Gestión de Conflictos entre Aspectos mediante un Sistema Experto de Reglas.

Sandra I. Casas
Unidad Académica Río Gallegos, Universidad Nacional de la Patagonia Austral,
Río Gallegos, Argentina, 9400
lis@uarg.unpa.edu.ar

J. Baltasar García Perez-Schofield
Departamento de Informática, Universidad deVigo,
Orense, España, 32004
jbgarcia@uvigo.es

Claudia A. Marcos
Instituto de Sistemas de Tandil, Universidad Nacional del Centro
Tandil, Argentina, 7000
cmarcos@exa.unicen.edu.ar

Abstract.

The Aspect Oriented Programming is a new programming paradigm that aims to achieve a greater modularization and decomposition of units in the design and the implementation of software systems, the goal is that the applications will be easier to implement, maintain and reuse. The conflicts occurrence among aspects is a consequence of the decomposition of the software systems in the aspect oriented development. This phenomenon is independent to the tools and it requires special attention and treatment since the activation of certain conflicts could cause unwanted, inconsistency and inexactness behaviours in the software systems execution. The work herein outlines a rules expert system approach to solve the problem of conflicts among aspects in an integral and simultaneous form.

Keywords: Aspect Oriented Programming, Aspects, Aspects Conflicts.

Resumen.

La Programación Orientada a Aspectos se postula como un nuevo paradigma de programación, que aspira a lograr una mayor modularización y descomposición de unidades en el diseño y la implementación de sistemas software, con el objeto que las aplicaciones sean más fáciles de implementar, mantener y reusar. La ocurrencia de conflictos entre aspectos es una consecuencia de la descomposición de los sistemas software en el desarrollo orientado a aspectos. Dicho fenómeno es independiente a las herramientas y requiere una especial atención y tratamiento ya que la activación de ciertos conflictos puede provocar comportamientos no deseados, inconsistentes e imprecisión en la ejecución de los sistemas software. El presente trabajo plantea un enfoque basado en un sistema experto de reglas para resolver el problema de conflictos entre aspectos en forma integral y simultánea.

Palabras claves: Programación Orientada a Aspectos, Aspectos, Conflictos entre Aspectos,

1. INTRODUCCIÓN

La programación orientada a aspectos [1] (POA) es un nuevo paradigma para el desarrollo de sistemas software que proporciona mecanismos y abstracciones para la implementación de las propiedades no funcionales o la funcionalidad secundaria (logging, seguridad, persistencia, concurrencia, registro de actividades, etc.), de manera separada y aislada. Es decir, la orientación a aspectos, es una técnica que permite aplicar el principio de Separación de Concern [2] y de esta

forma, obtener una mayor y mejor modularización del código. El enfoque resulta muy prometedor y atrayente ya que supera las características indeseables producida por el efecto de "tiranía de la descomposición dominante" [3] que tiene como consecuencias negativas la generación de código mezclado y diseminado. La POA provee mecanismos que permiten aplicar una mayor descomposición modular de los sistemas, proporcionando el beneficio que las mismas resultan más fáciles de diseñar, codificar, mantener y reusar.

Últimamente un nuevo tópico es considerado como área de investigación y problemática del paradigma a resolver "*el fenómeno de los conflictos entre aspectos"*, también denominados en la literatura como "*interacciones*" [4] o "*interferencias*" [5]. En ciertos casos estas interacciones, pueden tener por resultado un comportamiento contradictorio, nulo o innecesario. De esta manera, surgen los conflictos entre aspectos y la necesidad de construir mecanismos y estrategias para su adecuado tratamiento.

El presente trabajo aborda la problemática de conflictos entre aspectos y proporciona una propuesta de solución basada en un sistema experto de reglas. En esta primera instancia del estudio, se proyecta una herramienta de desarrollo POA que brinde mecanismos y soporte para el tratamiento de conflictos entre aspectos, superando los enfoques actuales. El resto del artículo se organiza como sigue: se sintetizan los mecanismos utilizados por distintas herramientas POA y trabajos más específicos referentes a conflictos, para describir a continuación la estrategia basada en un sistema experto de reglas; finalmente se discute la arquitectura de la futura herramienta que implementará esta aproximación, para entonces exponer las conclusiones.

2. CONFLICTOS ENTRE ASPECTOS: DETECCIÓN Y RESOLUCIÓN

Un conflicto puede ocurrir cuando dos o más aspectos compiten por su activación [6], o "la interacción entre aspectos ocurre cuando varios aspectos afectan el mismo punto de programa (ejecución o estructura)" [7]. En términos de codificación, se reconoce que un objeto de funcionalidad básica puede ser asociado a más de un aspecto, cada uno de los cuales tiene su propio objetivo de comportamiento. Si los aspectos son ortogonales [8], el sistema se ejecutará sin problemas. Pero, el comportamiento del sistema se torna impredecible si los aspectos que compiten no son independientes. En estos casos, el desarrollador debe ser informado y poder controlar estas potenciales situaciones para determinar la ejecución deseada de acuerdo al tipo de conflicto y/o el dominio de la aplicación, determinando las prioridades y políticas de activación de los aspectos.

