Especificación de Componentes MDA para Patrones de Diseño

Liliana Martinez
Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina
lmartine@exa.unicen.edu.ar

y

Liliana Favre[1]
Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina
lfavre@arnet.com.ar

Abstract

The Model Driven Architecture (MDA) promotes the use of models and model transformations for developing software systems. The idea behind MDA is to manage the evolution from Platform Independent Models to Platform Specific Models that can be used to generate executable components and applications. The concepts of metamodels and metamodel-based model transformations are critical in MDA.
In this paper, we analyze how to specify reusable components for design patterns in a way that fit MDA very closely. To define families of reusable components we describe a "megamodel" that refers to metamodels and model transformations organized into an architectural framework. We propose an integration of formal and semiformal specifications to specify MDA mega-components. Our formalization focuses on interoperability of formal languages in Model Driven Development (MDD).

Keywords: Model-Driven Architecture, Components, Design patterns, Formal specifications, Reusability

Resumen

La Arquitectura Model Driven (MDA) promueve el uso de modelos y transformaciones de modelos para desarrollar sistemas de software. La idea central de MDA es manejar la evolución de modelos independientes de la plataforma a modelos específicos a la plataforma que pueden ser usados para generar componentes ejecutables y aplicaciones. En el contexto MDA los metamodelos y las transformaciones basadas en metamodelos son esenciales.
En este artículo describimos como especificar componentes para patrones de diseño alineados a MDA. Para definir familias de componentes reusables presentamos un "megamodelo" que integra metamodelos y refinamientos organizados en un *framework* arquitectural. Proponemos integrar especificaciones semiformales y formales para especificar mega-componentes MDA. Nuestra formalización se centra en la interoperabilidad de lenguajes formales en el desarrollo Model Driven (MDD).

Palabras claves: Model-Driven Architecture, Componentes, Patrones de diseño, Especificaciones formales, Reusabilidad

[1] Comisión de Investigaciones Científicas de la Provincia de Buenos Aires

1 INTRODUCCIÓN

La arquitectura Model-Driven (Model-Driven Architecture o MDA) es un *framework* para el desarrollo de software definido por el Object Management Group (OMG) [14]. Propone elevar el nivel de abstracción en el que se desarrollan sistemas complejos separando la especificación de la funcionalidad de un sistema de su implementación en una plataforma tecnológica específica.

MDA promueve el uso de modelos y transformaciones de modelos para el desarrollo de sistemas de software. Distingue cuatro clases de modelos: *modelo independiente de la computación* (Computation Independent Model o CIM), *modelo independiente de la plataforma* (Platform Independent Model o PIM), *modelo específico a la plataforma* (Platform Specific Model o PSM) y *modelo específico a la implementación* (Implementation Specific Model o ISM). Los modelos son especificados por metamodelos que pueden ser expresados usando MOF (Meta Object Facility) que define una forma común de capturar todos los estándares y construcciones de intercambio [16]. Los metamodelos MOF se basan en los conceptos de entidades, interrelaciones y sistemas y se expresan como una combinación de diagramas de clases UML y restricciones OCL [20, 17].

El proceso de desarrollo Model-Driven (Model-Driven Development o MDD) en el contexto de MDA es llevado a cabo como una secuencia de transformaciones de modelos que incluye al menos los siguientes pasos: construir un PIM, transformar el PIM en uno o más PSMs y construir componentes ejecutables y aplicaciones directamente a partir de un PSM. El éxito de esta propuesta depende en gran medida de un alto grado de automatización de las transformaciones de modelo a modelo y de librerías de componentes que tengan un impacto significativo sobre las herramientas que proveen soporte a MDA. Como consecuencia estas ideas se propone una técnica de metamodelado para la definición de componentes MDA para patrones de diseño [8]. Se optó por éstos dada su amplia difusión, aceptación y uso, ya que presentan soluciones a problemas de diseño recurrentes. Se presenta también un “megamodelo” para definir familias de componentes por medio metamodelos y refinamientos entre ellos.

El desarrollo de componentes reusables requiere poner énfasis sobre la calidad del software, por lo cual las técnicas tradicionales para la verificación y validación son esenciales. En el contexto de los procesos MDD, los formalismos pueden ser usados para detectar inconsistencias tanto en los modelos internos o entre el modelo origen y el modelo destino que ha sido obtenido a través de transformaciones de modelos. En esta dirección, se propone especificar componentes MDA en NEREUS [4] a través de un sistema de reglas de transformación. NEREUS es un lenguaje algebraico alineado con los metamodelos MOF que permite especificar metamodelos basados en los conceptos de entidad, relaciones y sistemas. La necesidad de la formalización se debe a que MOF está basado en UML y OCL, los cuales son imprecisos y ambiguos cuando se trata de la simulación, verificación, validación y predicción de las propiedades del sistema [18,19].

En [6] se presentó una formalización de metamodelos basada en versiones previas de UML. El presente trabajo actualiza y extiende estos resultados con una formalización de mega-componentes MDA que integran niveles de PIM, PSM y código. Estos resultados son parte de los incluidos en [13].

Este artículo está organizado como sigue. La sección 2 describe un “megamodelo” para definir familias de componentes MDA para patrones de diseño. La sección 3 muestra cómo se especificaron los componentes para patrones en una perspectiva MDA y la sección 4 describe cómo formalizar dichos componentes. La sección 5 presenta algunos trabajos relacionados. La sección 6 considera las conclusiones y futuros trabajos.

