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

Manuel Muñoz Lara
Universidad de los Andes
Departamento de Ingeniería de Sistemas y Computación
man-muno@uniandes.edu.co

Resumen

Este trabajo propone una manera de solucionar problemas de preocupaciones transversales en contextos de *workflows*, usando conceptos que se encuentran en sistemas de orientados por aspectos. A su vez, se propone una solución al problema de interferencias entre aspectos. Se propone el uso de metamodelos ejecutables, utilizando el modelo de objetos abiertos. Éste trabajo se encuentra enmarcado dentro del proyecto Cumbia. Un proyecto que hace parte del grupo de construcción de software de la Universidad de los Andes.

Palabras Clave: Cumbia, Objectos Abiertos, Metamodelos Ejecutables Extensibles, BPEL, Aspectos, Interferencia entre Aspectos.

1. Introducción

Los inicios de los sistemas de *workflow* se remontan a los años setenta, cuando nacio la necesidad de automatizar los procesos empresariales. Con el transcurrir del tiempo, se han identificado otros contextos donde la necesidad de automatizar procesos se ha hecho evidente.

Un factor que ha sido determinante para la evolución de los sistemas de *workflow* ha sido la introducción de Internet. Al comienzo, fue utilizado como un ambiente donde publicar datos. Actualmente, 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. 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*.

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*). 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 había sido identificado en los lenguajes de programación. Como solución a esto, se planteó el uso de AOP (*Aspect-Oriented Programing*). 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 as-

pectos, utilizando a BPEL como lenguaje de *workflow* enmarcado en el proyecto Cumbia.

2. BPEL

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[20].

Un proceso de *workflow* BPEL consiste de actividades que interactúan con los Web services que participan en la composición, junto con actividades donde se específica 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[28].

2.1. Conceptos Básicos

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

2.1.1. Elementos Básicos

Los elementos básicos se dividen en variables y colaboradores.

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[28].

Colaboradores

Representan las partes con quien el proceso BPEL interactúa, cómo 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[27].

2.1.2. Actividades

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

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*.

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.

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*.

2.2. 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.

2.2.1. 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*.

2.2.2. 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 específica 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.

2.2.3. Compsensation 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.

2.2.4. Correlation Sets

Los *correlation sets* son un mecanismo que permite identificar a que 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.

2.2.5. Event Handler

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

3. Aspect-Oriented Programming

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 desarrollo AOP o *Aspect-Oriented Programming*[8]. 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[7].

Existen varias aproximaciones de cómo se deben modelar los sistemas de programación orientada por aspectos, cinco[1] de ellos fueron descritos en [9] basándose en herramientas existentes, el más utilizado de ellos es el modelo de AspectJ.

3.1. AspectJ

El modelo de AspectJ define cuatro conceptos básicos que son necesarios para definir un aspecto[7].

3.2. 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

[1] Modelo de Pointcuts y Advices basado en AspectJ. Modelo de Recorridos basado en la ley de Demeter o el principio del menor conocimiento[14]. Modelo de Composición de Clases basado en la programación orientada por temas (*Subject-Oriented Programming*[29]). Modelo de Clases Abiertas. Modelo de Navegador Basado en Consultas basado en QJBrowser.

programa, como llamados a métodos, llamados a constructores, escritura y lectura de atributos, etc.

3.3. 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 utlizando 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 caracteristicas. 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 [1], 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.

3.4. 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 ése llamado, etc.

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

Código 1: Ejemplo de AspectJ.

En el código mostrado 1 se ejemplifica un aspecto de monitoreo usando AspectJ. Este aspecto define un punto de corte llamado *loggableMethods*, el cuál específica 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*.

3.5. 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[5] 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*.

3.6. 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 [10]. 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.

3.7. Interferencia Semántica Entre Aspectos

Para ilustrar los conflictos de interferencia semántica, se presenta un ejemplo en AspectJ de dos preocupaciones transversales, tomado de [23].

3.7.1. Ejemplo

La figura 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.