En la mayoría de las herramientas POA, la identificación, control y resolución de conflictos es una tarea absolutamente manual. Concretamente, si dos o más aspectos presentan una potencial situación de conflicto, el tejedor de aspectos procede sin ningún tipo de inconveniente, ni aviso y/o comunicación previa al desarrollador.

La detección de conflictos puede ser una tarea sencilla si la aplicación maneja una reducida cantidad de unidades (clases y aspectos), pero la complejidad de la tarea crece en la medida que aumentan las unidades componentes de la aplicación. Además, existen otros factores que hacen esta tarea aun más ardua y compleja: (a) la herencia de aspectos puede introducir potenciales situaciones conflictivas que pueden pasar inadvertidas y cuya identificación puede resultar costosa; (b) las tareas de mantenimiento de las aplicaciones que requieren la adición, remoción y modificación de los componentes pueden introducir nuevas situaciones conflictivas, como demandar que determinadas políticas sean desactivadas ante la eliminación de conflictos inexistentes; (c) el uso de ciertas construcciones (por ejemplo los comodines en AspectJ [10]) empleadas para abreviar y agrupar puntos de unión, hace menos legible la identificación de conflictos; (d) la codificación de una aplicación realizada por un grupo de desarrolladores, reduce las posibilidades de llevar el control de conflictos e impacta negativamente al momento de efectuar la identificación manual de los mismos.

Como se ha observado [9] la mayoría de las herramientas POA ofrecen mecanismos muy restringidos y pobres para la resolución de conflictos. La mayoría de éstas ofrece como resolución un mecanismo que consiste en la fijación de orden, prioridad o precedencia, para lo cual los lenguajes proporcionan alguna semántica especial. Por ejemplo, AspectJ [10] proporciona la cláusula "declare precedence" para ordenar la ejecución de los avisos de los aspectos que provocan el conflicto; AspectC++ [11] provee un orden de declaración mediante la cláusula "order"; DJ Aspect [12] emplea la instrucción "dominates" y AOP/ST [13] maneja la relación de orden de composición, mediante un mecanismo sencillo por el cual se establecen las prioridades de composición/ejecución asignando un valor numérico a cada aspecto, así los aspectos con alta precedencia (valor numérico más alto) se ejecutan primero. Se han propuesto enfoques diferentes como ser Alpheus [14], Astor [15] y Reflex [7]. Alpheus es una herramienta da soporte a la especificación de asociaciones y conflictos entre los componentes. Aquí es importante el concepto de abstracción de conflictos al más alto nivel de granularidad, permitiendo declaraciones en niveles de funcionalidad y no sólo entre aspectos. Soporta un manejo flexible de conflictos con la introducción del concepto de planos (agrupación de aspectos relacionados), de esta manera se otorga facilidad en la definición y mantenimiento, siendo los niveles de granularidad de conflictos soportados: de aspecto-aspecto (determina un conflicto entre dos aspectos específicos); de aspecto-plano (establece conflictos entre un aspecto y plano); de plano-plano (especifica un conflicto entre todos los aspectos de un plano con respecto a todos los

aspectos de otro plano); y de aspecto-todos (permite la especificación de conflictos "uno a muchos", donde un aspecto específico tiene un conflicto con todos los otros aspectos). Pero lo mas destacado de Alpheus es que soporta un manejo amplio de conflictos que son clasificados en: de orden, opcional, exclusivo, nulo o dependiendo del contexto. Permite que el usuario defina los conflictos potenciales entre aspectos o planos, especificando la política a seguir si el conflicto es detectado; esto se realiza identificando los componentes involucrados (por ejemplo, aspecto-aspecto, aspecto-plano), el tipo de conflicto (por ejemplo, en orden) y, en el caso de conflictos dependientes del contexto, el código específico según el conflicto detectado. Esta última característica sólo existe en Alpheus. Sus inconvenientes son: manejo de conflictos de a pares; restricción en la generación de código cuando se detecta un conflicto entre dos aspectos para los cuales el conflicto no ha sido especificado. Astor [15] es un prototipo que propone una serie de mecanismos y estrategias para mejorar el tratamiento de conflictos en AspectJ [10]. Los mismos se soportan mediante la adición de Administrador de Conflictos que cumple principalmente con las funciones de detectar automáticamente conflictos y aplicar estrategias de resolución más amplias que las que AspectJ tiene por defecto, en forma semiautomática. La detección de conflictos actúa por una clasificación de los mismos por niveles de semejanza [16] y la resolución se efectúa siguiendo las directrices de una taxonomía que proporciona seis categorías de resolución [14]. La implementación del prototipo esta basada en el pre-procesamiento de código AspectJ. Reflex [7] es una herramienta que facilita la implementación y composición de diferentes lenguajes orientados a aspectos. Según los autores, lo atrayente del modelo es que provee un alto nivel de abstracción para implementar los nuevos lenguajes de aspectos y soportar la detección y resolución de conflictos. Reflex consiste básicamente en un kernel cuya arquitectura dispone de 3 capas, una de ellas es la capa de composición para la detección y resolución de interacciones. La detección de interacciones sigue el esquema propuesto por [17] y se limita a una aproximación estática de la interacción de aspectos, solo se detectan las interacciones que no ocurren en tiempo de ejecución. Las dos formas de resolver una interacción son: (1) elegir de las interacciones el aspecto que se va a aplicar en la ejecución (2) ordenar y anidar los aspectos para la ejecución. Este trabajo anticipa que una herramienta POA debe manejar conflictos y en consecuencia provee una capa específica del kernel para este propósito, pero le impone métodos de resolución muy restringidos.

