Transformación de un Modelo de Dominio y Diagramas de Comunicación en un Diagrama de Clases de Diseño

Andrés Vignaga

3 de septiembre de 2006
Tarea 4
CC71S - Modelamiento de Software
Departamento de Ciencias de la Computación - Universidad de Chile

Resumen

En este trabajo se reportan los resultados del desarrollo de una transformación de modelos realizada en Kermeta. Dicha transformación recibe un Modelo de Dominio y los Diagramas de Comunicación que contienen el diseño de operaciones del sistema correspondientes a un cierto caso de uso, y produce el Diagrama de Clases de Diseño que expresa la estructura de clases necesarias en el sistema para que las interacciones especificadas en los Diagramas de Comunicación puedan realizarse. Con ese fin, se definieron metamodelos específicos para los modelos de entrada y para el modelo de salida. En este documento se detallan los pasos seguidos para realizar la transformación, se presenta el diseño de la misma y se discute cada uno de sus elementos. El funcionamiento de la transformación es demostrado mediante su aplicación a un caso de estudio conocido, el cual resulta de una complejidad mayor a la de aquellos ejemplos encontrados comúnmente en la bibliografía.

Palabras clave: Transformación de modelos, Model Driven Engineering, Kermeta, metamodelado, UML, Diagrama de Clases de Diseño

1 Introducción

La transformación de modelos es una técnica que recientemente ha ganado gran atención en la comunidad de la Ingeniería de Software. En ella, la Ingeniería Dirigida por Modelos (Model Driven Engineering o MDE [10]) se presenta como un área amplia donde la noción de modelo es una construcción de primera clase y en torno a la cual se entiende el desarrollo de software. Es por tal razón que la transformación de modelos es estudiada en el contexto de MDE.

La esencia de esta tendencia a trabajar en base a modelos, en última instancia, es la necesidad de manejar la complejidad que significa en la actualidad el desarrollo de sistemas computacionales. En realidad, dicha complejidad estuvo siempre presente desde que el porte de los sistemas solicitados superó un cierto umbral; desarrollar un sistema artesanalmente es de por sí tan complejo que

hace que la real complejidad no pueda ser percibida. A medida que se gana comprensión de las técnicas que dan buen resultado y que la construcción de software se torna una actividad realmente ingenieril, más y más aspectos deben se manejados de manera adecuada, dando la sensación de que es cada vez más complejo desarrollar software de calidad. Razonar al nivel de las tecnologías de implementación desde hace tiempo probó no ser un buen enfoque para el desarrollo. Esto se debe principalmente a la inestabilidad de las mismas, que atenta tanto contra la mantenibilidad de las aplicaciones, así como contra la posibilidad de reusar partes de sistemas, y por sobre todo a que el nivel de abstracción de las construcciones provistas es en general demasiado bajo. El modelamiento de software provee el mecanismo de abstracción necesario para poder atacar un problema complejo en una forma compatible a cómo los seres humanos suelen manejar las cosas complejas: mediante refinamientos sucesivos. La práctica usual era, o es, atacar un problema construyendo un modelo de alto nivel del mismo, e irlo refinando a medida que se gana comprensión. Este proceso finaliza cuando estando ya cerca del nivel de implementación se migra el trabajo a dicha dimensión, continuando a trabajar en ella para irremediablemente desechar los modelos que condujeron a la misma. Esto ocurre usualmente por falta de mantenimiento. Dicho en otras palabras los modelos terminan siendo un medio volátil para llegar al "confortable" bajo nivel. Una vez que se produce esa mutación en el ambiente en el cual se razona la suerte esta echada; cualquier intento de vuelta atrás para poder valerse de aquel nivel intermedio de abstracción para, por ejemplo, poder manejar en forma más sencilla un cambio considerable en el problema, se ve frustrado por elementos desactualizados, sobreviviendo al paso de la evolución únicamente aquellos de más alto nivel que resultan inadecuados para el razonamiento. Entre otras cosas, esto causa que el mantenimiento del software termine realizándose principalmente en el nivel de la implementación. MDE propone cambiar radicalmente el propósito de los modelos en el desarrollo de software, buscando que ese nivel "confortable" sea el (previo) de manipulación de modelos y no el de artefactos de implementación. Con este enfoque, se presume que el esfuerzo mayor será puesto en la generación, mantenimiento y sincronización de modelos, los cuales constituirán una parte fundamental de los insumos en el desarrollo. De esta forma es posible razonar sobre los modelos a un nivel más alto de abstracción. El tratamiento manual de modelos puede ser inmanejable por la cantidad de información involucrada, o incluso por la carencia de una semántica bien definida de los mismos. Asimismo, puede resultar tedioso dada la naturaleza repetitiva de algunas de las actividades más comunes. En ese contexto, la transformación de modelos puede resultar una herramienta de mucho valor.

Las transformaciones de modelos buscan, cuando sea posible, la generación de nuevos modelos en forma idealmente automática, que de otra forma sería manual. Esta técnica tiene la potencialidad de ahorrar una gran cantidad de trabajo a los desarrolladores, así como la de evitar errores, entre otras cosas. Sin embargo, el estudio masivo de transformaciones aplicadas a modelos es relativamente reciente. Existen diversos aspectos básicos aún no comprendidos completamente como para poder desarrollar todo el potencial que este enfoque promete. No existe una definición de los tipos de transformaciones que podrían realizarse sobre modelos. Uno de los ejemplos clásicos de transformaciones es el impulsado por MDA [14] (Model Driven Architecture) del Object Management Group [13] y que refiere a transformar un modelo que represente una solución abstracta a un problema en otro que represente una solución concreta al mismo problema pero desde el punto de vista de una tecnología de implementación. Una transformación que lleve una versión menos detallada a otra más detallada de esa solución abstracta podría o no entenderse como del mismo tipo de transformación que la ya descrita. Asimismo, existen diferentes categorías de enfoques

[9] para llevar a cabo transformaciones, sean del tipo que sean. Algunas de dichas categorías han perdido fuerza recientemente, en cambio otras han cobrado impulso, sin embargo no queda claro si existe una que sea preferible para todos los casos, y más aún, cuál sería la adecuada para cada paso particular. Las aplicaciones, es decir, transformaciones concretas, constituyen un insumo que puede resultar de utilidad para identificar y comprender la naturaleza de los tipos de transformaciones, así como para analizar la aplicabilidad y soporte de las diferentes categorías de enfoques. En el mercado se cuenta con una gran cantidad de ejemplos de transformaciones de las denominadas modelo-a-texto, es decir, generación de código a partir de modelos. De hecho, las clásicas herramientas CASE implementan este tipo de transformación. A pesar de que en [9] se identifican posibles transformaciones denominadas modelo-a-modelo, como por ejemplo la ya mencionada PIM-a-PSM *à la* MDA, refinamiento de modelos, ingeniería reversa, aplicación de patrones de diseño, refactorización de modelos, derivación de productos en una línea de productos, etc., no es común encontrar casos prácticos. El ejemplo utilizado típicamente [1, 6, 8, 9, 18] es la transformación de un modelo estático orientado a objetos (expresado mediante un diagrama de clases de UML) a un Modelo Entidad-Relación (Entity-Relationship Model o ERM [2]).

En este documento se reportan los resultados de experimentos realizados en la definición de una transformación mediante la cual se genera un Diagrama de Clases de Diseño a partir de un Modelo de Dominio y un conjunto de Diagramas de Comunicación, todos expresados en UML. En esta transformación, los Diagramas de Comunicación (término introducido por UML2 y que denota los Diagramas de Colaboración definidos por UML1.x) expresan el diseño de cada una de las operaciones del sistema correspondientes a un mismo caso de uso, y el Diagrama de Clases de Diseño resultante expresa la estructura estática requerida para que dichas interacciones puedan efectivamente ocurrir. Esta transformación es compatible con en enfoque metodológico propuesto por RUP [7]. Una versión de alto nivel de esta transformación fue sugerida por Larman en [11].

Los experimentos realizados para este trabajo fueron realizados siguiendo el enfoque correspondiente a dos de las categorías identificadas en [9]. En particular se cuenta con una implementación en C# [4], correspondiente a la categoría de *lenguajes de programación de propósito general*, y en Kermeta [8], correspondiente a la categoría de *herramientas de matamodelado*. Sin embargo, el énfasis en la presentación de los resultados estará hecho respecto a la segunda de las implementaciones.

Este trabajo se organiza de la siguiente forma. En la sección 2 se explica el enfoque correspondiente a las herramientas de metamodelado, se presenta Kermeta, y se describen los insumos necesarios para la implementación y ejecución de la transformación, los cuales son tratados en mayor profundidad en el resto del documento. La sección 3 presenta una descripción de los metamodelos definidos específicamente para esta transformación y la relación entre ellos. En la sección 4 se introduce el caso de estudio mediante el cual se demuestra el funcionamiento de la transformación. La sección 5 explica propiamente la transformación; se describe el método base, las restricciones aplicadas y el resultado de la transformación aplicada al caso de estudio. Asimismo describe la estructura general de la transformación y su funcionamiento, para finalmente presentar un análisis de la misma. La sección 6 concluye.

2 Herramientas de Metamodelado

Un enfoque posible para la implementación de transformaciones es el de entender a una transformación como un modelo de un programa que transforma modelos. En particular, el lenguaje para expresar las transformaciones es el mismo que se utiliza para expresar los modelos a transformar. En esta sección se presentan las ideas generales detrás de este enfoque, así como una descripción de una herramienta específica de dicho enfoque y que fuera utilizada para realizar la transformación tratada en el trabajo.

2.1 Transformaciones como Modelos

Esta técnica en lugar de buscar la implementación de un programa que transforme modelos, se basa en la idea de construir modelos orientados a objetos de transformaciones; donde una transformación es el modelo de un "programa orientado a objetos" que transforma modelos. En otras palabras, desarrollar una transformación no requiere de la implementación de un programa concreto, sino en cambio, especificar un modelo del mismo. Aplicar la transformación es, por lo tanto, ejecutar el modelo que la especifica.