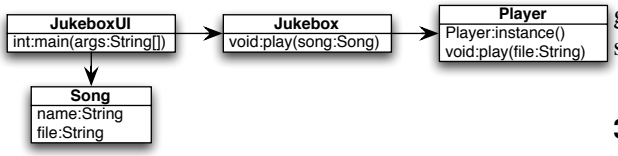

Figura 1: Sistema de reproducción de música.

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 ();
         proceed ();
9      }
       else
11     {
         throw new NotEnoughCreditsException ();
13     }
     }
15 }
```

Código 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      Playlist . instance (). enqueue ( song );
       return ;
7    }
   }
```

Código 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.

3.7.2. Clasificación de Interferencias

De acuerdo a [10], 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.

3.8. 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[24], o aproximaciones desde la etapa de modelamiento[25]. Otras aproximaciones van un poco más allá e introducen dependencias más complejas y relaciones de orden entre aspectos[10]. Otras aproximaciones detectan los conflictos entre los aspectos gracias a una especificación más completa del comportamiento de los aspectos, realizada por quien los programa[17, 23].

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

3.9. 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*[24]. 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 4.

```
aspect CreditsAspect
{
  declare precedence : CreditsAspect,
      PlaylistAspect;

  void around () : call ( public void Jukebox
      . play ( Song ))
  {
    if( Credits . instance (). enoughCredits
        ())
    {
      Credits . instance (). withdraw ();
      proceed ();
    }
    else
    {
      throw new NotEnoughCreditsException ();
    }
  }
}
```

Código 4: Declaración de precedencia entre aspectos en AspectJ.

Otra propuesta para realizar la definición de precedencia de aspectos, es la implementada para Motorola WEAVR, un plugin para manipular modelos ejecutables UML en Telelogic TAU G2[13]. Motorola WEAVR 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 2, tomada de [13]. En éste diagrama, se definen relaciones estereotipadas entre aspectos, que dictan la politica de cómo van a ser tejidos al modelo base. Los estereotipos definidos son *«follows»*, *«hidden_by»* y *«dependent_on»*, los cuales corresponden a tres maneras diferentes de manejar la interferencia.

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 despues de las instancias de *Aspect1*.

Hidden By - Cuando en 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*.

3.10. Detección de Conflictos

La idea principal de la propuesta[22] 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

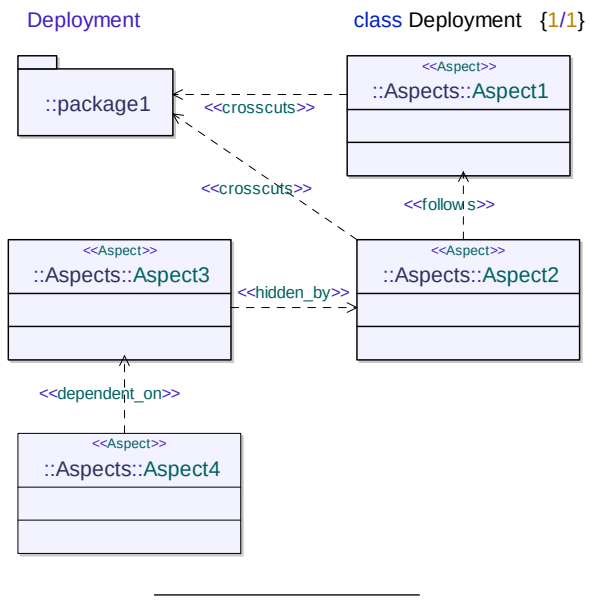

Figura 2: Diagrama de *deployment* mostrando los estereotipos para aspectos.

(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 especifico 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. Cómo 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 la especificación del comportamiento es:

```
JukeBox: enqueue;end
```

Código 5: Especificación del comportamiento del aspecto PlaylistAspect

Para el aspecto de cobra por escuchar una canción:

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

Código 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 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.

3.10.1. 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.

3.10.2. 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 8: Secuencia de operaciones para el recurso *JukeBox*

3.10.3. 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.