El problema de la detención de conflictos entre aspectos ha recibido mayor tratamiento. Existen varios y diversos trabajos enfocados únicamente en la identificación o análisis de los mismos pero que no incluyen aportes al problema de la resolución como ser [18], [19], [5] y [20].

Los tejedores dinámicos µDyner para C++ [21] y microDyner [22] son herramientas aún más restringidas en cuanto al tratamiento de conflictos que las mencionadas, ya que éstas no admiten que un punto de unión sea asociado a más de un aviso o punto de corte o aspecto.

Hasta el momento los trabajos y contribuciones se han alineados principalmente en las siguientes direcciones: (a) tratamiento parcial del problema de conflictos entre aspectos, ya que apuntan únicamente al análisis y detección del conflicto y no se suma ningún aporte a la resolución; (b) una cantidad importante de los trabajos son específicos para AspectJ; (c) las propuestas que tienden a ser más generales aún ofrecen escasa capacidad y creatividad en cuanto la resolución. En consecuencia, los objetivos de estudio de la presente investigación son: i) elaborar una propuesta "integral", es decir, que aborde el tratamiento de conflictos abarcando sus dos dimensiones: la detección y la resolución; ii) la resolución de conflictos puede ser más amplia y flexible que el simple esquema de orden, prioridad o precedencia, como se ha demostrado en Alpheus y Astor; iii) la propuesta deben tener carácter más general. En las próximas secciones se describen las estrategias y mecanismos que apuntan a lograr estos objetivos.

3. GESTIÓN DE CONFLICTOS MEDIANTE UN SISTEMA EXPERTO DE REGLAS

Un sistema experto basado de reglas es un enfoque muy interesante para el tratamiento de conflictos entre aspectos, ya que esta estrategia permitirá detectar y resolver conflictos de forma automática y en forma simultánea. La detección es un problema relativamente sencillo, en cambio la resolución es un tanto más compleja. La resolución de cada conflicto depende exclusivamente de la semántica de la aplicación, y en este sentido es el desarrollador de la aplicación quien conoce y sabe cómo debe resolverse cada conflicto en particular, así se constituye en "el experto", que especifica reglas de resolución para cada conflicto. Mediante un sistema experto, el desarrollador puede establecer reglas, generales o específicas, y solo sobre aquellos conflictos que en particular considere deban ser atendidos. El objetivo principal es que a partir del conjunto de reglas definidas por el experto, el resto del proceso (detección y aplicación de las reglas) se efectúe en forma automática y transparente.

En un sistema experto de reglas, la siguiente expresión permitirá no solo determinar la existencia de un determinado conflicto, sino además aplicar un método para resolverlo que será incluido en el proceso de tejido.

SI <*existe conflicto X*>
ENTONCES <*aplicar resolución Y*>

Donde la expresión <*existe conflicto X*> es una expresión que involucra un conjunto de condiciones que han de ser diferentes para cada conflicto en particular. Y donde la expresión <*aplicar resolución Y*> es una función que permite aplicar una categoría de resolución, (también simple o compleja) que implica algún tipo de modificación en el tejido o composición de los aspectos involucrados en el conflicto. De esta forma, si todas las asociaciones entre aspectos y clases son convertidas o mapeadas en la base de conocimientos del sistema experto, pueden ser utilizadas en una regla tanto para la detección como para la resolución del conflicto que plantean. Por ejemplo la siguiente regla R1, codificada en un lenguaje de sistema experto de reglas como CLIPS [25], verificará la existencia de un conflicto entre los aspectos Persistence y ActvityRecord vinculados al mismo punto de unión (Clase Account, método extract):

```
R1: (and (asociación (Persistence  call-after Account extract))
         (asociación (ActivityRecord call-after Account extract))
=>
    S1
```

3.1 Resolución Amplia de Conflictos

Una estrategia "amplia" de resolución de conflictos significa que existen múltiples y variados métodos para solucionar un conflicto. En este sentido se pueden aplicar un conjunto de categorías de resolución, denominadas básicas, (algunas de éstas están inspiradas en la taxonomía propuesta en Alpheus [23] y utilizada en Astor [21]) o alguna categoría combinada.

