REVISTA COLOMBIANA DE COMPUTACIÓN
Volumen 1, número 1
Págs. 73-90

Relaciones entre Casos de Uso en el Unified Modeling Language

Roxana S. Giandin Claudia F. Pons
LIFIA, Universidad Nacional de La Plata
Calle 50 esq.115, 1er.Piso, (1900) La Plata, Argentina Tel/Fax: (+54 221) 422 8252
giandini@sol.info.unlp.edu.ar

Resumen

El Unified Modeling Language (UML) es un lenguaje gráfico, semiformal, que ha sido aceptado como estándar para describir sistemas de software orientados a objetos. UML define varios tipos de diagramas que se utilizan para describir diferentes aspectos o vistas de un sistema. En particular, los diagramas de Casos de Uso se utilizan para capturar los requerimientos de los sistemas y guiar su proceso de desarrollo. Los distintos Casos de Uso que se definen a lo largo de un proceso de desarrollo no son independientes sino que es posible establecer relaciones entre ellos. Las principales relaciones consideradas por UML son: Generalización (*Generalization*), Inclusión (*Include*) y Extensión (*Extend*). Estas relaciones, tanto como el resto de las construcciones de UML, están definidas semiformalmente, dando lugar a interpretaciones ambiguas e inconsistencias.

Este trabajo presenta una formalización de las principales relaciones entre Casos de Uso aportando precisión en su definición. Además, con base en esta formalización se estudia la composición de estas relaciones en la etapa de evolución, mostrando en qué casos esta combinación es aplicable y cuándo se producen situaciones conflictivas.

Palabras Claves: Ingeniería de Software, Análisis y Diseño Orientado a Objetos, Lenguajes gráficos de modelado, Casos de Uso, Semántica formal.

Relations Between Use Cases in the Unified Modeling Language

Abstract

The Unified Modeling Language (UML) is a semi-formal graphical language that has been accepted as standard to model object-oriented software systems. This language

defines various kinds of diagrams which are used to describe different aspects or views of a system. In particular, Use Cases diagrams are used to capture the requirements of the systems and to guide their development process. The different Use Cases defined throughout a development process are not independent but it is possible to set relations between them. The main relations considered by UML are the following: *Generalization, Include and Extend*. These relations as well as the remaining UML constructs are semi-formally defined, giving place to ambiguous interpretations and inconsistancies.

This paper presents a formalization that gives precision to the definition of the main relations between Use Cases. Also, this work studies -based on the formalization- the composition between these relations during the evolution phase, showing in which cases this combination can be applied and when it may be conflicting.

Keywords: Software Engineering, Object-Oriented Analysis and Design, Graphical Modeling Languages, Use Cases, Formal Semantics

1. Introducción

El *Unified Modeling Language* (UML) [17] es una expresiva y poderosa notación basada en diagramas que ha sido aceptada como estándar para describir sistemas de software orientados a objetos. El UML define varios tipos de diagramas que se utilizan para describir diferentes aspectos o vistas de un sistema.

En UML, una de las herramientas principales para modelar comportamiento es la construcción Caso de Uso (*Use Case*), introducida originariamente por Jacobson en [7] y posteriormente desa-rrollada en trabajos como [8] entre otros. Un Caso de Uso especifica una manera de usar un sistema sin revelar la estructura interna del mismo. Los Casos de Uso han sido adoptados casi universalmente para capturar los requerimientos de los sistemas de software; sin embargo, los Casos de Uso son más que una herramienta de especificación ya que tienen una gran influencia sobre todas las fases del proceso de desarrollo tales como el diseño, la implementación y las pruebas del sistema.

En general, los procesos de desarrollo de software (por ejemplo el *Unified Process* [9]), son iterativos e incrementales, repitiendo una serie de iteraciones sobre el ciclo de vida de un sistema. Cada iteración consiste de un paso a través de las etapas de requerimientos, análisis, diseño, implementación y prueba. El resultado de cada iteración representa un incremento sobre cada uno de los modelos construidos en las etapas anteriores. Los distintos Casos de Uso que se definen a lo largo del proceso de desarrollo no son independientes sino que es posible establecer relaciones de dependencia entre ellos. Las principales relaciones consideradas por UML son:

- Generalización (*generalization*): es una relación que amplía la funcionalidad de un Caso de Uso o refina su funcionalidad original mediante el agregado de nuevas operaciones y/o atributos y/o secuencias de acciones.

- Inclusión (*include*): es una relación mediante la cual se re-usa un Caso de Uso encapsulado en distintos contextos a través de su invocación desde otros Casos de Uso.

- Extensión (*extend*): es una relación que amplía la funcionalidad de un Caso de Uso mediante la extensión de sus secuencias de acciones.

Estas relaciones y el resto de las construcciones de UML están definidas en el documento de especificación de UML [17]. Esta descripción es semiformal, ya que algunas partes del documento están expresadas en lenguaje formal, mientras que otras partes están escritas en lenguaje natural. La sintaxis abstracta de las diferentes construcciones se especifica con la notación gráfica de diagramas de UML, mientras que las reglas de buena formación de UML se dan en OCL [12], un lenguaje orientado a objetos para expresar restricciones. Sin embargo, la semántica de UML se describe en inglés. Por lo tanto, la definición de la estructura del lenguaje es rigurosa, mientras que la descripción de su semántica es informal.

La especificación formal de un lenguaje es necesaria para que no sea ambiguo y pueda ser entendido y utilizado apropiadamente. Además, la semántica de un lenguaje debe ser precisa para que resulte posible aplicar herramientas que realicen operaciones inteligentes sobre modelos expresados en el lenguaje, como chequeo de consistencia y transformaciones entre modelos.

Existe un monto importante de trabajos teóricos que brindan una descripción precisa de los principales conceptos del lenguaje gráfico de modelado UML y proveen reglas para analizar sus propiedades: [1, 3, 4, 5, 10, 11, 15]. Específicamente en el área de diagramas de comportamiento de UML, en [13] se formalizan Casos de Uso y en [14] Colaboraciones; sin embargo, en todos estos casos las definiciones incluyen elementos semánticos del sistema, tales como objetos ejecutando tareas del sistema. Por otra parte, el trabajo de Back [2] analiza los diagramas de Casos de Uso y los formaliza mediante contratos definiendo previamente las clases que forman el sistema y su comportamiento.

