REGLAS PARA LA GENERACIÓN AUTOMÁTICA DE CÓDIGO DEFINIDAS SOBRE METAMODELOS SIMPLIFICADOS DE LOS DIAGRAMAS DE CLASES, SECUENCIAS Y MÁQUINA DE ESTADOS DE UML 2.0

RULES FOR AUTOMATED CODE GENERATION DEFINED OVER SIMPLIFIED METAMODELS OF CLASS, SEQUENCE AND STATE MACHINE DIAGRAMS OF UML 2.0

ANDRÉS MUÑETÓN
Escuela de Sistemas, Facultad de Minas, Universidad Nacional de Colombia, sede Medellín. afmuneto@unal.edu.co

CARLOS M. ZAPATA
Escuela de Sistemas, Facultad de Minas, Universidad Nacional de Colombia, sede Medellín. cmzpata@unal.edu.co

FERNANDO ARANGO
Escuela de Sistemas, Facultad de Minas, Universidad Nacional de Colombia, sede Medellín. farango@unal.edu.co

Recibido para revisar febrero 15 de 2007, aceptado abril 04 de 2007, versión final mayo 05 de 2007

RESUMEN: La generación automática de código a partir de modelos ha sido una de las promesas parcialmente cumplidas en el desarrollo de software. La experiencia de las herramientas CASE, aún distante del automatismo absoluto, se complementa con algunos trabajos teóricos que se alejan de los estándares de modelamiento. En este artículo se proponen reglas para la generación de código a partir de metamodelos de diagramas de clases, secuencias y máquina de estados de UML. Las reglas están definidas en lógica de primer orden, permitiendo una especificación donde se evitan las ambigüedades y la necesidad de aprender un lenguaje de programación específico. Mediante un caso de estudio se representa la aplicación de las reglas de transformación, generando el código fuente de una clase en el lenguaje orientado a objetos Java.

PALABRAS CLAVE: UML, diagrama de clases, diagrama de secuencias, diagrama de estados, reglas de transformación, generación de código, metamodelos.

ABSTRACT: Automatic code generation from models has been one of the partially accomplished promises in software development. CASE Tools experiences, even so far from complete automatism, are complemented by some theoretic works, which torn apart modeling standards. In this paper we propose code generation rules from metamodels of the UML class, sequence, and state machine diagrams. The rules are defined on first-order logic, in order to allow the construction of a specification where both ambiguity and the need of learning a programming language are avoided. We also represent the application of transformation rules by means of a case study, and we generate source code of a class in the Java object-oriented programming language.

KEYWORDS: UML, class diagram, sequence diagram, state machine diagram, transformation rules, code generation, metamodels..

1. INTRODUCCIÓN

El ciclo de vida del software contempla una serie de fases, denominadas: definición, análisis, diseño, implementación, despliegue, pruebas y mantenimiento. En las cuatro primeras fases, se crean conjuntos de diagramas con niveles de abstracción que decrecen a medida que avanza el proyecto y que paulatinamente se van transformando en el código del software; este proceso se suele denominar "refinamiento". A pesar de las diferencias en los niveles de abstracción, los diagramas de las diferentes fases deben representar el mismo sistema, de manera que los conceptos del dominio del sistema que fueron identificados en la fase de definición deben conservarse aún en la implementación (véase la Figura 1). En síntesis, debe existir consistencia entre los diagramas de las diferentes fases del proyecto.

En teoría, si las fases de un proyecto son consistentes, cualquier modificación que se realice en un modelo podría verse reflejada en los demás; de esta forma, sería posible modificar el código fuente al hacer cambios en los diagramas de diseño.

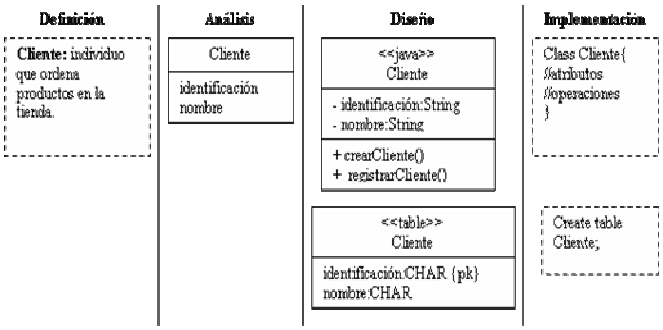

Figura 1. Consistencia en el Refinamiento de Modelos a través de las fases de desarrollo de software
Figure 1. Consistency-in-Refinement of model throughout the software development classes

La directiva MDA (Model-Driven Architecture—Arquitectura Orientada por Modelos) del Object Management Group (OMG) propone dos tipos de modelos, uno para la fase de análisis y otro para la fase de diseño, este último derivado del primero (Mellor et al., 2004). En MDA se utiliza el lenguaje unificado de modelado UML para especificar los modelos. El modelo de la fase de análisis es el PIM (Platform Independent Model), que es un modelo independiente de la plataforma, lo cual significa que los diagramas que lo componen son vistas del sistema construidas específicamente en términos del dominio. El PIM es la base para la construcción del modelo de la fase de diseño, que se conoce como PSM (Platform Specific Model), el cual está compuesto por diagramas construidos siguiendo los lineamientos no sólo de estructura y comportamiento del sistema, sino también de una plataforma específica.
La Figura 1 muestra en la fase de análisis un PIM y en la fase de diseño un PSM con diagramas de la plataforma Java y una plataforma de bases de datos. Nótese que en el PSM los conceptos del dominio del sistema (Cliente con su identificación y nombre) son acompañados de términos propios de cada plataforma utilizada, como los tipos de datos String y CHAR. Esto permite que el PSM pueda servir como marco de referencia al código fuente. Por ejemplo, la fase de implementación de la Figura 1 muestra el código que podría obtenerse del PSM.

A pesar de la relación estrecha que existe entre el PSM y la plataforma, existen diversas causas de inconsistencias que dificultan la generación de código a partir del PSM. Algunas tienen que ver con modelos mal planteados, desarrolladores incapaces de interpretar los modelos, o que no se acogen a estos para generar el código. Otras razones son la ambigüedad sintáctica y semántica de UML (Grossman et al., 2005) y la falta de reglas de transformación en las cuales existan asociaciones entre los elementos de UML y los elementos o estructuras del lenguaje de programación.
Las herramientas CASE (Computer-Aided Software Engineering) han evolucionado y se han convertido paulatinamente en herramientas que generan código de manera parcial, pero aún muy distante de un código completo que pueda ser ejecutado en una plataforma específica. Otros trabajos han definido de manera teórica cómo podría ser el código resultante de un conjunto de modelos, pero no han llegado a su implementación.
En este artículo se propone un conjunto de reglas de transformación para la obtención automática de código utilizando como punto de partida modelos PSM; se utilizan metamodelos de los diagramas de clases, secuencias y máquina de estados de la versión 2.0 de UML, como marco de referencia

conceptual para la descripción de los diagramas y expresión de las reglas de transformación. Las reglas se describen en Lógica de Predicados de Primer Orden (Morgan, 1998) y se ejemplifica su uso en un caso de estudio.

Este artículo está organizado así: en la Sección 2 se presenta el estado del arte en generación de código fuente y las limitaciones que se presentan. A partir de la Sección 3 se ofrece una propuesta de generación de código a partir de metamodelos del lenguaje unificado de UML, sobre los cuales se generan reglas de transformación a código, presentadas en la Sección 4. Estas reglas son aplicadas a un caso de estudio en la Sección 5. Finalmente, en la Sección 6 se discuten las conclusiones y trabajo futuro.

2. GENERACIÓN AUTOMÁTICA DE CÓDIGO

La generación automática de código ha sido parte de la historia de la programación. El primer compilador significó el inicio de la carrera en la búsqueda de herramientas para posibilitar la programación de computadores, en un nivel de abstracción cada vez más asequible a la interpretación humana. De esta forma, el ensamblador permitió la transición de los unos y ceros del lenguaje de máquina, a la manipulación de registros mediante operaciones identificadas con palabras de fácil recordación para el programador. A su vez, no siendo suficientemente sencillo de utilizar, el ensamblador dio paso a los lenguajes de alto nivel y por ende a herramientas que permitieron la transformación automática de los programas escritos con estos lenguajes en un programa interpretable por la máquina.