Dado que una transformación es entendida como un modelo, el lenguaje en que éstas son expresadas es pues un metamodelo. Dicho lenguaje, es el mismo lenguaje en que se expresan los metamodelos de los modelos a transformar. De esta forma, manipular un elemento de un modelo (tanto origen como destino) es igual a manipular objetos en la trasformación misma. Los lenguajes de metamodelado usualmente comprenden únicamente construcciones estructurales, por lo que el usado para especificar transformaciones debe estar enriquecido con acciones para que sus instancias puedan ser ejecutadas.

Ejemplo: Para ilustrar estas ideas, considérese el siguiente ejemplo. Supóngase que se cuenta con un diseño físico de una base de datos y se desea modificarlo de forma de eliminar las referencias 1-n de todas las tablas, reemplazando a cada una por una nueva tabla de asociación. En este ejemplo particular, el *tipo* de la entrada coincide con *tipo* de la salida; un esquema de una base de datos, es decir:

$$transformation : DBSchema \rightarrow DBSchema$$

Un elemento de tipo *DBSchema* contiene tablas, atributos y los demás elementos usuales, y no los datos almacenados en los registros, como en una aplicación de bases de datos típica. Por lo tanto *DBSchema* es un metamodelo para expresar esquemas de bases de datos. Sea $\mathcal{M}$ un lenguaje hipotético mediante el cual expresar la estructura del tipo *DBSchema*. El lenguaje $\mathcal{M}$ es pues un lenguaje para escribir metamodelos. La definición de *DBSchema* en $\mathcal{M}$ podría tener la siguiente forma:

```
Metamodel DBSchema is
   Element Table has
      feature name of Name [1]
      feature columns of Column [1..*]
      ...
```

```
   end Table
   Element Column has
      feature name of Name [1]
      feature type of Name [1]
   end Column
   ...
End DBSchema
```

Donde el tipo `Column` fue definido en el propio `BDSchema` y `Name` es un tipo predefinido de $\mathcal{M}$. Luego, una instancia `db` de tipo `DBSchema` podría contener los elementos *Customer* y *Provider*, ambos de tipo `Table`. Notar que esto es una estructura de datos para almacenar el esquema de la base de datos y no los datos correspondientes a las tablas `Customer` y `Provider`.

En este contexto, ¿dónde y cómo se define la transformación? Como se explicó antes, la transformación se define en el mismo nivel que el metamodelo *DBSchema*. Por tal razón, se debe definir un modelo que sea la propia transformación. Por supuesto, este modelo será escrito en $\mathcal{M}$. Un modelo de transformación en $\mathcal{M}$ podría definirse mediante la palabra clave `Transformation` en lugar de `Metamodel` como en el caso de `DBSchema`. De esta forma, la transformación podría escribirse en $\mathcal{M}$ como:

```
Transformation AddAssocTables is
   require DBSchema
   Element Trans has
      feature transformation (schema-in of DBSchema) of BDSchema is
         -- handle tables and columns of schema-in to produce the result
         -- (i.e.  instances of Table and Column)
      end transformation
      ...
   end Trans
   ...
End AddAssocTables
```

A este punto conviene notar varias cosas. Primero, `AddAssocTables` es realmente un modelo, a pesar de que su expresión textual haga parecer otra cosa. Segundo, dicho modelo es extremadamente sencillo, puesto que contiene solamente un elemento (en terminología orientada a objetos, se diría el modelo tiene una clase con una operación). Tercero, el tipo de la operación `transformation` coincide con el expresado antes, y en dicha operación se manipulan instancias de elementos definidos en `DBSchema` así como instancias de elementos definidos en `AddAssocTables`. Por último, para llevar adelante la transformación es necesario ejecutar el modelo, esto consiste en (contando con una instancia de `DBSchema` para ser transformada), instanciar el elemento `Trans` y sobre él ejecutar la operación `transformation` (con el argumento adecuado); el resultado de la operación es el modelo resultante de la transformación. □

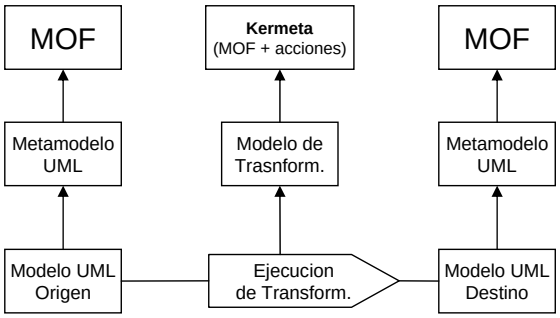

Figura 1: Transformación con KerMeta

Un beneficio que presenta este enfoque es que se puede entender que el desarrollo de transformaciones de modelos en gran escala no es diferente al desarrollo de software. Por lo tanto las técnicas del modelado orientado a objetos aplican a la definición de transformaciones: análisis y diseño orientado a objetos, aplicación de patrones de diseño, adaptabilidad mediante interfaces y despacho dinámico, manejo de excepciones, generación de casos de test, etc.

2.2 Kermeta

En la actualidad existen diferentes ambientes de definición de transformaciones que conforman con el enfoque explicado arriba. Entre otros se puede mencionar a MetaEdit+ [12] de MetaCase o XMF-Mosaic de Xactium [23]. Otra realización concreta de este enfoque se presenta en KerMeta [8]. Kermeta es un lenguaje específico de dominio (Domain Specifica Language o DSL) para ingeniería de metamodelos (construcción, ejecución, etc.). Cuenta con un ambiente de manejo de metamodelos que está integrado a Eclipse [3]. Por ser un lenguaje de metamodelado, Kermeta brinda soporte a la especificación de sintaxis abstracta, semántica estática y semántica dinámica de metamodelos. Entre sus capacidades se incluye además la simulación y prototipado de modelos y metamodelos, y la transformación de modelos. En este contexto, Kermeta fue utilizado en el desarrollo de la transformación presentada en este trabajo.

En términos del lenguaje en sí, Kermeta consiste en:

- Una extensión de EMOF (Essential MOF), que es una parte de MOF 2.0 [15]
- Una extensión imperativa de OCL [16] la cual incluye un conjunto de acciones que le dan a sus instancias la característica de ser ejecutables

De esta forma, la sintaxis abstracta de un metamodelo es expresada según las construcciones heredadas de EMF, la semántica estática con las de OCL, y las acciones permiten especificar su semántica dinámica.

Un escenario de uso de Kermeta es el siguiente: especificación de transformación de modelos de tipo modelo-a-modelo, en particular de modelos cuyo metamodelo esté basado en MOF, como por ejemplo UML. Recordar de la sección anterior que el lenguaje para expresar los metamodelos y transformaciones deben ser el mismo (como era el caso del lenguaje hipotético $\mathcal{M}$). En este caso, el lenguaje de los metamodelos es EMOF y el lenguaje de las transformaciones es Kermeta.

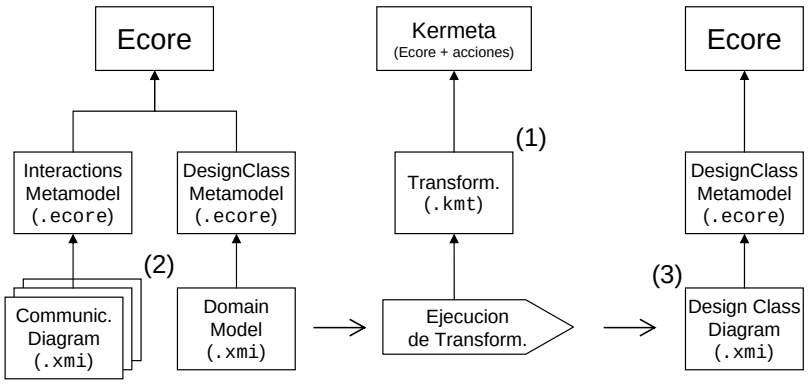

Figura 2: Insumos para la generación de Diagramas de Clases de Diseño

Dado que Kermeta incluye a EMOF, la compatibilidad es completa. Incluso, podría especificarse un metamodelo basado en MOF (como el de UML) en forma equivalente usando Kermeta. Un programa en Kermeta se encuentra por lo tanto al nivel del metamodelo de UML según se ilustra en la figura 1.

En términos prácticos, en el ambiente de desarrollo de Kermeta los metamodelos se expresan en Ecore, que es una variante de EMOF introducida en el ámbito de Eclipse. Kermeta naturalmente soporta la manipulación de metamodelos expresados en Ecore. Una alternativa a ello es expresar los metamodelos en la sintaxis concreta de Kermeta y mediante una herramienta traducirlos a Ecore. Notar que cualquier aspecto imperativo incluido en el metamodelo para expresar su comportamiento se perderá durante la traducción, dado que Ecore no los soporta. Sin embargo, la versión traducida mantiene dichos aspectos como anotaciones, las cuales pueden ser reconstruidas ante una vuelta a la sintaxis de Kermeta.

En lo que refiere a este trabajo, se podría modificar la figura 1, sustituyendo las ocurrencias de MOF por Ecore. En la siguiente subsección se describen los insumos utilizados para la generación de la transformación.

2.3 Insumos para la Generación de Diagramas de Clases de Diseño

En la figura 2, que es una versión más específica de la figura 1, se describen todos los ingredientes necesarios para realizar una transformación de modelos UML, organizados en tres niveles diferentes. En el nivel superior se encuentran Ecore y Kermeta como los lenguajes para expresar los metamodelos de los modelos UML y el modelo de la transformación respectivamente (recordar que Kermeta brinda soporte completo a Ecore). Estos elementos están dados y son las herramientas utilizadas para elaborar el resto.

