AAC: Agente Administrador de Conflictos entre Aspectos en AspectJ

Sandra I. Casas y Verónica L. Vanoli

Unidad Académica Río Gallegos. Universidad Nacional de la Patagonia Austral
Lisandro de la Torre 1070. CP 9400. Río Gallegos. Santa Cruz. Argentina
Tel/Fax: +54-2966-442313/17.

lis@uarg.unpa.edu.ar, vvanoli@uarg.unpa.edu.ar

Resumen. *Un conflicto ocurre cuando dos o más aspectos compiten por su activación, es decir, cuando un módulo funcional (clase o componente) es entrecruzado por más de un aspecto. Si dichos aspectos no son independientes y ortogonales, la ejecución descontrolada puede provocar comportamientos inestables e inesperados. El tratamiento de conflictos resulta ser una tarea compleja, ya que en la mayoría de las herramientas (como AspectJ), la identificación de conflictos es manual y los mecanismos de resolución de conflictos son restringidos. Este trabajo propone un Agente Administrador de Conflictos (AAC) que de forma transparente al desarrollador de aplicaciones AspectJ, detecta los conflictos automáticamente, recomienda métodos de resolución y asiste finalmente a la resolución de los mismos. La propuesta es soportada por un prototipo denominado "IntAOP".*

Palabras claves*: Conflictos entre Aspectos, Agentes, AspectJ, AOP.*

1. Introducción

La Programación Orientada a Aspectos (AOP) [Kiczales 1997] ha recibido considerable interés particularmente como extensión de la Programación Orientada a Objetos (OOP). Este enfoque propone encapsular los crosscutting concerns en módulos separados denominados aspectos. Un aspecto es un tipo entrecruzado que atraviesa diversos módulos de un sistema evitando que el código se mezcle y se disemine explícitamente, mediante mecanismos de composición implícita. Además de la unidad aspecto la AOP incorpora nuevas abstracciones como ser los pointcuts, advices, join-points, etc. Un proceso denominado tejedor de aspectos realiza la composición de clases y aspectos para generar la aplicación ejecutable. Bajo este esquema varias herramientas, lenguajes y frameworks que dan soporte de implementación al enfoque han sido presentados [Hirschfeld01] [HAspectC] [HAspectC++] [Gal02] [Piveta01] [HPythius] [HAspectR] [HAspectWerkz] [HHyper/JTM] [HJboss] [Schult03] entre otras, resultando ser el más maduro y consolidado el lenguaje AspectJ [Kiczales01].

Un conflicto ocurre cuando dos o más aspectos compiten por su activación [Pryor02], es decir, cuando un módulo funcional (clase o componente) es entrecruzado por más de un aspecto. Si dichos aspectos no son independientes y ortogonales, la ejecución descontrolada puede provocar comportamientos inestables e inesperados. Esta problemática se ha instalado fuertemente debido a que el soporte para el tratamiento de conflictos entre aspectos ofrecido por las herramientas AOP es realmente débil. La identificación de conflictos es manual y los mecanismos de resolución de conflictos son restringidos. En otras palabras, la administración de conflictos es una tarea que recae en el desarrollador resultando compleja y costosa.

Lo cierto es que la ocurrencia de conflictos resulta ser una característica inherente a la AOP, es decir, es válido que dos o mas aspectos se asocien o relacionen al mismo módulo funcional (join-point) y es pausible que éstos sean dependientes. Este hecho exige la investigación e indagación de estrategias y mecanismos para la administración de conflictos entre aspectos. Desde este punto de vista, la teoría de agentes puede aportar soluciones, si se considera que una de las principales utilidades de los agentes de software es hacer aplicaciones que ahorren tiempo y esfuerzo al usuario.

Este trabajo propone un Agente Administrador de Conflictos (AAC) que de forma transparente al desarrollador detecta, de forma automática, conflictos entre aspectos codificados en AspectJ, asiste a la resolución de los mismos y recomienda los métodos de resolución. AAC es un agente reactivo con estado interno (basado en modelos) [Russell95], que dado sus roles se convierte en un dispositivo de monitorio, asistencia y de recomendación, a la vez. La propuesta es soportada por un prototipo denominado “IntAOP”. No existen antecedentes que revelen el empleo de agentes de software como estrategia para la administración de conflictos entre aspectos.

Este trabajo se organiza de la siguiente manera. En la Sección 2 se plantea el problema de los conflictos en AspectJ. En la Sección 3 se describe el contexto del AAC incorporada al prototipo IntAOP. En la Sección 4 se definen los roles, responsabilidades y servicios que definen al AAC. La Sección 5 presenta el diseño e implementación del AAC. La Sección 6 plantea una discusión y trabajo futuro. En la Sección 7 se presentan los trabajos relacionados. Y finalmente, en la Sección 8 se exponen las conclusiones del trabajo propuesto.

2. El Problema: Conflictos en AspectJ

AspectJ [HAspectJ] extiende de Java proporcionando una nueva clase de módulos llamados aspectos. Los aspectos cortan las clases, interfaces y a otros aspectos mejorando la separación de competencias y haciendo posible localizar en forma limpia los conceptos de diseño. El tejedor de aspectos de AspectJ realiza la composición de aspectos y clases, y compila la aplicación, generando código objeto conforme a la especificación Java byte-code, ejecutable por la JVM (Maquina Virtual de Java). Los principales elementos estructurales de AspectJ son: “join-points”, “pointcuts”, “advices” e “introductions”.