Específicamente en este trabajo:

- Presentamos definiciones rigurosas y reglas de buena formación para poder precisar sin ambigüedad cuándo las operaciones entre Casos de Uso están bien definidas y cómo es el resultado de aplicarlas, definiendo así un álgebra para Casos de Uso. Estas definiciones se basan únicamente en elementos sintácticos del sistema, marcando una ventaja respecto a los trabajos citados ya que no se involucran distintos niveles de desarrollo (por ejemplo especificación y ejecución). Consideramos además, las versiones actualizadas de UML para describir relaciones entre Casos de Uso.

- Realizamos una comparación semántica entre las relaciones de Inclusión y Extensión.

- Utilizando la formalización estudiamos la evolución de Casos de Uso mediante la combinación de operaciones aplicables independientemente sobre ellos. Consideramos que el incremento sobre un Caso de Uso en una iteración a través de las etapas del proceso de desarrollo de software, puede constar de más de una operación primitiva. Esta composición puede producir situaciones conflictivas que deben ser detectadas formalmente para poder garantizar la consistencia del sistema de software durante la evolución.

Este artículo se organiza de la siguiente forma: la sección 2 introduce el concepto de Caso de Uso y su representación; las secciones 3, 4 y 5 definen, ejemplifican y formalizan las relaciones *Include*, *Extend* y *Generalization* respectivamente. La sexta compara dos de estas relaciones, mientras que la séptima enfoca el tema de composición de operaciones en la evolución de Casos de Uso. Por último se presentan conclusiones y líneas de trabajo futuro.

2. Casos de Uso

Un Caso de Uso describe un servicio provisto por un sistema, es decir un modo específico de usarlo. El conjunto completo de Casos de Uso especifica todas las posibles maneras en las que el sistema puede ser usado, sin revelar cómo esto es implementado por el sistema. Esto hace a los Casos de Uso apropiados para definir requerimientos funcionales en etapas tempranas del desarrollo del sistema, donde la estructura interna de éste aún no fue definida. Debido a que los Casos de Uso no se manejan con elementos dentro del sistema sino que se centran en cómo el sistema es percibido desde el exterior, son útiles en discusiones con usuarios finales para asegurar que hay concordancia con los requerimientos realizados sobre el sistema, sobre sus limitaciones, etc. Más precisamente, un Caso de Uso especifica un conjunto de secuencias completas de acciones que el sistema puede realizar. Cada secuencia es iniciada por un usuario del sistema. Esto incluye la interacción entre el sistema y su entorno como también la respuesta del sistema a estas interacciones.

2.1 Ejemplo

Presentamos el modelo de un sistema para mantener una Biblioteca. Los socios de esta biblioteca comparten una colección de libros. El sistema les debe permitir retirar libros, devolverlos o renovar un préstamo. Al devolver o al renovar el préstamo de un libro, el socio debe pagar una cuota. En caso de no pago de esta cuota, el socio no podrá retirar un nuevo libro o renovar un préstamo. Las figuras 1 y 2 describen el Caso de Uso RenewLoan. Este Caso de Uso modela la funcionalidad del sistema, solicitada por un socio de la biblioteca, para la renovación de un préstamo.

Un Caso de Uso puede describirse en varias formas. En la práctica, se utiliza frecuentemente texto común describiendo las conversaciones [8] entre usuario y sistema, que muestra en un alto nivel las acciones del usuario y las actividades del sistema en respuesta a estas acciones. La Figura 2 muestra una conversación entre un actor (socio de la librería) y el sistema. La conversación considera la secuencia normal de acciones y también secuencias alternativas (por ejemplo, el caso en que el libro no está disponible).

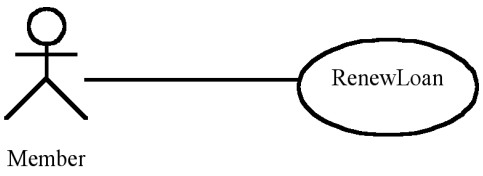

Figura 1. El Caso de Uso RenewLoan

En UML un Caso de Uso es un tipo de *Classifier* y por lo tanto posee una colección de operaciones (con sus correspondientes métodos) describiendo su comportamiento. Las operaciones de un Caso de Uso describen que mensajes puede recibir una instancia de Caso de Uso, mientras que los métodos describen la implementación de las operaciones en términos de secuencias de acciones que son ejecutadas por instancias del Caso de Uso. Por ejemplo, el Caso de Uso RenewLoan puede recibir un único mensaje, por lo tanto se describe con una única operación. El método asociado a esta operación contiene el conjunto de las secuencias de acciones que se forman teniendo en cuenta los casos normales y los casos alternativos para resolver la operación.

Acciones del usuario	**Respuesta del sistema**
Actor: Member 1. ask for renew loan	2. validate member identification 3. validate book availability 4. ask for debt 5 renew loan
Alternativas: 1.member identification is not valid -> reject loan	
2.book is not available -> reject loan	
3.member has debt -> payFee, then renew loan	

Figura 2. Conversación del Caso de Uso RenewLoan

Sea uc el Caso de Uso definido anteriormente. La definición de uc, usando la notación estándar y el metamodelo de UML ([17], págs. 2-14, 2-114), es como sigue:

uc.name= RenewLoan

uc.operation[1] = {op1}

op1.name=ask for renew loan

op1.method.body[2] =
{< validate member identification, validate book availability, ask for debt, renew loan>,
< validate member identification, reject loan>,
< validate member identification, validate book availability, reject loan>,
< validate member identification, validate book availability, ask for debt, pay fee, renew loan >}

3. La relación Include entre Casos de Uso

Como se establece en [17], una relación include entre dos Casos de Uso indica que el comportamiento definido en el Caso de Uso a adicionar, es incluído en un lugar dentro de la secuencia del comportamiento realizado por una instancia del Caso de Uso base. Cuando una instancia del Caso de Uso «llega al lugar» donde el comportamiento de otro Caso de Uso debe ser incluído, ejecuta todo el comportamiento descripto por el Caso de Uso incluído y luego continúa de acuerdo a su Caso de Uso original. El Caso de Uso incluído no depende del Caso de Uso base. En este sentido, el Caso de Uso incluído representa comportamiento encapsulado que puede ser reusado en varios Casos de Uso.

3.1 Ejemplo

En el ejemplo anterior, con el fin de completar el modelo de Casos de Uso, podemos agregar un nuevo Caso de Uso llamado PayFee, encargado de la tarea del pago de la deuda. La figura 3 muestra al Caso de Uso RenewLoan relacionado con el Caso de Uso agregado mediante la relación *include*.