Categorías de Resolución básicas:

- Orden: esta categoría establece un orden de ejecución para los aspectos integrantes del conflicto.
- Opcional: esta categoría establece condiciones para que se ejecuten opcionalmente los aspectos integrantes del conflicto. Estas condiciones se puede definir de acuerdo a alguna política interna del sistema o aleatoria.
- Exclusión: esta categoría establece la exclusión (eliminación) de la ejecución de algunos de los cortes de los aspectos integrantes del conflicto.
- Anulación: esta categoría establece la anulación (eliminación) de la ejecución de todos los cortes de los aspectos integrantes del conflicto.
- Integración: esta categoría establece que los cortes de los aspectos integrantes del conflicto se funden en uno solo.

Siguiendo el ejemplo anterior, al conflicto detectado entre los aspectos Persistente y ActivityRecord se puede resolver aplicando la categoría de orden. Acción que influye en el posterior proceso de tejido de los aspectos.

```
R1: (and (asociación (Persistence  call-after Account extract))
         (asociación (ActivityRecord call-after Account extract))
=>
    (order ActivityRecord Persistence)
```

La regla R2 especifica que si existe un conflicto entre los aspectos Persistence y ActivityRecord, la acción que se ejecutará es opcional, depende de la evaluación de una condición. Por ejemplo, en este ejemplo se considera que si el importe a extraer no es superior a 100 no es necesario realizar el registro de actividades.

```
R2: (and (asociación (Persistence  call-after Account extract))
         (asociación (ActivityRecord call-after Account extract))
=>
    If (amount > 100)
       (order ActivityRecord Persistence)
     Else
       (exclude ActivityRecord)
```

Combinación de categorías básicas

Cuando el conflicto esta integrado por más de dos asociaciones, puede ser necesario resolver el conflicto aplicando más de una categoría de resolución básica. De esta forma, nuevas categorías se definen a partir de la combinación de las

básicas, y así se brindan mayores posibilidades de resolución. La siguiente lista enumera un conjunto de combinaciones válidas:

- Exclusión – Orden
- Integración - Orden
- Opcional – Orden
- Opcional – Orden- Anulación
- Opcional – Orden – Exclusión
- Opcional – Orden – Integración
- Opcional – Integración - Exclusión
- Opcional – Interacción – Anulación
- Opcional – Anulación – Exclusión
- Opcional - Orden – Anulación – Exclusión
- Opcional - Orden – Anulación – Integración
- Opcional - Orden – Exclusión – Integración
- Opcional - Exclusión – Anulación - Integración
- Opcional – Orden – Anulación – Exclusión -Integración

Por ejemplo, la regla R1, aplica como acción la combinación las categorías de Exclusión y Orden. La regla R2, es un tanto más compleja, combina Opcional – Exclusión y Orden.

```
R1: (and (asociación (Persistence  call-after Account extract))
         (asociación (ActivityRecord call-after Account extract))
         (asociación ( Security call-after Account extract )))
=>
    (exclude ActivityRecord)
    (order Security Persistence)

R2: (and (asociación (Persistence  call-after Account extract))
         (asociación (ActivityRecord call-after Account extract))
         (asociación ( Security call-after Account extract )))
=>
    if (time_System < 15 )
        (exclude ActivityRecord)
        (order Security Persistence)
    else
        (order Security Persistence ActivityRecord)
```

El proceso de detección y resolución se efectúa en forma conjunta o simultánea, solo se requieren los siguientes elementos: la base de conocimientos (reglas y hechos) y las asociaciones. Se prescinde totalmente de las clases y aspectos, lo que independiza a la detección y resolución de manera tal que es específica de cada aplicación en particular y conjunto de reglas.

4. DISEÑO ARQUITECTÓNICO DEFINIDO

A continuación se presenta el diseño de una herramienta de desarrollo de sistemas orientados a aspectos que extiende al lenguaje Java e incorpora un sistema experto para la gestión de conflictos entre aspectos. El diseño arquitectónico que se plantea (Figura 1) esta basado en la definición de tres módulos de software. La división de los mismos responde al momento en que sus componentes han de ejecutarse.

- *Development Module:* Este módulo dispone de todos los componentes necesarios para la especificación y/o codificación de una aplicación orientada a aspectos.

- *Compiler Module:* Este módulo dispone de un conjunto de componentes que preparan a las distintas unidades (clases y aspectos) para su carga, tejido y ejecución. En esta etapa se aplica el proceso de identificación y resolución de conflictos.

- *Execution Module:* Este módulo es responsable del tejido de los aspectos (libres de conflictos) y las clases y ejecutar la aplicación.

En la Figura 1 se puede observar, que el sistema experto de reglas esta absolutamente embebido en la herramienta. A continuación se describe cada módulo de la arquitectura y sus componentes principales.