Adicional a los lenguajes de alto nivel, actualmente los equipos de desarrollo cuentan con lenguajes de modelado, los cuales posibilitan la construcción de modelos que, a diferencia del código, se convierten en herramientas de comunicación de fácil interpretación. Estos lenguajes de modelado son serios candidatos a contribuir en la evolución de la generación de código.

UML es el lenguaje de modelado de mayor uso en la actualidad. Con UML es posible modelar los aspectos estáticos y dinámicos de un sistema, y en diferentes niveles de detalle, como puede suceder con MDA y sus modelos PIM y PSM.

Ya que el PSM tiene una fuerte relación con la plataforma, puede ser un modelo de considerable utilidad en la generación automática de código a partir de un lenguaje de una abstracción superior, como lo es UML, a los lenguajes de alto nivel de la actualidad.

Kepple *et al.* (2003) definen los elementos necesarios en la transformación de un lenguaje a otro, que son un par de conjuntos que representan al lenguaje fuente y al lenguaje objetivo de la transformación, y que contienen los elementos de estos lenguajes. Además, deben existir relaciones entre los elementos de ambos conjuntos, de forma tal que todo elemento del conjunto fuente pueda encontrar su representación en al menos un elemento del conjunto objetivo. En el caso particular de UML, parte de las relaciones necesarias se han presentado en herramientas CASE comerciales y académicas, y trabajos de orden académico.

En el primer grupo, el de las herramientas CASE, se pueden mencionar Togeteher™ (Together, 2006), Rational Rose™ (Rose, 2006) y Fujaba (Fujaba, 2006; Geiger y Zündorf, 2006), esta última desarrollada en la Universidad de Panderborn, Alemania. Como característica común, estas herramientas permiten la generación de código a partir del diagrama de clases con resultados muy similares, apreciables en la estructura del código, la cual está formada por la declaración de las clases, atributos y operaciones. La obtención de código a partir de diagramas de comportamiento no es posible en Rational Rose™; mientras tanto, en Together™ se logra con diagramas de secuencias y en Fujaba con los Story Diagrams, que fueron creados por el grupo Fujaba y son la combinación de diagramas de actividades y diagramas de comunicación.

En Together™ y Fujaba™, el código generado a partir de un diagrama de comportamiento es relacionado con elementos propios de la implementación de las operaciones de las clases, esto es, el cuerpo de los métodos. Esto se debe a que en el comportamiento se modela la interacción entre los objetos del sistema, lo cual implica la creación de objetos, la invocación de métodos de otros objetos o del objeto mismo, y justamente este tipo de acciones son parte del cuerpo de los métodos. El diagrama de secuencias es un diagrama de comportamiento de UML que por su estructura y sus elementos, especialmente los fragmentos combinados que ofrecen la representación de decisiones, ciclos, excepciones, entre otras, se utiliza como una especie de "algoritmo gráfico" que

permite la generación de estructuras *if-then-else*, ciclos *for* o *while*, además de la creación de objetos y las llamadas a métodos de otros objetos o del objeto mismo.

En el segundo grupo, en Long *et al.* (2005) y Long y Zhiming (2005) se define rCOS (Relational Calculus for Object Systems), una formalización de un lenguaje orientado a objetos que permite especificar formalmente modelos UML. De manera similar en Laleau y Mammar (2005), se utiliza el Método-B como elemento de representación formal de los modelos y como punto intermedio a la generación de código. Al igual que las herramientas CASE, estos tipos de trabajo ofrecen transformaciones a partir del diagrama de clases y a partir de diagramas de comportamiento, específicamente del diagrama de secuencias.

Si se pretende generar código fuente a partir de modelos PSM, entonces tanto el PSM como el código, siendo éste el modelo de la fase de implementación, deben representar el sistema bajo estudio, en sus aspectos estáticos y dinámicos. Esto implica que un modelo PSM no debe estar compuesto únicamente por diagramas de clases, sino también por diagramas de comportamiento, como pueden ser los diagramas de secuencias y estados. Adicionalmente, las reglas de transformación de modelo a código deben estar definidas en una especificación que no de lugar a ambigüedades y deben poder ser programables para su uso. Tanto Together™ como Fujaba™, utilizan diagramas de comportamiento y de estructura para la generación de código fuente; ambas herramientas incluyen adiciones a los diagramas, no pertenecientes al estándar UML, que tienen como finalidad facilitar la obtención de código de ciertas estructuras de programación como los ciclos de repetición, las estructuras de decisión, o como ocurre en Fujaba™, sentencias como la declaración de variables y operaciones aritméticas entre variables, entre otras. El uso de elementos por fuera del estándar, ocasiona que el equipo de desarrollo dependa únicamente de la herramienta que implemente estos elementos y que se deba adaptar a los cambios sintácticos ocasionados en el lenguaje debido a estas adiciones (la migración a otra herramienta podría implicar la repetición de todos los modelos), lo cual puede complicar la comunicación con los miembros del equipo de desarrollo o miembros externos de apoyo al proyecto.

Las reglas de transformación ofrecidas por los trabajos enfocados en la formalización de UML se enfocan en sistemas de bases de datos, por ejemplo Laleau y Mammar (2005), y se utilizan para obtener código genérico, como la inserción de datos en una tabla; en otros casos, estas reglas están definidas en términos del lenguaje utilizado para formalizar UML y no sobre el lenguaje mismo, dificultando el entendimiento de las reglas por parte de los desarrolladores. Adicionalmente, no se ofrecen aún herramientas que permitan validar las reglas presentadas.

En este artículo se proponen reglas de transformación para PSMs compuestos de diagramas de carácter estructural, como el diagrama de clases, y de carácter dinámico como el diagrama de secuencias y el diagrama de máquina de estados. Las reglas están definidas sobre el metamodelo de UML, con lo cual se dispone de una estructura abstracta, independiente de cualquier modelo de usuario, y que además ofrece vínculos entre los diferentes diagramas utilizados en la generación de código. Estas reglas se especifican en cálculo de predicados de primer orden (Morgan, 1998), lo que las hace fáciles de entender y suficientemente genéricas para ser programadas y útiles en la generación de código en diferentes lenguajes de programación.

3. METAMODELOS UML COMPRIMIDOS

La especificación de UML, compuesta por la superestructura (OMGa, 2006) y la infraestructura (OMGb, 2006), define la sintaxis abstracta del lenguaje de modelado, presenta restricciones definidas en OCL que se aplican a la sintaxis y describe en lenguaje natural la semántica de UML.

La sintaxis del lenguaje, definida en UML, se conoce como metamodelo de UML y sus componentes como metaclases. Estas metaclases están relacionadas unas con otras, indicando qué elementos puede tener cada diagrama UML y cómo pueden relacionarse estos elementos. El metamodelo es complementado con restricciones especificadas en OCL.

La sintaxis de UML ofrece una extensa jerarquía de herencia de metaclases, en la cual se agregan atributos, relaciones y restricciones en cada nivel. Esto quiere decir que para conocer todos los

posibles atributos y relaciones que puede tener una metaclase, por ejemplo la metaclase que representa las clases, es necesario revisar cada uno de los niveles de la jerarquía de generalización (que puede ser múltiple), y las relaciones entre las metaclases que aparecen en cada nivel. Por ejemplo, el atributo *name* de la metaclase *Class*, es un atributo heredado de la metaclase *NamedElement*, la cual está cuatro niveles arriba en la jerarquía.

Una solución a esta complejidad es comprimir los metamodelos de los diagramas UML seleccionando un conjunto de metaclases representativas (según criterios mencionados adelante en esta Sección) de cada diagrama y que contengan los atributos y relaciones de las metaclases que especializaron, esto es, de sus metaclases madres. A estos nuevos metamodelos, propuestos en este artículo para facilitar la traducción de los modelos a código, se les denominará en adelante "Metamodelos Comprimidos".

A continuación se proponen los metamodelos comprimidos de los diagramas de clases, secuencias y máquina de estados que servirán como fuente de definición de las reglas de transformación al código.