2 UN "MEGAMODELO" PARA DEFINIR COMPONENTES MDA

Para definir familias de componentes para patrones de diseño se presenta un "megamodelo" que encapsula metamodelos y refinamientos en un framework arquitectural (Figura 1). Se definieron metamodelos en tres niveles de abstracción vinculados a través de refinamientos. Un refinamiento vertical transforma un modelo origen en un modelo destino, ambos en diferentes niveles de abstracción. En este contexto un refinamiento es una especificación detallada de un modelo que conforma a otro más abstracto. Está asociado a un modelo origen y a un modelo destino y está compuesto de parámetros, precondiciones y postcondiciones. La precondición establece las condiciones que deben ser cumplidas por el modelo origen para que la transformación sea aplicada. La postcondición establece propiedades en el modelo destino que la transformación garantiza cuando es aplicada. Las clases *Metamodelo-PIM*, *Metamodelo-PSM* y *Metamodelo-ISM* describen familias de PIMs, PSMs e ISMs respectivamente, mientras que las clases *Refinamiento-PIM-PSM* y *Refinamiento-PSM-ISM* describen familias de refinamientos entre los metamodelos PIM y PSM y entre los metamodelos PSM e ISM respectivamente.

La Figura 2 muestra una instancia del "megamodelo", donde pueden observarse instancias concretas (aparecen subrayadas, siguiendo la notación UML) de los metamodelos del patrón de diseño *Observer*, refinamientos y links entre ellos. Este podría ser visto como un mega-componente que define una familia de componentes reusables que integra instancias de PIMs, PSMs e ISMs.

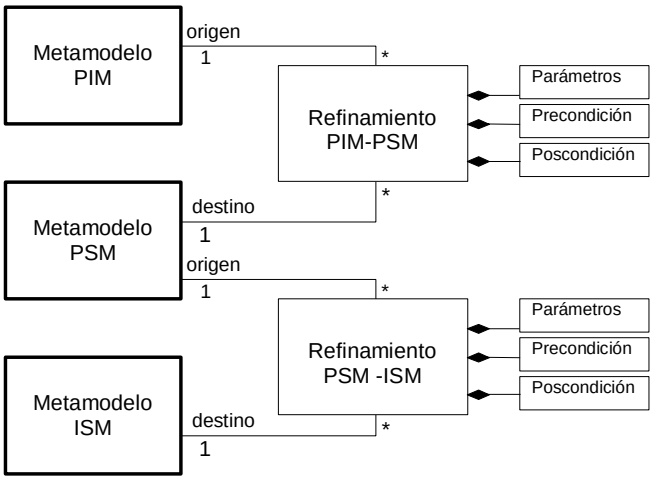


Figura 1: Un megamodelo para componentes MDA para patrones de diseño

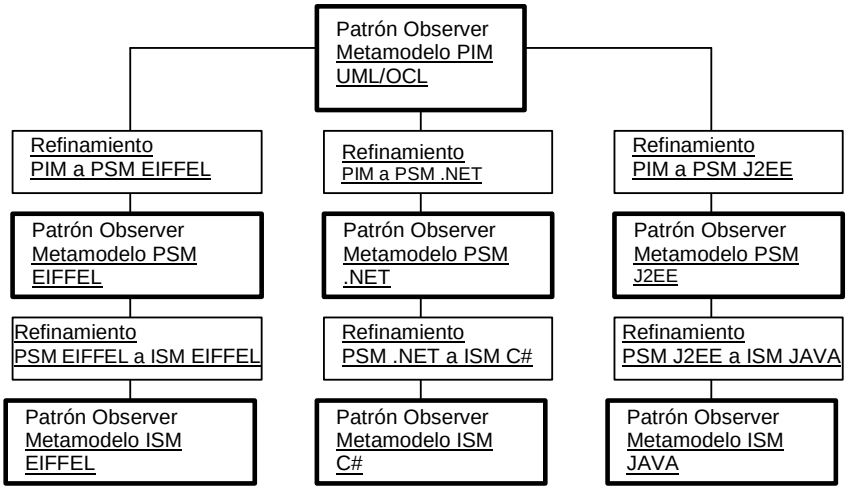


Figura 2: Una instancia del megamodelo. El Componente Patrón *Observer*

3. ESPECIFICACIÓN DE COMPONENTES PARA PATRONES DE DISEÑO

3.1 Especificación de los metamodelos

El metamodelo de un patrón de diseño describe la familia de modelos que forman el espacio de soluciones de dicho patrón. Para especificar un patrón de diseño en distintos niveles de abstracción, fue necesario construir metamodelos tanto para cada una de las plataformas como para cada lenguaje de programación utilizado. Todos estos metamodelos fueron especificados como una especialización del metamodelo UML (versión 2.1.2). Cada metamodelo describe la vista estructural (diagrama de clases) y es especificado como una combinación de diagramas UML, restricciones OCL y lenguaje natural. Los metamodelos de patrones se definieron en tres niveles de abstracción:

