Contratos XML: un enfoque independiente para la detección y resolución de interacciones aspectuales.

Sandra Casas[1] y Claudia Marcos[2]

[1]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.

[2]ISISTAN Research Institute. Facultad de Ciencias Exactas. UNICEN
Paraje Arroyo Seco. CP 7000. Tandil. Buenos Aires. Argentina
Tel/Fax: + 54–2293–440362/3.

lis@uarg.unpa.edu.ar, cmarcos@exa.unicen.edu.ar

Resumen: *Este trabajo propone aplicar las ideas del Diseño por Contratos como enfoque para el manejo de interacciones aspectuales. A partir de la propuesta teórica AICs (Aspects Integration Contracts), se proporciona un formato e implementación de los contratos basada en XML. El soporte XML de los contratos facilita la detección y resolución de interferencias y conflictos. La noción de AICs se extiende con los contratos de composición que siguen la línea de formato e implementación propuesta. El enfoque es implementado en un prototipo denominado XAC (XML- Aspect-Contracts). El trabajo se presenta en forma preliminar ya que deja abierta futuras líneas de trabajo que tienen por objeto integrar los contratos XML con la posterior implementación del código de los aspectos o la ejecución de los aspectos en lenguajes POA específicos.*

1. Introducción

La Programación Orientada a Aspectos [Kiczales et all., 1.997] (POA) es un nuevo paradigma para el desarrollo de software que proporciona mecanismos y abstracciones para la implementación de requerimientos no funcionales y transversales ("crosscuting concerns"), de manera separada y aislada de la funcionalidad base. Es decir, la orientación a aspectos, es una técnica que permite aplicar el principio de Separación de Concerns [Dijkstra, 1976] [Hürsch and Lopes, 1995] y de esta forma, superar los problemas ocasionados por el código mezclado y diseminado.

La POA propone implementar los requerimientos no funcionales en unidades individuales denominadas aspectos. Los aspectos están equipados de mecanismos especiales (pointcuts, join-points y advices) que permiten que un proceso posterior de composición (tejido) [Piveta and Zancanella, 2003] genere la aplicación final. En resumen, los requerimientos transversales se identifican, diseñan e implementan como aspectos y el proceso de tejido compone la funcionalidad base con los aspectos.

Sin embargo, la composición de los aspectos con la funcionalidad base realizada por el tejedor, tiene como efecto colateral la generación de interacciones aspectuales que potencialmente pueden ocasionar comportamientos inciertos e impredecibles durante la ejecución del software. Ciertos tipos de interacciones hacen que el software pierda confiabilidad y estabilidad. Se distinguen dos tipos de interacciones: componente-aspecto a las que denominamos *interferencias* y las interacciones aspecto-aspecto, que denominamos *conflictos*.

El Diseño por Contrato (DbC) [Meyer, 1992] es una técnica poderosa para la construcción de software confiable y estable. Los elementos claves del DbC son las aserciones (precondiciones, postcondiciones e invariantes). La idea original es que las aserciones sean utilizadas para establecer bajo que condiciones debe invocarse un método y a partir de ello establecer cual es el resultado de su ejecución. La definición de estas condiciones se denomina contratos. Los contratos deben ser explícitos y formar parte del software en sí mismo.

El presente trabajo surge de la premisa que sí el DbC es una técnica para diseñar y construir software confiable y estable, entónces las ideas del DbC pueden ser aplicadas para resolver el problema de las interacciones aspectuales. A partir de la noción de contratos AICs [Lagaisse et all., 2004], se propone: (i) una implementación y formato independiente para los contratos respecto de los lenguajes de programación; (ii) detección de interferencias y conflictos a partir de los contratos; (iii) completar la noción de contratos AICs con los contratos de composición/resolución y de esta forma dar soporte a la resolución de conflictos. La herramienta XAC (XML-Aspect-Contract) encarna en parte estas nuevas ideas.

Este trabajo se estructura de la siguiente manera: en la Sección 2 se presentan los trabajos relacionados y un análisis de sus principales limitaciones; en la Sección 2 se describen el soporte XML para contratos AICs; en la Sección 4 se definen los procesos de detección y resolución de interacciones aspectuales; en la Sección 5 se presenta el prototipo XAC y en la Sección 6 se exponen las conclusiones y el trabajo futuro.

