, para este caso el orden de ejecución es el *advice7*, luego el *advice1*, luego el *advice5*, luego el *advice11* y finalizando con el *advice4*. En caso que el árbol no tenga ninguna rama con todos los *advices*, ese aspecto no puede ser ejecutado. Por simplicidad se muestra solo una porción del árbol resaltando una rama que tiene todos los *advices* definidos para el ejemplo. Ver figura 7.15.

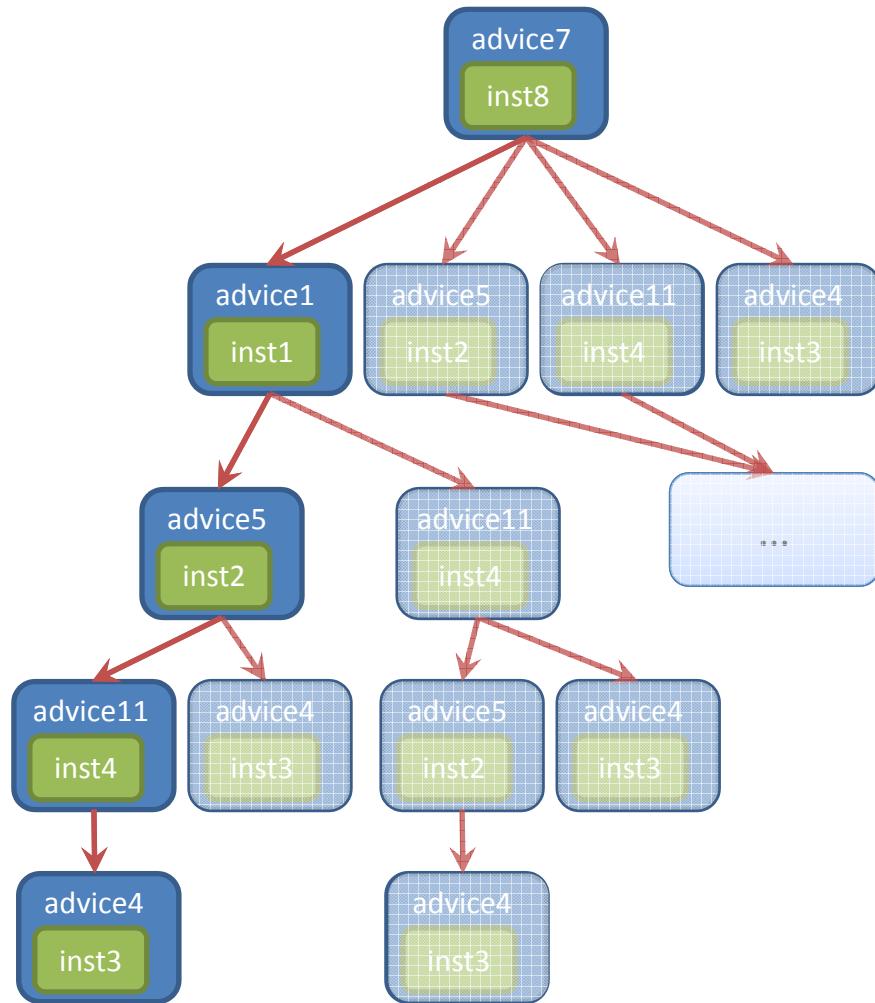


FIGURA 7.15: Árbol terminado resaltando una rama con todos los *advices*.

7.8. Pruebas

Para probar *AspectCaffeine* se utilizó el mismo *framework* de pruebas descrito en la sección 6.4.1.

Escenarios de Prueba

Para *AspectCaffeine* se desarrollaron 7 escenarios de prueba, donde cada uno de ellos prueba que la composición de los elementos sea correcta y que los *advices* que fueron definidos como conflictivos no se ejecuten secuencialmente.

Lenguaje de Animación

Para el lenguaje de animación de *AspectCaffeine* se extendió el lenguaje de animación definido para las pruebas de *Caffeine*.