En el nivel intermedio se encuentran el metamodelo de UML y la trasnformación. El metamodelo de UML se encuentra presentado en la especificación de UML [19] utilizando sintaxis MOF. Sin embargo dada la cantidad de elementos incluidos en dicho metamodelo que no eran relevantes para este trabajo se optó por crear una versión específica, utilizando Ecore, que contemplase únicamente aquellos aspectos relevantes para los experimentos. De esa forma, se definió un paquete denominado *Metamodels* que contiene dos paquetes con los metamodelos, a saber, *In-*

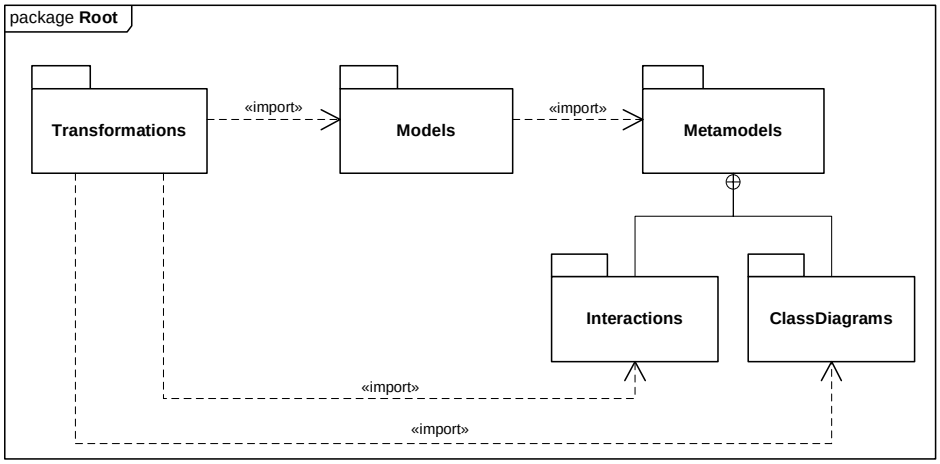

Figura 3: Estructura de paquetes de la propuesta

teractions y *ClassDiagrams*, que contienen la estructura de los Diagramas de Comunicación en el primer caso, y tanto la de los Diagramas de Clases de Diseño y Modelos de Dominio en el segundo. En la Sección 3 se presentan ambos metamodelos. También se definió un paquete *Transformations*, que contiene el modelo de la transformación, y su contenido es presentado en la Sección 5.

En el nivel inferior se encuentran los modelos propiamente, y la instancia (ejecución) de la transformación. En particular, los modelos son un Modelo de Dominio expresado mediante un Diagrama de Clases de UML, y las interacciones para cada operación del sistema, expresadas mediante Diagramas de Comunicación también de UML. Dado que en todos los casos se trata de modelos UML, el formato en que son almacenados los modelos (tanto los de entrada como el de salida) es XMI [20], y son contenidos en un paquete denominado *Models*. Los modelos concretos utilizados para ejemplificar la transformación propuesta se presentan en la Sección 4. Para fijar ideas, en la figura 2 puede entenderse que (1) es el programa que fue desarrollado en este trabajo, (2) son los datos de entrada del programa, y (3) son los datos de salida del programa. Aunque pueda resultar confuso en un comienzo, la analogía es completa: los modelos son datos, y el modelo es un programa. La relación entre todos los paquetes mencionados se ilustra en la figura 3.

3 Metamodelos

Las transformaciones según el enfoque elegido (utilizando herramientas de meramodelado) trabajan sobre modelos, pero tomando sus metadatos de repositorios estructurados. Dichos repositorios toman la forma de metamodelos de los modelos a manipular. En esta sección se presentan los metamodelos que fueron definidos específicamente para poder generar y manipular los modelos de entrada y de salida de la transformación presentada, y la relación entre ellos.

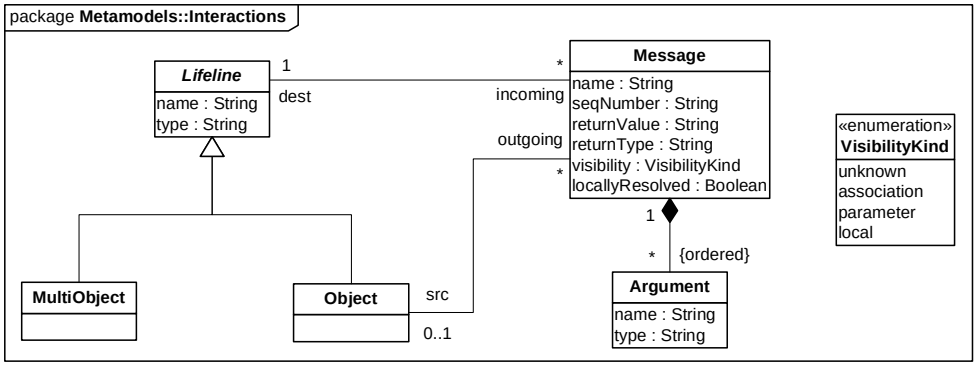

Figura 4: Metamodelo de interacciones

3.1 Interacciones

En esta sección se presenta el metamodelo correspondiente a las interacciones. Primeramente se ilustra su sintaxis abstracta y su estructuración en el nivel raíz en términos de las metaclases descritas, posteriormente se describen cada una de las metaclases involucradas.

3.1.1 Sintaxis Abstracta

El metamodelo ilustrado en la figura 4 es utilizado para generar los modelos que expresan el diseño de las operaciones del sistema a tratar, y que son representados gráficamente mediante Diagramas de Comunicación de UML.

3.1.2 Estructuración

En el paquete Metamodels::InteractionsMetamodel se define una metaclase Interaction implícita que representa una interacción. Una interacción está compuesta en su nivel raíz por los siguientes elementos:

- lifelines : Lifeline[1..*] Los objetos participantes de la interacción. Una interacción con sólo un objeto es trivial ya que contiene solamente el punto de entrada sin especificar como éste es resuelto.
- messages : Message[1..*] Los mensajes participantes de la interacción. Una interacción con sólo un mensaje es trivial ya que dicho mensaje es el punto de entrada.

Los objetos y los mensajes se encuentran interrelacionados entre sí. Las composiciones sobre las clases involucradas en el nivel raíz habilitan la inclusión en las interacciones de otro tipo de elementos (i.e. los elementos componentes). Por ejemplo, un argumento puede incluirse en una interacción si está asociado a un mensaje.

3.1.3 Descripción de Metaclases

A continuación se describe cada una de las metaclases incluidas en el metamodelo presentado arriba, detallando el significado de sus atributos y asociaciones en las que participa.

Lifeline

Representa un objeto tipado, y posiblemente anónimo, que participa en la interacción. Su participación es por medio de la recepción de mensajes.

Atributos

- name : String El nombre (opcional) del objeto.
- type : String El nombre de la clase de la cual el objeto es instancia.

Asociaciones

- incoming : Message[*] El conjunto de mensajes que el objeto recibe.

Object

Es una especialización concreta de lifeline la cual está habilitada además a enviar mensajes, a diferencia de su hermano MultiObject.

Asociaciones

- outgoing : Message[*] El conjunto de mensajes que el objeto envía.

MultiObject

Es una especialización concreta de lifeline la cual está habilitada únicamente a recibir mensajes, a diferencia de su hermano Object. Representa a un objeto que es un contenedor de objetos y usualmente implementa tanto a un diccionario como a un iterador. Todos sus mensajes entrantes se resuelven localmente por lo que no es un emisor de mensajes.

UML2 eliminó el concepto de multiobjeto para los Diagramas de Interacción, por lo que esta metaclase no se ajusta al nuevo estándar. La alternativa que se propone allí para manejar contenedores de objetos es utilizar un clase estructurada que contenga un conjunto de objetos [21]. De esta forma un mensaje a la colección se corresponde con un mensaje a una instancia de esta clase estructurada. Se entiende que esto agrega una complejidad innecesaria al metamodelo al hacer explícita la contención de los elementos en la colección. Por otra parte, como se verá en la Sección 5, las colecciones tienen relevancia para la generación de multiplicidades, por lo que se entiende que la noción clásica de multiobjeto es aún adecuada.

Message

Un mensaje representa la invocación de una operación de un objeto (de tipo Object) sobre otro (cualquier especialización de Lifeline).

Atributos

- name : String El nombre de la operación invocada.
- seqNumber : String El número que determina el orden en que el mensaje será enviado dentro de la interacción.
- returnValue : String El nombre (opcional) de la variable a la cual se le asigna el valor retornado por la operación.
- returnType : String El nombre del tipo del valor retornado por la operación, si corresponde.
- visibility : VisibilityKind El tipo de visibilidad que el objeto origen del mensaje tiene sobre el objeto destino, de forma de poder realizar el envío. Puede tratarse de un valor *unknown*, típicamente utilizado cuando el mensaje es el punto de entrada de la interacción, *association* cuando el destino está asociado al origen, *parameter* cuando el destino fue recibido como parámetro, o *local* cuando el destino es un objeto local a la operación. Ver [11] por más detalles sobre tipos de visibilidades.
- locallyResolved : Boolean Indica si el mensaje será resuelto por el destino utilizando únicamente información contenida en él, o si deberá delegar parte de la resolución del mensaje a otro objeto.

Asociaciones

- src : Object[0..1] El objeto que es origen del mensaje. Si el mensaje es el punto de entrada de la interacción, no hay objeto origen.
- dest : Lifeline El objeto destino del mensaje.
- args : Argument[*] La secuencia de argumentos pasados en el mensaje.

Argument

Representa información que es pasada al destinatario del mensaje cuando se realiza el envío.

Atributos

- name : String El nombre de argumento.
- type : String El nombre (opcional) del tipo del argumento.

3.2 Estructura Estática de Clases

En esta sección se presenta el metamodelo correspondiente a la estructura estática de clases. Primeramente se ilustra su sintaxis abstracta y su estructuración en el nivel raíz en términos de las metaclases descritas, posteriormente se describe cada una de las metaclases involucradas.

3.2.1 Sintaxis Abstracta

El metamodelo ilustrado en la figura 5 es utilizado para generar tanto un Modelo de Dominio como para generar un Diagrama de Clases de Diseño, y que son representados gráficamente mediante Diagramas de Clases de UML. El uso de un único metamodelo para ambos casos es porque las construcciones utilizadas para expresar un Modelo de Dominio es un subconjunto de aquellas utilizadas para expresar un Diagrama de Clases de Diseño. Al describir las metaclases se detallará aquellos elementos no aplicables a un Modelo de Dominio.