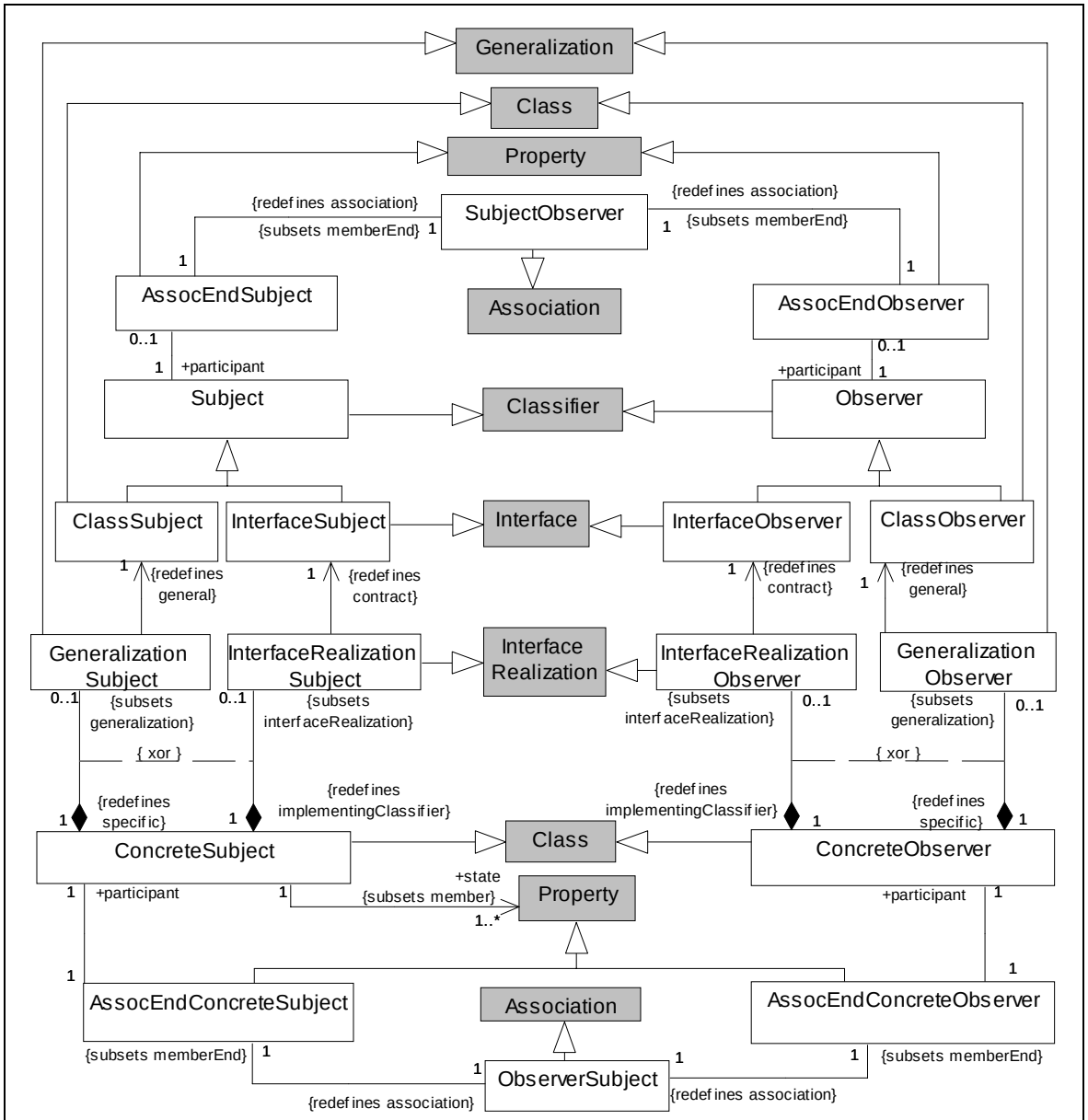
- *Nivel de modelos independientes de la plataforma*: metamodelos definidos de manera independiente de cualquier plataforma o tecnología específicas. Fueron especificados como una especialización del metamodelo UML.
- *Nivel específico a la plataforma*: metamodelos que especifican modelos de patrones vinculados a una plataforma determinada. Se especificaron como una especialización de los metamodelos específicos a sus plataformas.
- *Nivel de código*: cada metamodelo en este nivel depende de un lenguaje específico. Se especificaron como una especialización de los metamodelos específicos a sus lenguajes.

Para ejemplificar, la Figura 3 muestra parcialmente el metamodelo del patrón de diseño *Observer* a nivel PIM. Las metaclases de color gris corresponden a metaclases del metamodelo UML, las restantes a su especialización. El metamodelo debe especificar cada uno de los participantes en un modelo del patrón de diseño *Observer* (el sujeto abstracto, el observador abstracto, el sujeto concreto y el observador concreto), sus atributos, sus responsabilidades y cómo se relacionan entre sí.

El metamodelo establece por ejemplo que una instancia de un sujeto puede aparecer en un modelo como una clase o una interfaz. En el caso de ser una clase estará vinculado a cada sujeto concreto a través de una relación de generalización donde el sujeto cumple el rol de padre y el sujeto concreto el rol de hijo en dicha relación. En caso contrario, si el sujeto es una interfaz estará vinculado a un sujeto concreto a través de una relación de realización donde este último implementará el contrato definido por el sujeto abstracto. Las mismas relaciones se establecen para el observador y el observador concreto. Tanto el sujeto y el observador abstractos, como el sujeto y el observador concretos se vinculan por medio de asociaciones binarias, especificadas por las metaclases *SubjectObserver* y *ObserverSubject* respectivamente. Un sujeto concreto deberá tener un estado conformado por un atributo o un conjunto de atributos, los cuales serán objeto de observación, estos pueden ser propios o heredados.