AspectJ carece de mecanismos para detectar y resolver posibles conflictos entre aspectos. En cuanto a la detección, el tejedor-compilador “ajc” no advierte las posibles situaciones conflictivas en forma automática y procede en forma normal, tal como si no existiera ningún conflicto. En la Figura 1 los aspectos Logging y Security están en conflicto, ya que ambos ejecutan sus avisos cuando se invoca el método *getISBN()* de la clase *Book*, pero esta situación no es advertida por el tejedor. Es tarea y responsabilidad del programador llevar el control de los conflictos que se generan.

```
aspect Logging                          aspect Security
   {                                       {
   pointcut open():                        pointcut consult():
            call(void Book.getISBN());              call(void Book.getISBN());
   before(): open()                        before(): consult()
      {                                       {
      …                                       …
      }                                       }
   }                                       }
```

Figura 1: Conflicto entre dos aspectos.

Para la resolución de conflictos, AspectJ proporciona un esquema muy restringido basado en precedencias (prioridades u orden). Múltiples avisos pueden aplicarse al mismo join-point, en tales casos, el orden de ejecución de los avisos esta basado en la precedencia de los aspectos. Por defecto, en AspectJ no existe ningún orden definido, por lo que, si se precisa ejecutar los avisos en un determinado orden, es necesario especificarlo con las cláusulas *declare precedence*. Por ejemplo: **declare precedence: Logging, Security;** (la semántica es que si el aspecto Logging precede al aspecto Security, entonces los avisos del aspecto Logging tienen mayor prioridad y se ejecutan antes que los avisos del aspecto Security).

3. Prototipo IntAOP: el Contexto del AAC

El prototipo IntAOP es un entorno de programación que brinda facilidades para codificar, compilar y ejecutar aplicaciones desarrolladas en AspectJ. El esquema funcional de IntAOP es el típico, el desarrollador crea un proyecto al cual asocia las unidades de programa (clases, interfaces y aspectos) que codifica. El concepto de proyecto permite manejar todas las unidades individuales como un todo, al abrir, guardar, compilar o ejecutar un proyecto. A medida que se codifican las unidades, de manera transparente para el usuario se realiza un análisis léxico y sintáctico parcial que permite interactuar con el desarrollador, simplificando la tarea de codificación y disminuyendo los posteriores errores de tejido y compilación. Para esto, internamente el proyecto se comporta como un contenedor de instancias de unidades. Las unidades pueden ser clases, interfaces y aspectos. En el caso particular de un aspecto, el objeto (ICAspect) describe el nombre y cada uno de sus elementos (pointcuts, advice, inter-types, métodos, atributos, etc.).

La interacción entre el desarrollador e IntAOP se efectúa mediante un conjunto de componentes de interfaz gráfica que lo convierten en un entorno sencillo y amigable. En la Figura 2 se puede observar el esquema visual general de la herramienta.

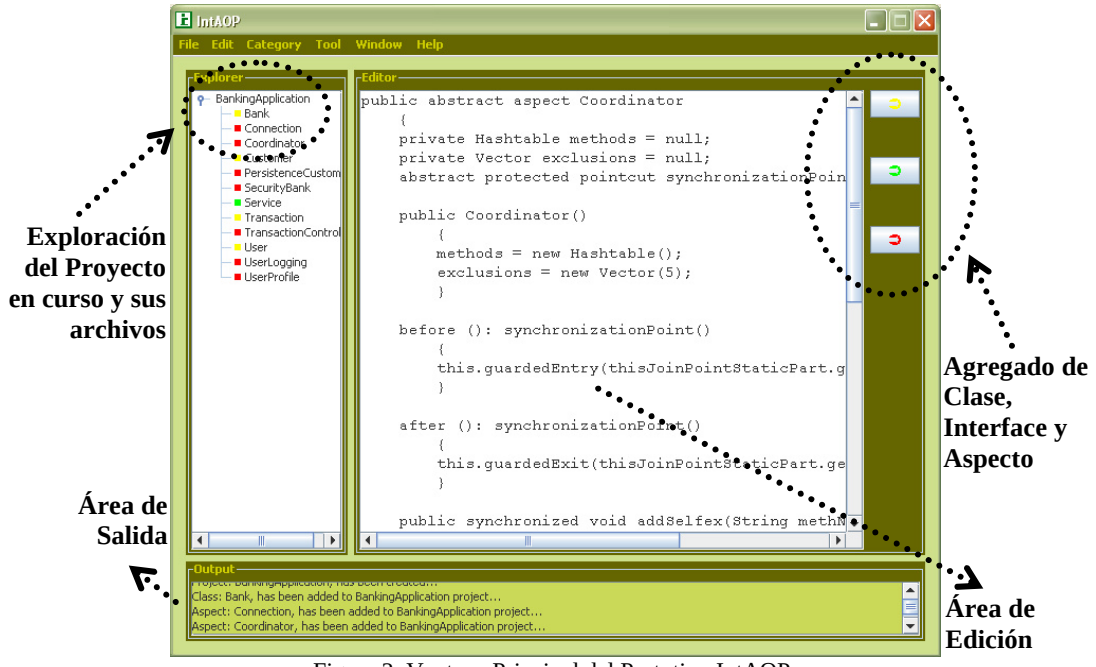

Figura 2: Ventana Principal del Prototipo IntAOP.