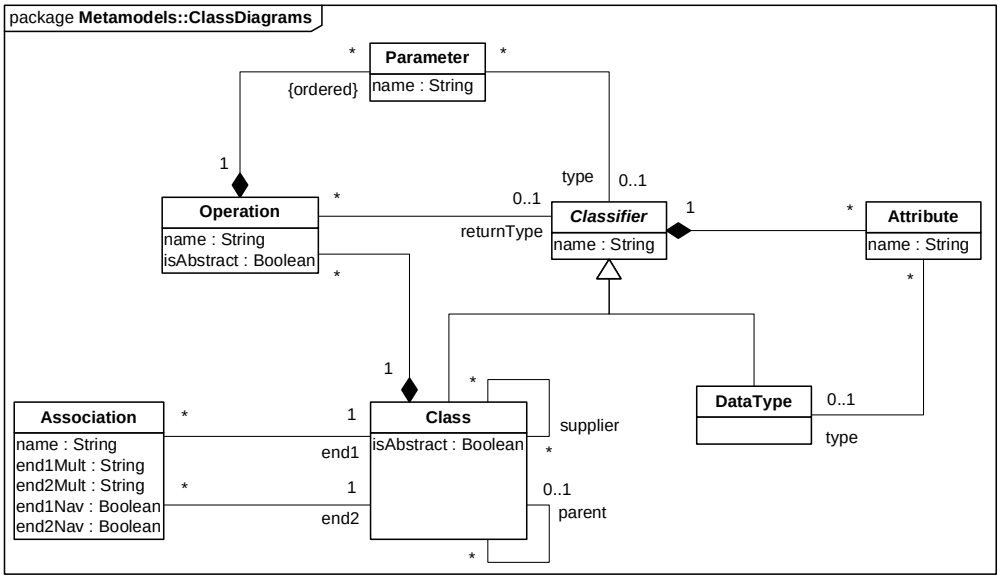

Figura 5: Metamodelo de estructura estática de clases

3.2.2 Estructuración

En el paquete Metamodels::ClassDiagramMetamodel se define una metaclase ClassModel implícita que representa una estructura de clases. Una estructura de clases está compuesta en su nivel raíz por los siguientes elementos:

- classifiers : Classifier[1..*] Las clases y datatypes participantes de la estructura.
- associations : Association[*] Las asociaciones entre clases participantes en la estructura.

Las clases y asociaciones se encuentran interrelacionadas entre sí. Al igual que en el caso anterior, las composiciones habilitan la inclusión de otros elementos en la estructura. Por ejemplo una operación puede incluirse en una estructura si está asociada a una clase.

3.2.3 Descripción de Metaclases

A continuación se describe cada una de las metaclases incluidas en el metamodelo presentado arriba, detallando el significado de sus atributos y asociaciones en las que participa.

Classifier

Representa un elemento que puede ser instanciado y que contiene atributos.

Atributos

- name : String El nombre del elemento.

Asociaciones

- attrs : Attribute[*] El conjunto de atributos del elemento.

DataType

Es una especialización concreta de classifier y representa un tipo de datos empleado para tipar atributos de otros classifiers. Puede ser un tipo atómico, en cuyo caso no tendrá atributos, o puede ser un tipo compuesto, en cuyo caso sí tendrá atributos.

Class

Es una especialización concreta de classifier y representa una clase en la estructura, a partir de la cual se crearán objetos.

Atributos

- isAbstract : Boolean Indica si la clase es abstracta o concreta, es decir, si será instanciable o no.

Asociaciones

- opers : Operation[*] El conjunto de operaciones que son propiedad de la clase.
- suppliers : Class[*] El conjunto de clases de la cual la clase depende. Que la clase Y pertenezca a suppliers de X indica que X depende de Y. No aplica a los modelos de dominio.
- parent : Class[0..1] La clase (opcional) de la cual la clase es una especialización. Que la clase X sea el parent de Y indica que Y es una clase derivada de X. No se soporta herencia múltiple.

Attribute

Representa un compartimiento tipado asociado a un classifier, de forma que sus instancias puedan almacenar información ahí.

Atributos

- name : String El nombre del atributo.

Asociaciones

- type : DataType[0..1] El tipo de datos (opcional) que determina los valores posibles del atributo.

Operation

Representa una transformación o consulta que un objeto de una cierta clase puede ser llamado a ejecutar. Puede recibir parámetros y devolver un resultado. No aplica a Modelos de Dominio.

Atributos

- name : String El nombre de la operación.

- isAbstract : Boolean Indica si la operación tiene un método asociado en la clase o no.

Asociaciones

- params : Parameter[*] La secuencia de parámetros que la operación tiene.
- returnType : Classifier[0..1] El tipo (opcional) del objeto o valor devuelto por la operación. Puede ser tanto una clase como un tipo de datos.

Parameter

Representa información que el objeto, sobre el cual se invoca la operación a la que el parámetro está asociado, recibirá. Por estar contenido en una operación, no aplica a Modelos de Dominio.

Atributos

- name : String El nombre del parámetro.

Asociaciones

- type : Classifier[0..1] El tipo (opcional) del parámetro. Puede ser tanto una clase como un tipo de datos.

Association

Representa una relación semántica entre clases. Indica la posibilidad de conectar instancias de las clases relacionadas. Solamente se consideran asociaciones binarias.

Atributos

- name : String El nombre de la asociación.
- end1Mult : String La multiplicidad correspondiente al primer extremo de la asociación. Indica cuántas instancias diferentes de la clase correspondiente a ese extremo podrán conectarse con una misma instancia de la clase en el otro.
- end2Mult : String La multiplicidad correspondiente al segundo extremo de la asociación.
- end1Nav : Boolean La navegabilidad correspondiente al primer extremo de la asociación. Indica si es posible que una instancia de la clase correspondiente a ese extremo podrá ser visible por las instancias (de las clases correspondientes al otro extremo) conectadas mediante la asociación. Esta visibilidad es la que habilita la invocación de operaciones. La multiplicidad correspondiente a un extremo de asociación no navegable es irrelevante. No aplica a modelos de dominio.
- end2Nav : Boolean La navegabilidad correspondiente al segundo extremo de la asociación. No aplica a Modelos de Dominio.

Asociaciones

- end1 : Class La primera de las clases que es relacionada con otra por la asociación.
- end2 : Class La segunda de las clases que es relacionada con la anterior a través de la asociación.

3.3 Relación entre los Metamodelos

A pesar de estar definidos en forma independiente, existe una relación entre los metamodelos presentados. De hecho, dicha relación es la que hace posible el presente trabajo. La transformación propuesta genera un modelo de clases a partir del cual es posible definir una configuración de objetos tal que las interacciones dadas puedan ocurrir. En términos prácticos, si las interacciones expresan el diseño de las operaciones del sistema correspondientes a un caso de uso, entonces la configuración de objetos mencionada representa la *colaboración* que realiza dicho caso de uso. En este sentido, las interacciones usadas como entrada constituyen el comportamiento de la colaboración.

Una interacción y una estructura de clases pueden referir a aspectos diferentes de *una misma* aplicación, por lo que es natural que exista alguna clase de conexión entre ellas. Una estructura de clases representa un "catálogo" de plantillas a partir de las cuales se podrán generar objetos. Una interacción representa un cierto intercambio de mensajes que persigue un cierto fin, entre objetos ya creados. Eso significa que para que una interacción pueda ocurrir en tiempo de ejecución, los objetos participantes de la misma deben haber sido creados previamente a partir del catálogo. El catálogo pues, debe contener todas las plantillas requeridas para la creación de los objetos participantes de la interacción. A su vez, para que un objeto pueda enviar un mensaje a otro, la clase del primero debe estar debidamente relacionada con la del segundo, y además, la clase del segundo debe contener una operación que pueda ser invocada como consecuencia del envío del mensaje.

De esta forma surge la noción de *consistencia* entre una interacción y una estructura de clases, la cual resulta fundamental para comprender la transformación y las propiedades de los resultados que entrega (ver sección 5.6). Supóngase la interacción I y la estructura de clases E, entonces para que I sea consistente con E se deben cumplir los siguientes invariantes:

- El tipo de un lifeline de I debe ser el nombre de alguna clase de E.
- El nombre de un mensaje de I debe ser el nombre de una operación en E perteneciente a la clase dada por el tipo del objeto destino del mensaje, y a su vez:
  - El tipo de retorno de un mensaje en I debe ser en E el nombre del classifier asociado como tipo de retorno de la operación correspondiente al mensaje.
  - El nombre y tipo de los argumentos de un mensaje en I se deben corresponder en el mismo orden con los parámetros de la operación en E.
- Para que un objeto pueda enviar un mensaje a otro en I, le debe ser posible al primero ganar algún tipo de visibilidad sobre el segundo. Las visibilidades locales o de parámetro no requieren en E elementos adicionales a los ya mencionados en los puntos anteriores. Si se trata de una visibilidad por asociación, en E debe existir una asociación navegable desde la clase del objeto origen del mensaje hacia la clase del objeto destino. Si además el objeto destino es un multiobjeto, se requiere que, en el extremo correspondiente a su clase en la asociación mencionada, el máximo de la multiplicidad sea "*" de forma de habilitar la presencia de la colección.

La transformación propuesta puede entenderse como un programa que genera, a partir de un conjunto de interacciones, una estructura de clases minimal, de forma tal que cada interacción

sea consistente con ella, es decir, preservando los invariantes presentados arriba. Se dice que la estructura de clases es minimal en el sentido de que la eliminación de uno de sus elementos provoca la pérdida de consistencia con al menos una de las interacciones de entrada. Dicho en otras palabras, solamente contiene lo indispensable para que todas las interacciones puedan producirse. Los detalles de dicha transformación son dados en la Sección 5.

4 Caso de Estudio

La lógica de la transformación definida en este trabajo es ilustrada mediate su aplicación a caso de estudio. En esta sección se presenta el problema a partir del cual se extrajeron los modelos que se utilizan con tal fin. Primeramente se presenta el contexto del problema y se describe el caso de uso a atacar. Luego se presenta el Modelo de Dominio y las interacciones que contienen el diseño de cada una de las operaciones del sistema de dicho caso de uso.