Agregar un Aspecto - Ésta instrucción permite decirle a *AspectCaffeine* que agrege un nuevo aspecto, sin asociarlo a ninguna instancia específica de ningún proceso, de ésta manera es posible verificar que los puntos de corte sean interpretados correctamente. Por defecto todos los aspectos son habilitados para su ejecución.

```
<addAspect file="./data/aspects/models/aspects/aspect2.xml" enabled="true"/>
```

CÓDIGO 7.12: Instrucción para agregar un aspecto

Remover un Aspecto - Ésta instrucción permite decirle a *AspectCaffeine* que se quiere quitar la definición de un aspecto para que no siga siendo ejecutado.

```
1 <removeAspect name="aspect1"/>
```

CÓDIGO 7.13: Instrucción para quitar un aspecto

Deshabilitar un Aspecto - Ésta instrucción permite decirle a *AspectCaffeine* que se quiere deshabilitar la ejecución de un aspecto para una instancia específica de un proceso específico. Existe una variación de ésta que deshabilita la ejecución del aspecto para todas las instancias de todos los posibles procesos.

```
1 <disableAspect name="aspect1" processName="HelloWorld" processID="0" instanceID="0"/>
```

CÓDIGO 7.14: Instrucción para deshabilitar un aspecto

Sensores

Los sensores utilizados en *AspectCaffeine* también extienden los sensores definidos en la sección 6.4.1, debido a que se necesitan sensores que actúen sobre los elementos BPEL definidos en los *advices* y nuevos sensores que se colocan sobre los elementos de

AspectCaffeine para monitorear las máquinas de estado de los elementos, de ésta manera es posible conocer cuáles fueron los elementos que se activaron y en qué orden, así es posible realizar aserciones acerca del orden de ejecución de esos elementos.

Capítulo 8

Conclusiones y Trabajo Futuro

8.1. Conclusiones

A lo largo de este trabajo se presentó uno de los problemas, que teniendo en cuenta la cantidad de herramientas que lo soportan, ha sido muy poco explorado. Las preocupaciones transversales y la modularización de ellas son una cara de las aplicaciones orientadas a *workflow* que presenta una gran importancia debido a las ventajas que representa contar con mecanismos que permitan tanto definir nuevas funcionalidades sobre procesos como encapsularlas.

El enfoque de este trabajo de tesis fue la construcción de un motor de BPEL que no solo permitiera la definición de nuevos comportamientos encapsulados, sino también una aproximación para resolver el problema de interferencia entre aspectos, el cual afecta a los lenguajes orientados por aspectos.

Gracias a que se hizo uso de las propuestas realizadas por el proyecto Cumbia, es posible tener un motor de BPEL funcional y a su vez realizar la fácil integración con otro modelo que representa los comportamientos transversales, sin perder la clara diferenciación entre los elementos que hacen parte de las preocupaciones transversales y los elementos del proceso, otorgándole a los comportamientos transversales identidades de primer orden.

Debido a que se utilizaron objetos abiertos, es posible hacer un fácil monitoreo de los aspectos que están siendo ejecutados en cierto momento en el tiempo o conocer fácilmente cuales son los aspectos que afectan directamente instancias de proceso específicas. Además, gracias a la flexibilidad intrínseca de utilizar modelos ejecutables extensibles, es posible cambiar fácilmente las implementaciones aquí propuestas, de tal manera que se acomoden a las necesidades específicas de diferentes contextos a muy bajo costo. Un

ejemplo claro de esto es la posibilidad de definir nuevos *transition points*, donde la manera de ordenar los *advices* sea una estructura que se acomode mejor a las necesidades del negocio, en vez de un grafo dirigido.

Es posible diseñar componentes extra que permitan enriquecer al motor de BPEL que ayuden a manejar requerimientos no funcionales, como la transaccionalidad de los procesos o la seguridad en el intercambio de mensajes. A su vez, la fácil extensión de los motores, permite desarrollar componentes para contextos donde la lógica del negocio está definida usando reglas de negocio.

8.2. Trabajo Futuro

Como parte del trabajo futuro se propone realizar la misma experimentación sobre otros de los activos existentes del proyecto, como por ejemplo el motor de BPMN.

También se propone hacer una extensión sobre el lenguaje de puntos de corte para tener en cuenta puntos que representan, por ejemplo, cuando una actividad x es ejecutada después de una actividad z o involucrar otros dominios, como el de recursos, para tener expresiones que puedan definir puntos de corte donde cierto participante ejecute cierta actividad. También puntos de corte donde se monitoree cuando una variable es modificada o leída.