En la vista que se encuentra a la izquierda de la ventana (denominada “Explorer”) se disponen jerárquicamente las unidades de programa del proyecto en curso. Cada unidad se distingue por un nombre y un color que indica su tipo (clase: amarillo, aspecto: rojo o interface: verde). La vista del área de edición de código

(denominada “Editor”) ocupa una dimensión mayor dispuesta en la zona central de la ventana. Dicha vista cuenta con el área de edición propiamente dicha y 3 botones. Los mismos permiten salvar (asociar al proyecto actual) la unidad de programa editada. La vista horizontal que se encuentra en la parte inferior (denominada “Output”) constituye el área de salida, en la cual se visualizan los mensajes enviados al desarrollador (tareas realizadas, errores de acción, edición y compilación, etc.). En este estadio, IntAOP es una herramienta que no soporta el tratamiento de conflictos entre aspectos. En las secciones siguientes se describe la estrategia por la cual se le ha adicionado a IntAOP soporte automático para la administración de conflictos mediante un agente de software.

4. Roles, Responsabilidades y Servicios del AAC

La definición de los roles de un agente permite establecer sus responsabilidades y los servicios que debe ofrecer. En el caso particular del AAC se identifican tres roles: Monitoreo de Conflictos, Asistente de Resolución de Conflictos y Consejero de Resolución. A continuación se analiza cada uno de los roles.

Rol: Monitoreo de Conflictos

- Análisis: AAC debe estar permanentemente alerta ante la posibilidad de la ocurrencia de un conflicto entre aspectos. Aquí se plantea la creación de aspectos como un evento que se produce en el ambiente (IntAOP) y que el agente debe percibir. Por cada proyecto AAC debe mantener una lista de aspectos en su memoria.

- Responsabilidades: Mantener una lista de aspectos por proyecto. Detectar cuando se origina un conflicto entre dos aspectos automáticamente.

- Servicios: Brindar la información del conflicto ocurrido al desarrollador.

Rol: Asistente de Resolución de Conflictos

- Análisis: La acción a realizar cuando un conflicto es detectado debe garantizar el correcto funcionamiento de la aplicación. En algunos casos será necesario indicar un orden específico de ejecución de los aspectos involucrados. En otros casos, un conflicto puede requerir que uno o ambos aspectos no sean activados. AAC proporcionará seis métodos de resolución de conflictos:

  En Orden: Establece un orden de ejecución para los aspectos en conflicto.

  Orden Inverso: Establece un orden inverso de ejecución para los aspectos en conflicto.

  Opcional: Establece la ejecución condicional de los aspectos en conflicto.

  Exclusión: Establece la exclusión de uno de los aspectos en conflicto. No se ejecutará uno de los aspectos conflictivos.

  Nulidad: Establece la anulación de los aspectos en conflictos. No se ejecutará ninguno de los aspectos conflictivos.

  Dependiente del Contexto: Es una combinación entre *En Orden* y *Opcional*. El orden de ejecución se estable según una condición.

- Responsabilidades: Facilitar la asociación de un método de resolución al conflicto detectado, evitando que el desarrollador deba modificar el código de los aspectos involucrados.

- Servicios: Aplicar automáticamente el método de resolución sobre los aspectos en conflicto.

Rol: Consejero de Resolución

- Análisis: Se puede generar un consejo o recomendación que ayude a resolver un conflicto usando información histórica. Es decir, buscando en los métodos de resolución de conflictos aplicados anteriormente en casos similares. Para esto se define como mecanismo la clasificación de los aspectos específicos en categorías generales. A priori se define un conjunto de categorías genéricas aspectuales: Logging, Persistence, Authentification, Activity Record, etc., estas categorías aducen a cierto tipo de funcionalidad transversal. Cuando se implementa un proyecto en particular, es necesario que los aspectos específicos sean asociados a una categoría genérica existente. Esta vinculación no implica ninguna modificación de la lógica o código del aspecto. La categorización de aspectos permite analizar los métodos de resolución aplicados a nivel categoría. Esto implica que AAC debe mantener información de los métodos de resolución aplicados a largo plazo. Esta información debe persistir a lo largo del desarrollo de los distintos proyectos. Complementariamente y para una mejor orientación del desarrollador es posible analizar conflictos idénticos y conflictos similares. Dos conflictos

son idénticos si coinciden en categoría genérica, corte y aviso. Mientras que dos conflictos son similares solo si coinciden en categorías genéricas.

- Responsabilidades: Mantener un historial de los métodos de resolución de conflictos aplicados a nivel categoría.

- Servicios: Cuando se detecta un conflicto entre dos aspectos particulares generar una recomendación basada en las categorías genéricas involucradas que incluya el análisis de conflictos idénticos y conflictos similares.