3.2 Especificación de las transformaciones

La definición de una transformación de modelos es la especificación de un mecanismo para convertir los elementos de un modelo, que es una instancia de un metamodelo particular, en elementos de otro modelo, que es instancia de otro metamodelo. Las transformaciones de modelos fueron especificadas como contratos OCL en términos de precondiciones y postcondiciones. Se incluyen también parámetros (elementos de los metamodelos origen y destino) y operaciones locales utilizadas para facilitar la escritura de la regla. Pueden intercalarse líneas de comentarios precedidas por --. La Figura 4 muestra parcialmente la transformación entre un PIM y un PSM-Eiffel del patrón *Observer*. Las postcondiciones establecen relaciones entre los elementos de los modelos origen y destino.



context AssocEndConcreteObserver inv:
*--Este extremo de asociación tiene una multiplicidad 0..**
self.lower >= 0 **and** self.upper >= 1

context AssocEndConcreteSubject inv:
--Este extremo de asociación tiene una multiplicidad 0.. y es navegable.*
self.lower >= 0 **and** self.upper >= 1 **and**
self.isNavigable()

context ConcreteSubject inv:
--Una instancia de un sujeto concreto no debe ser una clase abstracta.
not self.isAbstract **and**
--state es un conjunto de propiedades que son atributos y no extremos de asociación.
self.state->forAll(p | p.association-> isEmpty())

context ObserverSubject inv:
--El número de extremos de asociación miembros es dos, por ser una asociación binaria.
self.memberEnd ->size() =2

Figura 3: Metamodelo PIM del patrón de diseño *Observer*

```
Transformation PIM-UML to PSM-EIFFEL {
parameters
    sourceModel: Observer-PIM-Metamodel:: Package
    targetModel: Observer-PSM-EIFFEL-Metamodel:: Package
postconditions
post:
    -- sourceModel y targetModel tienen el mismo número de clasificadores.
    targetModel.ownedMember-> select(oclIsTypeOf(EiffelClass))-> size()   =
            sourceModel.ownedMember-> select(oclIsTypeOf(Class))-> size() +
            sourceModel.ownedMember-> select(oclIsTypeOf(Interface))-> size()
post:
    -- Para cada interface 'sourceInterface' en sourceModel existe una clase 'targetClass' en targetModel tal que:
    sourceModel.ownedMember -> select(oclIsTypeOf(Interface))-> forAll ( sourceInterface |
            targetModel.ownedMember-> select(oclIsTypeOf(EiffelClass))-> exists ( targetClass |
        -- 'targetClass' se corresponde con ' sourceInterface'.
            targetClass.oclAsType(EiffelClass).classInterfaceMatch  (sourceInterface.oclAsType(Interface))  ) )
 post:
    -- Para cada clase 'sourceClass' en sourceModel existe una clase 'targetClass' en targetModel tal que:
    sourceModel.ownedMember -> select(oclIsTypeOf(Class))-> forAll ( sourceClass |
        targetModel.ownedMember -> select(oclIsTypeOf(EiffelClass))-> exists ( targetClass |
            -- 'targetClass' se corresponde con 'sourceClass'.
            targetClass.oclAsType(EiffelClass).classClassMatch (sourceClass.oclAsType(Class))  ) )
local operations
    Observer-PSM-EIFFEL-Metamodel::EiffelClass::classClassMatch (aClass: Observer-PIM-Metamodel::Class):Boolean
        classClassMatch (aClass)  = self.name = aClass.name and self.isDeferred = aClass.isAbstract and
    ...
```

Figura 4: Especificación de transformaciones en OCL

4 FORMALIZACIÓN DE COMPONENTES MDA

El desarrollo de componentes reusables requiere poner énfasis sobre la calidad del software, por lo cual las técnicas tradicionales para la verificación y validación son esenciales para lograr atributos de calidad en el software tales como consistencia, correctitud, robustez, reusabilidad, eficiencia y confiabilidad. UML y OCL son imprecisos y ambiguos cuando se trata de la simulación, verificación, validación y predicción de las propiedades del sistema y más aún si se trata de generar modelos o implementaciones a través de transformaciones. Una especificación formal clarifica el significado deseado de los metamodelos y de las transformaciones de modelos basadas en metamodelos ayudando a validarlos y proveyendo una referencia para la implementación.

La integración de especificaciones semiformales UML/OCL y los lenguajes formales ofrece lo mejor de ambos mundos. En esta dirección, se propone la utilización de la notación de metamodelado NEREUS junto con un sistema de reglas de transformación que permiten transformar metamodelos MOF a especificaciones NEREUS. Se optó por la utilización de esta notación por los siguientes motivos:

- Está alineado con MOF, ya que ambos tienen mecanismos de estructuración similares los conceptos de los metamodelos basados en MOF pueden ser traducidos a NEREUS en una forma directa.
- NEREUS puede ser visto como una notación intermedia abierta a muchos otros lenguajes formales, lo que facilita la interoperabilidad con lenguajes formales clásicos. En particular se definió una forma de traducir automáticamente cada construcción NEREUS a CASL [5] lo que permitiría el uso de las herramientas que dan soporte al mismo.
- La integración de OCL con un lenguaje como NEREUS permite una sólida definición del sistema de tipos, relaciones de herencia, subtipo y refinamientos entre especificaciones.