2. Trabajos Relacionados

En los últimos años, la investigación en la POA ha avanzado en la temática de interacciones aspectuales y ha producido diversos resultados, que si bien significan un avance en el estudio aún distan de ser una solución definitiva al problema.

Uno de los primeros trabajos relacionado directamente con la detección y resolución de conflictos aparentemente ha sido [Duoence et all., 2002]. Los autores sostienen que el tratamiento de los conflictos entre aspectos debe realizarse en forma separada a la definición de los aspectos. Se propone un modelo de tres-fases para la programación de múltiples aspectos: (i) Programación: Los aspectos que son parte de una aplicación son escritos independientemente, posiblemente por diferentes programadores; (ii) Análisis de conflictos: Una herramienta automática detecta las interacciones entre aspectos y retorna los resultados al programador; (iii) Resolución de conflictos. El programador resuelve las interacciones usando un lenguaje de composición dedicado. El resultado de esta fase puede ser chequeada nuevamente en la fase (ii). La solución esta basada en un framework genérico para POA, que se caracteriza por un lenguaje de cortes cruzados muy expresivo, análisis de conflictos estáticos y un soporte lingüístico para la resolución de conflictos. Las posibilidades de resolución de conflictos son limitadas.

Un enfoque para detectar y analizar las interferencias causadas por las capacidades que AspectJ [Kiczales et all., 2001] proporciona para modificar la estructura jerárquica de las clases (declaración declare parents) e introducir nuevos miembros a las clases (métodos y atributos), fue presentado en [Storzer and Krienkle, 2003]. Este trabajo esta basado en técnicas de análisis de programas tradicionales y sólo esta orientado a la detección de interferencias.

El modelo de precedencia de AspectJ (secuencial), utilizado para establecer el orden de ejecución de los avisos, cuando están asociados al mismo join-point, es mejorado y optimizado en [Yu and Kienzle, 2004]. 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 [ROOTS, 2005] provee análisis de las interacciones aspecto-aspecto para AspectJ que incluye capacidades para: (a) identificar una clase bien definida de interacciones, (b) determinar el orden de ejecución libre de conflicto (c) determinar el algoritmo de tejido más conveniente para un conjunto de aspectos dado. El análisis de conflictos 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 resuelve el problema de la detección de conflictos.

Astor [Casas et all., 2005] es una herramienta 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 analizar los conflictos entre aspectos, ya que puede reducir las partes de código que se necesitan analizar para entender los efectos de cada aspecto. [Monga et all., 2003] Este trabajo sólo da cobertura al problema de la detección.

Un trabajo muy interesante es Reflex [Tanter and Noye, 2005], 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: (i) una capa de transformaciones encargada del tejido básico con soporte para la modificación estructural y de comportamiento de programas bases; (ii) una capa de composición para la detección y resolución de interacciones (conflictos) y (iii) una capa de lenguaje, para la definición modular del lenguaje de aspectos. La detección de interacciones sigue el esquema propuesto por [Duoence et all., 2002] y se limita a una aproximación estática de la interacción de aspectos, sólo se detectan las interacciones que no ocurren en tiempo de ejecución. Las dos formas de resolver una interacción son: (i) elegir de las interacciones el aspecto que se va a aplicar en la ejecución y (ii) 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.

Un método para validar formalmente el orden de precedencia entre aspectos que comparten un mismo join-point, es presentado en [Pawlak et all., 2005]. Este trabajo introduce un lenguaje simple, CompAr, en el cual el usuario expresa el efecto de los avisos que es importante en la interacción entre aspectos y las propiedades que deben ser verdaderas luego de la ejecución de un aviso. El compilador CompAr chequea que dado un orden de avisos no invalide una propiedad de un aviso.

Los Filtros de Composición se utilizan para analizar interacciones en [Durr et all., 2005]. 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 uso de reglas como estrategia o mecanismo para manejar conflictos ha sido propuesto recientemente en varios trabajos, como ser [Kessler and Tanter, 2006]. Los autores presentan una exploración inicial basada en programación lógica donde los hechos y reglas se definen 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.

En [Nagy et all., 2006] se propone un enfoque declarativo basado en restricciones para especificar la composición de aspectos ante un join-point compartido ("shared join-points"). Las restricciones pueden ser de orden o control, y pueden aplicarse en forma independiente y no combinarse. La implementación de esta modelo

requiere la extensión del lenguaje POA en varios aspectos: constructores de join-point, constructores de avisos, sentencias de declaración, etc. En consecuencia, las restricciones del lenguaje POA impactan sobre el modelo.