5. Diseño e Implementación del AAC

Jennings y Wooldridge definen el término agente como "un programa autocontenido capaz de controlar su proceso de toma de decisiones y de actuar, basado en la percepción de su ambiente, en persecución de uno o varios objetivos" [Jennings98]. Russell y Norvig proponen el siguiente concepto: "Un agente puede ser visto como algo que percibe su ambiente a través de sensores y actúa contra este ambiente a través de efectores" [Russell95]. Desde estas perspectivas se concibieron el diseño e implementación del agente AAC como un agente reactivo con estado interno para la administración de conflictos entre aspectos en AspectJ. En la próxima sección, el diseño abstracto y la implementación del AAC se describen con más detalle para su mejor comprensión.

5.1. Diseño Abstracto del AAC

El esquema abstracto presentado en la Figura 3, representa de manera simplificada las interacciones de AAC con su ambiente o entorno. AAC permanece activo durante todo el desarrollo de un proyecto, esperando que los eventos que son de su interés ocurran y en consecuencia provoquen su reacción. En particular, el evento externo que concierne al AAC es la codificación de un aspecto. Cuando un aspecto es codificado y asociado al proyecto en curso (como objeto ICAspect), AAC reacciona. El sensor es un elemento de la interfaz gráfica. Primero se actualiza la Tabla de Aspectos que el agente mantiene en su memoria. Para ello, lo único que el agente requiere es la instancia de ICAspect que se ha generado.

En forma automática, al actualizarse la Tabla de Aspectos, AAC razona e infiera. Esto significa que analizará si el aspecto percibido plantea un conflicto con los aspectos previamente almacenados en su memoria. Si AAC infiere que se ha producido un conflicto, crea una instancia de ICConflict. ICConflict es una entidad que será devuelta por el agente al ambiente y que contiene toda la información referente a la interferencia. La acción que a continuación realiza AAC es buscar en su historial de resoluciones de conflictos aplicadas una recomendación o consejo. De acuerdo a las categorías genéricas de los aspectos en conflicto genera una estadística de las resoluciones aplicadas en el pasado. La estadística generada es encapsulada en una instancia de ICAgentRecommendation, que también es devuelta al ambiente.

Las percepciones son almacenadas en distinta forma por ACC, aquí se distingue un conjunto que es manejado como memoria a corto plazo y otro conjunto que es manejado como memoria a largo plazo. La detección de un conflicto es una operación que requiere la lista de los aspectos particulares del proyecto en curso. Esta lista de aspectos no es válida para otros proyectos. Esta información es la que se ha denominado memoria a corto plazo del ACC. Cada vez que se resuelve un conflicto entre dos aspectos, la resolución específica aplicada es percibida por el agente que la almacena a "largo plazo". De esta forma, las resoluciones aplicadas van conformando un historial que se utilizará también en futuros proyectos para generar las recomendaciones o consejos.

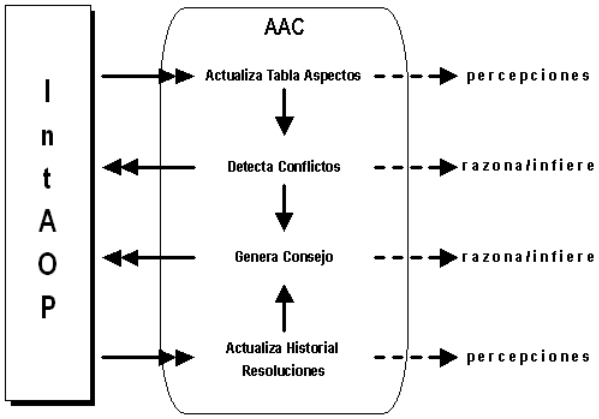

Figura 3: Diseño Abstracto del AAC.

Los tres roles del agente con sus responsabilidades y servicios es materializada en la interfaz que se ha diseñado especialmente para que el desarrollador interactúe con el AAC. Esta se visualiza en la Figura 4. El formulario se divide en tres secciones principales: (B) la vista denominada “Actual Conflict Description” que informa las características del conflicto planteado (los aspectos, categorías y pointcuts, y el componente funcional afectado). Esta información se recupera de la instancia ICConflict generada por AAC; (A) la vista denominada “Summary of Resolute Conflicts” que presenta la información de recomendación (los métodos aplicados para la resolución de conflictos idénticos y conflictos similares) que se obtiene de la instancia ICAgentRecommendation creada por ACC. La recomendación puede visualizarse en forma tabular (como muestra la Figura 4.a en dicha vista con la pestaña “Conflicts Table”) o gráfica (como muestra la Figura 4.b, si se hubiera seleccionado la pestaña “Conflicts Graphics Representation”). Por último, (C) la vista denominada “Conflicts Resolution” que proporciona asistencia automática para la resolución del conflicto generado, según los seis métodos de resolución vistos en la Sección 4.