NEREUS consiste de construcciones para expresar clases, asociaciones y paquetes. La sintaxis de las especificaciones NEREUS es la siguiente:

CLASS className [<parameterList>]
IMPORTS <importsList>
INHERITS <inheritsList>
IS-SUBTYPE-OF <subtypeList>
GENERATED-BY <basicConstructors>
ASSOCIATES<associatesList>
DEFERRED
TYPES <typesList>
FUNCTIONS <functionList>
EFFECTIVE
TYPES <typesList>
FUNCTIONS <functionList>
AXIOMS <varList>
<axiomList>
END-CLASS

ASSOCIATION <relationName>
IS <constructorTypeName>
[…: Class1;…: Class2;
…: Role1; …: Role2;
…: mult1; …: mult2;
…: visibility1; …: visibility2]
CONSTRAINED-BY <constraintList>
END

PACKAGE packageName
IMPORTS <importsList>
INHERITS <inheritsList>
<elements>
END-PACKAGE

Distingue herencia de subtipo (INHERITS, IS-SUBTYPE-OF). Las cláusulas EFFECTIVE y DEFFERRED permiten declarar nuevos tipos y operaciones de manera completa e incompleta respectivamente. Las operaciones son declaradas en la cláusula FUNCTIONS.

Provee una jerarquía de constructores de tipos que permite clasificar las asociaciones de acuerdo a su tipo, su navegabilidad y su conectividad. ASSOCIATION permite construir nuevas asociaciones a través del mecanismo de instanciación.

PACKAGE permite agrupar clases y asociaciones.

4.1 Formalización de metamodelos MOF

Los metamodelos MOF y NEREUS tienen construcciones y mecanismos de estructuración similares. Cada paquete, clase o asociación en el metamodelo es traducido en un paquete, una clase o una asociación en NEREUS.

El proceso de traducción de especificaciones OCL a NEREUS es soportado por un sistema de reglas de transformación. En los metamodelos, las especificaciones OCL pueden aparecer como precondiciones, poscondiciones o invariantes de clases, restricciones de atributos y restricciones de asociaciones. Analizando especificaciones OCL se pueden derivar axiomas que serán incluidos en las especificaciones NEREUS. Las precondiciones escritas en OCL se usan para generar precondiciones en NEREUS. Las poscondiciones e invariantes permiten generar axiomas en la especificación NEREUS. Las restricciones de las asociaciones se traducen a restricciones de la especificación de las asociaciones.

La Figura 5 muestra parcialmente la formalización del metamodelo presentado en la Figura 3.

4.2 Formalización de refinamientos

La especificación de los refinamientos de un componente puede ser vista como la especificación de un grafo dirigido acíclico **G = (V , A)** (Figura 6), donde:

- **V** es el conjunto de vértices formado por los metamodelos que participan en los refinamientos. Están divididos en clases de equivalencias V_{PIM}, V_{PSM} y V_{ISM} donde:
  - V_{PIM}: es el conjunto de los metamodelos a nivel PIM
  - V_{PSM}: es el conjunto de los metamodelos a nivel PSM
  - V_{ISM}: es el conjunto de los metamodelos a nivel de código
- **A** es un conjunto de arcos que representan refinamientos entre metamodelos. Cada arco es una relación binaria entre los elementos de V de manera tal que:
  - $\forall$ arco $(m_1, m_2) \in A$: $(m_1 \in V_{PIM} \wedge m_2 \in V_{PSM}) \vee (m_1 \in V_{PSM} \wedge m_2 \in V_{ISM})$

PACKAGE ObserverMetamodel
IMPORTS Kernel, Interfaces, Dependencies

-- Especificación de las metaclases

CLASS AssocEndSubject
IS-SUBTYPE-OF Property
ASSOCIATES <<AssocEndSubject-SubjectObserver>>, <<AssocEndSubject-Subject>>
GENERATED_BY create
TYPES AssocEndSubject
FUNCTIONS
create: ∗ → AssocEndSubject
AXIOMS assEnd: AssocEndSubject
get_lower(assEnd) >=0 and (get_upper(assEnd) >= 1
END-CLASS

CLASS ConcreteSubject
IS-SUBTYPE-OF Class
ASSOCIATES
<<AssocEndConcreteSubject-ConcreteSubject>>, <<ConcreteSubject-GeneralizationSubject>>, …
FUNCTIONS
create: ∗ → ConcreteSubject
AXIOMS sub:ConcreteSubject; CP: ConcreteSubject-Property; AP: Association-Property; …
not isAbstract(sub)
$forAll_p$ (get_state (CP,sub), [isEmpty(get_association(AP,p)])
END-CLASS

CLASS ObserverSubject
IS-SUBTYPE-OF Association
ASSOCIATES
<<AssocEndConcreteSubject-ObserverSubject>>, <<AssocEndConcreteObserver-ObserverSubject>> …
FUNCTIONS
create: ∗ → ObserverSubject
AXIOMS a: ObserverSubject; AP: Association-Property
size(get_memeberEnd(AP,a)) = 2
END-CLASS

-- Especificación de las asociaciones

ASSOCIATION AssocEndConcreteSubject-ConcreteSubject
IS Bidirectional-1 [AssocEndConcreteSubject: Class1; ConcreteSubject: Class2; assocEndConcreteSubject:role1; participant:role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
END

ASSOCIATION AssocEndConcreteSubject-ObserverSubject
IS Bidirectional-1 [AssocEndConcreteSubject: Class1; ObserverSubject: Class2; assocEndConcreteSubject:role1; association:role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
CONSTRAINED_BY
assocEndConcreteSubject: subsets memberEnd
observerSubject: redefines association
END

ASSOCIATION ConcreteSubject-Property
IS Unidirectional-2 [ConcreteSubject:Class1; Property:Class2; concreteSubject:role1; state:role2; 1:mult1; 1..*:mult2; +:visibility1; +:visibility2]
CONSTRAINED-BY state: subsets member
END

ASSOCIATION ConcreteSubject-GeneralizationSubject
IS Composition-1 [ConcreteSubject: Whole; GeneralizationSubject: Part; concreteSubject: role1; generalizationSubject: role2; 1:mult1; 0..1:mult2; +:visibility1; +:visibility2]
CONSTRAINED-BY
generalizationSubject: subsets generalization
concreteSubject: redefines specific
END

…
END PACKAGE

Figura 5. Formalización de Metamodelos

La Figura 7 muestra el esquema de un componente en NEREUS, el cual será instanciado con el nombre. Esta especificación del componente permite agregar y borrar metamodelos y links, obtener el conjunto de metamodelos de cada nivel, etc. C-Metamodel es un metamodelo de componente, es decir, que permite la definición de modelos a nivel PIM, PSM o código.