3.1 Metamodelo del Diagrama de Clases

La Figura 2 presenta el metamodelo comprimido del diagrama de clases de UML. Los elementos más comunes de este diagrama son las clases, con sus atributos y operaciones, y las relaciones entre clases, que incluyen la generalización y la asociación, con las variaciones que ésta puede tener como es la composición y la agregación. Por este motivo, se eligieron como elementos representativos del diagrama de clases las metaclases presentadas en la Figura 2.

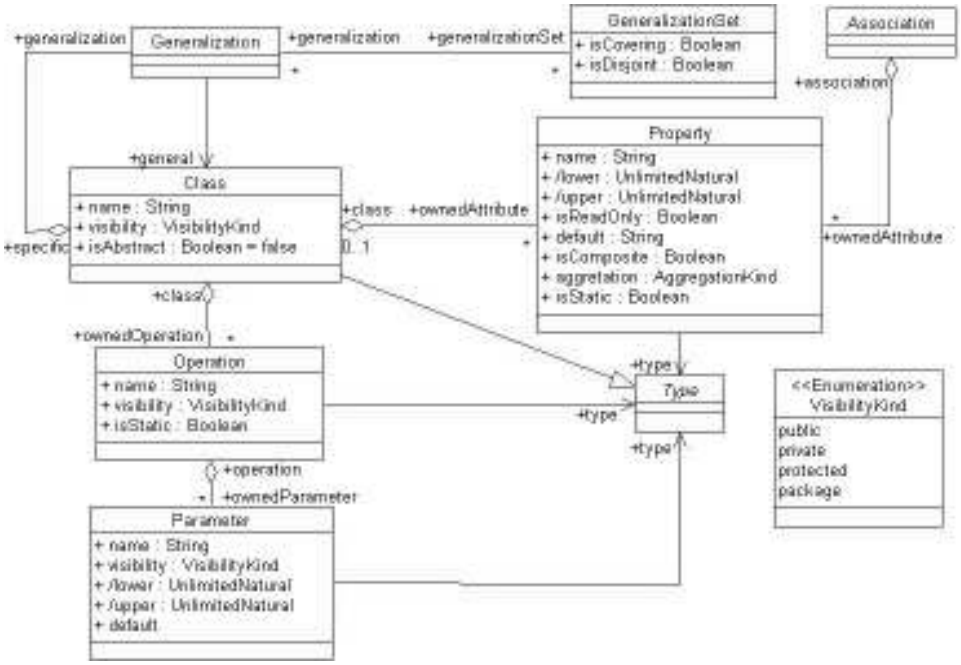

Figura 2. Metamodelo comprimido del Diagrama de Clases
Figure 2. Compressed Metamodel of the Class Diagram

Es importante tener en cuenta los atributos (multiplicidad y rol) de las relaciones entre las metaclases, y así se verá más adelante en las reglas de transformación. Por el momento, es necesario considerar que:

- Toda instancia de la metaclase *Class* tiene asociado un conjunto de atributos y otro de operaciones, a los cuales se puede acceder utilizando los atributos de *Class ownedAttributes* y *ownedOperations* respectivamente.
- Si una clase es la especialización de otra, la relación entre ambas clases se dará mediante una instancia de la metaclase *Generalization*. Por la navegabilidad de las relaciones, una clase general (véase rol *general* en la Figura 2) no puede tener acceso a las clases que la

especializan pero, en cambio, las clases especializadas pueden tener acceso a la clase general utilizando los atributos *generalization* y *general* de la relación entre instancias del metamodelo así:

ClaseGeneral = self.generalization.general

Donde *self* es la clase especializada que desea saber cuál es su clase madre.

- Si hay más de una clase especializada, la metaclase *GeneralizationSet* pemite identificar el conjunto de clases que hacen parte de una misma generalización.

El metamodelo de UML no establece una relación de generalización directa entre las metaclases *Type* y *Class*, pero se presenta acá como tal, dando a entender que el tipo de un atributo o una operación puede ser una clase.

La Figura 3 presenta una instancia del metamodelo del diagrama de clases. La instancia a1 de la metaclase *Association* representa la relación entre las clases *Customer* y *Order*. Nótese que la asociación no se relaciona directamente con las clases sino con propiedades cuyos tipos son las clases de la relación. Estas propiedades, *p5* y *p6*, equivalen a los roles en el diagrama de clases.

En la Figura 3 también se puede ver que la clase *Order* tiene una operación llamada *setNumber*, con un parámetro llamado *number* de tipo *String*.

3.2 Metamodelo del Diagrama de Secuencias

El metamodelo de la Figura 4, muestra las metaclases del diagrama de secuencias necesarias para generar modelos de interacción entre objetos mediante el paso de mensajes para la invocación de operaciones y el uso de fragmentos combinados. Estos fragmentos, representados por la metaclase *CombinedFragment*, permiten modelar interacciones especiales como ciclos, decisiones, excepciones, entre otras.

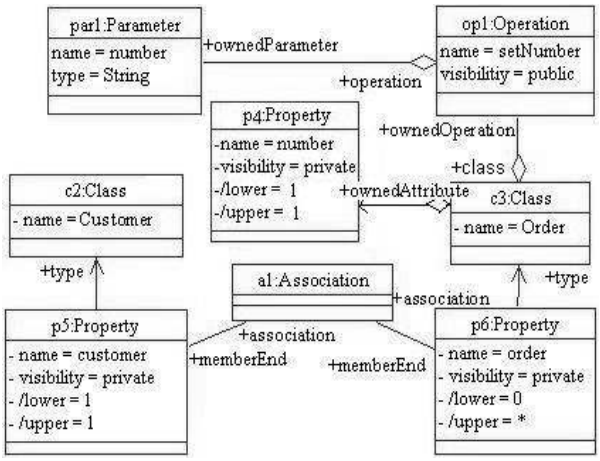

Figura 3. Instancia del metamodelo del diagrama de clases

Figure 3. Metamodel Instante of the Class Diagram

Las líneas de vida del diagrama de secuencias están representadas por las instancias de la metaclase *Lifeline*, vinculadas con los mensajes que componen la interacción mediante la metaclase *OcurrenceSpecification*. Las instancias de esta metaclase son los puntos de enlace entre el mensaje y la línea de vida, por lo que, como se ve en la Figura 4, todo mensaje tiene relación con dos *OcurrenceSpecification*, una que indica el origen del mensaje y otra el destino.

La Figura 5 muestra una instancia del metamodelo del diagrama de secuencias. La interacción representada tiene dos líneas de vida, *lf1* y *lf2*, las cuales están interactuando mediante el mensaje *m1*, que transporta la operación *op1*. El valor del parámetro de *op1* está dado por el atributo *symbol* de la instancia de la metaclase *Expression*.

Debido a que la línea de vida *lf1* está relacionada con la *OcurrenceSpecification* marcada con el rol *sendEvent*, se puede deducir que esta línea de vida es quien está enviando el mensaje *m1*. Similarmente, se deduce que *lf2* es la línea de vida que recibe el mensaje por tener relación con *oc2*, la instancia de *OcurrenceSpecification* marcada con el rol *receiveEvent*.

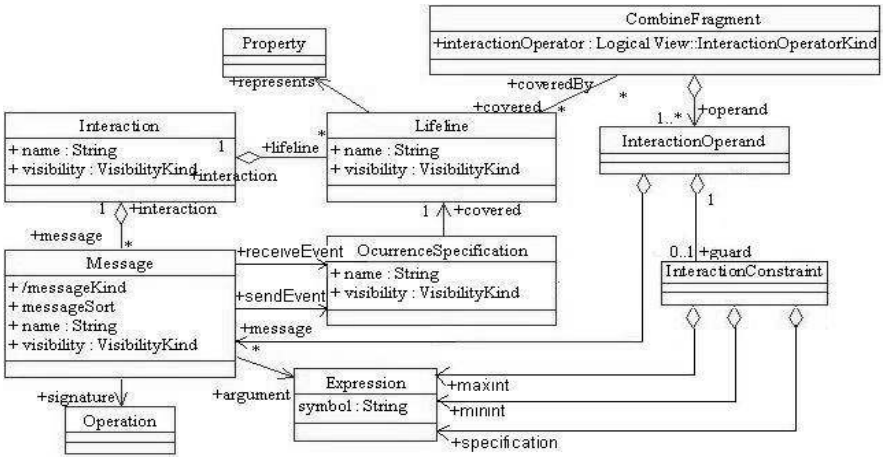

Figura 4. Metamodelo Comprimido del Diagrama de Secuencias
Figure 4. Compressed Metamodel of the Sequence Diagram

Es importante anotar el vínculo formado entre los metamodelos de secuencias y clases gracias a la presencia en ambos de las metaclases Property y Operation. Al existir esta relación, es posible conocer las clases propietarias de las operaciones transportadas en los mensajes y/o validar la consistencia entre los modelos. Por ejemplo, en la Figura 5 aparecen las instancias c2 y c3 de *Class*, también presentes en la instancia del metamodelo del diagrama de clases presentado en la Figura 3.