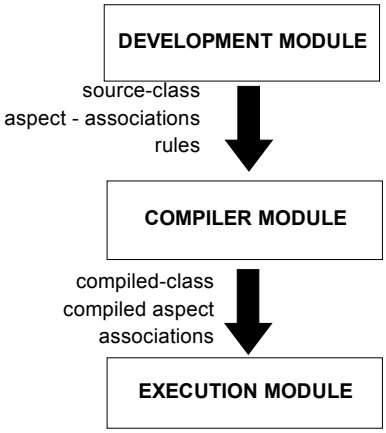

Figura 1: Arquitectura General Propuesta.

4.1 Development Module (Módulo de Desarrollo)

El objetivo principal de este módulo es capturar toda la información que se requiere como entrada para el desarrollo de un sistema software: clases, interfaces, aspectos, asociaciones y reglas. Esto implica no solo una interfaz adecuada para cada entidad sino además adicionar una importante carga de funcionalidad destinada a la consistencia y validación de la información que se ingresa. En la Figura 2 se ilustran los componentes de este módulo.

El componente Units Manager gestiona la codificación de las unidades funcionales (clases e interfaces) y no funcionales (aspectos) de una aplicación.

El componente Association Manager gestiona las relaciones de asociación entre unidades funcionales y no funcionales (clases y aspectos). Una asociación es una entidad que relaciona a un elemento de un aspecto (especie de método) con un elemento de una clase (punto de unión). Así, una asociación define con exactitud un punto de corte. Para ello, una asociación especifica en forma precisa: el punto de unión que se corta (clase, método y/o atributo), aspecto y método del mismo que se insertará y tipo de corte o asociación (por ejemplo call-before). Entonces un elemento de aspecto puede tener N asociaciones a elementos de clases y a su vez un elemento de clase puede participar de N asociaciones. La forma en que un aspecto puede asociarse a un elemento a una clase se denomina tipo de asociación. El componente Associations Manager debe efectuar además una serie de validaciones que garanticen la consistencia de las asociaciones y permita que la información que se extraiga de ésta posteriormente sea correcta.

El componente Rules Manager gestiona la especificación de las reglas para la detección y resolución de conflictos de manera fácil y sencilla. Las reglas se especifican para detectar y resolver conflictos de asociaciones (puntos de cortes). En este esquema un conflicto ocurre cuando dos o más asociaciones coinciden en aspecto, elemento de aspecto, punto de unión (método/atributo de la misma clase) y tipo de asociación. Cada regla es una entidad que describe las siguientes propiedades: un nombre que la identifica de manera univoca, una condición y una acción asociada si la condición se cumple. Las reglas especificadas requieren ser verificadas para asegurar su correcta y válida aplicación, ésta es otra responsabilidad del componente Rules Manager. Las reglas generadas correctas y válidas son almacenadas en la base de conocimiento para efectuar el proceso de resolución de conflictos en tiempo de compilación.

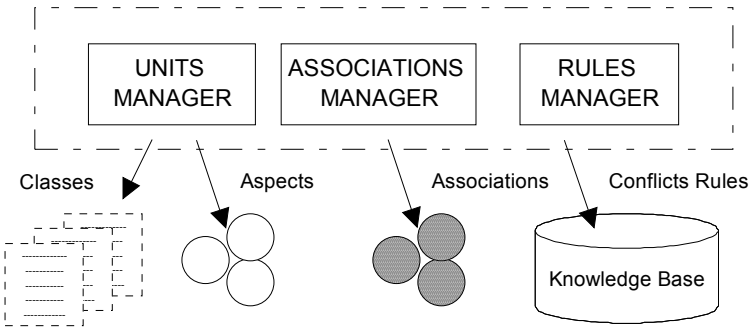

Figura 2: Componentes del Módulo de Desarrollo.

4.2 Compiler Module (Módulo de Compilación)

El propósito de este módulo es realizar todos aquellos procesos y funciones necesarios que preparen al sistema software desarrollado para su ejecución. Los componentes principales de este módulo son: Aspect Manager, Conflict Manager y Class Compiler (Figura 3).

El componente Aspect Manager toma como entrada los aspectos y asociaciones generados en el módulo de desarrollo y realiza dos tareas fundamentales. La primer tarea consiste en extraer los datos necesarios para generar la base de hechos de asociaciones (base de conocimiento) que se utilizarán para la detección y resolución de conflictos, más adelante. Esto significa efectuar una especie de mapeo de las asociaciones a hechos. Este mapeo se hace conforme a la definición del hecho asociación que se requiere almacenar en la base de conocimientos. El componente Aspect Manager genera un hecho por cada asociación existente que indica un punto de corte a un determinado punto de unión para la inserción del código de un determinado aspecto. La segunda tarea fundamental es transformar los aspectos en clases y preparar el código fuente en lenguaje Java para que el componente Class Compiler efectúe luego la compilación.