Las instancias de los refinamientos se pueden traducir a especificaciones NEREUS por medio de la instanciación de esquemas reusables. La Figura 8 muestra el esquema para la clase Transformation en NEREUS usada por el esquema de la Figura 7. Cada transformación es instanciada con un nombre, metamodelos origen y destino junto con precondiciones y postcondiciones. La función $Translate_{NEREUS}$ *(transformation.precondition)* que aparece en el esquema de la transformación después la palabra "pre:" de la operación *create*, traduce a NEREUS la precondición de la transformación. La función $Translate_{NEREUS}$ *(transformation.postcondition)* que aparece en los axiomas traduce a NEREUS las postcondiciones de la transformación.

La Figura 9 muestra parcialmente el resultado de la instanciación del esquema de transformación para la transformación PIMtoPSMEiffel entre un metamodelo del patrón Observer a nivel PIM y un metamodelo del mismo patrón a nivel PSM-EIFFEL mostrada en la Figura 4.

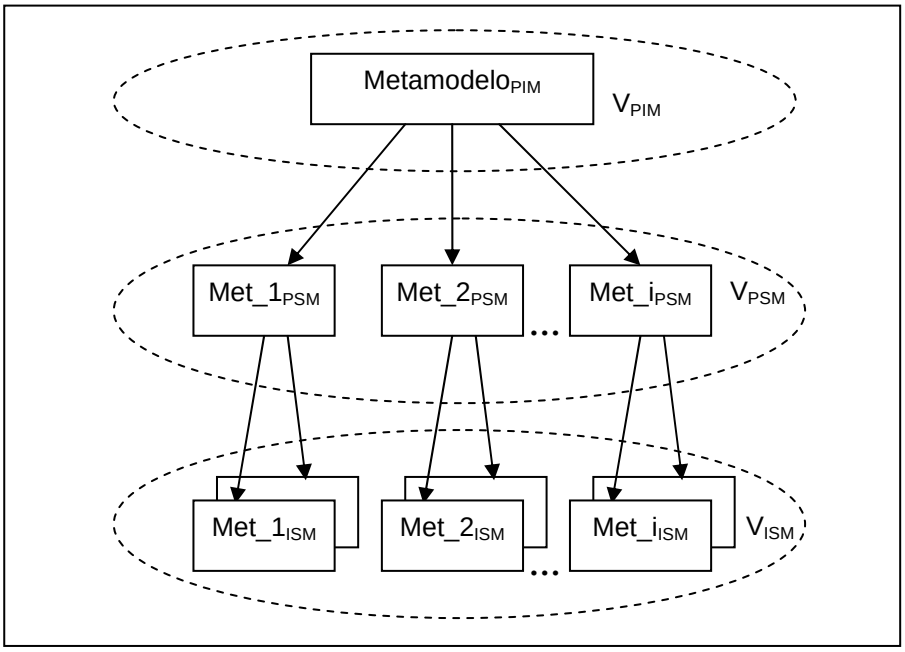


Figura 6: Formalización de Refinamientos

CLASS *Component-name*
IMPORTS Set [C-Metamodel], Transformation
GENERATED-BY create, addLink, addMetamodel
EFFECTIVE
TYPE *Component-name*
FUNCTIONS
create: → *Component-name*
addLink: *Component-name* (c) *x* Transformation (t) → *Component-name*
pre: ((get-source(t) ∈ get_V_{PIM} (c) and get-target (t) ∈ get_V_{PSM}(c)) or
(get-source(t) ∈ get_V_{PSM} (c) and get-target (t) ∈ get_V_{ISM}(c))) and not existsLink (c, t)
addMetamodel: *Component-name* (c) x C-Metamodel (m) → *Component-name*
pre: not existsMetamodel (c, m)
removeLink: *Component-name* (c) *x* Transformation (t)→ *Component-name*
pre: existsLink (c, t)
removeMetamodel: *Component-name* (c) *x* C-Metamodel (m)→ *Component-name*
pre: existsMetamodel (c, m)
existsLink*: Component-name x* Transformation →Boolean
existsMetamodel*: Component-name x* C-Metamodel →Boolean
get_V_{PIM}: *Component-name* → Set [C-Metamodel]
get_V_{PSM}: *Component-name* → Set [C-Metamodel]
get_V_{ISM}: *Component-name* → Set [C-Metamodel]

AXIOMS c: *Component-name*; t, t1, t2: Transformation*;* m,m1,m2: C-Metamodel;
s1,s2,s3: Set [C-Metamodel]
removeLink (addLink (c,t1),t2)= **IF** equal (t1,t2) **THEN** c **ELSE** addLink(removeLink(c,t2),t1) **ENDIF**
removeLink (addMetamodel (c,m),t) = addMetamodel(removeLink (c,t), m) …
existsLink (create(), t) = false
existsLink (addLink (c, t1), t2) = **IF** equal(t1,t2) **THEN** true **ELSE** existsLink (c, t2) **ENDIF**
existsLink (addMetamodel (c,m),t) = existsLink (c, t) …
get_V_{PIM} (create()) = createSet()
get_V_{PIM}(addLink(c,t))= get_V_{PIM}(c)
get_V_{PIM}(addMetamodel(c,m))= **IF** equal(type(m),'PIM') **THEN** add (get_V_{PIM}(c),m)
…
END-CLASS