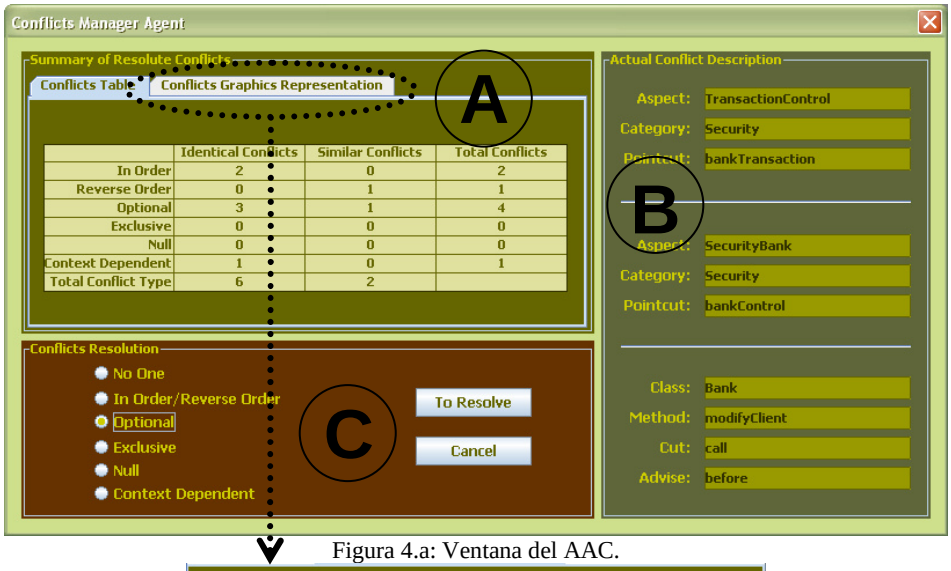

Figura 4.a: Ventana del AAC.

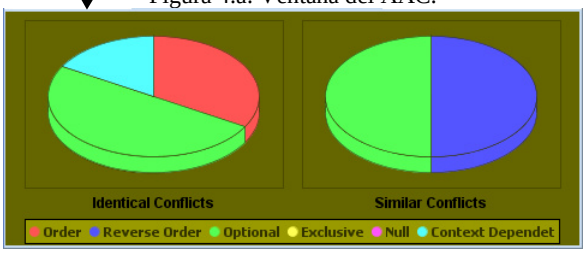

Figura 4.b: Ventana del AAC, utilizando Graphs para visualizar.

5.2. Implementación del AAC

La implementación del AAC responde a un esquema híbrido, dado que combina el enfoque declarativo con el enfoque orientado a objetos. Así AAC se compone de la clase ICAgent y el motor de inferencia Jess [Friedman-Hill03] (Figura 5). ICAgent es un hilo que se ejecuta en forma concurrente, e interactúa integrada y sincronizadamente con la interfaz gráfica de IntAOP y con Jess. Los métodos principales de ICAgent tienen el siguiente comportamiento:

- **ICAgent()**: crea e inicializa el motor de inferencia de Jess (atributo engine de tipo Rete).

- **event(ICAspect)**: el método es invocado cuando se ha creado una instancia de ICAspect en IntAOP. Esta instancia es mapeada como un conjunto de hechos pointcut y aspcat en la memoria de trabajo (MT) de Jess.

- **hasDetectConflict()**: este método es invocado cuando la regla conflict (detecta conflictos entre dos aspectos) de la máquina de inferencia Jess fue disparada o activada. Se obtiene la información de la MT (hecho conf) y se crea una instancia de ICConflict.

- **initialization()**: carga las plantillas, reglas e historial de resoluciones en la MT de Jess.

- **recommendation()**: invocado luego que se detectó un conflicto entre aspectos. Realiza consultas a la base de hechos para obtener las resoluciones de conflictos entre las mismas categorías. Analiza conflictos idénticos y conflictos similares. Crea una instancia de ICAgentRecommendation.

- **updateResolution()**: método que es invocado cuando se resuelve un conflicto entre aspectos. El método de resolución aplicado es representado mediante un objeto ICSolve. Este es mapeado como hechos resol e ingresado en la MT de Jess. La regla mapResol crea los hechos rescat automáticamente.

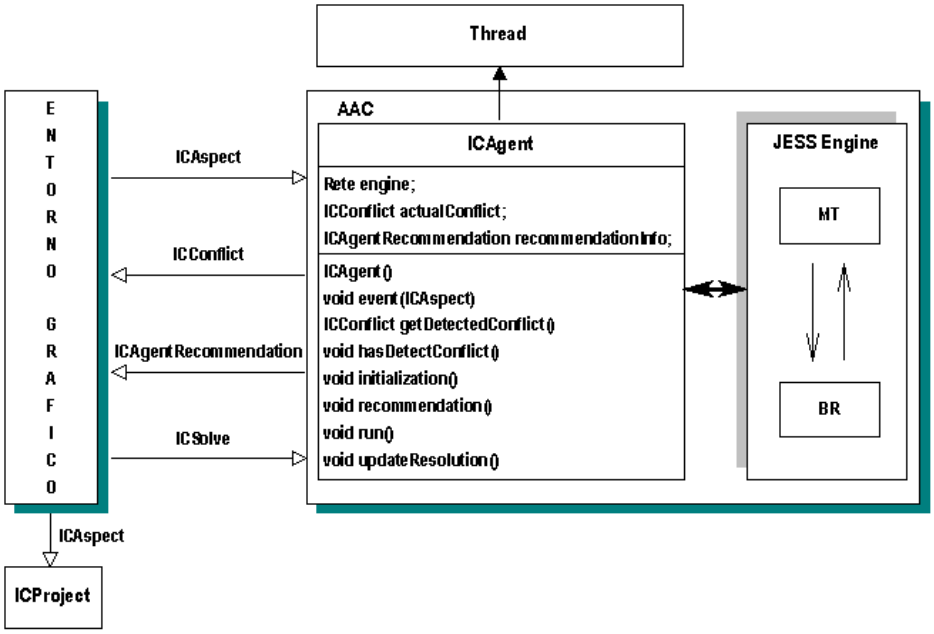

Figura 5: Implementación del AAC.