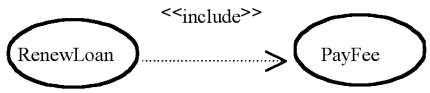

Figura 3. La relación Include entre Casos de Uso

El Caso de Uso PayFee consta de una operación, llamada *payfee*, con una única secuencia de acciones:

payFee.actionSequence= {< input payment, validate payment, modify debt>}

La relación *include* entre los Casos de Uso RenewLoan y PayFee se establece de la siguiente manera:

El Caso de Uso PayRenewLoan resultante de aplicar la inclusión tendrá el comportamiento de RenewLoan adicionado con el de PayFee.

La representación textual de PayRenewLoan es:

PayRenewLoan.operation = {op1´}

op1´.name= ask for renew loan

op1´.actionSequence= {<validate member identification, validate book availability, ask for debt, renew loan>,< validate member identification, reject loan>,< validate member identification, validate book availability, reject loan>,< validate member identification, validate book availability, ask for debt, input payment, validate payment, modify debt, renew loan >}

3.2 Formalizando la inclusión (include) entre Casos de Uso

En este apartado presentamos una formalización para la inclusión entre Casos de Uso, definida en UML mediante la relación *include*.

Definición 1: Inclusión de Casos de Uso

Un Caso de Uso UC es el resultado de incluir UC2 dentro de UC1; es decir UC = UC1 $^{3}_{inc}$ UC2 si se cumple lo siguiente:

a- Aplicabilidad: dentro de UC1 existe una llamada a UC2
$oUC1.operation . $utoo.actionSequence . UC2.nameGut

b- Completitud: UC contiene todas las posibles formas de incluir UC2 dentro de UC1
« o1UC1.operation . $oUC.operation . (o.name=o1.name Ÿ(« s1b1.actionSequence. inclusions(s1,UC2) Õ o.actionSequence))

c- Corrección: todas las secuencias de acciones de UC provienen de incluir a UC2 dentro de alguna secuencia de acciones de UC1.
« oUC.operation . $o1UC1.operation . (o.name=o1.name Ÿ(« soo.actionSequence. $s1b1.actionSequence. sinclusions(s1,UC2)))

Definición 2:

isIncluible: ActionSequence x UseCase

El predicado isIncluible(s,uc) es verdadero si la secuencia de acciones s contiene alguna invocación al Caso de Uso uc y está definido de la siguiente forma:
« s:ActionSequence « uc:UseCase . isIncluible(s,uc) ´ uc.name s

Definición 3:

inclusions: ActionSequence x UseCase -> Set(ActionSequence)

La función inclusions(s, uc) retorna el conjunto de todas las posibles inclusiones del Caso de Uso uc en la secuencia s y esta definida por casos, de la siguiente forma.

Case 1: ÿisIncluible(s, uc)

inclusions(s, uc)={s}

Case 2: isIncluible(s, uc)

inclusions(s, uc)={ before(s,a);s2;after(s,a) / a=uc.name Ÿ s2uc.actionSequence }

3.3 Aplicando la formalización al ejemplo

Presentamos a continuación parte de la instanciación de las fórmulas que definen la inclusión entre Casos de Uso, sobre el ejemplo dado en 3.1 y utilizando la representación textual del caso de Uso introducido en 2.1. La finalidad es mostrar que el modelo del ejemplo satisface dichas fórmulas, es decir es una inclusión de Casos de Uso completa y correcta.

Por **Definición 1**, debemos mostrar que: PayRenewLoan = RenewLoan β_{inc} PayFee, o sea se deben cumplir los puntos de esta definición:

a) Aplicabilidad: dentro de RenewLoan existe una llamada a PayFee
$o∈RenewLoan.operation . $ut∈o.actionSequence . PayFee.name=ut Dado que la única operación op1 en RenewLoan tiene una secuencia de acciones: < validate member identification, validate book availability, ask for debt, pay fee, renew loan > que incluye la acción *payFee* y considerando que PayFee es un Caso de Uso con una sola operación llamada *payFee*, la condición de aplicabilidad queda probada.

b- Completitud: PayRenewLoan contiene todas las posibles formas de incluir PayFee dentro de RenewLoan
« o1∈RenewLoan.operation. $ o ∈PayRenewLoan.operation . (o.name= o1.name Ÿ (« s1∈o1.actionSequence. inclusions(s1, PayFee) Õ o.actionSequence)) Como RenewLoan y PayRenewLoan tienen una única operación cada uno -op1 y op1´ respectivamente- con el mismo nombre, sólo resta verificar la inclusión:
(« s1∈op1.actionSequence. inclusions(s1, PayFee) Õ op1´.actionSequence).

En el ejemplo, las secuencias de acciones son las siguientes:

op1.actionSequence={ut1, ut2, ut3, ut4}

op1'.actionSequence={ut1, ut2, ut3, ut4'}

Por Definition 2, sólo la secuencia ut4 de op1 -ver punto a)- satisface el predicado isIncluible. Por Definition 3:
inclusions(ut4, PayFee) = {<validate member identification, validate book availability, ask for debt, input payment, validate payment, modify debt, renew loan >}. Luego, inclusions(ut4, PayFee) Õ op1´.actionSequence. El resto de las secuencias existen en op1´ en su forma original, por lo que la inclusión queda probada.

De manera similar puede instanciarse la fórmula de Corrección y mostrar que se satisface, al igual que las definiciones presentadas más adelante.

4. La relación Extend entre Casos de Uso

La relación *extend* establece que un Caso de Uso puede ser extendido con algún comportamiento adicional definido en otro Caso de Uso. La relación contiene una condición y referencia una secuencia de puntos de extensión en el Caso de Uso base. Una vez que una instancia del Caso de Uso ejecuta un comportamiento referenciado por el primer punto de extensión de la relación, la condición es evaluada. Si se cumple, la secuencia de la instancia se extiende para incluir la secuencia del Caso de Uso extensión. Las diferentes partes del Caso de Uso extensión son insertadas en los lugares definidos por la secuencia de puntos de extensión de la relación: una parten cada punto de extensión referenciado.