Figura 7: Un esquema de Componente

CLASS *TransformationName*
GENERATED-BY create
EFFECTIVE
TYPE *TransformationName*
FUNCTIONS
create: *sourceMetamodel* x *targetMetamodel* → *TransformationName*
pre: *Translate*$_{NEREUS}$ *(Transformation.precondition)*
get-source: *TransformationName* → *sourceMetamodel*
get-target: *TransformationName* → *targetMetamodel*
equal: *TransformationName* x *TransformationName* → Boolean
AXIOMS
m1: *sourceMetamodel*, m2: *targetMetamodel*; t1,t2 : *TransformationName*
get-source (create(m1,m2,)) = m1
get-target (create(m1,m2,)) = m2
equal (t1,t2) = **IF** get-source(t1)=get-source(t2) and get-target(t1)=get-target(t2)
THEN true **ELSE** false
Translate$_{NEREUS}$ *(Transformation.postcondition)*
END-CLASS

Figura 8: Un esquema de transformación

CLASS PIM-UML_to_PSM-EIFFEL
GENERATED-BY create
EFFECTIVE
TYPE PIM-UML_to_PSM-EIFFEL
FUNCTIONS
create:Observer-PIM-Metamodel x Observer-PSM-EIFFEL-Metamodel→PIM-UML_to_PSM-EIFFEL
get-source: PIM-UML_to_PSM-EIFFEL → Observer-PIM-Metamodel
get-target: PIM-UML_to_PSM-EIFFEL → Observer-PSM-EIFFEL-Metamodel
equal: PIM-UML_to_PSM-EIFFEL x PIM-UML_to_PSM-EIFFEL → Boolean
-- operaciones locales (privadas) …
AXIOMS
m1: Observer-PIM-Metamodel, m2: Observer-PSM-EIFFEL-Metamodel;
t1,t2 : PIM-UML_to_PSM-EIFFEL; PP : Package-PackageableElement;
e: Observer-PSM-EIFFEL-Metamodel::EiffelClass; c: Observer-PIM-Metamodel::Class; …
get-source (create(m1,m2)) = m1
get-target (create(m1,m2)) = m2
equal (t1,t2) = **IF** get-source(t1)=get-source(t2) and get-target(t1)=get-target(t2) **THEN** true **ELSE** false
-- TranslateNEREUS (Transformation.postcondition):
-- sourceModel y targetModel tienen el mismo número de clasificadores.
size (select$_{elem}$(get_ownedMember(PP,m2), [oclIsTypeOf(elem,EiffelClass)])) =
size (select$_{elem}$(get_ownedMember(PP,m1), [oclIsTypeOf(elem, Class)])) +
size (select$_{elem}$(get_ownedMember(PP,m1), [oclIsTypeOf(elem, Interface)])) and
-- Para cada interface 'sourceInterface' en sourceModel existe una clase 'targetClass' en targetModel tal que:
forAll$_{sourceInterface}$(select$_{elem}$(get_ownedMember(PP,m1) , [oclIsTypeOf(elem, Interface)]),
[exists$_{targetClass}$ (select$_{elem}$(get_ownedMember(PP,m2) , [oclIsTypeOf(elem, EiffelClass)]),
-- sourceInterface y targetClass se corresponden
[interfaceClassMatch(oclAsType(targetClass, EiffelClass), oclAsType(sourceInterface, Interface))])]) and
-- Para cada clase 'sourceClass' en sourceModel existe una clase 'targetClass' en targetModel al que: ...
-- operaciones locales
classClassMatch(e,c) = equal(name(e), name(c)) and equal (isDeferred(e),isAbstract(c)) and …
END-CLASS

Figura 9: Una instanciación del esquema de transformación

5 TRABAJOS RELACIONADOS

Desde la aparición de los patrones de diseño ha habido innumerables trabajos sobre especificación de patrones de diseño usando técnicas formales y semiformales [10, 11, 7, 3]. El objetivo de algunos fue la documentación y definición, el de otros la aplicación automática de patrones, su detección automática y/o la generación de código a partir de ellos. Nuestra propuesta es englobar todos estos objetivos a través de la especificación de componentes para patrones de diseño enmarcados en MDA, los metamodelos de patrones fueron especificados en tres niveles de abstracción: PIM, PSM y código. Esta especificación permitiría tanto la detección de un patrón en los tres niveles de abstracción, como así también la generación de código a partir de un modelo en más de un lenguaje.

Entre los trabajos vinculados a componentes para reuso en el contexto de patrones de diseño cabe mencionar a [15, 1]. El primero establece las principales propiedades que debe tener un modelo de componente de alta calidad y apunta a la producción de componentes que garanticen dichas propiedades. Nosotros tomamos estas ideas y contribuimos con una técnica de metamodelado para construir componentes de software reusables en una perspectiva MDA. Para lograr la confiabilidad de los componentes se propuso la formalización de los mismos. El segundo trabajo analiza los patrones propuestos por Gamma para identificar cuáles pueden transformarse en componentes reusables y los describe. Si bien en este trabajo la reusabilidad está dada en términos de código, fue una inspiración para pensar en patrones de diseño en términos de componentes MDA.