Respecto de la parte declarativa e inferencial, como se ha indicado se ha implementado íntegramente con JESS. Las plantillas de definición de los hechos más relevantes son:

```
aspcat : relaciona un aspecto con su categoría
genérica
   ida: identificación del aspecto
   idcat: categoría genérica del aspecto
(Logging, persistence, etc.)

pointcut : definición de un pointcut
   idp: identificación del pointcut
   ida: identificación del aspecto
   cut: corte primitivo (call, execution, etc.)
   advise: aviso (before, after, etc.)
   idc: clase afectada
   idm: método afectado

resol: resolución aplicada a un conflicto
  ida1: identificación de aspecto
  ida2: identificación de aspecto
  cut: corte
  advise: aviso
  res: método de resolución aplicado (In Order,
Reverse Order, Optional, Exclusive, Null o
Context Dependent)
```

```
rescat: resolución aplicada a nivel
categoría.
  idc1: categoría genérica
  idc2: categoría genérica
  cut: corte
  advise: aviso
  res: método de resolución aplicado

conf: representa un conflicto entre dos
aspectos
  ida1: identificación de aspecto
  ida2: identificación de aspecto
  cut: corte
  advise: aviso
  idc: clase afectada
  idm: método afectado
```

La Figura 6 presenta el código de la regla conflict. Esta regla detecta conflictos entre dos aspectos, básicamente analiza si dos pointcuts (p1 y p2) de dos aspectos diferentes (a1 y a2) coinciden en el aviso, corte (w y z) y join-points (c y m).

```
(defrule conflict
  (pointcut (idp ?p1)(ida ?a1)(advise ?w)(cut ?z)(idc ?c)(idm ?m))
  (pointcut (idp ?p2)(ida ?a2)(advise ?w)(cut ?z)(idc ?c)(idm ?m))
  (test (neq ?y1 ?y2))
  (not (conf(ida1 ?y2)(ida2 ?y1)(advise ?w)(cut ?z)(idc ?c)(idm ?m)))
=>
  (assert (conf(ida1 ?y1)(ida2 ?y2)(advise ?w)(cut ?z)(idc ?c)(idm ?m))))
```

Figura 6: Regla conflict.

La Figura 7 presenta el código de la regla mapResol, que se dispara cada vez que se agrega un hecho resol a MT. Un hecho resol representa una resolución aplicada a un conflicto entre dos aspectos (a1 y a2), a continuación aparea las categorías de dichos aspectos (ic1 e ic2) y crea el hecho rescat, que mapea la resolución a nivel categoría genérica.

```
 (defrule mapResol
  (resol(ida1 ?a1)(ida2 ?a2)(cut ?c)(advise ?ad)(res ?r))
  (aspcat(ida ?a1)(idcat ?ic1))
  (aspcat(ida ?a2)(idcat ?ic2))
=>
  (assert (rescat (idc1 ?ic1)(idc2 ?ic2)(cut ?c)(advise ?ad)(res ?r))))
```

Figura 7: Regla mapResol.

La generación de la recomendación se implementa a través de consultas. Por ejemplo, en la Figura 8 se presenta la consulta para identificar resoluciones a conflictos idénticos, donde las variables c1 y c2 representan a las categorías genéricas, la variable *c* representa al corte primitivo y la variable *a* representa al aviso:

```
(defquery search-identical
       (declare (variables ?c1 ?c2 ?c ?a))
       (rescat (idc1 ?c1)(idc2 ?c2)(cut ?c)(advise ?a)(res ?r)))
```

Figura 8: Recomendación para conflictos idénticos.

6. Discusión y Trabajo Futuro

AAC maneja con habilidad y capacidad las situaciones conflictivas planteadas por dos aspectos. La limitación del ACC estaría dada en principio cuando un conflicto es "replanteado". Es decir, un conflicto entre dos aspectos detectado y resuelto se "replantea" cuando un tercer aspecto se asocia al mismo componente funcional. Según el enfoque presentado, ACC identificará tres conflictos, para lo cual suministrará tratamiento por separado a cada uno, resultando inadecuado ya que se trata de un único conflicto en el cual participan tres aspectos. Esto significa que un enfoque más completo sería aquel que contemple las situaciones de "replanteo de conflictos". Este nuevo escenario, requiere que el agente posea mayores capacidades de inferencia tanto para la detección como para la recomendación. Precisamente, en este sentido se orienta el trabajo futuro, en particular se analiza la posibilidad de emplear una arquitectura de mayor complejidad que aborde una solución desde un enfoque multi-agente. En este nuevo esquema distintos agentes cumplen distintos roles, responsabilidades y servicios y se comunican para administrar conflictos entre varios aspectos.