4.1 Ejemplo

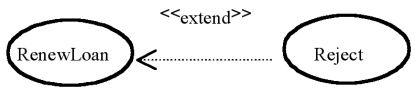

Figura 4. La relación extend entre Casos de Uso

El Caso de Uso descripto en la sección 2 puede ser extendido con el fin de contabilizar cuántas renovaciones de libros se rechazan por identificación inválida de socio. Esta extensión puede realizarse sin modificar el Caso de Uso original, mediante una relación *extend* y un nuevo Caso de Uso especificando el incremento de comportamiento. La Figura 4 muestra esta relación entre los Casos de Uso. El punto de extensión en este caso será la acción de rechazo de un préstamo. La condición de la extensión es que la identificación del socio es inválida.

Sea CountReject el Caso de Uso que especifica el incremento de comportamiento. El Caso de Uso CountReject tiene una operación con una única secuencia de acciones:

countReject.actionSequence={<updateRejectCounter>}

La relación *extend* ext se define:

ext.base= RenewLoan

ext.extension= CountReject

ext.condition= *the member identification is not valid*

ext.extensionPoint= { reject loan }

Sea CountRejectRenewLoan el Caso de Uso obtenido de RenewLoan por aplicación de la relación de extensión ext, es decir CountRejectRenewLoan = RenewLoan $^{3}_{ext}$ CountReject. La representación textual de CountRejectRenewLoan es:

countRejectRenewLoan.actionSequence=
{<validate member identification, validate book availability, ask for debt, renew loan >,
< validate member identification, reject loan, updateRejectCounter >,
< validate member identification, validate book availability, reject loan>,
< validate member identification, validate book availability, ask for debt, pay fee, renew loan >}

4.2 Formalizando la extensión (extend) entre Casos de Uso

Como en el caso de la inclusión, presentamos una formalización para la operación de extensión entre Casos de Uso, definiendo las características del Caso de Uso resultante. En UML la relación de extensión es un elemento especial de modelado, por lo tanto la relación aparece explícitamente en las siguientes definiciones.

Definición 4: Extensión de Casos de Uso

Un Caso de Uso UC es la extensión de UC1 por UC2 a través de una relación "extend" *ext*; es decir, UC = UC1 $^{3}_{ext}$ UC2 si se cumple:

a- **Aplicabilidad**: El Caso de Uso UC1 es extendible por *ext* si se cumple la siguiente condición:

Para cada punto de extensión de *ext* debe existir una acción que coincida con él dentro de las secuencias de acciones del Caso de Uso:

« i Œext. extensionPoint**.** $uo ŒUC1.operation **.** $uotŒuo.actionSequence **.** i.location Œuot

b- **UC1-Completitud**: cada secuencia de acciones en UC1 es extendida en todas las formas posibles:
« o1 ŒUC1.operation . $o ŒUC.operation .
(o.name=o1.name Ÿ(‘s1Œ1.actionSequence. extensions(s1,ext,UC2) Õ o.actionSequence))

c- **UC1-Corrección**: cada secuencia de acciones en UC es una extensión de alguna secuencia de acciones en UC1.
« oŒUC.operation .$o1Œ UC1.operation . (o.name=o1.name Ÿ(« sŒ.actionSequence. $s1Œ1.actionSequence. sŒextensions(s1,ext,UC2)))

Definición 5:

isExtensible: ActionSequence x Extend

El predicado es verdadero si la secuencia de acciones contiene algún punto de extensión de la relación *extend* y está definido de la siguiente forma:
« s:ActionSequence « ext:Extend.
isExtensible(s,ext) ´ $ iŒ*ext*.extensionPoint . i.locationŒs

Definición 6:

extensions: ActionSequence x Extend x UseCase -> Set(ActionSequence)

La función extensions(s,ext,uc) retorna el conjunto de todas las posibles extensiones de la secuencia s dadas por la relación ext y el Caso de Uso uc. La función se define por casos de la siguiente forma:
Case 1: ÿisExtensible(s,ext)

extensions(s,ext,uc)={s}

Case 2: isExtensible(s,*ext*)

extensions(s,ext,uc)={before(s,i.location); s2; after(s, i.location) /
iŒ*ext*.extensionPoint Ÿi.locationŒs Ÿs2Œuc.actionSequence }

Definición 7: UC extiende a UC1 si existe un Caso de Uso UC2 tal que UC es la extensión de UC1 por UC2 a través de una relación ext:
UC *extends*$_{ext}$ UC1 ´ $ UC2:UseCase . (UC = UC1 ³$_{ext}$ UC2)

5. La relación Generalization entre Casos de Uso

Una relación de generalización entre Casos de Uso implica que el Caso de Uso hijo hereda todos los atributos, secuencias de comportamiento, puntos de extensión y relaciones definidos en el Caso de Uso padre. El Caso de Uso hijo puede definir nuevas operaciones, como también redefinir o enriquecer con nuevas secuencias de acciones operaciones ya existentes en el Caso de Uso padre. Para distinguir si la especialización está redefiniendo una operación del padre o agregándole secuencias de acciones, sugerimos la inclusión de un estereotipo (elemento de UML) <<*redefine*>> para el primer caso o <<*enrichment*>> para el segundo, en la operación en cuestión.

5.1 Ejemplo

El Caso de Uso descripto en la sección 2 puede ser especializado con el fin de contabilizar cuántas personas renuevan el préstamo

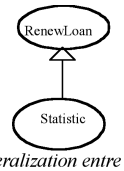

Figura 5.Generalization entre Casos de Uso

de un libro técnico. Esta especialización puede realizarse sin modificar el Caso de Uso original, mediante una relación *generalization* entre el Caso de Uso RenewLoan y un nuevo Caso de Uso que lo especializa especificando el comportamiento adicional. La Figura 5 muestra la relación de generalización entre los Casos de Uso.