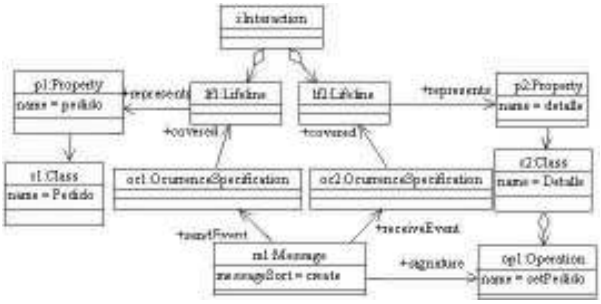

Figura 5. Instancia Metamodelo del Diagrama de Secuencias
Figure 5. Metamodel Instance of the Sequence Diagram

El metamodelo de la Figura 4, además de comprimido, se modificó adicionándole la relación entre las metaclases *InteractionOperand* y *Message*. La relación entre estas metaclases, la cual no existe en la especificación de UML (OMG1, 2006) se propone como necesaria para la generación de código a partir de modelos que incluyan fragmentos combinados. En un diagrama de secuencias, un fragmento combinado actúa como agrupador de mensajes sobre los cuales tendrá efecto de acuerdo a su tipo. Por ejemplo, si es un fragmento combinado de tipo *alt*, entonces se tendrán dos conjuntos mutuamente excluyentes de mensajes; uno será tenido en cuenta sí y sólo si se cumple la condición booleana asociada al fragmento combinado, y el otro en el caso contrario (véase la Figura 10).

El metamodelo del diagrama de secuencias relaciona las metaclases *CombinedFragment* e *InteractionOperand*, lo cual permite especificar cuántos conjuntos de mensajes conforman un fragmento combinado (cuántas instancias de *InteractionOperand* están asociadas con el *CombinedFragment*) y las condiciones que representan a cada conjunto (mediante la relación entre *InteractionOperand* e *InteractionConstraint*). A pesar de esto, el metamodelo no ofrece una relación que permita deducir cuáles mensajes hacen parte de estos conjuntos.

El único vínculo entre *InteractionOperand* y *Message* se presenta mediante la metaclase *Gate*. Las instancias de *Gate* actúan como reemplazo de *OcurrenceSpecification* si un mensaje tiene alguno de sus extremos en un fragmento combinado; en otras palabras, dichas instancias sirven de vínculo entre el fragmento combinado y su entorno. Debido a esto, no es posible considerar los mensajes contenidos en el fragmento combinado.
En conclusión, es necesario establecer una relación entre la metaclase que representa cada conjunto de mensajes del fragmento combinado, la cual es *InteractionOperand*, y los mensajes, representados por la metaclase *Message*.

3.3 Metamodelo del Diagrama de Máquina de Estados

La Figura 6 muestra un diagrama de máquina de estados general, con dos estados: Estado0 y Estado1 y los elementos que pueden tener estos estados y la transición que los vincula.

La Figura 7 muestra el metamodelo comprimido del diagrama de máquina de estados. Este metamodelo dispone de las metaclases necesarias para desarrollar modelos con los elementos de la Figura 6. Los estados están representados por la metaclase *State*, excepto por el estado final que es un pseudoestado de tipo *initial*, y el estado final que es instancia de la metaclase *FinalState*, una especialización de *State*. Los bloques *entry*, *doActivity* y *exit* son instancias de la metaclase *FunctionBehavior* y las transiciones son instancias de la metaclase *Transition* o *ProtocolTransition*. Si la transición tiene una lista de operaciones, entonces es instancia de *ProtocolTransition*; de lo contrario, es instancia de *Transition*. La metaclase *Trigger* representa los eventos y *Constraint* el guarda de la transición. La metaclase *Region* es un contenedor de estados y transiciones, y un conjunto de regiones componen una máquina de estados representada por la metaclase *StateMachine*.

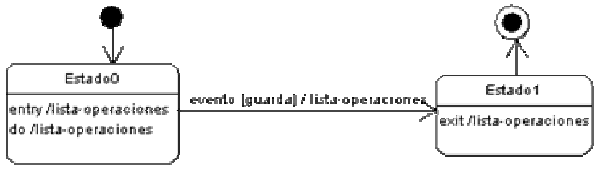

Figura 6. Diagrama de máquina de estados general
Figure 6. General State Machine Diagram

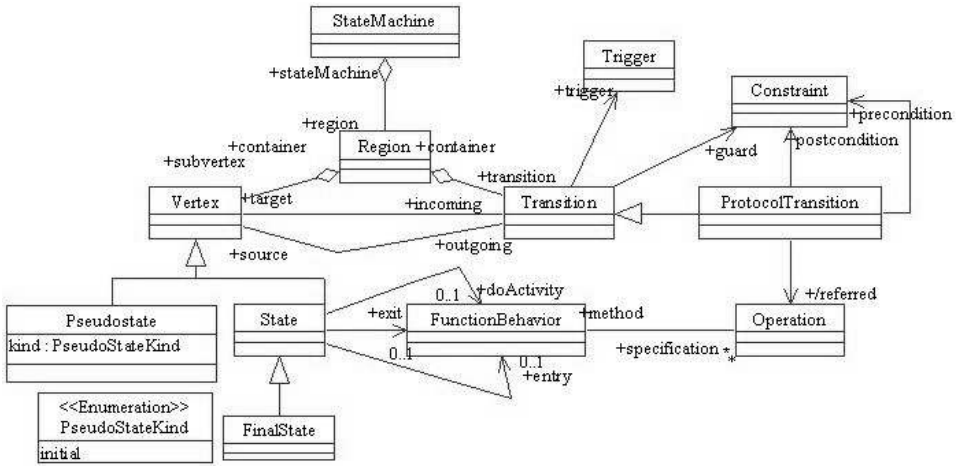

Figura 7. Metamodelo Comprimido del Diagrama de Máquina de Estados
Figure 7. Compressed Metamodel of the State Machine Diagram

Nótese que, al igual que el metamodelo del diagrama de secuencias, este metamodelo también está vinculado con el metamodelo del diagrama de clases mediante la metaclase *Operation*, posibilitando nuevamente el chequeo de consistencia con los demás diagramas.

4. REGLAS DE TRANSFORMACIÓN Y DE INTEGRACIÓN

A partir de los metamodelos presentados en la Sección anterior, se proponen reglas que permiten relacionar cada instancia del metamodelo, es decir,

cada diagrama convencional, con el código fuente que puede obtenerse a partir de éste. Las reglas son expresadas en lógica de primer orden con el fin de ofrecer una solución general adaptable a diferentes lenguajes de programación orientado a objetos como Java o C#.

Cada regla está compuesta por relaciones condicionales, en las cuales el consecuente representa el código que se puede obtener del elemento sobre el cual fue aplicada la regla. Este código se presenta para cada diagrama en los elementos que lo componen utilizando la sintaxis definida por el OMG para los diagramas UML.

4.1 Reglas de Transformación a Partir del Diagrama de Clases

El principal elemento del diagrama de clases es la clase. Por este motivo, las reglas de transformación se definen a partir de este elemento. La Figura 8 presenta una clase con la representación tradicional de UML, pero el contenido de cada casilla (nombre de la clase, atributos, operaciones) se especifica con la sintaxis definida por el OMG.