En la Tabla 1 se comparan los aportes alcanzados entre los trabajos indicados.

Trabajo	Detección de conflictos	Resolución de conflictos	Interferencias	Específico de AspectJ
[Duoence et all., 2002]	X	X		
[Storzer and Krienkle, 2003]			X	X
[Yu and Kienzle, 2004]		X		X
[ROOTS, 2005]	X			X
[Casas et all., 2005]	X	X		X
[Monga et all., 2003]	X			
[Tanter and Noye, 2005]	X	X		
[Pawlak et all., 2005]		X		
[Kessler and Tanter, 2006]	X			
[Nagy et all., 2006]		X		
[Lagaisse et all., 2005]			X	

Tabla 1: Comparación de los aportes alcanzados.

Este análisis comparativo muestra lo mucho que aún resta por hacer en la búsqueda de soluciones generales e integrales, ya que éstos trabajos presentan las siguientes limitaciones:

(i) Manejo parcial de interacciones: ya que sólo abarcan el problema de conflictos o el problema de interferencias y no ambos problemas.

(ii) Tratamiento acotado de interacciones: ya que en general proponen sólo estrategias para el proceso de detección o para el proceso de resolución, pero no ambos.

(iii) Aplicables a dominios lingüísticos específicos: el fuerte predominio de AspectJ marca una tendencia en el desarrollo de mecanismos y estrategias de manejo de interacciones aspectuales.

A partir de estas premisas, se sientan las bases de nuestra propuesta:

(i) Plantear mecanismos y estrategias que detecten y resuelvan conflictos e interferencias.

(ii) Proponer soporte automático para los mecanismos y estrategias planteados.

(iii) Formular métodos independientes de herramientas y lenguajes de programación.

3. Soporte XML para AICs

Las ideas originales de DbC son extendidas a Diseño de Contratos Firmados (Design by Signed Contract) [Rausch, 2002] para dar un mejor soporte al desarrollo de sistemas basado en componentes. Los contratos firmados permiten que se especifique no sólo lo que un proveedor proporciona a su entorno, sino también lo que un cliente necesita de su entorno. Los contratos firmados garantizan que las necesidades del cliente estarán satisfechas por las correspondientes propiedades proporcionadas el proveedor. Aspect Integration Contracts (AICs) [Lagaisse et all., 2004] esta basado en las ideas del diseño de contratos firmados, y se formula como una propuesta teórica para manejar las interferencias semánticas que se pueden producir por la superposición entre aspectos y

componentes. Los autores identifican 4 tipos de interferencias a resolver: exposición de datos, modificación de datos, exposición del comportamiento y modificación del comportamiento. Los AICs básicamente plantean extender los contratos para especificar 3 nuevos elementos: (i) los requerimientos de los aspectos; (ii) la funcionalidad y efectos de los aspectos y; (iii) las interferencias permitidas. En este sentido los autores además incluyen la notación de la sintaxis de estos elementos. La sintaxis de la notación propuesta se transcribe en la Figura 1.

```
<AIC Component> ::= IContract ( <aspects> ) { <body> }
<aspect > ::= aspect_name | aspect_name,<aspects >
<body> ::= permits : <permission>*
<permission> ::= <action> <members > ;
<action> ::= expose | modify
<members> ::= <member> | <member>, <members>
<member> :: = own_method_signature | own_class_var
-----
<Requiriements Aspect> ::= Require From class_name { <body> }
<body> ::= require : <require_clause>*
<require_clause> ::= <action> <members> ;
<action> ::= expose | modify
<members> ::= <member> | <member>, <members>
<member > :: = method_signature | class_var_name
```

Figura 1: Notación propuesta para AICs.

En este sentido AIC es una propuesta teórica, ya que el DbC se basa en la premisa que los contratos puedan ser definidos (codificados) como parte de software y corroborados durante la ejecución del software. Este ha sido un factor crítico de la técnica, ya que sólo algunos lenguajes de programación han proporcionado soporte para el DbC [Meyer, 1991][Karaorman et all., 1998][Plath, 2000][Rangarajan, 2000][Kramer, 1998].