Otra propuesta es hacer algo similar a lo que hace *Padus*, que permite definir *advices* de tipo *in*, los cuales son colocados dentro de una actividad específica, por ejemplo dentro de un *flow*. También es posible tener en cuenta otros tipos de *advices* que son utilizados en *AO4BPEL* que pueden ser ejecutados en paralelo al elemento sobre los cuales están definidos.

Bibliografía

- [1] OASIS. History of bpel. Publicación Web, Octubre 2008. URL <http://bpel.xml.org/history>.
- [2] OASIS Web Services Business Process Execution Language (WSBPEL) TC. Web services business process execution language version 2.0. Publicación Web, Abril 2007. URL <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>.
- [3] Weerawarana S., Curbera F., Leymann F., Storey T., and Ferguson D. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More.* Prentice Hall PTR, March 2005. ISBN 0131488740. URL <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0131488740>.
- [4] Active Endpoints. Bpel open source. Publicación Web, Octubre 2008. URL <http://www.activevos.com/community-open-source.php>.
- [5] Active Endpoints. Publicación Web, Octubre 2008. URL http://www.activevos.com/indepth/b_capabilities/d_DeployingVignette/Deploying.html.
- [6] Active Endpoints. Publicación Web, Octubre 2008. URL http://www.activevos.com/indepth/b_capabilities/e_ReportngVignette/Reporting.html.
- [7] Active Endpoints. Publicación Web, Octubre 2008. URL http://infocenter.activevos.com/infocenter/ActiveVOS/v60/topic/com.activee.bpel.doc/aeDataSource_UserReports.pdf.
- [8] Active Endpoints. Publicación Web, Octubre 2008. URL http://www.activevos.com/indepth/b_capabilities/j_activeVOS5Designer/ActiveVOSDemonstration.html.
- [9] Active Endpoints. Active Endpoints, Octubre 2008. URL <http://www.activebpel.org/infocenter/ActiveVOS/v50/index.jsp?topic=/com.activee.bpel.doc/html/UG20-5.html>.

- [10] Oracle. Oracle bpel process manager 10.1.2.0.x quick start tutorial. Publicación Web, Octubre 2008. URL <http://download.oracle.com/otndocs/products/bpel/quickstart.pdf>.
- [11] Oracle. Oracle bpel process manager. Publicación Web, Octubre 2008. URL www.oracle.com/technology/products/ias/bpel/pdf/oracle_bpel_process_manager_datasheet.pdf.
- [12] Oracle. Bpel console reports. Publicación Web, Octubre 2008. URL <http://www.oracle.com/technology/products/ias/bpel/pdf/bpelreportsfeaturepreview.pdf>.
- [13] Oracle. Bpel console reports. Publicación Web, Octubre 2008. URL <http://www.oracle.com/technology/products/ias/bpel/pdf/bpelunitestwebconference.pdf>.
- [14] Matjaz J., Benny M., and Poornachandra S. *Business process execution language for web services : an architect and developer's guide to orchestrating web services using BPEL4WS*. Packt Publ., 2nd ed. edition, 2006. ISBN 1-904811-81-7=978-1-904811-81-7. URL http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+509825540&sourceid=fbw_bibonomy.
- [15] Apache ODE. Developing, deploying and running a hello world bpel process with the eclipse bpel designer and apache ode. Publicación Web, Octubre 2008. URL <http://people.apache.org/~vanto/HelloWorld-BPELDesignerAndODE.pdf>.
- [16] Web services human task (ws-humanTask). Publicación Web, Octubre 2008. URL http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf.
- [17] Ws-bpel extension for people (bpel4people). Publicación Web, Octubre 2008. URL http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/BPEL4People_v1.pdf.
- [18] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., and Irwin J. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997. URL <http://citeseervx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.6608>.
- [19] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Griswold W. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4.