Se presenta una única clase, sin asociaciones con otras clases, considerando que las asociaciones son modificadoras de la definición de las clases que vinculan. Así, la relación de herencia modifica la definición de la clase y una relación de asociación con otra clase implica la aparición de un nuevo atributo en alguna o todas las clases relacionadas.

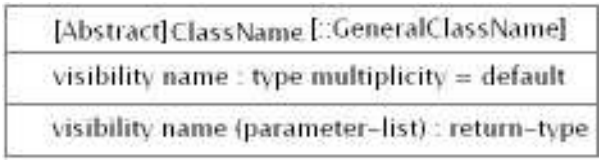

Figura 8. Sintaxis de una clase
Figure 8. Syntax of a Class

Cada casilla está compuesta por elementos sintácticos. La primera casilla, por ejemplo, indica que el nombre de una clase tiene un identificador llamado *ClassName* que puede estar acompañado de la palabra *Abstract* si la clase es abstracta, y/o del símbolo :: seguido del nombre de la clase general de la clase actual, en caso de existir una relación de generalización.

Las reglas que permiten obtener, para un diagrama de clases dado, la estructura sintáctica presentada en la Figura 8 son las siguientes:

R1_ABS - Regla esAbstracta: Todas las instancias de la metaclase *Class* cuyo atributo *isAbstract* sea verdadero, serán clases abstractas.

$\forall c \in Class(c.isAbstract \Rightarrow$ 'abstract')

R2_NOMC - Regla NombreClase: El atributo *name* de las instancias de la metaclase *Class*, será el nombre de la clase.

$\forall c \in Class \Rightarrow c.name$

R3_GEN - Regla claseGeneral: Si la clase *c* es el extremo *specific* de una relación de generalización *g*, entonces *c* tiene una clase general y esta clase es el extremo *general* de la generalización *g*.

$\forall c \in Class((\exists g \in Generalization \cdot g.specific = c) \Rightarrow g.general)$

R4_ATTRP - Regla atributos propios de la clase: Los atributos propios de una clase son aquellos que se obtienen por la relación entre las superclases *Class* y *Property*. Una propiedad *p*, instancia de la metaclase *Property*, es atributo de una clase *c* si el atributo *class* de *p* es igual a *c*. La multiplicidad, visibilidad, tipo y nombre de *p* están dados por sus atributos *upper*, *visibility*, *type* y *name* respectivamente.

Si la multiplicidad es mayor que 1, la clase *c* tendrá un conjunto de atributos *p*. Si la multiplicidad es igual a 1, entonces la clase *c* tendrá un sólo atributo *p*.

$\forall c \in Class \forall p \in Property \cdot p.class = c \Rightarrow$
(
//si la multiplicidad es 1, el tipo permanece sin cambios
p.upper=1 $\Rightarrow$(p.visibility p.type.name p.name ';')
$\vee$
//si la multiplicidad es mayor que 1, el tipo es un conjunto de elementos de Type
p.upper>1 $\Rightarrow$ (p. visibility (p.type.name)$\mathbb{P}$ p.name ';'
)

R5_ATTRA - Regla atributos por asociación

Considere el modelo de la Figura 9. La relación entre las clases Pedido y Cliente indica que la clase Remisión tendrá un atributo de tipo Cliente cuyo nombre es cliente, y de igual forma, la clase Cliente tendrá un atributo de tipo Pedido cuyo nombre es pedido. Sin embargo, nótese en la parte inferior de la Figura 9, que no hay relación entre la clase de un extremo de la asociación y el atributo del otro extremo. La especificación UML, por lo tanto, no considera ni gráficamente, ni mediante restricciones sobre las clases, la relación acá señalada; por ello, se deben obtener los atributos de una clase derivados de una relación, indirectamente navegando sobre las relaciones entre las instancias que forman la relación.

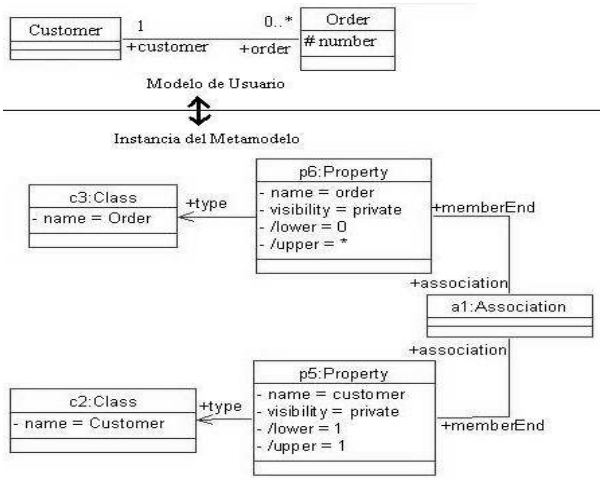

Figura 9. Relación entre un modelo de usuario y una instancia del metamodelo

Figure 9. Relationship between a user model and a metamodel instance

Utilizando la relación entre las propiedades y la asociación, es posible obtener para cada clase un atributo cuyo tipo es la clase del extremo contrario. Por ejemplo, el atributo cliente de la clase Pedido se obtendrá a partir de la asociación a y su relación con la propiedad *p2* llamada *ownedEnd*. Teniendo *p2* es posible tener su nombre, tipo (relación *type*), visibilidad, y demás elementos requeridos para obtener atributos de una clase. Esto se representa de forma general así:

$\forall c \in Class(\exists a \in Association \cdot (\exists p \in a.ownedAttributes).type = c \wedge |a.\ ownedAttributes|=2)$

$\Rightarrow$

(

//si la multiplicidad es 1, el tipo permanece sin cambios

p'.upper=1 ⇒(p'.visibilidad p'.Type.name p'.name ';')

$\vee$

//si la multiplicidad es mayor que 1, el tipo es un conjunto de elementos de Type

p'.upper>1 ⇒ (p'.visibilidad (p'.Type.name)$\mathbb{P}$ p'.name ';'

) ·(p' = ownedAttributes - {p})

En la regla anterior, se selecciona una asociación en la cual al menos uno de sus dos extremos tiene como tipo la clase actual. Luego, para obtener la visibilidad, tipo, nombre y valor por defecto del atributo, se obtiene el otro extremo de la asociación mediante la especificación p' = *ownedAttributes* - {p}, donde *ownedAttributes* es el conjunto de propiedades relacionadas con la asociación.

En esta regla no se consideran las clases de asociación, motivo por el cual se evalúa inicialmente que la asociación tenga dos extremos.

R6_OP - Regla operaciones propias de la clase: Las operaciones de una clase se obtienen a partir de la relación entre las metaclases *Class* y *Operation*. El tipo de la operación se obtiene de la relación entre las metaclases *Operation* y *Type*; y los parámetros se obtienen de la relación entre *Operation* y *Parameter*.

Una operación *o*, instancia de la metaclase *Operation*, es operación de la clase *c* si el atributo *class* de *o* es igual a *c*. La visibilidad, tipo y nombre de la operación *o* está dado por sus atributos *visibility*, *type* y *name* respectivamente.

Un parámetro *ownedParameter*, instancia de la metaclase *Parameter*, es parámetro de la operación *o* si el atributo *operation* de *ownedParameter* es igual a *o*. El tipo y nombre de *ownedParameter* están dados por sus atributos *type* y *name* respectivamente.

Para todas las operaciones *o* de *c,* se selecciona la visibilidad, tipo y nombre de la operación. Además, se seleccionan todos los parámetros de la operación *o*.

$\forall c \in Class \forall o \in Operation \cdot o.class = c \Rightarrow$

(

```
//visibilidad tipo nombre (parámetros)
o.visibility o.type.name o.name '('
        ∀ownedParameter∈Parameter ·
        ownedParameter.operation = o ⇒
        //tipo parámetro nombreParametro
        ownedparameter.type.name
        ownedparameter.name'){'
'}'
)
```

4.2 Reglas de Transformación a Partir del Diagrama de Secuencias.

R7_NEWO - Regla Creación de Objetos: La creación de objetos se representa en el diagrama de secuencias con los mensajes de tipo *createMessage*, que gráficamente se representan con la palabra clave <<*create*>>. La siguiente regla permite generar código de creación de objetos. El tipo y nombre del objeto que se desea crear se obtiene a partir de las líneas de vida, o instancias de la metaclase *Lifeline*.