7. Trabajos Relacionados

En esta sección se indican algunos trabajos y aportes relacionados específicamente con la problemática de detección y resolución de conflictos entre aspectos.

Un enfoque para detectar y analizar las interferencias causadas por las capacidades que AspectJ proporciona para modificar la estructura jerárquica de las clases (declaración: *declare parents*) e introducir nuevos miembros a las clases (métodos y atributos), fue presentado por [Storzer03]. Este trabajo esta basado en técnicas de análisis de programas tradicionales y sólo esta orientado a la detección de los conflictos de tipo clase-aspecto y no ofrece cobertura a los conflictos de tipo aspecto-aspecto.

El modelo de precedencia de AspectJ (secuencial), utilizado para establecer el orden de ejecución de los avisos, cuando están asociados al mismo punto de unión, es mejorado y optimizado en [Yu04]. La representación del modelo en un grafo de precedencias, conduce a un modelo de precedencia concurrente. Este trabajo es sólo una propuesta de mejora para el mecanismo de resolución de conflictos específica para AspectJ.

LogicAJ [Roots05] provee análisis de las interferencias aspecto-aspecto para AspectJ que incluye capacidades para: (a) identificar una clase bien definida de interferencias, (b) determinar el orden de ejecución libre de interferencia, (c) determinar el algoritmo de tejido más conveniente para un conjunto de aspectos dado. El análisis de interferencias es independiente de los programas base a los que los aspectos se refieren (sólo los aspectos son necesarios para el análisis) e independiente de anotaciones especiales del analizador de aspectos. Este trabajo sólo enfoca el problema de la detección de conflictos y es específico para AspectJ.

Astor [Casas05] es un prototipo que propone una serie de mecanismos y estrategias para mejorar el tratamiento de conflictos en AspectJ. Los mismos se soportan mediante la adición de un componente 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 y

la resolución se efectúa siguiendo las directrices de una taxonomía que proporciona seis categorías de resolución. La implementación del prototipo esta basada en el pre-procesamiento de código AspectJ, siendo además éste el único requisito para su uso. Las limitaciones de Astor están dadas por las restricciones de AspectJ.

"Programme Slicing" es una técnica que apunta a la extracción de elementos de programa relacionados a una computación en particular. Este enfoque es propuesto para crear modelos que puedan ser utilizados para analizar las interacciones entre aspectos, ya que puede reducir las partes de código que se necesitan analizar para entender los efectos de cada aspecto [Monga03]. Pero debido a la inadecuada información generada por los modelos no es lo suficientemente capaz de extraer información significativa. Este enfoque requiere de información adicional para posibilitar que el análisis de conflictos sea más apropiado. Este trabajo sólo da cobertura al problema de la detección de conflictos.

Un método para validar formalmente el orden de precedencia entre aspectos que comparten un mismo punto de unión, es presentado en [Pawlak05]. Este trabajo introduce un lenguaje simple, CompAr, en el cual el usuario expresa de manera abstracta el efecto de los avisos que es importante en la interacción entre aspectos y las propiedades (restricciones) que deben ser verdaderas luego de la ejecución de un aviso. El compilador CompAr detecta automáticamente los conflictos y chequea que dado un orden de avisos no invalide una propiedad de un aviso "around". La mayor contribución de este trabajo es el alto nivel de abstracción que el lenguaje ofrece para definiciones genéricas de aspectos. Su principal limitación es que sólo es aplicable a avisos de tipo "around" y la resolución esta basada en el esquema de orden.

Los Filtros de Composición se utilizan para analizar interacciones en [Durr05], este enfoque se basa en el modelo formal para detectar conflictos semánticos entre aspectos. En este trabajo se detecta cuando un aspecto precede la ejecución de otro aspecto, y se chequea que una propiedad especificada de traza sea realizada por un aspecto. El mecanismo cuenta con la definición de flujos de trabajo ("workflows") los cuales definen como los filtros interactúan. Sin embargo, es complicado definir un flujo de trabajo determinado y hacer seguro que éste será válido y usable en cualquier caso. Además, el enfoque elegido hace que la implementación sea bastante difícil de leer y mantener.

El uso de reglas como estrategia o mecanismo para manejar conflictos ha sido propuesto recientemente en varios trabajos, como ser [Kessler06]. Los autores presentan una exploración inicial basada en programación lógica donde los hechos y reglas son definidos para la detección de interacciones en Reflex. El valor de este trabajo no se puede apreciar por tratarse de una exploración, sin resultados específicos, al momento de la publicación. Sin embargo, dado que se proyecta sobre Reflex, las posibilidades de resolución estarán limitadas a las soportadas por la herramienta POA.

En [Nagy06] se propone un enfoque declarativo basado en restricciones para especificar la composición de aspectos ante puntos de unión compartidos ("shared join-points"). Las restricciones pueden ser de orden o control, y pueden aplicarse en forma independiente y no combinarse. La implementación de este modelo requiere la extensión del lenguaje POA en varias formas: constructores de puntos de unión, constructores de avisos, sentencias de declaración, etc.

8. Conclusiones

El tratamiento de conflictos entre aspectos es una tarea que debe ser realizada para garantizar el correcto funcionamiento del software. Esta puede ser compleja y costosa si es manual o simple y rápida si es automática.