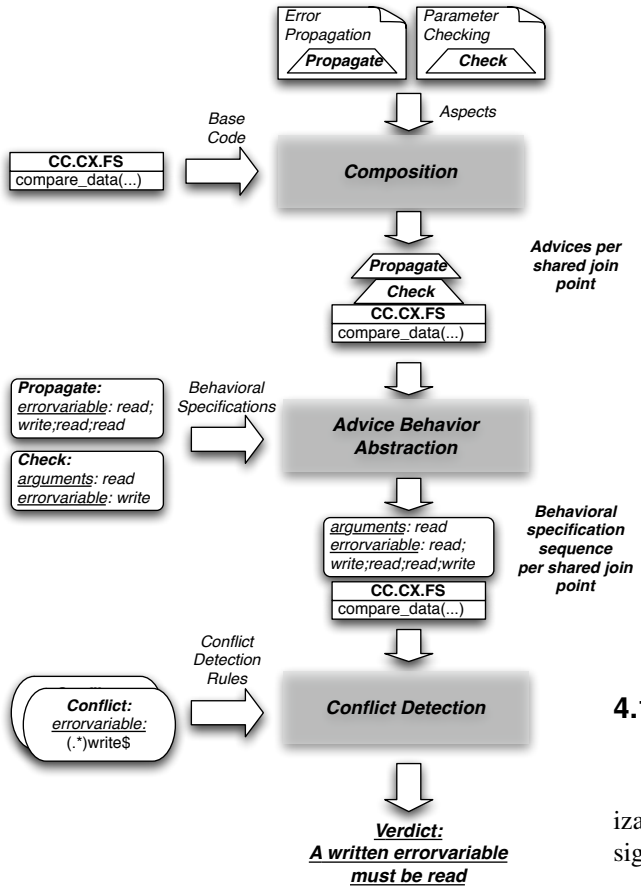

Figura 3: Proceso de detección de conflictos.

4. Aspect-Oriented Workflow Languages

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 inyectar 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.

4.1. Problemas de Modularización de Lenguajes de Workflow

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

4.1.1. 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[15] y Midomi[18].

En la figura 4 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.

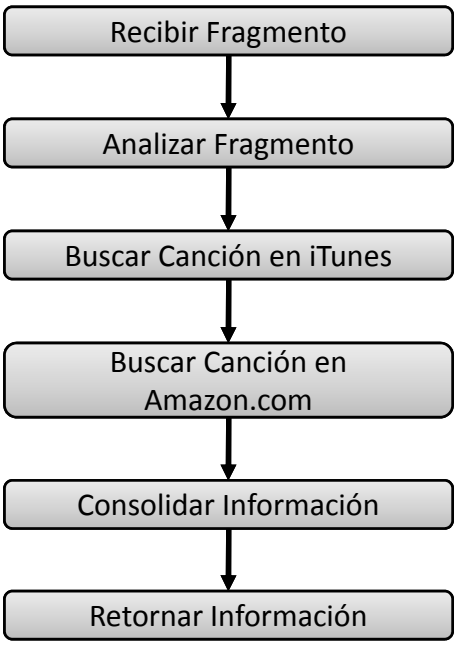

Figura 4: Ejemplo de un proceso workflow.

4.1.2. 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 [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 5. Para poder implementar éste cambio en BPEL, es

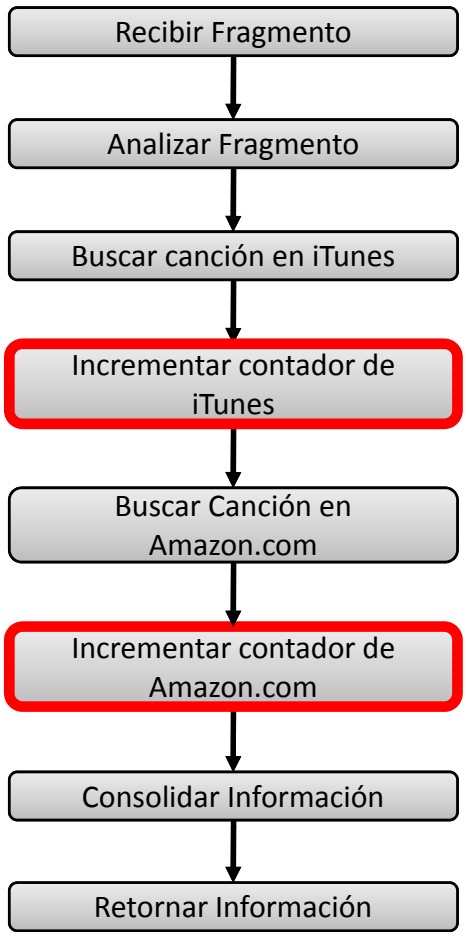

Figura 5: Ejemplo de recolección de información.