Si *m*, instancia de la metaclase *Message*, es de tipo *createMessage*, entonces se debe crear un objeto cuyo tipo y nombre se obtienen a partir del atributo *covered.represents* de *m*. Los argumentos del constructor del objeto son los argumentos de *m* dados por su atributo *argument*.

```
∀m∈Message · m.messageSort =
MessageSort.createMessage ⇒
//tipo variable
m.receiveEvent.covered.represents.type.name
//nombre objeto
m.receiveEvent.covered.represents.name
//operador new
'= new '
//constructor
m.receiveEvent.covered.represents.type.name '('
//argumentos del constructor
∀e∈Expression · e = m.argument ⇒ e.symbol ');'
```

R8_CALLOP - Regla invocación de operaciones: La operación invocada por un objeto puede pertenecer al objeto mismo o a otro. En este último caso, la regla valida si la visibilidad de la operación es privada, lo cual impide que sea accedida por objetos diferentes al que la contiene. No se considera el alcance de los modificadores de acceso *public* y *protected*, por lo cual se considera como accesible toda operación que tenga alguno de estos modificadores.

Un mensaje *m* es enviado desde una línea de vida referenciada por el atributo *sendEvent* de *m*, y es recibido por una línea de vida referenciada por el atributo *receiveEvent* de *m.*

Si las líneas de vida de envío y recepción de *m* son diferentes, entonces la operación *o,* enviada por *m* pertenece al objeto representado por la línea de vida de recepción, es decir, por *receiveEvent*. Si son iguales, entonces el objeto representado por las líneas de vida es el mismo.

El tipo y nombre de la operación *o* están dados por sus atributos *type* y *name* respectivamente. Los argumentos pasados a *o* son los argumentos de *m* dados por su atributo *argument*. Si la operación tiene tipo, entonces retorna un valor cuyo tipo es el mismo tipo de la operación.

```
∀m∈Message ·
(
//operación de otro objeto
~(m.sendEvent = m.receiveEvent) ∧ (∃o∈
m.receiveEvent.covered.represents.type.ownedOperat
ion · o = m.signature) ∧ ~(o.visibility=private) ⇒
(~(o.type = Ø) ⇒
//tipo
o.type.name 'var_'+o.type.name '=') ∧
//invocación de la operación
m.receiveEvent.covered.represents.name '.'
m.signature.name '('
//parámetros de la operación invocada
∀e∈Expression · e ∈ m.argument ⇒
e.symbol'); '
)
V
 (
//operación del objeto actual
(m.sendEvent = m.receiveEvent ∧ (∃o∈
m.receiveEvent.covered.represents.type.ownedOperat
ion · o = m.signature)) ⇒
~(o.type = Ø) ⇒
//tipo
o.type.name 'var_'+o.type.name '=') ∧
//invocación de la operación
self '.' m.signature.name '('∀e∈Expression · e ∈
m.argument ⇒
e.symbol'); '
)
```

La disyunción de la regla permite discernir entre la invocación de operaciones de otro objeto, o del objeto mismo que envía el mensaje, caso en el cual se utiliza el operador *self* para hacer referencia al objeto invocador.

R9_IF - Regla Sentencias de Decisión (if – then – else): La Figura 10 muestra un diagrama de secuencias en el cual se utiliza un fragmento combinado alt para decidir si se debe invocar la operación *op1()* o la operación *op2()* de acuerdo a una condición booleana dada por *guard*.

La siguiente regla aprovecha las características del fragmento combinado *alt* para generar código que contenga la estructura de decisión *if-then-else*.

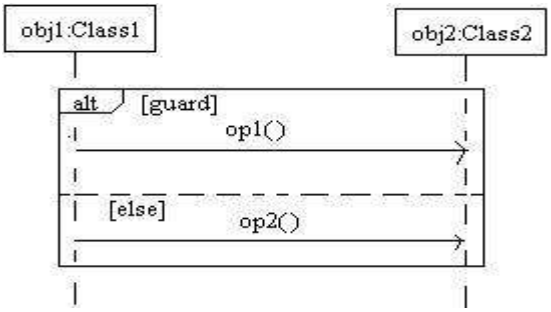

Figura 10. Fragmento Combinado "Alt"
Figure 10. The "Alt" Combined Fragment

El tipo de un fragmento combinado *f* está dado por su atributo *interactionOperator*. Si el tipo de *f* es *alt*, entonces *f* estará compuesto por instancias *o, o', ..., o^n*, de la metaclase *InteractionOperand*, que representan la estructura *if-then-else*. La condición *if* está representanda por una expresión booleana, llamada guarda que es referenciada por el atributo *guard* de *InteractionOperand*.

Las operaciones de cada bloque de la estructura *if-then-else* son obtenidas de los mensajes enviados entre las líneas de vida.

∀f∈CombinedFragment · f.interactionOperator =
InteractionOperadorKind.alt ⇒
(∃o∈InteractionOperand · (o∈f.operand ∧
~(o.guard.specification.symbol = 'else')) ∧
∃o'∈InteractionOperand · (o∈f.operand ∧
o.guard.specification.symbol = 'else')) ⇒ 'if('
o.guard.specification.symbol ') {'
//mensajes contenidos en el if
∀m∈Message · m∈o.message ⇒
< R8_CALLOP>
'}'
'else {'
//mensajes contenidos en el else
∀m'∈Message · m'∈o'.message ⇒
< R8_CALLOP>
'}'

Nótese que luego de seleccionar el fragmento combinado *alt*, la regla hace una distinción entre el conjunto de mensajes que hacen parte del *if* y el conjunto de mensajes que hacen parte del *else*. Esto se logra gracias a que el fragmento combinado está compuesto por dos *InteractionOperand*, uno que representa al *if*, y otro que representa al *else*, y cada *InteractionOperand* está relacionado con el conjunto de mensajes que lo componen.

La Figura 11 muestra la interacción contenida en el fragmento combinado de la Figura 10, como instancia del metamodelo del diagrama de secuencias. La instancia *m1* de *Message*, está asociada con el *InteractionOperand* que tiene la restricción *guard*, mientras el mensaje *m2* está relacionado con el *InteractionOperand* que representa al *else*.

R10_LOOP - Regla Ciclos de Repetición: Para obtener los ciclos de repetición *while* y *for* se sigue la misma lógica de la regla para obtener estructuras de decisión, excepto por el fragmento combinado, que ahora será *loop*, y que en el caso específico del *for*, deben seleccionarse los límites inferior y superior de la iteración, utilizando las relaciones *minint* y *maxint* entre *InteractionConstraint* y *Expression*.

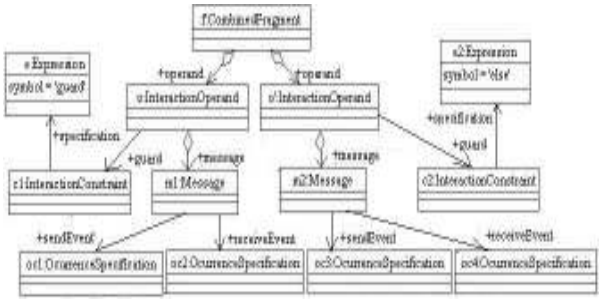

Figura 11. Instancia del metamodelo del diagrama de secuencias incluyendo el Fragmento Combinado "Alt"
Figure 11. Metamodel Instance of the Sequence Diagram including the "Alt" Combined Fragment

4.3 Reglas de Transformación a Partir del Diagrama de Máquina de Estados

La regla de transformación a partir del diagrama de máquina de estados permite generar el código necesario para administrar la transición de estados de los objetos de una clase particular y lo que esta transición implica, como es la verificación de las condiciones de guarda y la ejecución de las operaciones indicadas en la transición y en los estados actual y nuevo.

En la regla se parte del supuesto de que se ha generado un evento y continúa con los siguientes pasos:

- Determina si el evento generado pertenece al estado s actual y además, que la condición guarda sea verdadera.
- Selecciona las operaciones *exit* del estado actual, que deben ser ejecutadas antes del cambio de estado.
- Establece el nuevo estado del objeto, verificando que la transición que contiene al evento disparado es la transición entrante al nuevo estado.
- Selecciona las operaciones *entry* y *doActivity* del nuevo estado.