Nuestro enfoque plantea especificar e implementar los contratos en un formato independiente del lenguaje de programación, pero que no pierda el poder expresivo del contrato en si mismo. A la vez, este formato de representación debe poder ser utilizado para identificar las interferencias y los conflictos. Un método independiente consiste en representar los contratos AICs en formato XML. En la Figura 2 se exponen las DTD (Data Type Definition) correspondientes a la notación original.

```
<ELEMENT aic_component (aspect+)>
<ELEMENT aspect (permit+)>
<ELEMENT permit (expose*, modifiy*)>
<ELEMENT expose (#PCDATA)>
<ELEMENT modify (#PCDATA)>
<!ATTLIST aic_component component CDATA #REQUIRED>
<!ATTLIST aspect asp_name CDATA #REQUIRED>
-----
<ELEMENT aspect_requeriments (require+)>
<ELEMENT require (expose*, modifiy*)>
<ELEMENT expose (#PCDATA)>
<ELEMENT modify (#PCDATA)>
<!ATTLIST aspect_requeriments asp_name CDATA #REQUIRED>
<!ATTLIST require comp_name CDATA #REQUIRED>
```

Figura 2: DTD para AICs.

En la Figura 3 se presenta un ejemplo de la implementación de AICs en formato XML. En este ejemplo se han especificado los AICs contratos para el componente Account y el aspecto Logging. Los permisos que el componente Account otorga al aspecto Logging quedan definidos en los elementos *aic_component* y los requerimientos del aspecto Logging sobre el componente Account quedan definidos en los elementos *aspect_requerimients*.

```
<aic_component comp_name="Account">
 <aspect asp_name= "Logging">
  <permit>
   <expose>public void setBalance(float)</expose>
   <modify>public void setBalance(float)</modify>
  </permit>
 </aspect>
</aic_Component>
<aspect_requeriments asp_name="Logging">
<require comp_name="Account">>
  <expose>public void setBalance(float)</expose>
  <modify>public void setBalance(float)</modify>
  </require>
</aspect_requeriments>
```

Figura 3: Ejemplo de AICs en XML.

La definición de estos contratos puede ser realizada en una etapa de desarrollo anterior a la implementación, lo más conveniente sería durante el diseño avanzado del software. De esta forma, previamente a la implementación del código en un lenguaje POA específico pueden detectarse conflictos e interacciones en forma automática.

4. Detección y Resolución de Interacciones basadas en contratos XML

Una interacción aspectual ocurre cuando un componente de funcionalidad básica es afectado por la invocación implícita de un aspecto y tiene por resultado un comportamiento anómalo. El objetivo del aspecto es modificar el comportamiento de dicho componente funcional, en general para adicionar funcionalidad secundaria.

Ciertas interacciones aspectuales producen un comportamiento anómalo, si la ejecución del software que resulta no es la esperada, por lo tanto no es confiable y/o estable. Se distinguen dos tipos de interacciones aspectuales: interferencias y conflictos. Una interferencia ocurre cuando la interacción anómala es entre componente base-aspecto. Un conflicto ocurre cuando la interacción anómala es aspecto-aspecto. En principio este tipo de interacciones deben ser detectadas, y en consecuencia convenientemente resueltas.

A partir de la representación XML de contratos propuesta en la Figura 2 se establecen las condiciones que provocan que una interferencia o un conflicto ocurran (Tabla 2).

Una interferencia entre el componente C y el aspecto A, ocurre y puede ser detectada cuando:	Un conflicto entre el aspecto A y el aspecto B ocurre y puede ser detectado cuando:
(a) A tiene requerimientos sobre C y C no ha definido permisos para A; (b) A tiene requerimiento de modificación sobre C y en C se ha definido permiso de acceso para A.	(a) A y B tienen requerimientos del mismo tipo de permiso sobre el mismo componente; (b) A y B tienen requerimientos de distinto tipo de permiso sobre el mismo componente.

Tabla 2: Condiciones de ocurrencia de interferencias y conflictos.