En la actualidad la propuesta MDA está tomando cada vez más fuerza en el desarrollo de software. Desde su aparición han surgido numerosos trabajos relacionados a la especificación de las transformaciones entre modelos [9, 12]. A diferencia de los trabajos mencionados, proponemos transformaciones de modelos como contratos OCL basadas en los metamodelos origen y destino e integradas como parte de un componente.

6 CONCLUSIONES Y FUTUROS TRABAJOS

En este artículo se presentó un “megamodelo” para definir familias de componentes para patrones de diseño a través de una técnica de metamodelado que permite alcanzar un alto nivel de reusabilidad y adaptabilidad en una perspectiva de MDA. La definición de los componentes reusables se hizo a través de la especificación de metamodelos y transformaciones de modelos. Los metamodelos se especificaron como una extensión del metamodelo UML. Las transformaciones se especificaron como contratos OCL en términos de precondiciones y postcondiciones.

Para obtener componentes confiables se propuso integrar especificaciones semiformales y formales. En el contexto de MDA las especificaciones formales permitirían detectar inconsistencias tanto dentro de un modelo, como inconsistencias entre un modelo origen y un modelo destino obtenido a través de una transformación. Se propuso formalizar los componentes usando la notación de metamodelado NEREUS a través de la formalización de los metamodelos MOF y de las transformaciones de modelos basadas en metamodelos. Se mostró un esquema para la formalización de un mega-componente como un todo que integra metamodelos de componente y refinamientos.

La propuesta fue ilustrada usando el patrón de diseño *Observer*.

Uno de los objetivos propuestos es completar el catálogo de componentes para patrones especificando metamodelos para la vista de comportamiento. Un problema crucial es cómo hacer para detectar qué parte de un diagrama se corresponde con un patrón, el metamodelado puede ayudar en la identificación de patrones de diseño por medio de un *matching* de signaturas y semántico. Completar dicho *matching* es otra de las metas propuestas. Por último, se prevé la validación de la propuesta integrándola con alguna de las herramientas CASE UML [2] existentes.

REFERENCIAS

[1] Arnout, K.. From Patterns to Components. Tesis Doctoral, Swiss Institute of Technology (ETH Zurich), 2004.

[2] CASE TOOLS. Disponible en: www.objectsbydesign.com/tools/umltools_byCompany.html

[3] Elaasar, M., Briand L. y Labiche, Y. A Metamodeling Approach to Pattern Specification and Detection. Lecture Notes Computer Science 4199. 484-498, 2006.

[4] Favre, L. Foundations for MDA-based Forward Engineering. Journal of Object Technology (JOT). 4(1): 129-153. 2005.

[5] Favre, L. A Rigorous Framework for Model Driven Development. Keng Siau (ed.). Advanced Topics in Database Research, Vol. 5. Chapter I, IGP, USA, 1-27, 2006.

[6] Favre, L. y Martinez, L. Formalizing MDA Components. Lecture Notes in Computer Science 4039, Springer Verlag Berlin Heidelberg, ISSN 0302-9743. 326-339, 2006.

[7] France, R., Kim, D., Ghosh, S. y Song, Eunjee. A UML-Based Pattern Specification Technique. IEEE Transactions on Software Engineering. IEEE Computer Society, 30(3): 193-206, 2004.

[8] Gamma, E., Helm, R., Johnson, R. y Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[9] Judson, S., France, R. y. Carver D. Specifying Model Transformations at the Metamodel Level. 2003. Workshop in Software Model Engineering, UML'03 Conference, 2003. Disponible en: www.metamodel.com/wisme-2003/19.pdf

[10] Kim, D, France, R., Ghosh, S. y Song, E. A UML-Based Metamodeling Language to Specifying Desing Patterns. Workshop in Software Model Engineering,Wisme-UML. USA, 2003. Disponible en: www.metamodel.com/wisme-2003/01.pdf

[11] Kim, D., France, R., Ghosh, S. y Song, E. A Role-Based Metamodeling Approach to Specifying Desing Patterns. 27th Annual International Computer Software and Applications Conference (COMPSAC'03). IEEE Computer Society, 452-457, 2003.

[12] Kuster, J., Sendall, S. y Wahler, M. Comparing Two Model Transformation Approaches. Workshop de OCL and Model Driven Engineering. Portugal, 114-127, 2004.

[13] Martinez, L. "Componentes MDA para Patrones de Diseño". Tesis de Magister en Ingeniería de Software. Facultad de Informática. UNLP. Argentina, 2008.

[14] MDA. Model-Driven Arquitecture. Disponible en www.omg.org/mda

[15] Meyer, B. The Grand Challenge of Trusted Components. 25th International Conference on Software Engineering (ICSE), Portland, Oregon. IEEE Computer Press, 660-667, 2003.

[16] MOF. Meta Object Facility. Documento: formal/2006-01-01. Disponible en www.omg.org

[17] OCL.Object Constraint Language. Documento: formal/06-05-01. Disponible en www.omg.org

[18] Richters, M. A Precise Approach to Validating UML Models and OCL Constraints. Ph.D. thesis, Universit¨at Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[19] Stevens, P. On Associations in the Unified Modeling Language. Lecture Notes Computer Science 2185 (M. Gogolla y C. Kobryn editores.). Springer-Verlag, 361-375, 2001.

[20] UML. Unified Modeling Language Specification. Documento: formal/07-02-05. Disponible en www.omg.org