El componente Conflict Manager es en si mismo el sistema experto de reglas cuyo objetivo es detectar y resolver los conflictos. Una vez que la base de conocimientos esta generada (con los hechos que representan a las asociaciones y las reglas definidas por el desarrollador), puede lanzarse la ejecución del motor de inferencia. Como resultado algunas reglas se "dispararán" (o activan) produciendo como acción asociada, la resolución de determinados conflictos. La resolución de los conflictos implica la "modificación" de la estructura de los objetos asociaciones. El Conflict Manager genera un objeto "Solver" que encapsula la resolución del conflicto y vincula este objeto a las asociaciones participantes del conflicto (Figura 3). Es de notar, que la resolución de un conflicto puede afectar más de una asociación. La resolución de un conflicto consiste en una especie de directiva o indicación que será interpretada y aplicada en el proceso de tejido-composición de los aspectos y clases.

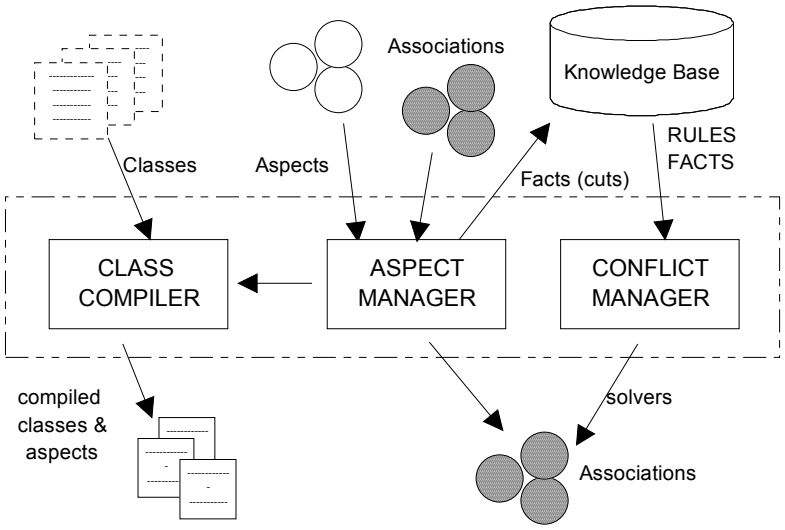

Figura 3: Componentes del Módulo de Compilación.

4.3 Execution Module (Módulo de Ejecución)

Este módulo es el responsable de ejecutar la aplicación desarrollada. Antes de la ejecución, las clases del sistema software se cargan en demanda, para la posterior interpretación de la JVM (Java Virtual Machine). Un ClassLoader personalizado permitirá detectar previamente a las clases que son afectadas por alguna asociación y en consecuencia requieren ser tejidas, como se indica en la Figura 4. En estos casos, se derivará el control al componente Aspect Weaver para que efectúe las transformaciones (tejido) necesarias. Cuando el componente Aspect Weaver toma el control obtiene toda la información de dicha asociación y crea un método "derivador" por cada par (punto de unión, tipo de asociación) afectado, en una clase denominada "derivadora". Es decir, si para el punto de unión X existen dos asociaciones (a aspectos diferentes) de tipo call-before, se crea un único método derivador. En cambio si el punto de unión X esta asociado a dos aspectos pero una asociación es de tipo call-before y la otra es de tipo call-after se crean dos métodos derivadores. La clase derivadora es creada en este proceso. Los métodos derivadores incluyen los mensajes a los métodos de los aspectos definidos en las asociaciones correspondientes. En el caso que el método derivador corresponda a dos asociaciones, lo cual expresa una situación conflictiva, el método derivador incluirá la directiva o indicación definida en el objeto "solver" (creado previamente por el componente Conflict Manager) vinculado a las asociaciones en cuestión.

Para aquellos casos que para la situación conflictiva no se haya definido una regla, y por lo tanto no existe un objeto solver que indique la forma de composición, las invocaciones de los aspectos se tejen en el orden en que el Aspect Weaver las toma de la colección de asociaciones. A continuación se describe el proceso mediante el gráfico de la Figura 5. En el gráfico se muestran tres puntos de unión: clase Transaction – método setAmount; clase Account – métodos debit y getBalance. Los aspectos ActivityRecord y Persistence se han transformado en clases. Durante el proceso de tejido o composición se realizan: (a) la creación de la clase derivadora y (b) relación (enlace) de los puntos de unión con la clase derivadora. En la Figura 5, el método setAmount, antes de su ejecución invoca al método derivador D1, que a su vez invoca al método register del aspecto ActivityRecord. Luego de la ejecución del método register, continuará la ejecución del método setAmount normalmente. Dado que el código del aspecto se ejecuta *antes* que el punto de unión, significa que el tipo de asociación definida entre éste y el aspecto era call-before. Si la asociación definida hubiere sido de tipo call-after, la invocación al método derivador se habría insertado *después* de la ejecución del método setAmount.