R11_STATES – Regla de Transformación del diagrama de máquina de estados: Al generarse un evento *e*, se selecciona la transición de salida *t* del estado *s* que contenga al evento generado. El evento de una transición es referenciado por su atributo *trigger*.

Si *t* tiene una condición para el cambio de estado, y ésta es verdadera, entonces se ejecutan las operaciones de salida de *s*, referenciadas por el atributo *exit* de *s*.

Las operaciones de la transción *t* están dadas por el atributo *referred* de *t*.

El estado de destino de una transición se obtiene evaluando la igualdad entre las transiciones de salida de un estado *s* y las transiciones de entrada de un estado *s′*. Las operaciones de entrada del nuevo estado *s′* y las operaciones *doActividy* están dadas por los atributo *entry* y *doActivity* de *s′* respectivamente.

Sea *e* un evento *Event* generado, entonces:

$\forall s \in State \bullet (\exists t \in (s.outgoing\ \mathbb{P}) \bullet t.trigger = e \wedge t.guard=true) \Rightarrow$
//operaciones exit
$\forall o \in Operation \bullet o \in (s.exit.specification\ \mathbb{P}) \Rightarrow$
//regla de creación de operaciones
<R6_OP>
//invocación de la operación creada
s.exit..specification.name'(' ');'
//nuevo estado
$\exists s' \in State \bullet (\exists t' \in (s'.incoming\ \mathbb{P}) \wedge t' = t)$
$\Rightarrow$
//operaciones de la transición
$\forall o' \in Operation \bullet o' \in (t.referred\ \mathbb{P}) \Rightarrow$
//regla de creación de operaciones
<R6_OP>
//invocación de la operación creada
t.referred.name '(' ');'

//operaciones entry del nuevo estado
$\forall o'' \in Operation \bullet o'' \in (s'.entry.specification\ \mathbb{P}) \Rightarrow$
//regla de creación de operaciones
<R6_OP>
//invocación de la operación creada
s'.entry.specification.name '(' ');'

//operaciones doActivity
$\forall o''' \in Operation \bullet o''' \in (s'.doActiviy.specification\ \mathbb{P}) \Rightarrow$
//regla de creación de operaciones
<R6_OP>
//invocación de la operación creada
s'.doActivity.specification.name '(' ');'

En la anterior regla se utilizó la regla de transformación del diagrama de clases que permite la generación de código para operaciones. Gracias a la relación, señalada en la sección "Metamodelo del Diagrama de Máquina de Estados", entre el diagrama de clases y el diagrama de máquina de estados, es posible enlazar reglas de diferentes diagramas.

5. CASO DE ESTUDIO

Utilizando las reglas de transformación propuestas en este artículo, se puede obtener el código fuente a partir de los diagramas de clases, secuencias y máquina de estados de un sistema particular; el caso de estudio corresponde al manejo de una pizzería. El

código fuente se representa utilizando el lenguaje orientado a objetos Java aunque las reglas fueron expresadas de manera genérica para cubrir otros lenguajes orientados a objetos.

Inicialmente, se obtiene la estructura del sistema en código, lo cual se hace utilizando el diagrama de clases. Para lograr esto, se utiliza una plantilla con la sintaxis del lenguaje, la cual se completa con los resultados de las reglas de transformación. En la plantilla se combinan elementos propios del lenguaje y los consecuentes de las reglas de transformación, estos últimos reemplazados por sus equivalentes en los modelos UML.

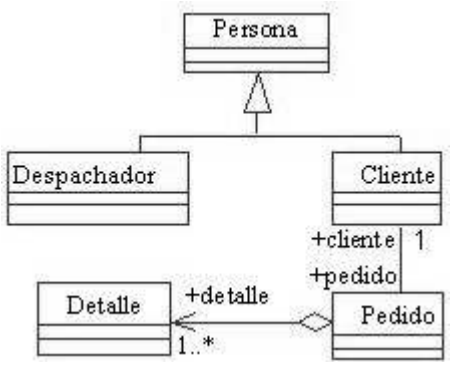

Figura 12. Diagrama de clases del servicio a domicilio de un restaurante (porción)
Figure 12. Class Diagram of a restaurant's domicile service (fragment)

La plantilla para las clases Java es:

```
public class $c.name{
        //atributos
        $p.visibility $.type.name $p.name;

        //operaciones
        $o.visibility $o.type.name $o.name
($ownedParameter.type.name
$ownedParameter.name)
        {
                //método
        }
}
```

La Tabla 1 muestra las reglas de transformación que debe ser aplicadas para completar la plantilla de la clase.

Tabla 1. Relación código-reglas de transformación del diagrama de clases
Table 1. Relationship between code and Class Diagram Transformation Rules

$c.name	R2_NOMC
$p.visibility $p.type.name $p.name	R4_ATTRP y R5_ATTRA
$o.visibility $o.type.name $o.name ($ownedParameter.type.name $ownedParameter.name)	R6_OP

La Figura 3 es la representación del diagrama de la Figura 12 como instancia del metamodelo del diagrama de clases, correspondiente a las clases Pedido y Cliente. Al aplicar las reglas de transformación para el diagrama de clases sobre la instancia *c3* de la metaclase *Class*, la cual hace referencia a la clase Pedido, el código generado para esta clase, de acuerdo a la plantilla y las relaciones presentadas en la Tabla1 es:

```
public class Pedido{
    //atributos
    private String number;
    private Cliente cliente;

    //operaciones
    public void agregarDetalle(){
    //método
    }
}
```

El método o implementación de la operación *agregarDetalle*, se obtiene utilizando un diagrama de secuencias y su representación como instancia del metamodelo. Para este caso se emplea la instancia de la Figura 5 que representa la llamada a la operación *setPedido* del diagrama de la Figura 13.

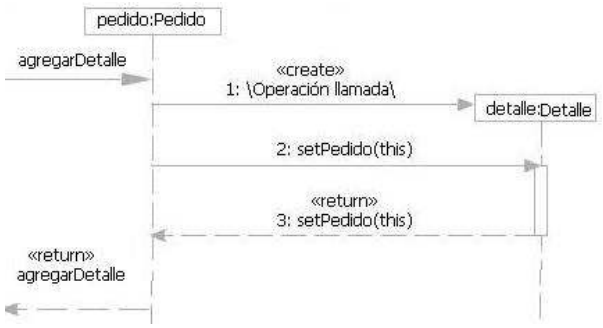

Figura 13. Diagrama de Secuencias del servicio a domicilio de un restaurante
Figure 13. Sequence Diagram of a restaurant's domicile service

Aplicando las reglas de transformación para creación de objetos e invocación de operaciones del diagrama de secuencias, se obtiene el siguiente código:

```
public void agregarDetalle()
{
    //creación de objeto
    Detalle detalle = new Detalle();
    //llamada a método
    detalle.setPedido(this);
}
```

De acuerdo con las reglas de transformación, el nombre del objeto y su tipo fueron obtenidos a partir de la sentencia *m.receiveEvent.covered.represents*, que hace referencia a la propiedad detalle. El operador *new* es utilizado como operador de creación de objetos. Finalmente, el constructor es obtenido a partir de la misma sentencia, agregando los parámetros del constructor que en este caso es un conjunto vacío.

Nótese que la palabra clave *self*, ha sido reemplazada por *this*, su equivalente en Java.

En el caso del diagrama de secuencias no es necesario utilizar una plantilla como la utilizada para la declaración de la clase. Es suficiente recorrer el diagrama, como instancia del metamodelo, y agregar el código como método de la operación referenciada por el diagrama.

La Tabla 2 muestra las reglas utilizadas en la generación del método de la operación *agregarDetalle*.