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 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 qué 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 cuáles son los elementos usados para satisfacer las preocupaciones transversales.

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[1]. 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, 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 6. Una posible implementación en 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.

4.1.3. 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 [1].

De acuerdo a [1] 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.

Figura 6: Ejemplo monitoreo de tiempo de ejecución.

- **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. Estos ejemplos son basados en [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 7.

Figura 7: 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.

4.2. 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[1]. 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[7], se ha propuesto aplicar está 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:

4.2.1. 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.

De acuerdo a [1], 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 *joint 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.

4.2.2. 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.

4.2.3. 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 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*[16]. De acuerdo al modelo discutido en 3.1 los *advices* pueden ser ejecutados antes, después, o en vez de los *join points* que son seleccionados por el punto de corte.

4.2.4. Tejido de Aspectos

El tejido de aspectos es la parte de la implementación que debe asegurar que el código de los *advices* y 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[1]. 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[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 3.1).

Una de las ventajas que tiene está 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.

Modificación del Motor para Verificación de Aspectos

En ésta aproximación, el motor tiene que ser modificado para verificar sí 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.

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 especifico. Éstos se conocen como aplicaciones basadas en control.

Múltiples factores influencian tanto los contextos como las aplicaciones que las soportan y no es común que las arquitecturas de dichas aplicaciones no son lo suficientemente flexibles para poder adaptarse a estos cambios[11].

El proyecto Cumbia 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 comu-

nican entre sí y uno de los componentes tiene como responsabilidad manejar el control de la aplicación[19].

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[2].

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[2].

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 éstar 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 [6] 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[12].

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 8) está compuesto por un objeto tradicional llamado entidad, una máquina de estados y un conjunto de acciones asociadas a las transiciones de los estados.

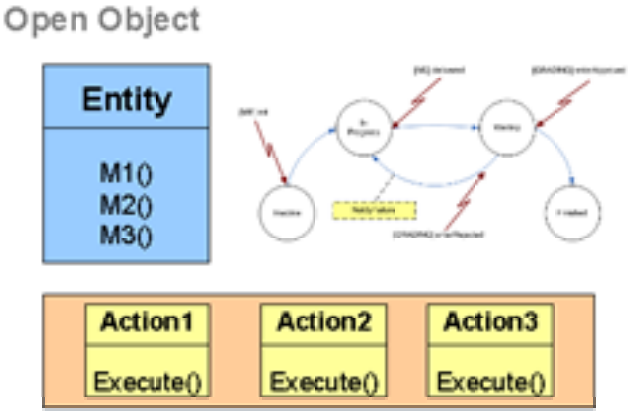

Figura 8: Estructura de un objeto abierto.

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[26].

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 entretejerse 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 entretejido 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[12].

Las ventajas de este tipo de construcciones son enumeradas en [12] 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 entretejer preocupaciones sobre aplicaciones que ya contienen otros modelos entretejidos.
- 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 9 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 especifíca 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.

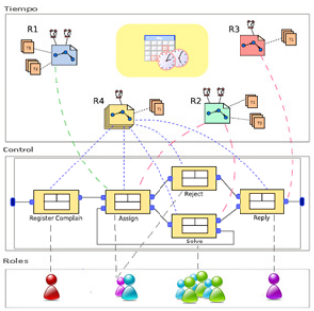

Figura 9: Composición de Modelos de Diferentes Dominios.

6. Caffeine v2.0: Motor BPEL sobre Cumbia

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[3]. 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.