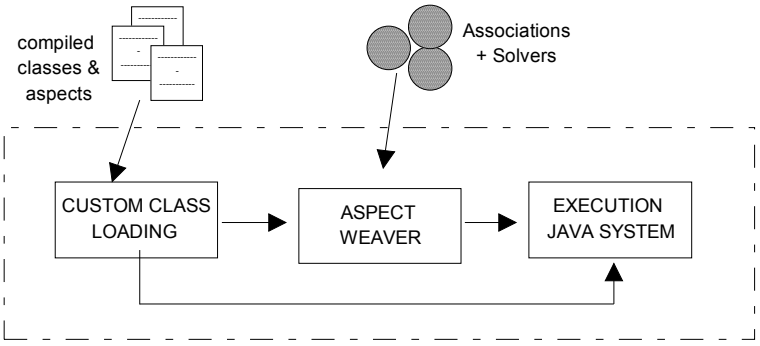

Figura 4: Componentes del Módulo de Ejecución.

El método derivador D2 relaciona al punto de unión (clase Account- método debit) con dos aspectos. Esto es así porque existían dos asociaciones del mismo tipo para este punto de unión. En la breve descripción del código del método derivador D2, se ha incluido una condición que involucra la ejecución del método register o save, esto significa que se ha aplicado la categoría de resolución de conflicto opcional.

En resumen, en el punto de unión (clase y método o atributo), el componente Aspect Weaver inserta una llamada a un método derivador, antes-después o en lugar del código de la misma (según el tipo de asociación). El método derivador, consiste en la invocación del método del aspecto asociado en forma directa, o puede incluir un conjunto de sentencias que aplican una categoría de resolución de conflicto. De esta forma las clases no tienen conocimiento de que aspecto las corta, los aspectos preservan su estado original y pueden ser asociados a cualquier otro punto de unión y la resolución de los conflictos queda escondida en la clase derivadora, siendo específica para una determinada aplicación.

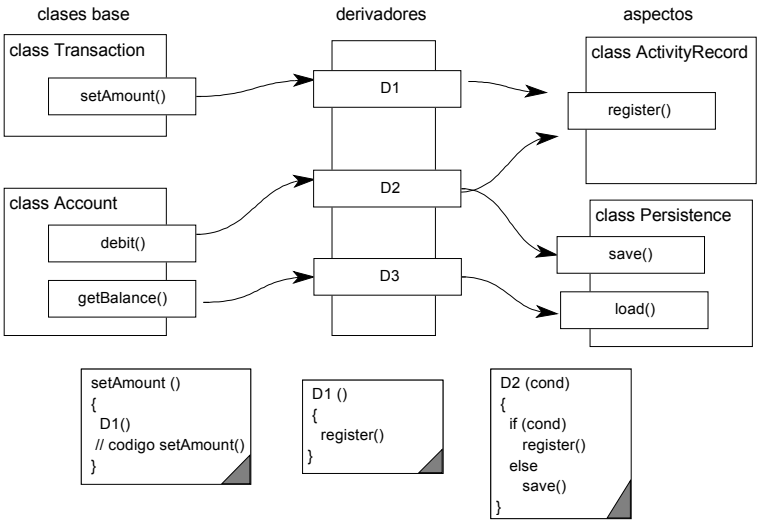

Figura 5: Tejido de Aspectos, Clases y Resolución de Conflictos.

5. CONCLUSIONES

Este trabajo propone una solución al problema de conflictos entre aspectos que satisface los tres objetivos de estudio identificados: i) es un enfoque integral: aborda en forma integrada y completa las dos dimensiones del problema de conflictos entre aspectos, la detección y resolución de los conflictos. Se propone un mecanismo que de manera automática detecta los conflictos entre aspectos y de manera semi-automática los resuelve; ii) es un enfoque amplio y flexible en cuanto a los métodos de resolución que propone: proporciona más de veinte formas de resolver un conflicto en contraposición al limitado y restringido enfoque convencionalmente soportado de orden, precedencia o prioridad; iii) es un enfoque general e independiente: la propuesta se aplicará a un lenguaje simple de aspectos y se construirá un tejedor específico a los efectos de demostrar la incidencia del proceso de resolución en la composición particular de una aplicación. Pero la propuesta puede adaptarse a otras herramientas y lenguajes con leves modificaciones y mínimo esfuerzo ya que lo único que se requiere es que se añadan métodos de especificación de reglas, mapeo automático de hechos y técnicas de composición que incorporen las resoluciones en el proceso de tejido. Estos dos últimos procesos pueden realizarse en compilación o ejecución. De esta forma, por ejemplo un esquema muy similar puede incorporarse en AspectJ.

La estrategia particular adoptada esta basada en la incorporación de un sistema experto de reglas que se funde en la arquitectura de manera natural y tiene las siguientes características: el conjunto de hechos se genera automáticamente mediante el mapeo de asociaciones; el conjunto de reglas es especificado por el desarrollador, las condiciones implican cualquier cantidad de asociaciones y la acción de una regla corresponde a una categoría de resolución básica o combinada; el proceso de detección y aplicación de resoluciones es simultáneo y automático.