4.1 Problema

El problema utilizado como ejemplo en este trabajo es el caso de estudio que en [11] guía la presentación de todos sus ejemplos. Este caso de estudio está basado en un sistema para las cajas registradoras de una tienda, también conocido como POS (Point-of-Sale). Se trata de un problema simple aunque realista porque en la actualidad se escriben sistemas para POS según el paradigma de objetos. Un sistema POS es un sistema computacional utilizado (en parte) para registrar ventas y manejar pagos, típicamente en una tienda. El caso de uso a tratar es *Process Sale* debido a que es central para el caso de estudio original. Mediante *Process Sale* se realiza una venta de uno o más productos, .

En este trabajo se ataca una versión extendida de dicho caso de uso, la cual incluye diferentes formas de pago para una venta. Una *sinopsis* de dicha versión del caso de uso podría ser la siguiente:

> Un cliente llega a la caja con productos para comprar. El cajero comienza una nueva venta. Por cada producto, el cajero ingresa el identificador del producto y la cantidad de elementos a comprar. El sistema presenta el total de la venta. El cajero comunica al cliente dicho total y solicita el pago. El cliente realiza el pago, en efectivo o mediante un cheque. El sistema registra el pago y guarda la venta. El cliente se retira con los productos.

Para este caso de uso se identificaron las siguientes operaciones del sistema. Debido a la extensión de considerar diferentes medios de pago, la operación makePayment(am : Float) que aparece en la versión original del caso de estudio debió ser reemplazada por las dos últimas operaciones que aparecen en la enumeración siguiente.

- makeNewSale() que crea una nueva venta quedando como la venta en curso
- enterItem(id : Integer, qty : Integer) que ingresa qty elementos del producto identificado por id a la venta en curso
- endSale() : Float que finaliza la venta en curso y entrega el total a pagar por la misma

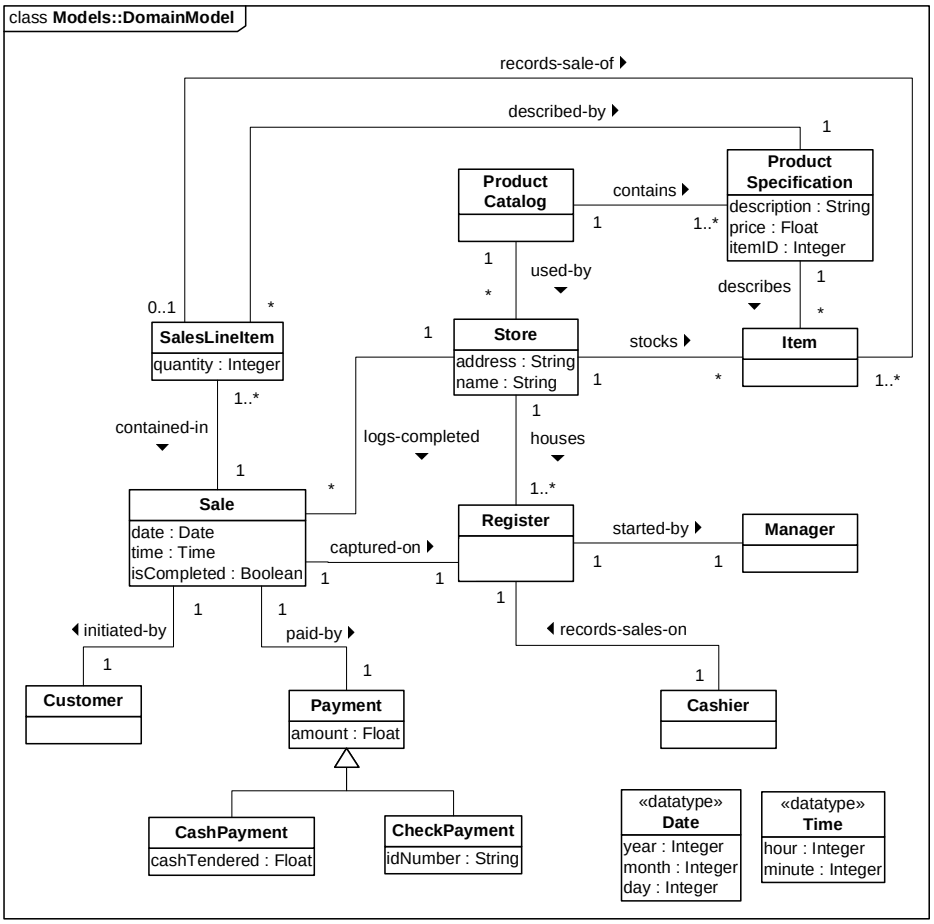

Figura 6: Modelo de Dominio del sistema para el POS

- makeCashPayment(am : Float, cash : Float) que registra el pago en efectivo de la venta en curso, dejando constancia de la suma de dinero recibida
- makeCheckPayment(am : Float, idN : String) que registra el pago con cheque de la venta en curso, dejando constancia del número de documento presentado por el cliente

En las siguientes subsecciones se presentan el Modelo de Dominio para el sistema del POS, así como las interacciones que son el resultado del diseño de cada una de las operaciones del sistema especificadas arriba.

4.2 Modelo de Dominio

El Modelo de Dominio presentado en la figura 6 ilustra los conceptos relevantes en la realidad de un POS, restringidos al caso de uso Process Sale. Esta versión es idéntica a la original con la

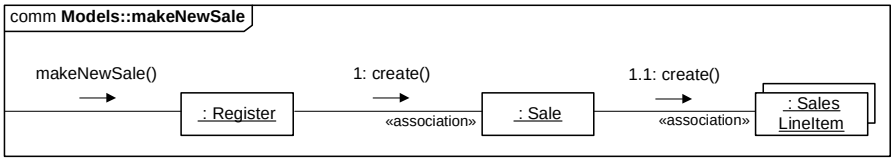

Figura 7: Diagrama de comunicación para makeNewSale

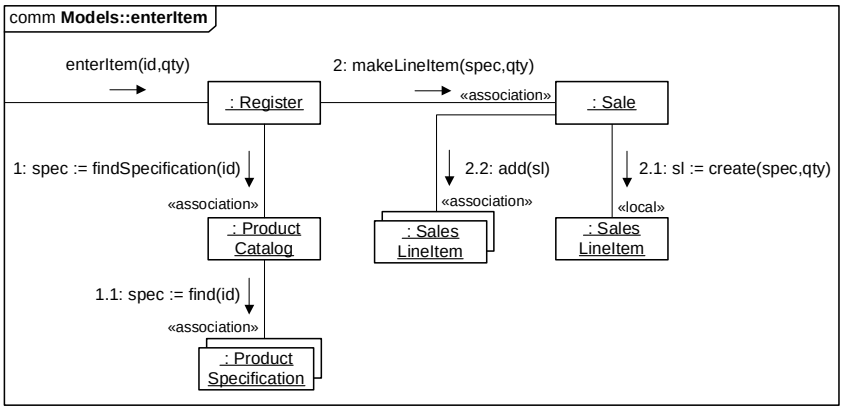

Figura 8: Diagrama de comunicación para enterItem

excepción de que se especializó el concepto de pago para contemplar pagos en efectivo y pagos con cheque, y que además el modelo incluye información de tipo para todos los atributos.

El concepto Store modela a la organización que posee los POS y registra las ventas finalizadas. Los POS son modelados por Register. La venta en curso es conectada con una caja mediante la asociación captured-on. Una venta cuenta con un SalesLineItem por cada tipo de producto comprado, para el cual se especifica la cantidad. Cada una de estas entradas tiene asociada la información del producto en cuestión (ProductSpecification), en particular su precio. Una venta finalizada tiene asociado un pago, que puede ser en efectivo o mediante cheque. En la siguiente subsección se presentan las interacciones que resuelven cada una de las operaciones del sistema ya identificadas.

4.3 Diseño de Interacciones

A continuación se presentan los Diagramas de Comunicación que ilustran el diseño realizado para cada una de las cinco operaciones del sistema identificadas antes. En todos los casos se definió a Register como el controlador del caso de uso, es decir, quien será responsable de manejar las operaciones del sistema.

El diseño de la operación makeNewSale() se presenta en la figura 7. Cuando el sistema recibe este mensaje, el controlador crea una nueva instancia de venta, quedando ésta como la venta en curso para esa caja e inicializando su colección de líneas de venta.

La figura 8 muestra el diseño de la operación enterItem(). Cuando el sistema recibe este

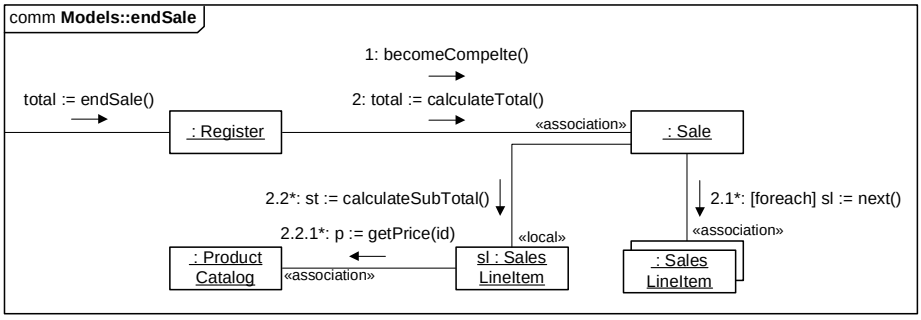

Figura 9: Diagrama de comunicación para endSale

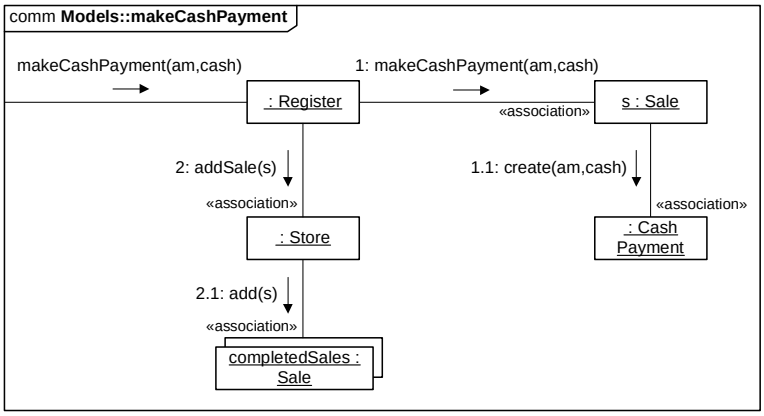

Figura 10: Diagrama de comunicación para makeCashPayment

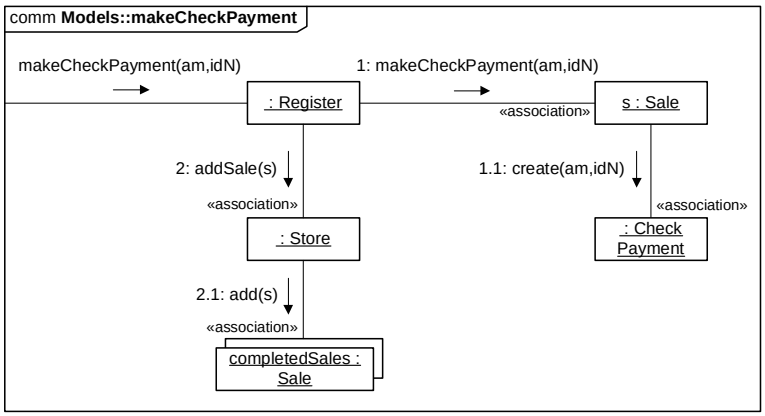

Figura 11: Diagrama de comunicación para makeCheckPayment

mensaje, el controlador busca en el catálogo de productos la especificación del producto que se está añadiendo a la venta. Esta información, junto con la cantidad de productos, es pasada a la venta en curso, la cual genera la nueva línea de venta correspondiente y la almacena en su colección asociada.