Sea Statistic el Caso de Uso que especifica el incremento de comportamiento. El Caso de Uso Statistic tiene una única operación que enriquece con nuevas secuencias de acciones (que contabilizan el caso en que la renovación se aplique a un libro técnico) a la única operación de RenewLoan:

Statistic.operation= {op1} op1.name=renewLoan op1.stereotype=<<*enrichment*>>
op1.actionSequence= {<validate member identification, validate book availability, ask for debt, renew loan, updateRenewsTechnicalCounter >, < validate member identification, validate book availability, ask for debt, pay fee, renew loan, updateRenewsTechnicalCounter >}

Sea StatisticRenewLoan el Caso de Uso obtenido por aplicación de la relación de *generalization*, es decir StatisticRenewLoan = RenewLoan $\triangleleft_{gen}$ Statistic. La representación textual de StatisticRenewLoan es:

StatisticRenewLoan.operation={op}

op.name=renewLoan

op.actionSequence=
{<validate member identification, validate book availability, ask for debt, renew loan >,
< validate member identification, reject loan >,
< validate member identification, validate book availability, reject loan>,
< validate member identification, validate book availability, ask for debt, pay fee, renew loan >,
<validate member identification, validate book availability, ask for debt, renew loan, updateRenewsTechnicalCounter >, < validate member identification, validate book availability, ask for debt, pay fee, renew loan, updateRenewsTechnicalCounter >}

5.2 Formalizando generalization entre Casos de Uso

Formalizamos en este apartado la operación de generalización entre Casos de Uso, definiendo las características del Caso de Uso resultante. Dado que los Casos de Uso son elementos generalizables de UML, por el mecanismo implícito de herencia, el Caso de Uso resultante contará con todos los atributos y asociaciones (por ejemplo con Actores fuera del sistema) del Caso de Uso padre y del hijo, considerando las redefiniciones hechas en este último. No queda explícito, en la descripción de generalización entre Casos de Uso de UML, lo que sucede con las operaciones que se enriquecen o redefinen, ni con las relaciones de extensión de los Casos de Uso participantes. La formalización que aquí presentamos apunta a agregar claridad y precisión en relación a esos aspectos.

Definición 8: Generalización de Casos de Uso

Un Caso de Uso UC es la generalización de UC1 por UC2, es decir UC = UC1 $\triangleleft_{gen}$ UC2 si se cumple:

a- **Aplicabilidad:** Toda operación de UC2 que no está en UC1 debe tener distinto nombre que cualquier acción en UC1. Si esto no ocurriera, se confundiría acción con operación en el Caso de Uso resultante.

« o2 ŒUC2.operation . (o2 œUC1.operation Æ « o1 ŒUC1.operation .
‘š1 Œo1.actionSequence . ‘a Œs1. o2.name<>a)

b- **UC1-Completitud**:

b.1- Las operaciones de UC1 que no están en UC2 y las de UC2 que no están en UC1, están en UC en su forma original:
« o1 ŒUC1.operation . ((« o2 ŒUC2.operation . o2.name<>o1.name)Æ o1ŒUC.operation)
« o2 ŒUC2.operation . ((« o1 ŒUC1.operation . o1.name<>o2.name)Æ o2ŒUC.operation)

b.2- Las operaciones que están en UC1 y en UC2, están en UC redefinidas o enriquecidas, según el estereotipo de cada una.
« o1 ŒUC1.operation .
($o2 ŒUC2.operation . o2.name=o1.name)Æ
((o2.stereotype = <<*redefine*>> Æ o2 ŒUC.operation) Ÿ
(o2.stereotype = <<*enrichment*>> Æ $oŒUC.operation . (o.name =o2.name Ÿ
o.actionSequence= o1.actionSequence » o2.actionSequence)))

b.3- *Herencia de relación de extensión*: Por cada relación de extensión definida para UC1 o para UC2, habrá una en UC con igual nombre y puntos de extensión.
« ext: Extend . (ext.base = UC1 ¦ ext.base = UC2 Æ $ext’. (ext’.base=UC Ÿext’.name=ext.name Ÿ ext’.extensionPoint=ext.extensionPoint))

c- **UC1-Corrección:**

Toda operación en UC está en UC1 o en UC2. Si está definida en ambos, estará en UC redefinida o enriquecida, según el estereotipo de la operación:
« oŒUC.operation .
((oŒUC1.operation Ÿ« o1ŒUC2.operation . o1.name <> o.name) ¦
(oŒUC2.operation Ÿ« o1ŒUC1.operation . o1.name <> o.name) ¦
(oŒUC2.operation Ÿo2.stereotype = <<*redefine*>>) ¦
($o1ŒUC1.operation . $o2ŒUC2.operation . (o1.name = o2.name Ÿ
o2.stereotype = <<*enrichment*>> Ÿo.name =o2.name Ÿ
o.actionSequence= o1.actionSequence » o2.actionSequence)))

Definición 9: UC es una generalización de UC1 a través del incremento especificado por UC2 si UC se obtiene componiendo UC1 con UC2 mediante una relación de *generalización*, es decir:
UC *generalice*$_{UC2}$ UC1 ´ (UC = UC1 ³$_{gen}$ UC2)

6. Comparación entre las relaciones *Include* y *Extend*

UML define las relaciones *extend* e *include* para Casos de Uso, marcando diferencias entre ellas. La primera cuenta con una construcción para definir la relación, la cual contiene puntos de extensión (coincidentes con acciones del Caso de Uso a extender con el comportamiento del otro) y una condición a ser evaluada al llegar al primero de esos puntos. La relación *include*, por otra parte, se define como la inclusión de un Caso de Uso en otro, al estilo “modularización de procedimientos”, ya que el “lugar” donde se produce la adición debe coincidir con el nombre del Caso de Uso a incluir. Este mecanismo es similar a una “invocación” entre procedimientos.

Luego de desarrollar ejemplos y llegar a una formalización de ambas relaciones entre Casos de Uso, presentada en secciones anteriores, observamos que no existen diferencias semánticas signifi-

cativas entre ellas. Más precisamente podemos concluir que la relación *include* puede considerarse un caso particular de la relación *extend*, donde:

- La secuencia de puntos de extensión es una secuencia unaria donde el único punto coincide con la invocación del Caso de Uso a incluir.
- La condición que debe cumplirse al alcanzar este punto siempre evalúa al valor verdadero.

6.1 Ejemplo

La inclusión entre RenewLoan y PayFee presentada en la sección 3, puede ser expresada mediante una relación *extend*, como mostraremos a continuación.

Sea ext´ la relación *extend* que se define de la siguiente forma:

ext´.base= RenewLoan ext´.extension= PayFee ext´.condition= *true*

ext´.extensionPoint= {pay fee}

Sea PayRenewLoan´ el use case obtenido de RenewLoan por aplicación de la relación de extensión ext´, es decir PayRenewLoan´ = RenewLoan $^{3}_{ext´}$ PayFee. La representación textual es:

payRenewLoan´.actionSequence=
{< validate member identification, validate book availability, ask for debt, renew loan>,
< validate member identification, reject loan>, < validate member identification, validate book availability, reject loan>, < validate member identification, validate book availability, ask for debt, input payment, validate payment, modify debt, renew loan >}

Puede observarse que los Casos de Uso resultantes PayRenewLoan (definido mediante la relación *include*, en la sección 3) y PayRenewLoan´ (definido mediante la relación *extend*) coinciden.