Este trabajo ha presentado un nuevo enfoque para administrar conflictos entre aspectos en AspectJ basado en agentes de software. Básicamente se ha diseñado e implementado un agente reactivo para monitorizar los conflictos sucedidos, asistir en la resolución de los mismos y recomendar métodos de resolución. Estas funciones son realizadas totalmente por el agente facilitando el tratamiento de conflictos

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

Referencias

[Casas05] Casas S., Marcos C., Vanoli V., Reinaga H., Saldivia C., Pryor J., Sierpe L. "ASTOR: Un Prototipo para la Administración de Conflictos en AspectJ", XIII ECC - JCC, Chile. 2005.

[Durr05] Durr P., Staijen T., Bergmans L., Aksit M. "Reasoning about semantic conflicts between aspects". In K. Gybels, M. D'Hondt, I. Nagy, and R. Douence, editors, 2nd European Interactive Workshop on Aspects in Software (EIWAS'05). Vrije Universiteit Brussel. Brussels, Belgium. Sept. 2005.

[Friedman-Hill03] Friedman-Hill E. "Jess in Action". Manning Publications, ISBN 1-930110-89-8. 2003.

[Gal02] Gal A., Scroder-Preikschat W., Spinczyk O. “AspectC++: Language Proposal and Prototype Implementation”. ACM International Conference Proceeding Series Proceedings of the Fortieth International Conference on Tools Pacific. Vol.10. Australia. 2002.

[HAspectC] Homepage de AspectC: http://www.cs.ubc.ca/labs/spl/projects/aspectc.html

[HAspectC++] Homepage de AspectC++: http://www.aspectc.org/

[HAspectJ] Homepage de AspectJ Xerox, PARC, USA http://aspectj.org/.

[HAspectR] Homepage de AspectR: http://aspectr.sourceforge.net/

[HAspectWerkz] Homepage de AspectWerkz, http://aspectwerkz.codehaus.org/

[HHyper/JTM] Homepage de Hyper/JTM, IBM T.J. Watson Research Centre: http://www.alphaworks.ibm.com/tech/hyperj

[Hirschfeld01] Hirschfeld R. “AspectS - AOP with Squeak”. In Proceedings of OOPSLA. Workshop on Advanced Separation of Concerns in Object-Oriented System. USA. 2001.

[HJboss] Homepage de Jboss: http://www.jboss.org/products/aop

[HPythius] Homepage de Pythius: http://sourceforge.net/projects/pythius/

[Jennings98] Jennings N., Wooldridge M. “Agent Technology - Foundations, Applications, and Markets”. Springer-UNICOM. 1998.

[Kessler06] Kessler B., Tanter E. “Analyzing Interactions of Structural Aspects”. Workshop IAD in 20th ECOOP. France, 2006.

[Kiczales97] Kiczales G., Lamping J., Menhdhekar A., Maeda C., Lopes C., Loingtier J-M., Irwin J. “Aspect-Oriented Programming”. In Proceedings ECOOP, LNCS 1241, pages 220–242. Finland. Springer-Verlag. June 1997.

[Kiczales01] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G. “An Overview of AspectJ”. In Proc. of the 15th ECOOP. Pp. 327-357. Budapest, Hungary. 2001.

[Monga03] Monga M., Beltagui F., Blair L. “Investigating Feature Interactions by Exploiting Aspect Oriented Programming”. Technical Report N comp-002-2003, Lancaster University, England. http://www.com.lancs.ac.uk/computing/aop/Publications.php

[Nagy06] Nagy I., Bergmans L., Aksit M. “Composing Aspects at Shared Join Point”. Workshop IAD in 20th. ECOOP. France, 2006.

[Pawlak05] Pawlak R., Duchien L., Seinturier L. “CompAr: Ensuring safe around advice composition”. FMOODS 2005, vol. 3535 of LNCS, pp 163–178, 2005.

[Piveta01] Piveta E., Zancanela L. “Aurelia: Aspect oriented programming using reflective approach”. Workshop on Advanced Separation of Concers ECOOP. 2001.

[Pryor02] Pryor J., Diaz Pace A., Campo M. “Reflection on Separation of Concerns”. RITA. Vol 10, Num 2. 2002.

[Roots05] Roots. “LogicAJ – A Uniformly Generic and Interference-Aware Aspect Language”; http://roots.ia.uni-bonn.de/researh/logicaj/, 2005.

[Russell95] Russell S., Norvig P. “Artificial Intelligence: A Modern Approach”. Prentice Hall. 1995

[Schult03] Schult W., Tröger P., “Loom.NET - an Aspect Weaving Tool,” Workshop on Aspect-Oriented Programming, ECOOP'03, Darmstadt, 2003. Homepage: http://www.dcl.hpi.uni-potsdam.de/research/loom/

[Storzer03] Storzer M., Krinkle J. “Interference Analysis for AspectJ”. FOAL: Foundations of Aspect-Oriented Languages, USA, 2003.

[Yu04] Yu Y., Kienzle J. “Towards an Efficient Aspect Precedence Model”. Proceeding of the DAW, pp 156-167, England, 2004.