Cuando todos los productos a comprar fueron ingresados, se invoca la operación endSale(). Su diseño puede verse en la figura 9. El controlador primero notifica a la venta que se encuentra finalizada. Luego le solicita el monto total para entregarlo como resultado. Una venta no tiene precalculado su total, por lo tanto debe sumar los subtotales de cada línea de venta. Estos subtotales se calculan como la multiplicación entre el valor unitario del producto y la cantidad de productos de la línea.

Para realizar el pago de una venta existen dos alternativas, una es con dinero en efectivo y la otra es con un cheque. La información manejada en cada caso es diferente, por lo que se diseñó una operación del sistema para atender a cada una de las alternativas, sin embargo la lógica de ambas es la misma como se puede apreciar en las figuras 10 y 11. El controlador le indica a la venta en curso que el pago se realiza mediante efectivo o mediante cheque, según sea el caso la venta crea una instancia del pago adecuada. Finalizado esto, el controlador entrega la venta a la tienda, la cual la almacena entre las ventas finalizadas y pagadas. De esta forma el controlador queda momentáneamente sin venta en curso hasta que otra venta sea iniciada.

5 Transformación

Los Diagramas de Comunicación especifican, en la forma de interacciones, el comportamiento interno que el sistema debe realizar para satisfacer cada una de las operaciones del sistema a lo largo de un caso de uso. Para que todas estas interacciones puedan efectivamente ocurrir, es necesario que todos los participantes de las mismas existan y estén conectados entre sí en forma adecuada. Por lo tanto, debe contarse con una especificación de clases que sea consistente con las interacciones dadas y que permita producir tales configuraciones de objetos. Esta especificación se encuentra dada en el Diagrama de Clases de Diseño y su generación es el objetivo de la transformación.

La propuesta metodológica de Larman [11] que está basada en RUP [7], sugiere un método para generar un Diagrama de Clases de Diseño a partir del Modelo de Dominio (posiblemente restringido al caso de uso que esté siendo atacado), y de los Diagramas de Comunicación correspondientes a cada una de las operaciones del sistema involucradas en él. Dicho método es sistemático, por lo que aparece como natural la idea de construir una transformación automática que lo implemente. Sin embargo, su propósito original es su aplicación manual, no contando con un nivel de detalle suficiente para su implementación directa. Por esta razón fue necesario realizar un refinamiento del mismo.

En esta sección se presenta el detalle de la transformación propuesta, en la que se incluye el refinamiento realizado al método de Larman, y se ejemplifica su aplicación al caso de estudio. Primeramente se describe el método general de transformación tal cual fuera propuesto por Larman, junto con una interpretación del mismo. A continuación se enumeran las restricciones que se asumen para la implementación de la transformación sobre los modelos de entrada. Luego se describe la transformación misma y se presenta el resultado de aplicarla al caso de estudio. Tomando este ejemplo como base, se presentan luego los detalles de cada paso de la transformación. Finalmente se discuten aspectos referentes a la correctitud y la completitud de los modelos a ser

obtenidos como resultado de la transformación.

5.1 Método Base

En Larman [11] se presenta la idea general de un método para generar un Diagrama de Clases de Diseño a partir del Modelo de Dominio y de los Diagramas de Comunicación que detallan el diseño de las operaciones del sistema. Consiste en una secuencia de pasos que involucran la inspección de los Diagramas de Comunicación para extraer la información necesaria para producir el Diagrama de Clases de Diseño, apoyada además por información contenida en el Modelo de Dominio.

La información extraída, típicamente refiere a la clase de los objetos que participan de las interacciones, así como el nombre, origen y destino de los mensajes. Dado que un Modelo de Dominio puede entenderse además como una primera aproximación al Diagrama de Clases de Diseño [11], para la generación del mismo, la información obtenida de las interacciones puede ser complementada con datos obtenidos del Modelo de Dominio. Algunos conceptos presentes en el Modelo de Dominio (del mundo del análisis) pueden dar lugar, durante las actividades de diseño, a clases (en el mundo del diseño). De esta forma puede ser de utilidad aprovechar información relativa a dichos conceptos y sus relaciones (en un Modelo de Dominio se trata únicamente de asociaciones y generalizaciones), por ejemplo atributos, multiplicidades, etc.

Como se puede apreciar a continuación, la propuesta de Larman es de alto nivel y está orientada a su aplicación manual, dejando omitidos muchos de los detalles necesarios para completar cada uno de los pasos. Sin embargo, algunos de ellos se pueden inferir del ejemplo utilizado para apoyar su presentación. Los pasos propuestos son:

1. Identificar todas las clases que participan de la solución de un caso de uso. Hacer esto analizando los Diagramas de Comunicación.
2. Incluir las clases en el Diagrama de Clases de Diseño.
3. Replicar los atributos de los conceptos correspondientes en el Modelo de Dominio, agregando aquellos nuevos que sean necesarios.
4. Agregar las operaciones correspondientes a cada clase analizando los Diagramas de Comunicación.
5. Agregar la información de tipos a los atributos y operaciones.
6. Agregar las asociaciones necesarias para permitir las visibilidades por asociación requeridas en los Diagramas de Comunicación.
7. Agregar navegabilidades para indicar la dirección de cada visibilidad por asociación.
8. Agregar dependencias para reflejar los demás tipos de visibilidades utilizados.

En los pasos 1 y 2 se sugiere ver el tipo declarado de cada objeto participante en *alguna* interacción y crear una clase con ese nombre. El paso 3 sugiere establecer una correspondencia entre las clases generadas en los pasos anteriores y los conceptos incluidos en el Modelo de Dominio; si una clase X se corresponde con un concepto Y (es decir, una instancia de X tiene el mismo propósito en el software que una instancia de Y en el mundo real), se pueden utilizar los atributos

de Y como inspiración para determinar los de X. Exista tal correspondencia o no, el paso 3 sugiere también identificar qué atributos se necesitan para una clase de forma que sus instancias puedan resolver todos los mensajes recibidos. Notar que para ello es necesario comprender la lógica de cada interacción, que, a pesar de poder estar "sugerida" en un Diagrama de Comunicación en algunos casos simples, no se encuentra incluida de forma explícita. El paso 4 sugiere para cada clase ya generada, identificar todos los mensajes recibidos por alguna instancia de dicha clase y agregarle una operación por cada uno de esos mensajes. Algunas veces, los Diagramas de Comunicación incluyen información del tipo de los argumentos de los mensajes, así como del valor retornado; cuando esta información está disponible se puede completar la especificación de las operaciones agregadas en el paso 4. Lo mismo ocurre con aquellos atributos tomados del Modelo de Dominio. Esto es sugerido en el paso 5. La decisión del tipo de visibilidad empleado en cada envío de mensaje se toma durante la elaboración del Diagrama de Comunicación en el que ocurre el mensaje. En algunos casos el diseñador anota el diagrama con esta información, en otros no. En caso de no contar con esa información, quien genere el Diagrama de Clases de Diseño debe inferirla del contexto, cosa que en ocasiones puede resultar complejo o incluso imposible. Cuando se detecta que una instancia de clase X envía un mensaje a otra de clase Y con visibilidad por asociación, se crea una asociación navegable desde la clase X hacia la clase Y. Esto aparece sugerido en los pasos 6 y 7. Al igual que antes, para el paso 8 es de utilidad contar con información de visibilidad asociada a cada mensaje. Cuando se cuenta con dicha información se agregan dependencias entre clases para especificar visibilidad por parámetro o local. Es decir, si una instancia de clase X envía un mensaje a una instancia de clase Y teniendo alguno de estos dos tipos de visibilidades, entonces se agrega una dependencia de X hacia Y.

A pesar de que la interpretación anterior del método de transformación parece completa, varios detalles o casos de borde fueron dejados de lado. Como por ejemplo, la determinación de las multiplicidades, manejo de generalizaciones en el Modelo de Dominio, y casos donde es necesaria una dependencia a pesar de no ser necesaria visibilidad alguna, etc. Todos estos detalles deben ser resueltos para desarrollar una transformación que opere en forma automática. En la sección 5.3 se presenta la estructura de la transformación propuesta, mientras que en la sección 5.5 se presentan los detalles del refinamiento realizado.

5.2 Restricciones

Para la versión de la transformación reportada en este trabajo se asumen condiciones particulares sobre los modelos de entrada. En algunos casos se trata de simplificaciones al problema, en otros se trata de requerimientos que aseguren la disponibilidad de un mínimo de información que permita una operación interesante de la transformación. En algunos casos la carencia de esta información se traduce en la imposibilidad de generar ciertas partes del resultado, en otros se traduce en un resultado menos detallado. Estos aspectos se discuten en la sección 5.6. En particular, para los modelos de entrada se asume lo siguiente:

1. Todos los modelos de entrada son tanto sintáctica como semánticamente correctos.
2. El Modelo de Dominio se encuentra en un nivel de abstracción similar al de los Diagramas de Comunicación. Por ejemplo, el Modelo de Dominio del caso de estudio podría haber sido de más alto nivel, omitiendo los tipos de pago, y los detalles de cómo se registran los productos en una venta, entre otros.