La definición y aplicación de políticas de resolución de conflictos (conjunto de reglas) es independiente de los aspectos y clases a los que se refiere, siendo específicas a la aplicación que los utiliza. Esta vinculación no altera la estructura de las unidades (clases y aspectos) originalmente desarrolladas, favoreciendo de este modo el reuso de las mismas.

El trabajo actual y futuro esta centrado en la implementación de la herramienta que de soporte al enfoque propuesto en este artículo.

Referencias

[1] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J: Aspect-Oriented Programming. In Proceedings of ECOOP (1.997).

[2] Hürsch W., Lopes C.: Separation of Concerns. Northeastern University, Technical Report NU-CCS-95-03, Boston (1.995).

[3] Tarr P., Ossher H., Harrison W. and Sutton Jr. S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. Proceedings of ICSE'99., IEEE Computer Society Press / ACM Press, (1.999) 107-119.

[4] Duoence R., Fradet P. and Südholt M.: "Detection and Resolution of Aspect Interactions", Technical Report Nº4435, INRIA, ISSN 0249-6399, Francia (2.002).

[5] ROOTS: LogicAJ – A Uniformly Generic and Interference-Aware Aspect Language. http://roots.iai.uni-bonn.de/researh/logicaj/ , (2.005).

[6] Pryor J., Diaz Pace A., Campo M.: Reflection on Separation of Concerns. RITA. Vol.9. Num.1 (2.002).

[7] Tanter E., Noye J.: A Versatile Kernel for Multi-Language AOP. Proceeding of ACM SIGPLAN/SIGSOFT – Conference on Generative Programming and Component Engineering (GPCE) LNCS, Springer-Verlag, Estonia, (2.005).

[8] Kienzle J., Yu Y., Xiong J.: On Composition and Reuse of Aspect. Foundations of Aspects Oriented Languajes, FOAL, USA (2.003).

[9] Casas S., Reinaga H., Sierpe L., Vanoli V., Saldivia C., Prior J.: Clasificación y Resolución de Conflictos entre Aspectos. VII Workshop de Investigadores en Ciencias de la Computación – Argentina (2.005).

[10] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W.: An Overview of AspectJ. ECOOP (2.001).

[11] Gal A., Schroder W., Spinczyk O.: AspectC++: Language Proposal and Prototype Implementation. ACM International Conference Series Proc. of the 40th International Conference on Tools Pacific. Vol.10. Australia. (2.002).

[12] Pryor J., Bastán N., Campo M.: A Reflective Approach to Support Aspect Oriented Programming in Java. In Proc. of the ASSE. 29 JAIIO. Argentina. (2.000).

[13] Boellert, K.: On Weaving Aspects. Proc. of the AOP Workshop at ECOOP (1.999).

[14] Pryor J., Marcos C.: Solving Conflicts in Aspect-Oriented Applications. Proceedings of the Fourth ASSE. 32 JAIIO. Argentina. (2.003).

[15] Casas S., Marcos C., Vanoli V., Reinaga H., Saldivia C., Pryor J. y Sierpe L.: ASTOR: Un Prototipo para la Administración de Conflictos en Aspecto. XIII Encuentro Chileno de Computación, (JCC 05), Chile (2.005).

[16] Casas S., Marcos C., Vanoli V., Reinaga H., Sierpe L., Prior J., Saldivia C.: Administración de Conflictos entre Aspectos en AspectJ. VI Argentine Symposium on Software Engineering in 34th JAIIO, Argentina (2.005).

[17] Duoence R., Fradet P. and Südholt M.: A Framework for the Detection and Resolution of Aspect Interaction. In Proceeding of GPCE 2.002, vol. 2487 of LNCS, USA, (2.002), 173-188.

[18] Storzer M., Krinkle J.: Interference Analisys for AspectJ. FOAL: Foundations of Aspect-Oriented Languages, USA (2.003).

[19] Tessier F., Badri M., Badri L.: A Model-Based Detection of Conflicts Between Crosscutting Concern: Towards a Formal Approach. International Workshop on Aspect – Oriented Software Development (WAOSD 04), China (2.004).

[20] Monga M., Beltagui F., Blair L.: Investigating Feature Interactions by Exploiting Aspect Oriented Programming. Technical Report N.comp-002-2.003, Lancaster University, Inglaterra (2.003).

[21] Chen Y., Aspect – Oriented Programming (AOP): Dinamic Weaving for C++. Master thesis, Vrije Universiteit Brussel and Ecole des Mines de Nantes, Francia (2.003).

[22] Segura-Devillechaise M., Meneaud J.: microDyner: efficient dynamic weaving of aspects in native running processs. Languages et Modeles a Objets, Francia (2.003) , 119-133.

[23] Homepage of CLIPS, http://www.ghg.net/clips/CLIPS.html

El presente trabajo fue parcialmente financiado por la Universidad Nacional de la Patagonia Austral, Santa Cruz, Argentina.