7. Composición de operaciones sobre Casos de Uso

Las relaciones que hemos formalizado pueden considerarse *operaciones* sobre Casos de Uso. Estas operaciones representan una evolución o incremento de comportamiento sobre el Caso de Uso base. Sin embargo, dicho incremento en un paso de iteración dentro del proceso de desarrollo, puede no ser primitivo o elemental sino que puede componerse de más de una operación. Si consideramos dos operaciones que en forma independiente son aplicables a un Caso de Uso, como se muestra en la figura 6, queremos ver si su composición (es decir su aplicación conjunta) es consistente y genera un nuevo Caso de Uso resultado o si genera algún tipo de inconsistencia o conflicto. Un análisis de conflictos similar fue desarrollado en [6] respecto a la evolución de componentes reusables.

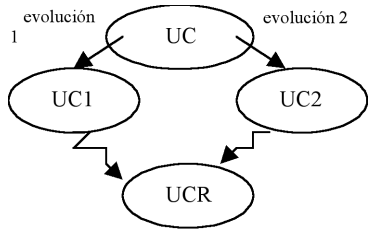

Figura 6. Conflictos de Evolución

7.1 Ejemplos

Consideramos dos ejemplos de evolución compuesta sobre un Caso de Uso base. En cada ejemplo, ambas operaciones son aplicables al Caso de Uso base en forma independiente. Sin embargo la composición en un caso es aplicable y en el otro,

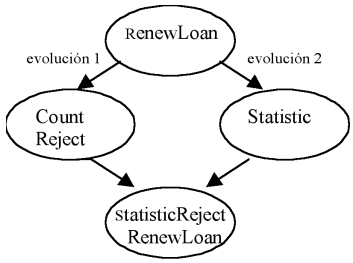

Figura 7. Composición no conflictiva

conflictiva. En el próximo apartado, analizamos y expresamos todos los casos de composición, utilizando la formalización de operaciones.

Composición no conflictiva

Consideremos el Caso de Uso RenewLoan, descripto en la sección 2, al que se le aplican dos evoluciones, como muestra la Figura 7:

Por un lado, la extensión descripta y ejemplificada en la sección 4, con el Caso de Uso CountReject, y por otro la generalización, de tipo *enrichment*, con el Caso de Uso Statistic, descripta en la sección 5. En este caso la composición es aplicable ya que la extensión actúa sobre secuencias de acciones ya existentes en el Caso de Uso RenewLoan y la generalización agrega nuevas secuencias de acciones. El resultado de la composición puede describirse en un nuevo Caso de Uso StatisticCountRejectRenewLoan, cuya representación textual es:
statisticCountRejectRenewLoan.actionSequence={<validate member identification, validate book availability, ask for debt, renew loan >,< validate member identification, reject loan, updateRejectCounter >,< validate member identification, validate book availability, reject loan>, < validate member identification, validate book availability, ask for debt, pay fee, renew loan>, <validate member identification, validate book availability, ask for debt, renew loan, updateRenewsTechnicalCounter >, < validate member identification, validate book availability, ask for debt, pay fee, renew loan, updateRenewsTechnicalCounter >}

Composiciones Conflictivas

Considerando nuevamente el Caso de Uso RenewLoan, supongamos que se desea, como **primer evolución**, redefinir su única operación en el sentido que no se valide la disponibilidad del libro, dándole prioridad al socio que ya lo tiene en su poder.

Sea NewRenewLoan el Caso de Uso que especifica este cambio de comportamiento:

NewRenewLoan.operation = {op1} op1.name= renewLoan op1.stereotype= <<*redefine*>>
op1.actionSequence= {< validate member identification, ask for debt, renew loan>,< validate member identification, reject loan >,< validate member identification, ask for debt, pay fee, renew loan >}

Supongamos que la **segunda evolución** está dada por una generalización, de tipo *enrichment*, con el Caso de Uso Statistic ya descripta. En este caso la composición no es aplicable ya que la redefinición modifica el comportamiento del Caso de uso RenewLoan y el *enrichment* agrega nuevas secuencias de acciones asumiendo el comportamiento anterior.

7.2 Identificación de conflictos de evolución

La Tabla 1 muestra la combinación entre evoluciones. La inclusión (*include*) no es considerada debido a que en la sección anterior se pudo concluir que esta relación es un caso particular de la extensión (*extend)*, pues no existen deferencias semánticas significativas entre ellas.

En este análisis de conflictos consideramos al Caso de Uso base compuesto por una o más operaciones y al Caso de Uso llamado por ejemplo UC1, que representa al incremento, por simplicidad, con una única operación. En adelante, utilizamos la notación *uc1.name* para referir al nombre de la única operación de UC1, *uc1.stereotype* para su estereotipo, etc.

Tabla 1. Conflictos en la composición de operaciones

evolución 1 / evolución 2		extend	generalization		
			redefine	enrichment	adding
extend		conflicto 1	conflicto 3	conflicto 6	conflicto 8
generalization	redefine	conflicto 2	conflicto 4	conflicto 7	sin conflicto
	enrichment	sin conflicto	conflicto 5	sin conflicto	sin conflicto
	adding	sin conflicto	sin conflicto	sin conflicto	conflicto 9

Incluimos a continuación las definiciones de los predicados que distinguen que tipo de generalización se está aplicando entre dos Casos de Uso, utilizadas luego al expresar los conflictos:

Definición 10:
isAdding: UseCase x UseCase Æ Bool

El predicado es verdadero si la operación que especializa tiene distinto nombre que las operaciones de UC:
« UC, UC1: UseCase .
isAdding(UC, UC1) ″ oŒUC.operation . uc1.name <> o.name

Definición 11:
isEnrichment: UseCase x UseCase Æ Bool

El predicado es verdadero si la especialización agrega secuencias de acciones a una operación de UC.
« UC, UC1: UseCase . isEnrichment(UC, UC1) ´ uc1.stereotype = <<*enrichment*>>

Definición 12:
isRedefine: UseCase x UseCase Æ Bool

El predicado es verdadero si la especialización redefine una operación de UC.
« UC, UC1: UseCase . isRedefine(UC, UC1) ´ uc1.stereotype = <<*redefine*>>

A continuación incluimos el desarrollo de cada situación conflictiva en particular:

[conflicto 1] La combinación de extensiones es aplicable, por lo que esta situación no es conflictiva sino de precaución: Si las secuencias de UC1 contienen puntos de extensión de ext2, el resultado de la combinación tendrá más extensiones que las esperadas.
Warning (UC ³ $_{ext1}$ UC1, UC ³ $_{ext2}$ UC2) ´ $s1Œuc1.actionSequence. $a Œs1 Ÿa Œext2.extensionPoint

[conflicto 2] La combinación es aplicable, pero se genera una situación de precaución: Si la operación de UC que se redefine contiene puntos de extensión de ext, el resultado de la combinación no será el esperado.
Warning (UC ³ $_{ext}$ UC1, UC ³ $_{gen}$ UC2) ´ (isRedefine(UC, UC2) Ÿ($oŒUC.operation . o.name= uc2.name . $sŒo.actionSequence. $a Œs Ÿa Œext.extensionPoint))

[conflicto 3] Para todo punto de extensión en ext debe existir una acción que coincida con él dentro del Caso de Uso generalizado, para que la extensión sea aplicable.
Conflict (UC ³ $_{gen}$ UC1, UC ³ $_{ext}$ UC2) ´ (isRedefine(UC, UC1) Ÿ($iŒext.extensionPoint. « oŒ (UC ³ $_{gen}$ UC1).operation . « sŒo.actionSequence. i œs))

En la composición de generalización con extensión, también se producen situaciones de precaución, definidas como caso 1 y caso 2 respectivamente:

[conflicto 3] Si en la operación que se redefine existe algún punto de extensión de ext, se produce una situación de precaución, pues la extensión, luego de la redefinición podría no dar los mismos resultados.

[conflicto 6 y 8] La combinación es aplicable, por lo que estas situaciones no son conflictivas sino de precaución: Si las secuencias que se agregan de UC1 contienen puntos de extensión de ext, el resultado de la combinación no será el esperado.

Warning (UC ³gen UC1, UC ³ext UC2) ´
Case1: (isRedefine(UC, UC1) Ÿ ($oCUC.operation . o.name= uc1.name . $sCo.actionSequence. $a Cs a CeExt.extensionPoint))
¦
Case2: ((isEnrichment (UC, UC1) ¦ isAdding (UC, UC1)) Ÿ($sCuc1.actionSequence. $a Cs Ÿa CeExt.extensionPoint))

[conflicto 4, 5 y 7] Precaución: Las operaciones de UC1 y UC2 que especializan a UC no deben tener el mismo nombre.

Warning (UC ³gen UC1, UC ³gen UC2) ´
((isRedefine(UC, UC1) ŸisRedefine(UC, UC2)) ¦
(isEnrichment(UC, UC1) ŸisRedefine(UC, UC2)) ¦
(isRedefine(UC, UC1) ŸisEnrichment(UC,UC2))) Ÿuc1.name=uc2.name

[conflicto 9] Las operaciones de UC1 y UC2 que especializan a UC no deben tener el mismo nombre.

Conflict (UC ³gen UC1, UC ³gen UC2) ´ ((isAdding(UC, UC1) ŸisAdding(UC, UC2)) Ÿuc1.name = uc2.name)

Composiciones no conflictivas

No se produce conflicto en los casos en que se combinan generalizaciones y una de las evoluciones agrega una nueva operación, ya que la otra, por ser aplicable, enriquece o redefine una operación ya existente. El caso en que se enriquece una operación en ambas evoluciones tampoco es conflictivo, aunque se trate de la misma operación.

ŸConflict (UC ³gen UC1, UC ³gen UC2) ´
((isAdding(UC, UC1) ŸisRedefine(UC, UC2)) ¦
(isAdding(UC, UC1) ŸisEnrichment(UC, UC2)) ¦
(isRedefine(UC, UC1) ŸisAdding(UC, UC2)) ¦
(isEnrichment(UC, UC1) ŸisAdding(UC, UC2)) ¦
(isEnrichment(UC, UC1) ŸisEnrichment(UC, UC2)))

Cuando la primer evolución extiende y la segunda agrega o enriquece una operación, no hay conflicto pues el incremento nunca es referenciable por la extensión.

ŸConflict(UC ³ext UC1, UC ³gen UC2) ´(isEnrichment(UC, UC2) ¦ isAdding (UC, UC2))

7.3 Análisis de Conmutatividad en la combinación de operaciones

En los casos que no se generan conflictos o solamente se detectan situaciones de precaución, es posible enunciar bajo que condiciones existe conmutatividad de operaciones ya que en la sección anterior pudo observarse que el orden de aplicación en la composición resulta relevante en la mayoría de los casos.

Conmutatividad de extensiones: La aplicación de dos extensiones ext1(por UC1) y ext2 (por UC2) sobre un Caso de Uso UC es conmutativa, si UC1 no contiene puntos de extensión de ext2 y UC2 no contiene puntos de extensión de ext1.

(UC ³$_{ext1}$ UC1, UC ³$_{ext2}$UC2) = (UC ³$_{ext2}$ UC2, UC ³$_{ext1}$UC1) ´ (« s1Œuc1.actionSequence. « a Œs1 . a œext2.extensionPoint) Ÿ(« s2Œuc2.actionSequence. « a Œs2 . a œext1.extensionPoint)

Conmutatividad de extensión con *enrichment* y con *adding*: La aplicación sobre un Caso de Uso de una extensión ext combinada con una adición o un *enrichment* es conmutativa, si el Caso de Uso que incrementa no contiene puntos de extensión de ext.

(UC ³$_{ext}$ UC1, UC ³$_{gen}$UC2) = (UC ³$_{gen}$ UC2, UC ³$_{ext}$UC1) ´
(isEnrichment(UC, UC2) ¦ isAdding (UC, UC2)) Ÿ(“s2Œuc2.actionSequence. “a Œs2 . a œ ext.extensionPoint)

8. Conclusiones y Trabajo Futuro

El Unified Modeling Language (UML) es un lenguaje gráfico, semi-formal, que ha sido aceptado como estándar para describir sistemas de software orientados a objetos. UML define varios tipos de diagramas que se utilizan para describir diferentes aspectos o vistas de un sistema. En particular, los diagramas de Casos de Uso se utilizan para capturar los requerimientos de los sistemas y guiar su proceso de desarrollo. Los distintos Casos de Uso que se definen a lo largo de un proceso de desarrollo no son independientes sino que es posible establecer relaciones entre ellos. Las principales relaciones consideradas por UML son: Generalización (*Generalization*), Inclusión (*Include*) y Extensión (*Extend*). Estas relaciones tanto como el resto de las construcciones de UML, están definidas semiformalmente, dando lugar a interpretaciones ambiguas e inconsistencias.