3. No existen operaciones o atributos de clase.

4. No se distingue entre propiedades públicas o privadas de una clase.

5. Las multiplicidades utilizadas son las "clásicas", es decir, toman los valores "0..1", "1", "*" o "1..*".

6. No existen automensajes.

7. No existen autoasociaciones. Esto es porque en el metamodelo no se consideran nombres de roles, imprescindibles para eliminar la ambigüedad del modelo.

8. No existen múltiples asociaciones entre un mismo par de clases. Al igual que en el caso anterior , esto es por no soportar nombres de roles y por que los nombres de asociaciones son opcionales. Haciendo que el nombre de una asociación sea obligatorio, es posible modificar el mecanismo actual de identificar asociaciones y así eliminar esta restricción.

9. No se consideran visibilidades de tipo "global" en diagramas de comunicación. Esto es poco común y en general implica el uso del patrón de diseño Singleton [5], el cual requiere de atributos y operaciones de clase.

10. El diseño de una operación del sistema se presenta en un único Diagrama de Comunicación. Eso quiere decir que el punto de entrada de cualquier diagrama se asume que es una operación del sistema.

5.3 Modelo de la Transformación

En la presente sección se describe la transformación propuesta. Se comienza describiendo su estructura, y a continuación su comportamiento.

5.3.1 Estructura

La estructura de la transformación se ilustra en la figura 12. La transformación consta de dos clases de Kermeta; Main y Transformation, las cuales son el contenido del paquete Transformations. La clase Main es responsable de inicializar los modelos de entrada de la transformación a partir del repositorio de modelos (mediante loadInteractions() y loadDomainModel()), así como de salvar en el repositorio el resultado (mediante saveClassModel()). Asimismo dispone de la operación main() que es el punto de entrada de toda la transformación, en forma equivalente a la operación main() de una aplicación Java o Main() de una aplicación de consola de C#. Esta clase depende de las clases Interaction y ClassModel explicadas en las secciones 3.1.2 y 3.2.2 respectivamente, tal cual se aprecia en la figura 12. Las operaciones mencionadas son las encargadas de instanciar estas clases e inicializarlas de acuerdo a la información almacenada en los archivos XMI. Una instancia de Main mantiene una referencia a una instancia de Transformation a quien le delega la transformación.

La clase Transformation es responsable de realizar la transformación propiamente. Una instancia de esa clase es inicializada con una secuencia de instancias de Interaction (es decir, todos los Diagramas de Comunicación), y una instancia de ClassModel (es decir, el Modelo de Dominio), hacia los cuales mantiene una referencia. La operación transform() de dicha clase crea una instancia de ClassModel, que será el Diagrama de Clases de Diseño resultante y mantiene una referencia

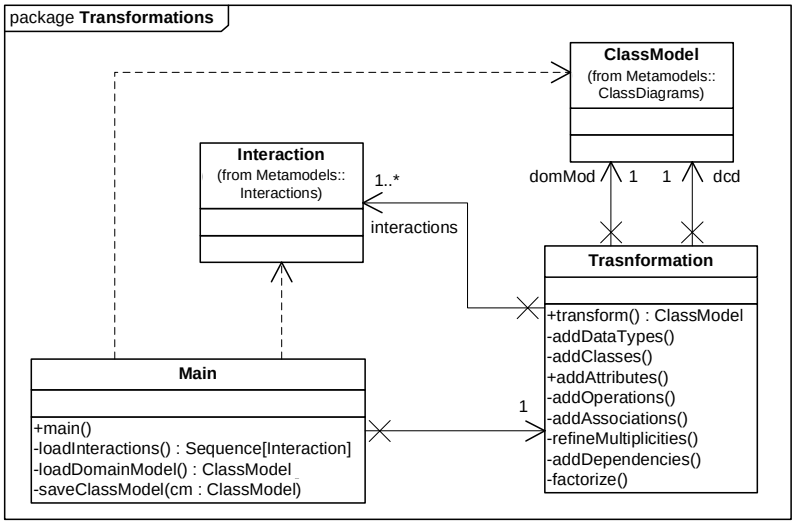

Figura 12: Modelo estructural de la transformación

hacia ella, denominada dcd. Esta instancia será poblada de elementos en forma sucesiva a través de las siguientes operaciones auxiliares, las cuales se corresponden casi directamente con los pasos descritos en la sección 5.1.

- addDataTypes() copia los tipos de datos del Modelo de Dominio asociado en el modelo de clases dcd.
- addClasses() agrega las clases necesarias al modelo de clases dcd.
- addAttributes() agrega los atributos necesarios al modelo de clases dcd.
- addOperations() agrega las operaciones necesarias al modelo de clases dcd.
- addAssociations() agrega las asociaciones necesarias al modelo de clases dcd.
- refineMultiplicities() refina las multiplicidades ya generadas para las asociaciones en el modelo de clases dcd.
- addDependencies() agrega las dependencias necesarias al modelo de clases dcd.
- factorize() abstrae a una superclase del modelo de clases dcd las propiedades presentes en todas sus subclases.

Estas operaciones auxiliares manipulan la estructura interna de las interacciones y los modelos de clases. Por tal razón, la clase Transformation depende también de todas las clases pertenecientes a los paquetes Metamodels::Interactions y Metamodels::ClassDiagrams. Todas estas clases son también clases de Kermeta y son generadas automáticamente por el ambiente de Kermeta a partir de los archivos Ecore que contienen los metamodelos.

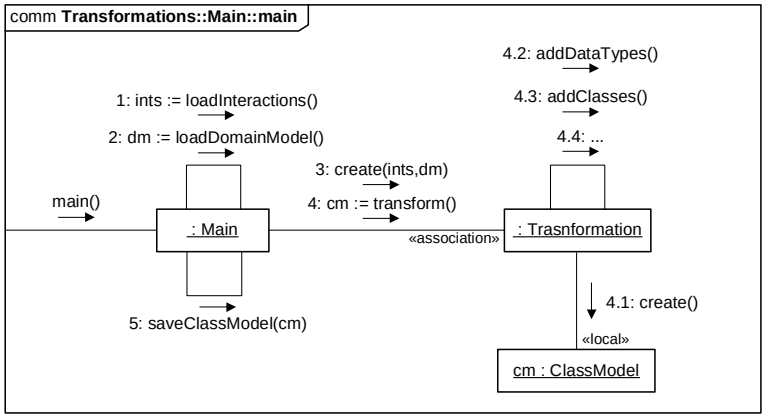

Figura 13: Comportamiento de la transformación

5.3.2 Comportamiento

La ejecución de la transformación es relativamente simple y su comportamiento se ilustra en la figura 13. La transformación comienza cuando se invoca la operación main() sobre una instancia de la clase Main. Como consecuencia de ello, los modelos de entrada son cargados, una instancia de Transformation es creada e inicializada con dichos modelos, y finalmente le es invocada la operación transform(). Dicha operación retorna el Diagrama de Clases de Diseño final, el cual es almacenado por la instancia de Main que lo solicitó.

La operación transform() crea un modelo clases vacío, poblándolo de elementos a partir de los modelos referenciados por el objeto implícito. La operativa de esta operación consiste en aplicar secuencialmente las operaciones auxiliares descritas en la sección anterior, cada una transformando el modelo de clases inicialmente vacío, en una suerte de arquitectura de *pipeline* [22], hasta obtener el modelo final. Dichas operaciones agregan elementos nuevos al modelo, o modifican elementos existentes. Los detalles de estas transformaciones son presentados en la sección 5.5.

5.4 Resultado del Caso de Estudio

Previo a pasar a los detalles de la transformación se presenta a continuación el modelo resultante de aplicar la transformación a los modelos del caso de estudio presentados en la Sección 4. Este modelo a su vez será utilizado en la sección 5.5 para ilustrar algunos aspectos específicos de la transformación.

El Diagrama de Clases de Diseño resultante de la transformación para el caso de uso *Process Sale* es mostrado en la figura 14. En líneas generales resulta idéntico al original presentado en [11], sin embargo exhibe algunas diferencias que se discuten a continuación.

En el modelo original se incluye inexplicablemente (de acuerdo al método propuesto) una asociación navegable uses desde la clase Store hacia la clase ProductCatalog. Esta asociación no se existe en el modelo generado puesto que ninguna interacción requiere visibilidad por asociación entre instancias de las clases mencionadas.