Teniendo en cuenta las condiciones establecidas, la detección de interferencias y conflictos surge del análisis de los contratos. En la Figura 4 se presenta un simple ejemplo, la representación de los contratos XML del proyecto Banking Application. Se han especificado los contratos del componente AccountManager y los aspectos Logging, Persistence y Statistic.

```
<project_contract project="Banking Application">
 <aic_Component comp_name="AccountManager">
   <aspect asp_name= "Logging">
     <permit>
       <expose>public void addAccount(Account)</expose>
       <expose> public void removeAccount(Account)</expose> ⇦
     </permit>
  </aspect>
  <aspect asp_name= "Persistence">
     <permit>
       <expose>public void addAccount(Account)</expose>
       <modify> public void removeAccount(Account)</modify>
     </permit>
  </aspect>
 </aic_component>
 <aspect_requeriments asp_name="Logging">
    <require comp_name="AccountManager">
      <expose> public void addAccount(Account)</expose> ⇦
      <modify> public void removeAccount(Account)</modify> ⇦ ←
    </require>
 </aspect_requeriments>
 <aspect_requeriments asp_name ="Persistence" >
     <require comp_name ="AccountManager">
       <expose> public void addAccount(Account)</expose> ⇦
       <modify> public void removeAccount(Account)</modify> ⇦
     </require>
 </aspect_requeriments>
 <aspect_requeriments asp_name="Statistic">
     <require comp_name ="AccountManager">
       <expose> public void addAccount(Account)</expose> ⇦ ←
     </require>
 </aspect_requeriments>
</project_contract>
```

Figura 4: Contratos del Proyecto Banking Application.

Detección de interferencia y conflictos

Existen en esta representación de contratos dos conflictos (indicados con flechas ⇦): un conflicto entre los aspectos Logging y Persistence ya que en sus correspondientes contratos requieren acceso al miembro public void removeAccount (Account) del componente AccountManager. El otro conflicto se plantea entre los contratos de los aspectos Logging, Persistence y Statistic ya que definen acceso al miembro public void addAccount(Account) del componente AccountManager.

Existen dos interferencias no permitidas (indicadas con flechas 🡐). El aspecto Statistic define requerimientos sobre el componente AccountMananger, mientras que este componente no define ningún tipo de permiso para este aspecto. El componente AccountManager establece un permiso de exposición sobre el miembro public void removeAccount(Account) para el aspecto Logging, mientras que este aspecto requiere un acceso de modificación sobre dicho miembro.

Resolución de interferencias y conflictos

La resolución de interferencias implica que los contratos de los componentes y aspectos deben ser compatibles. Cuando éstos no lo son, la resolución implica debe ajustarse alguno de los contratos en interferencia.

La resolución de conflictos entre aspectos, conduce a la especificación de contratos de composición/resolución. Un contrato de composición/resolución definirá la forma (orden) en que dos o más aspectos que están en conflicto deberán ser compuestos. Por ejemplo en la Figura 5 se define el contrato de composición que resuelve el conflicto entre los aspectos Logging y Statistic. El contrato establece que Logging debe ser tejido primero que el aspecto Statistic, ante el conflicto sobre el componente AccountManager.

```
<conflict_resolution>
  <component comp_name= “AccountManager”>
    <member= “addAccount”>
      <aspect> Logging </aspect>
      <aspect> Statistc </aspect>
    </member>
  </component>
</conflict_resolution>
```

Figura 5: Contrato de composición/resolución para el conflicto entre los aspectos Logging y Statistic.

5. XAC : XML Aspect-Contract

XAC es un editor de contratos XML que permite en estas instancias iniciales explorar las posibilidades de esta propuesta. Los objetivos principales definidos para XAC son: (i) especificación de contratos; (b) detección de conflictos e interferencias en forma automática; (c) resolución de conflictos mediante la generación de contratos de composición/resolución y (d) generación de todos los contratos en formato XML.

En XAC la especificación de cada contrato es independiente y se realiza mediante un formulario específico. A partir de la definición de todos los contratos se aplican los procesos de detección de conflictos e interferencias. En el caso de existir interferencias y/o conflictos el desarrollador debe resolver convenientemente cada

situación. En el caso de las interferencias debe corregir los contratos especificados y en el caso de los conflictos deben especificarse los contratos de composición. En función de generar una representación contractual libre de interferencias y conflictos XAC no genera la representación XML si las interacciones no han sido resueltas.

En la Figura 6 se puede observar la especificación de los contratos del proyecto Banking Application con la herramienta XAC. En el panel lateral izquierdo, se visualizan cada uno de los contratos en forma individualizada y según el tipo de contrato (AccountManager es un contrato de componente, Logging, Persistence y Statistic son contratos de aspectos y cr_LS es un contrato de composición/resolución). En el panel principal, se ha desplegado el contenido del archivo XML correspondiente. Este fichero XML se genera a partir de los contratos individuales.