En este trabajo presentamos definiciones rigurosas y reglas de buena formación para poder precisar sin ambigüedad cuándo las operaciones entre Casos de Uso están bien definidas y cómo es el resultado de aplicarlas, constituyendo así un álgebra para Casos de Uso. Esto permitirá chequear consistencia, tanto al incrementar el modelo de Casos de Uso en iteraciones dentro del proceso de desarrollo de software, como al relacionar este modelo con otros. Además, realizamos la comparación entre las relaciones *extend* e *include* y llegamos a la conclusión de que no presentan diferencias significativas. Utilizando la formalización desarrollada analizamos la composición y conmutatividad de operaciones al aplicarlas sobre un Caso de Uso base, logrando la detección de situaciones conflictivas en evolución.

Las relaciones consideradas en este trabajo se limitan a agregar comportamiento a un Caso de Uso extendiendo sus secuencias de acciones. Como continuación, resulta interesante analizar otras formas de evolución para Casos de Uso que impliquen cambios diferentes, como adicionar actores, objetos jugando nuevos roles, etc. Por otro lado, el documento de especificación de UML menciona que un Caso de Uso complejo puede ser refinado mediante un conjunto de Casos de Uso más simples. Tras este refinamiento, la funcionalidad especificada por el Caso complejo debe ser completa-

mente *mapeada* a sus subordinados. Esta es otra forma de evolución a considerar en el futuro.

La composición de operaciones sobre Casos de Uso podría aplicarse como forma de evolución similar sobre otros diagramas de UML, como las colaboraciones, resultando interesante analizar la incidencia que puedan ejercer estos cambios sobre el resto de los modelos. Al respecto, en [16] distinguimos y proponemos una descripción formal para distintos tipos de relaciones de dependencia entre modelos de UML. El objetivo es aportar fundamentos formales para herramientas que realicen un análisis inteligente sobre los modelos asistiendo al ingeniero de software a través del proceso de desarrollo.

Referencias

[1] J. Araújo. Formalizing Sequence Diagrams. In: L. Andrade, A. Moreira, A. Deshpande y Stuart Kent (ed), *Proc. OOPSLA´98 Wsh., Formalizing UML. Why? How? Vancouver*, 1998.

[2] R. Back, L. Petre and I. Porres Paltor. Analysing UML Use Cases as Contract. In: *Proceedings of the UML'99 Second International Conference. Fort Collins, CO, USA, October 28-30/99. Lecture Notes in Computer Science, Springer-Verlag*, 1999.

[3] R. Breu et al. Towards a formalization of the unified modeling language. In: *Proceedings* ECOOP'97., *Lecture Notes in Computer Science vol.1241, Springer*, 1997.

[4] A. Evans, et al. Towards a core metamodelling semantics of UML. Behavioral specifications of businesses and systems, H,Kilov editor, Kluwer Academic Publishers, 1999.

[5] A. Evans, et al. Developing the UML as a formal modeling notation, In: *Proceedings of the UML'98 Beyond the notation, Muller and Bezivin editors, Lecture Notes in Computer Science vol.1618, Springer-Verlag*, 1998.

[6] R. Giandini. Documentación y evolución de componentes reusables: Contratos de reuso con semántica de comportamiento. Tesis del Magister en Ingeniería de Software, Universidad Nacional de La Plata, Argentina, http://www-lifia.info.unlp.edu.ar/~giandini.Setiembre 1999.

[7] I. Jacobson. Object-Oriented Development in an Industrial Environment. En: *Proceedings OOPSLA' 87, special issue of SIGPLAN Notices. Vol 22, N°12, pp.183-191*, 1987.

[8] I. Jacobson et al. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1993.

[9] I. Jacobson; I. Booch and G. Rumbaugh J.. The Unified Software Development Process, Addison Wesley. ISBN 0-201-57169-2, 1999

[10] S. Kim; D. Carrington, Formalizing the UML Class Diagrams using Object-Z, In: *proceedings UML´99 Conference, Lecture Notes in Computer Sciencie 1723*, 1999.

[11] A. Knapp. A formal semantics for UML interactions, In: *Proceedings of the UML´99 conference <<UML>>´99 - The Unified Modeling Language. Beyond the Standard. R.France and B.Rumpe editors, , Colorado, USA,. Lecture Notes in Computer Science 1723, Springer*. (1999).

[12] OBJECT CONSTRAINT Language. version 1.3, July 1999. Part of Unified Modeling Language (UML) Specification. OMG, http://www.rational.com, 1999

[13] G.Övergaard; K. Palmkvist. A Formal Approach to Use Cases and Their Relationships. In: *P. Muller and J. Bézivin editors, Proceedings of the UML'98: Beyond the Notation, Lecture Notes in Computer Cience 1618. Springer-Verlag*, 1999.

[14] G. Övergaard. A Formal Approach to Collaborations in the Unified Modeling Language. In: *Proceedings of the UML'99 Second International Conference. Fort Collins, CO, USA, October 28-30/99.Lecture Notes in Computer Science, Springer-Verlag*, 1999.

[15] C. Pons, G. Baum and M. Felder. Foundations of Object-oriented modeling notations in a dynamic logic framework. *Fundamentals of Information Systems*, Chapter 1. T.Polle, T.Ripke and K.Schewe Editors, Kluwer Academic Publisher, 1999.

[16] C. Pons, R. Giandini, and G. Baum. Dependency relations between models in the Unified Process. In: *Proceedings of the IWSSD*. San Diego, California, IEEE Press, 5-7 Nov. 2000.
[17] OMG Unified Modeling Language Specification, v.1.3, UML Revision Task Force, http://www.rational.com. July 1999.

[1] Para un *Classifier C*, *C.operation* es el conjunto de las operaciones de *C*. Cada operación *op* contiene un nombre y un conjunto de secuencias de acciones, *op.name* y *op.method.body*, respectivamente. Abreviaremos op.method.body con op.actionSequence.

[2] El cuerpo de un método es una expresión Procedure que especifica una posible implementación de una operación. La definición de esta expresión está fuera del alcance de UML, nosotros la interpretamos como un conjunto de secuencias de acciones.