6.1. Elementos Presentes

En el trabajo desarrollado en [3], 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[3]. 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.

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

6.1.1. Elementos de Inicio

Para poder iniciar un proceso BPEL se deben definir elementos para que al recibir un mensaje se instancie el proceso[28], esos elementos se encuentran agrupados en ésta categoría.

Elemento *Process*

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

Elemento *StartingPick*

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

Elemento *StartingReceive*

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

6.1.2. Elementos de Interacción

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

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[28].

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.

Elemento *OnMessage*

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

Elemento *Receive*

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

Elemento *Reply*

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

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.

6.1.3. Elementos Estructuradores

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

Elemento *Sequence*

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

Elemento *Flow*

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

Elemento *While*

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

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.

6.1.4. Instrucciones

Elemento *Exit*

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

Elemento *Empty*

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

Elemento *Wait*

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

Elemento *Assign*

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

6.2. Arquitectura

En ésta sección se hablará de la arquitectura de Caffeine.

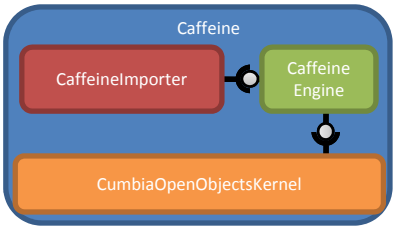

Figura 10: Arquitectura de Caffeine.

6.2.1. CumbiaOpenObjectsKernel

El trabajo con metamodelos ejecutables extensibles ha sido un trabajo constante en el proyecto Cumbia. A partir de la tesis de maestria de Pablo Barvo y Mario Sánchez[21] 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 9) donde se definen cuales son los elementos que componen el metamodelo. De igual manera en éste archivo se indica cuál es el descricptor que define la máquina de estados para cada elemento.

```
<metamodel name="BPEL" version="0.1">

  <!-- State machines used by the elments of
       the metamodel -->
  <state-machine-reference name="assign" file
       ="assign.xml" />
  ...
  <state-machine-reference name="while" file=
       "while.xml" />

  <!-- Elements of the metamodel -->
  <type name="Assign" class="uniandes.cumbia.
       bpel.elements.assign.Assign"
       statemachine="assign"/>
  ...
  <type name="Copy" class="uniandes.cumbia.
       bpel.elements.assign.copy.Copy"/>
  ...
  <type name="From" class="uniandes.cumbia.
       bpel.elements.assign.from.From"/>

  ...
</metamodel>
```

Código 9: Archivo de declaración de los elementos del metamodelo.

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 especifica, 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 10). Por último, se debe crear la implementación de todos los elementos que hacen parte del modelo y asociarlos al *Kernel*.

```
<?xml version="1.0" encoding="UTF-8"?>
<definition metamodel="BPEL" modelName="
     HelloWorld" version="0.1">
    <metamodel-extensions/>
    <elements>
      <open-object name="HelloWorld" typeName
           ="Process"/>
      ...
      <open-object name="replyOutput"
           typeName="Reply"/>
    </elements>
    <runtime/>
    <model>
      <process ...>

          <!-- List of services participating
               in this BPEL process -->
          <partnerLinks>
          ...
          </partnerLinks>

          <!-- List of messages and XML
               documents used as part of this
               BPEL process -->
          <variables>
          ...
          </variables>
```

```
            <!-- Orchestration Logic -->
24          <sequence name="sequence">
            ...
26          </sequence>
        </process>
28    </model>
</definition>
```

Código 10: Archivo de definición de los elementos del metamodelo.

6.2.2. 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 11.

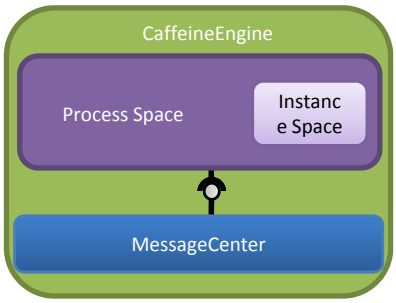

Figura 11: Arquitectura de CaffeineEngine.

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*.

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.

6.2.3. 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.