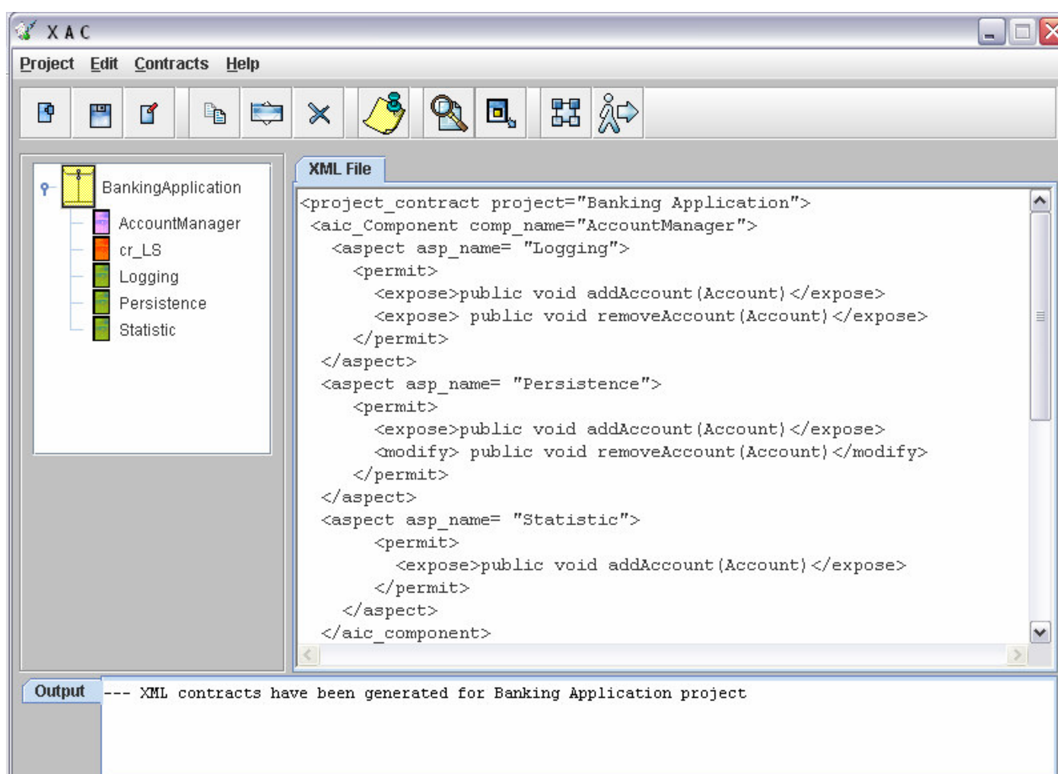

Figura 6: Edición de contratos y generación de contratos XML con XAC.

El archivo generado contiene contratos en el cual todas las posibles interferencias son permitidas y en el caso de los conflictos existe por cada uno, un contrato de composición/resolución. En esencia XAC genera un archivo XML que contiene los contratos de los componentes y aspectos, con interferencias permitidas y conflictos resueltos.

Siguiendo el mismo ejemplo planteado, en la Figura 7 se muestra el informe de detección de conflictos e interferencias efectuado por XAC (según las condiciones establecidas en la Tabla 2). En este caso, se ha detectado un conflicto y dos interferencias. Esta información permite al desarrollador resolver cada interacción como corresponda. Como ya se indicó anteriormente, en el caso de las interferencias deberán corregirse los contratos existentes y en el caso de los conflictos deberán crearse contratos de composición.