Tabla 2. Relación código - Reglas de Transformación del Diagrama de Secuencias
Table 2. Code - Sequence Diagram's Transformation Rules relationship

Detalle detalle = new Detalle();	R7_NEWO
detalle.setPedido(this);	R8_CALLOP

En las reglas del diagrama de secuencias, el mensaje actúa como principal elemento pues a partir de él se navega a través de la instancia del metamodelo. Por ejemplo, en la sentencia de la regla R7_NEWO:

m.receiveEvent.covered.represents.type

Se utiliza la instancia *m* de *Message* y su cadena de relaciones *receiveEvent*, *covered*, *represents* y *type*, para hacer referencia a la clase instanciada en la línea de vida que recibe el mensaje. De igual forma puede tenerse acceso a la clase instanciada por la línea de vida que envía el mensaje, reemplazando a *receiveEvent* por *sendEvent.*

Finalmente, el diagrama de máquina de estados se utiliza para complementar el código obtenido hasta el momento, con los posibles estados en los que puede estar un objeto y las reglas de transición entre estados.

El diagrama de máquina de estados de la Figura 14 muestra los dos posibles estados, Registrado o Cancelado, en los cuales puede encontrarse un pedido.

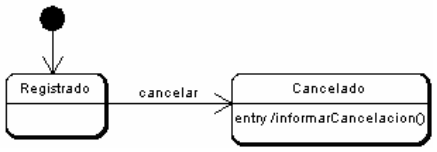

Figura 14. Diagrama de Máquina de Estados de la clase Pedido
Figure 14. State Machine Diagram of the Order class

La transición del estado Registrado a Cancelado sólo requiere de la generación del evento *cancelar*. Una vez el pedido ha sido cancelado, se ejecuta la operación *informarCancelación.*

Antes de utilizar la regla de transformación R11_STATES, es necesario obtener una representación del modelo de la Figura 14 como instancia del metamodelo comprimido del diagrama de máquina de estados (Véase la Figura 15).

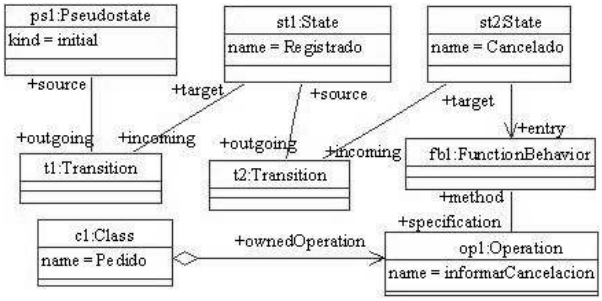

Figura 15. Instancia del metamodelo del diagrama de máquina de estados para el caso de estudio
Figure 15. Metamodel instance of the State machine diagram of the study case

A partir de la regla R11_STATES se genera el siguiente código:

```
if(Registrado.events(e) && e.transition.guard){
      //operaciones exit

      //operaciones transición

      //Nuevo estado
      estadoActual = Cancelado;

      //operaciones entry
      informarCancelacion();

      //operaciones doActivity
}
else if(Cancelado.events(e) && e.transition.guard){
      //operaciones exit

      //operaciones transición

      //Nuevo estado

      //operaciones entry

      //operaciones doActivity
}
```

Para cada estado se genera una condición *if* en la cual se evalúa si el evento generado hace parte del estado y si la condición de la transición es verdadera. Nótese que ésta es la primera condición de la regla R11_STATES, y que la regla obliga a generar una estructura de clases en código, similar a la presentada en el metamodelo. De esta forma, se deben haber generado las clases Java *Event*, *Transition*, *State* y demás, necesarias para el modelo bajo estudio.

También es importante anotar que a partir del diagrama de máquina de estados es posible obtener operaciones no consideradas en el diagrama de clases, como es el caso de *informarCancelacion* de la clase *Pedido*. Por esta razón, la regla R11_STATES, incluye además de la invocación de operaciones, la llamada a la regla de generación de código para operaciones de la clase.

Para el caso de estudio, lo anterior significa que el código de la clase Pedido, además de ser complementado con el código para el manejo de los estados, también lo será con la declaración de una nueva operación llamada *informarCancelación.*

El código generado para la clase Pedido luego de aplicar las reglas de transformación sobre los diagramas de clases, secuencias y máquina de estados es:

```
public class Pedido{
   //atributos
   private String number;
   private Cliente cliente;

   //operaciones
   public void agregarDetalle(){
         //creación de objeto
         Detalle detalle = new Detalle();
         //llamada a método
         detalle.setPedido(this);
   }

   Public void informarCancelacion(){
   }

   public void manejadorEventos(Event e){
      if(Registrado.events(e)&& e.transition.guard){
         //operaciones exit
         //operaciones transición

         //Nuevo estado
         estadoActual = Cancelado;

         //operaciones entry
         informarCancelacion();

         //operaciones doActivity
      }
     Else
      if(Cancelado.events(e)&& e.transition.guard){
         //operaciones exit
         //operaciones transición
         //Nuevo estado
         //operaciones entry
         //operaciones doActivity
      }
}
```

En el código anterior, el manejo de eventos fue agregado en una operación adicional, llamada *manejadorEventos.*

6. CONCLUSIONES Y TRABAJO FUTURO

En este artículo se presentó una propuesta para la generación de código fuente desde los metamodelos del diagrama de clases, secuencias, y máquina de estados de UML.

Dada la complejidad de la especificación de UML, se propusieron metamodelos comprimidos hasta disponer de las metaclases suficientes para la generación del código fuente en Java.

Las reglas de transformación fueron especificadas utilizando lógica de primer orden. Esta representación evita ambigüedades y una fácil interpretación de las reglas, ya que no obliga al lector a aprender un lenguaje particular. Adicionalmente, disponer de una representación no condicionada a un lenguaje particular, permite la generación de código en cualquier lenguaje a partir de plantillas sintácticas.

Como trabajo futuro, se pretende desarrollar una herramienta CASE que incluya los metamodelos comprimidos y las reglas de transformación presentadas.

También es necesario, incluir las restricciones sobre los metamodelos, como las especificadas en la especificación de UML, que permitan validar la consistencia de cada diagrama; y de esta forma lograr una herramienta que además del código permita modificar en un proceso de ingeniería inversa los modelos del usuario.

REFERENCIAS

[1] FUJABA, University of Paderborn, Software Engineering Group. Fujaba Tool Suite. En: http://wwwcs.uni-paderborn.de/cs/Fujaba/index.html

[2] GEIGER, LEIF Y ZÜNDORF, ALBERT, TOOL modeling with fujaba. Electronic Notes in Theoretical Computer Science. 2006, No 148, p. 173–186.

[3] GROSSMAN, MARTIN; ARONSON, JAY Y MCCARTHY, RICHARD. Does UML the grade? Insights from the software development community. Information and Software Technology, 2005. No. 47, p. 383–397.

[4] KEPPLE, ANNEKE; WARMER, JOS Y BAST, Wim. MDA Explained, The Model Driven Architecture: Practice and Promise. Indianápolis: Addison-Wesley, 2003.

[5] LALEAU R. y MAMMAR A. (2005), From a B formal specification to an executable code: applicational to the relational database domain. Information and Software Technology. In Press: 1–27.

[6] LIU, ZHIMING Y JIFENG, HE. Towards a Rigorous Approach to UML-Based Development. Electronic Notes in Theoretical Computer Science. No 130, p. 57–77

[7] LIU, ZHIMING Y JIFENG, HE LONG, LUANG; LIU, ZHIMING; Y LI, XIAOSHAN Y JIFENG, He. Consistent code Generation from UML models. Technical Report, UNU/IIST, 2005.

[8] MELLOR, STEPHEN; SCOTT, KENDALL; UHL, AXEL; WEISE, DIRK. MDA Distilled. Addison-Wesley Professional, 2004.

[9] MORGAN, CARROLL. Programming from Specifications, Second Edition. Prentice Hall International, 1998.

[10] OMGa (Object Management Group). Unified Modeling Language: Superstructure. Fecha de actualización: Septiembre 23 de 2005. Fecha de Consulta: Julio 3 de 2006. http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf

[11] OMGb (Object Management Group). Modeling Language (UML) Specification: Infraestructure. Fecha de actualización: Septiembre 23 de 2005. Fecha de Consulta: Julio 3 de 2006. http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-14.pdf

[12] ROSE, IBM Corporation. Rational Rose ArchitectTM. En: http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html

[13] TOGETHER™, BORLAND Software Corporation. Borland Together Architect. En: http://www.borland.com/us/products/together/index.html.