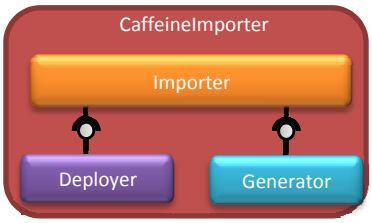

Figura 12: Arquitectura de CaffeineDeployer.

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 6.2.1. 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.

6.3. 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[26] por Sergio Moreno.

6.3.1. 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 Sincrónica - Ésta instrucción permite decirle al *MessageCenter* que envié 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 11: Instrucción sincrónica de envió 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.

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

Código 12: Instrucción sincrónica de envió 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.

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

Código 13: 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.

7. AspectCaffine: Extensión de Aspectos a Caffeine

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 4.2 para exponer como está construido *AspectCaffeine*.

7.1. 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.

7.2. Lenguaje de Puntos de Corte

De acuerdo con las limitaciones del lenguaje de puntos de corte de *AspectJ* descritas en la sección 3.3, 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.

```
*Invoke
```

Código 14: 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.

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

Código 15: 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.

```
*Invoke->ToCalculatingNextAdvice
```

Código 16: 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.

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

Código 17: 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.

```
*Invoke|Shazam
```

Código 18: 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.

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

Código 19: 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.

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

Código 20: 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 especifico 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.

```
*Invoke|
    Shazam:secuencia:InvocarServicioFacturac
```

Código 21: Seleccionar para un proceso dado su nombre un elemento dando su localización y su nombre.

- Seleccionar una transición de un elemento especifico 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.

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

Código 22: 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.

7.3. Lenguaje de *Advices*

Como se discutió en la sección 4.2.3, 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 específica sí el *advice* va a ser ejecutado antes, después o en vez del elemento que identifica el punto de corte.

7.3.1. 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 23.

```
<?xml version="1.0" encoding="UTF-8"?>
<aspect name="facturacion">
<transitionPoint name="TP1" pointcut="*Invoke
    |Shazam">
<advice name="invokeFacturacion" type="before
    ">
<partnerLinks>
<partnerLink name="recaudo"
partnerLinkType="tns:FacturacionShazam"
myRole="RecaudoRequester"
partnerRole="RecaudoProvider"
/>
</partnerLinks>
<variables>
<variable name="facturacionInfo" messageType=
    "tns:FacturacionMessage"/>
</variables>
<assign>
<copy name="copy">
<from name="from">bpel:getVariableData('input
    ', 'payload','/tns:userName')</from>
<to name="to" variable="facturacionInfo" part
    ="payload" query="/tns:result"/>
</copy>
</assign>
<invoke name="invoke" operation="initiate"
    partnerLink="recaudo" portType="
    tns:FacturacionShazam" inputVariable="
    facturacionInfo"/>
</advice>
</transitionPoint>
</aspect>
```

Código 23: 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.

7.4. 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 sé 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 ésta 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.

7.5. 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*, *TransitonPoint*, *Advice*, *Instruction*.

Elemento *Aspect*

Cómo 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*.

Elemento *Transition Point*

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

Elemento *Advice*

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

Elemento *Instruction*

Éste elemento representa una instrucción.

7.6. 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 *advices* que potencialmente pueden entrar 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"?>
<conflicts>
  <advice name="advice1">
    <conflictsWith name="advice4"/>
  </advice>
  <advice name="advice4">
    <conflictsWith name="advice7"/>
    <conflictsWith name="advice11"/>
  </advice>