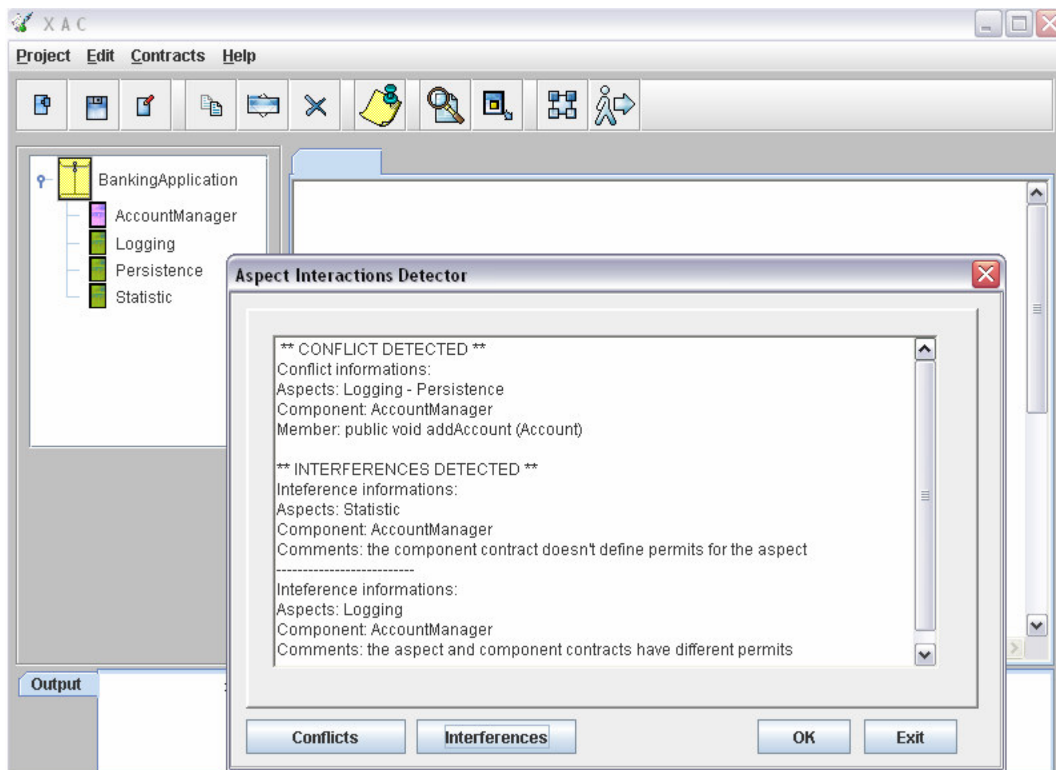

Figura 7: Detección de Conflictos e Interferencias con XAC.

En principio los contratos XML pueden ser utilizados para detectar y resolver interacciones aspectuales en forma automática. Pero los contratos XML pueden ser empleados en otros usos muy interesantes y beneficiosos para la fase de implementación como ser:

- *Verificación de la consistencia del código de los aspectos:* el código fuente de los aspectos pueden ser preprocesado a partir de los contratos XML para verificar que el código es consistente con los contratos.

- *Tejido Estático:* el proceso de tejido puede ser completado con los contratos de composición, por ejemplo tomando los contratos de composición para definir el orden de ejecución de los aspectos.

- *Tejido Dinámico*: los contratos XML pueden ser utilizados en ejecución para controlar que sólo se ejecuten interferencias permitidas y aplicar los contratos de composición a los conflictos que se sucedan, durante el tejido dinámico.

Los contratos XML pueden ser generados con un editor como XAC, pero también pueden formar parte del modelado avanzado, ya que pueden especificarse en forma paralela y complementaria a diagramas UML.

6. Conclusiones y Trabajo Futuro.

Las ideas originales del DbC pueden ser empleadas para manejar las distintas clases de interacciones aspectuales: interferencias y conflictos. Los contratos AICs constituyen un

interesante punto de partida, ya que al ser implementados en formato XML, los contratos se representan de manera natural sin perder las características básicas de la propuesta teórica original. Asimismo, los contratos XML pueden ser sometidos a procesos que automáticamente detecten interferencias y conflictos. Los conflictos se resuelven por medio de la especificación de contratos de composición/resolución. La herramienta XAC implementa estas ideas.

Lo valioso de la propuesta radica, que los contratos XML generados resultan ser una especificación independiente del lenguaje POA futuro, donde todas las posibles interferencias son permitidas y en el caso de los conflictos existe por cada uno, un contrato de composición/resolución. En esencia XAC genera un archivo XML que contiene los contratos de los componentes y aspectos, con interferencias permitidas y conflictos resueltos.

El trabajo futuro se orienta a emplear los contratos XML en la implementación y composición del software. En este sentido se proponen dos acciones de trabajo: (i) utilizar los contratos XML generados con XAC para comprobar mediante pre-procesamiento que el código fuente de los aspectos en AspectJ y/u otro lenguaje POA es consistente con los contratos; (ii) diseñar e implementar un tejedor sencillo que emplee y aplique los contratos XML.

El presente trabajo fue parcialmente financiado por la Universidad Nacional de la Patagonia Austral, Santa Cruz, Argentina y el proyecto PICT 32079 (ANPCYT).