- [20] Karl J. Lieberherr, Ian M. Holland, and Arthur J. Riel. Object-oriented programming: An objective sense of style. In *OOPSLA*, pages 323–334, 1988.
- [21] Harrison W. and Ossher H. Subject-oriented programming: a critique of pure objects. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, volume 28, pages 411–428. ACM Press, October 1993. doi: <http://dx.doi.org/10.1145/165854.165932>. URL <http://dx.doi.org/10.1145/165854.165932>.
- [22] Masuhara H. and Kiczales G. A modeling framework for aspect-oriented mechanisms. In *In Proc. of the 17th European Conference on Object-Oriented Programming (ECOOP)*, volume 2734 of LNCS, pages 2–28. Springer, 2003.
- [23] Charfi A. *Aspect-Oriented Workflow Languages: AO4BPEL and Applications*. PhD thesis, TU Darmstadt, Fachbereich Informatik, 2007.
- [24] Hilsdale E. and Hugunin J. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM. ISBN 1-58113-842-3. doi: <http://doi.acm.org/10.1145/976270.976276>.
- [25] Nagy I., Bergmans L., and Aksit M. Composing aspects at shared join points. In *NODE 2005, GSEM 2005, Erfurt, Germany, September 20-22, 2005*, volume p-69 of *Lecture Notes in Informatics 69*, pages 19–38, 2005. URL <http://doc.utwente.nl/54193/>.
- [26] Durr P.E.A., Staijen T., Bergmans L.M.J., and Aksit M. Reasoning about semantic conflicts between aspects. In *EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software*, 2005. URL <http://doc.utwente.nl/54430/>.
- [27] Miles R. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004. ISBN 0596006543.
- [28] Reddy Y. R., Ghosh S., France R., Straw G., Bieman J., McEachen N., Song E., and Georg G. Directives for composing aspect-oriented design class models. *Trans. Aspect-Oriented Software Development*, pages 75–105, 2006.
- [29] Sihman M. and Katz S. *Superimpositions and aspect-oriented programming*, volume 46. 2003.
- [30] Zhang J. Aspect interference and composition in the motorola aspect-oriented modeling weaver.
- [31] Durr P.E.A. *Resource-based Verification for Robust Composition of Aspects*. PhD thesis, Univeristy of Twente, Enschede, June 2008.

- [32] Shazam Entertainment Ltd. Shazam, Julio 2008. URL <http://phobos.apple.com/WebObjects/MZStore.woa/wa/viewSoftware?id=284993459>.
- [33] Mellodis. Midomi, Julio 2008. URL <http://phobos.apple.com/WebObjects/MZStore.woa/wa/viewSoftware?id=284972998&mt=8>.
- [34] Braem M., Verlaenen K., Joncheere N., Vanderperren W., Straeten V., Truyen E., and Joosen W. Isolating process-level concerns using padus. In *In Proceedings of the 4th International Conference on Business Process Management (BPM 2006)*. Springer-Verlag, 2006.
- [35] United States Copyright Office. The digital millennium copyright act of 1998. URL <http://www.copyright.gov/legislation/dmca.pdf>.
- [36] Villalobos J., Sánchez M., and Romero D. Executable models as composition elements in the construction of families of applications. In *6th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA)*, Lisboa, Portugal, 2007.
- [37] Calderón N. and Vega C. Composición y adaptación de modelos ejecutables extensibles para aplicaciones elearning. caso ims-ld. Master's thesis, Universidad de los Andes, 2008.
- [38] Jiménez C. Composición de modelos ejecutables extensibles en una fábrica de aplicaciones basadas en workflows. Master's thesis, Universidad de los Andes, 2007.
- [39] Bezivin et al. Towards a precise definition of the omg/mda framework. automated software engineering. In *Proceedings. 16th Annual International Conference*, pages 273 – 280, 2001.
- [40] Villalobos J., Sánchez M., and Barrero I. Enriching a process engine with flexible time management through model weaving. 2007.
- [41] Moreno S. A testing framework for dynamic composable executable models. Master's thesis, Universidad de los Andes, 2007.
- [42] Romero D. Modelos ejecutables extensibles como activos en una fábrica de motores de workflow: Caso bpel. Master's thesis, Universidad de los Andes, 2007.
- [43] Barvo P. and Sánchez M. Construcción de una línea de producción de motores de workflow basada en modelos ejecutables. Master's thesis, Universidad de los Andes, 2006.
- [44] Wilson D.B. Generating random spanning trees more quickly than the cover time. In *Proceedings of the Twenty-eighth Annual ACM Symposium on the Theory of Computing*, pages 296–303. ACM, 1996.