  <instruction name="inst1">
    <conflictsWith name="inst8"/>
```

```
    </instruction>
14  <instruction name="inst3">
      <conflictsWith name="inst1"/>
16    <conflictsWith name="inst2"/>
    </instruction>
18 </conflicts>
```

Código 24: 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*.

Advice11

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

Advice5

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

Advice7

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

Advice4

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

Al finalizar el grafo de conflictos se vería como se muestra en la figura **??**.

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.

7.6.1. Árbol de Recubrimiento

El árbol de recubrimiento se construye utilizando una variación del algoritmo de Wilson[4], 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 **??**, se va a escoger un vértice aleatorio. De escoger el vértice *advice4* al no tener sucesores, es descartado inmediatamente porque no es posible continuar armado el árbol.

   Como raíz será escogido el *advice7*.

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*.

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.

4. **Árbol Terminado** - El algoritmo continúa hasta llegar a que los vértices no tienen más sucesores. En la figura **??** 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.

7.7. Pruebas

Para probar *AspectCaffeine* se utilizó el mismo *framework* de pruebas descrito en la sección 6.3.1.

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 agregue un nuevo aspecto, sin

asociarlo a ninguna instancia especifica 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 25: 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 26: 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 27: Instrucción para deshabilitar un aspecto

*

Sensores

Los sensores utilizados en *AspectCaffeine* también extienden los sensores definidos en la sección 6.3.1, 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.

8. 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 especificas. 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 especificas 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.

9. 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, cómo 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 especifica, 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.

Referencias

[1] C. A. *Aspect-Oriented Workflow Languages: AO4BPEL and Applications*. PhD thesis, TU Darmstadt, Fachbereich Informatik, 2007.

[2] J. C. Composición de modelos ejecutables extensibles en una fábrica de aplicaciones basadas en workflows. Master's thesis, Universidad de los Andes, 2007.

[3] R. D. Modelos ejecutables extensibles como activos en una fábrica de motores de workflow: Caso bpel. Master's thesis, Universidad de los Andes, 2007.

[4] W. 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.

[5] H. E. and H. 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.

[6] B. et al. Towards a precise definition of the omg/mda framework. automated software engineering. In *Proceedings. 16th Annual International Conference*, pages 273 – 280, 2001.

[7] K. G., H. E., H. J., K. M., P. J., and G. 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.

[8] K. G., L. J., M. A., M. C., L. C., L. J., and I. J. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

[9] M. H. and K. 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.

[10] N. I., B. L., and A. 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.

[11] V. J., S. M., and R. 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.

[12] V. J., S. M., and B. I. Enriching a process engine with flexible time management through model weaving. 2007.

[13] Z. J. Aspect interference and composition in the motorola aspect-oriented modeling weaver.

[14] K. J. Lieberherr, I. M. Holland, and A. J. Riel. Object-oriented programming: An objective sense of style. In *OOPSLA*, pages 323–334, 1988.

[15] S. E. Ltd. Shazam, Julio 2008.

[16] B. M., V. K., J.Ñ., V. W., S. V., T. E., and J. W. Isolating process-level concerns using padus. In *In Proceedings of the 4th International Conference on Business Process Management (BPM 2006*. Springer-Verlag, 2006.

[17] S. M. and K. S. *Superimpositions and aspect-oriented programming*, volume 46. 2003.

[18] Mellodis. Midomi, Julio 2008.

[19] C.Ñ. and V. C. Composición y adaptación de modelos ejecutables extensibles para aplicaciones elearning. caso ims-ld. Master's thesis, Universidad de los Andes, 2008.

[20] OASIS. History of bpel. Publicación Web, Octubre 2008.

[21] B. P. and S. M. Construcción de una línea de producción de motores de workflow basada en modelos ejecutables. Master's thesis, Universidad de los Andes, 2006.

[22] D. P.E.A. *Resource-based Verification for Robust Composition of Aspects*. PhD thesis, Univeristy of Twente, Enschede, June 2008.

[23] D. P.E.A., S. T., B. L.M.J., and A. M. Reasoning about semantic conflicts between aspects. In *EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software*, 2005.

[24] M. R. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[25] R. Y. R., G. S., F. R., S. G., B. J., M.Ñ., S. E., and G. G. Directives for composing aspect-oriented design class models. *Trans. Aspect-Oriented Software Development*, pages 75–105, 2006.

[26] M. S. A testing framework for dynamic composable executable models. Master's thesis, Universidad de los Andes, 2007.

[27] W. S., C. F., L. F., S. T., and F. D. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, March 2005.

[28] O. W. S. B. P. E. L. W. TC. Web services business process execution language version 2.0. Publicación Web, Abril 2007.

[29] H. W. and O. 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.