Referencias

[Casas, 2005] Casas S., Marcos C., Vanoli V., Reinaga H., Saldivia C., Pryor J. y Sierpe L. (2005) “ASTOR: Un Prototipo para la Administración de Conflictos en AspectJ”. XIII Encuentro Chileno de Computación, XIII Jornadas Chilenas de Computación (JCC). Valdivia, Chile.

[Dijkstra, 1976] Dijkstra E. W. (1976) “A Discipline of Programming”. Prentice Hall.

[Duoence et all., 2002] Duoence R., Fradet P. and Südholt M. (2002) “Detection and Resolution of Aspect Interactions”. TR Nº4435, INRIA, ISSN 0249-6399, France.

[Durr et all., 2005] Durr P., Staijen T., Bergmans L. and Aksit M. (2005) “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.

[Hursch and Lopes, 1995] Hursch W. and Lopes C. (1995) “Separation of Concern”. TR. NU-CCS-95-03, Northeastern University.

[Lagaisse, 2004] Lagaisse B., Joosen W., De Win B. (2004). “Managing Semantic Interference with Aspect integration Contracts”. International Workshop on Software-Engineering Properties of Languages for Aspect Technologies, SPLAT, Lancaster, United Kingdom.

[Karaorman et all., 1998] Karaorman M., Olzle U., and Bruno J. (1998). “jContractor: A Refective Java Library to Support Design By Contract”. Technical report, Department of Computer Science, University of California.

[Kessler and Tanter, 2006] Kessler B. and. Tanter E. (2006) “Analyzing Interactions of Structural Aspects”. Workshop AID in 20th. European Conference on Object-Oriented Programming (ECOOP). France.

[Kiczales et all., 1997] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J. and Irwin J. (1997) “Aspect-Oriented Programming”. In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), Finland.

[Kiczales et all., 2001] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J. and Griswold W. (2001) “An Overview of AspectJ”. In J. L. Knudsen, editor, Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP), Num. 2072 in LNCS, pp 327–353, Springer-Verlag. Hungary.

[Kramer, 1998] Kramer R. (1998) “iContract | the Java Designs by Contract tool” In Proc. Technology of Object-Oriented Languages and Systems, TOOLS 26, Santa Barbara/CA, USA. IEEE Press.

[Meyer, 1991] Meyer B. (1991) “Eiffel: The Language”. Prentice-Hall.

[Meyer, 1992] Meyer B. (1992) “Applying Design by Contract". Computer, 25(10):40,51.

[Monga et all., 2003] Monga M., Beltagui F. and Blair L. (2003) “Investigating Feature Interactions by Exploiting Aspect Oriented Programming”.TR N comp-002- Lancaster University, England.

[Nagy et all., 2006] Nagy I., Bergmans L. and Aksit M. (2006) “Composing Aspects at Shared Join Point”. Workshop AID in 20th. European Conference on Object-Oriented Programming (ECOOP). France.

[Pawlak et all., 2005] Pawlak R., Duchien L., and Seinturier L.(2005) “CompAr: Ensuring safe around advice composition”. In FMOODS 2005, Vol. 3535 of LNCS, pp 163–178.

[Piveta and Zancanella, 2003] Piveta E. and Zancanella L.(2003) “Aspect Weaving Strategies”. Journal of Universal Computer Science, Vol.9, Num. 8.

[Plath, 2000] Plath M. (2000) “Trace-Zusicherungen in Jass | Erweiterung des Konzepts "Programmieren mit Vertrag". Master's thesis, Universit. at Oldenburg.

[Rangarajan, 2000] Rangarajan K. (2000) “Design by Contract for Java using JMSAssert”. http://www.mmsindia.com/JMSAssert.html

[Rausch, 2002] Rausch A.: “Design by Contract” + “Componentware” = “Design by Signed Contract” ,Journal of Object Technology, Vol. 1, No. 2, July-August 2002.

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

[Storzer and Krienkle, 2003] Storzer M. and Krinkle J. (2003) “Interference Analisys for AspectJ”. FOAL: Foundations of Aspect-Oriented Languages, USA.

[Tanter and Noye, 2005] Tanter E. and Noye J. (2005) “A versatile kernel for multi-language AOP”. In Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, Vol. 3676 of LNCS, pp 173–188, Springer-Verlag. Estonia.

[Yu and Kienzle, 2004] Yu Y. and Kienzle J. (2004) "Towards an Efficient Aspect Precedence Model". Proceeding of the 2004 Dynamic Aspects Workshop, pp 156-167, England.