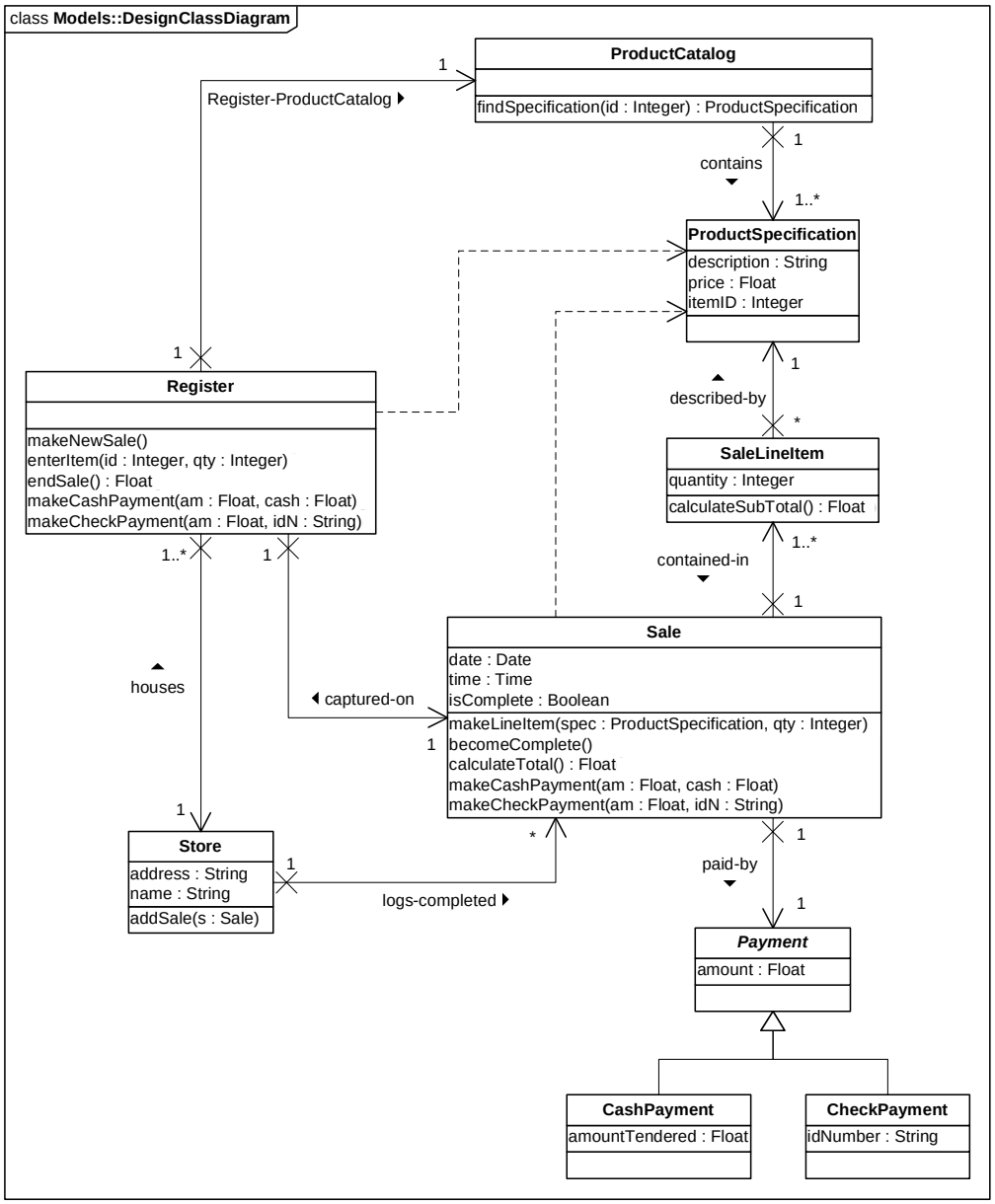

Figura 14: Representación del resultado de la transformación aplicada al caso de estudio

La asociación houses aparece en el modelo original con las navegabilidades invertidas, es decir, es navegable desde Store hacia Register, cuando en el modelo generado es navegable desde Register hacia Store. Esto último es lo correcto, puesto que existe en los diagramas correspondientes a las operaciones makeCashPayment() y makeCheckPayment() (ver figuras 10 y 11 respectivamente) un mensaje addSale() desde una instancia de Register a otra de Store, y no viceversa. Esta diferencia se atribuye a un error en la confección del diagrama presentado en [11].

Finalmente, otras diferencias corresponden a las operaciones makeCashPayment() y makeCheck-Payment() en las clases Register y Sale del modelo generado en lugar de makePayment() como en el modelo original. También el modelo generado contiene las clases CashPayment y CheckPayment, ambas subclases de Payment, que no existen en el modelo original. Esta diferencia, naturalmente se debe a que los diferentes mecanismos de pago no fueron considerados en el caso de estudio original.

5.5 Procesamiento de la Transformación

En esta sección se presentan los detalles de la transformación propuesta. El enfoque a aplicar no es mostrar el algoritmo que resuelve cada una de las operaciones de la clase Transformation, que fueran presentadas en la sección 5.3.1, sino que se visitará cada metaclase del metamodelo Metamodels::ClassDiagrams (ver figura 7), y se discutirán las situaciones bajo las cuales los valores de cada uno de sus elementos (metaatributos y metaasociaciones) son generados. Esto provee una forma más modular de explicar la transformación. Los algoritmos se pueden obtener de los fuentes Kermeta que pueden ser accedidos en `http://www.dcc.uchile.cl/~avignaga/transformation`.

5.5.1 Tipos de Datos

Los tipos de datos se asumen declarados en el Modelo de Dominio, por lo cual simplemente se copian todos hacia el Diagrama de Clases de Diseño. Antes de crear un nuevo tipo de datos, es necesario verificar que no exista ya. El nombre del tipo de datos lo identifica.

Nombre El nombre del nuevo tipo de datos es copiado del nombre del tipo de datos original.

Atributos Para cada atributo del tipo de datos original (si es que tiene), se toma el nombre, y se busca el tipo de datos correspondiente en el Diagrama de Clases de Diseño y se asigna como el tipo del atributo. Los tipos de datos se asumen en orden de dependencia, por lo cual si un atributo de un tipo T es de tipo T', necesariamente T' fue copiado antes que T al Diagrama de Clases de Diseño. De esta forma, la búsqueda de un tipo nunca falla.

Este esquema si bien es simple, tiene la desventaja de los tipos de datos utilizables en los Diagramas de Comunicación son aquellos declarados en el Modelo de Dominio. Por otra parte, si algún tipo de datos no fuese utilizado, y por lo tanto no necesitase ser copiado al Diagrama de Clases de Diseño, igual sería copiado. Una solución a ambos problemas puede ser una recorrida por todos los mensajes, inspeccionando los tipos de los argumentos y el tipo de retorno de cada uno. Para cada uno de ellos, si coincide con un tipo de datos del Modelo de Dominio, se copia al Diagrama de Clases de Diseño en forma completa (incluyendo los tipos del cual pueda depender), sino, es necesario distinguir si se trata de un nuevo tipo de datos o de una clase. Si se tratara de una clase, su nombre debería coincidir con el tipo de al menos un lifeline de una de las interacciones.

Si es el caso, por supuesto no se agrega como tipo de datos al Diagrama de Clases de Diseño, pero de no ser así, o bien se trata de una clase para la cual ninguna de sus instancias recibe o envía un mensaje, o bien se trata de un tipo de datos que no aparece en el Modelo de Dominio. Dado que la primera de las posibilidades es poco factible, se puede asumir que se trata de un nuevo tipo de datos. De esta forma, se puede crear un tipo de datos en el Diagrama de Clases de Diseño, pero del cual sólo se conoce su nombre.

5.5.2 Clases

Una clase en el Diagrama de Clases de Diseño es consecuencia de la existencia de al menos un objeto de dicha clase en al menos uno de los Diagramas de Comunicación. Por cada tipo distinto de objeto, se crea una clase nueva. Antes de crear una nueva clase, es necesario verificar que no exista ya. El nombre de la clase la identifica.

Nombre El nombre de una clase es tomado del valor type de un lifeline.

Atributos Si la nueva clase tiene un elemento P correspondiente (con el mismo valor de name) en el Modelo de Dominio, se copian todos los atributos de C a la nueva clase en forma similar al caso de los tipos de datos.

Adicionalmente, si alguna instancia de clase C recibe algún mensaje cuyo nombre sea de la forma *getX()* o *setX()*, eso sugiere el acceso a un atributo x (en minúsculas) en la clase C. Por lo tanto si tal atributo no existe en la clase, es agregado. El tipo del nuevo atributo es determinado por el tipo del argumento, en caso de ser un mensaje set, o del tipo del valor retornado, en caso de tratarse de un mensaje get.

En ese sentido, el mensaje calculateTotal() hacia una instancia de Sale en el caso de estudio original fue denominado getTotal(). Para el caso de estudio del presente trabajo dicho nombre fue cambiado porque hubiese sido tratado por la transformación como un acceso a un atributo. Esto habría causado, primero, que la clase Sale contase con un atributo total , lo cual contradiría la idea original de que el total de una venta fuese calculado, y segundo, que la operación correspondiente al mensaje getTotal() no apareciera entre las operaciones de la clase Sale (la razón de esto se discute en la sección 5.5.3). Una alternativa a esto podría haber sido que como el mensaje en cuestión debe tener la propiedad locallyResolved en falso, se podría modificar la transformación de forma de incluir la operación en la clase, así como el atributo, marcándolo además como *derivable*. Este enfoque requeriría de la modificación del metamodelo Metamodels::ClassDiagrams para incluir en la metaclase Attribute la propiedad isDerivable.

Instanciabilidad El valor de isAbstract se pone en falso, excepto cuando la nueva clase tiene un elemento correspondiente en el Modelo de Dominio con su valor de isAbstract en verdadero, o cuando a la clase se le agrega al menos una operación sin método (el valor de isAbstract de la operación es verdadero).

Clase base Cuando la nueva clase tiene un elemento C correspondiente en el Modelo de Dominio, y dicho elemento tiene a C' como clase base, en el Diagrama de Clases de Diseño se crea recursivamente la clase C' y se la designa como clase base de la nueva clase.

Proveedores La determinación de dependencias entre clases es discutido en la sección 5.5.5.

Operaciones La creación de operaciones para una clase es discutido en la sección 5.5.3.

Tomando como referencia a los Diagramas de Comunicación solamente es imposible conocer exactamente los atributos que una clase debe poseer. Como fuera discutido arriba, el acceso al valor de un atributo desde el exterior evidencia la existencia del atributo en la clase del objeto destino. Sin embargo, dado que no es objetivo de los Diagramas de Comunicación expresar un pseudocódigo de la forma en que los mensajes son resueltos, aquellos atributos que son utilizados únicamente para resolver mensajes (es decir, que no accedidos desde otros objetos) no aparecen mencionados en las interacciones. Por tal razón, no es posible predecir la correctitud del conjunto de atributos de una clase. En los casos en que una clase en el Diagrama de Clases de Diseño tiene un correspondiente en el Modelo de Dominio, podría llegar a ocurrir que los atributos determinados como de interés en este último coincidan con los necesarios para que la clase en el modelo de diseño lleve adelante sus responsabilidades (tal cual ocurrió en el caso de estudio). Sin embargo esa no es la situación más común. Este inconveniente se agrava cuando la clase de diseño no tiene un correspondiente en el modelo de análisis; en ese caso la presencia de un atributo dependerá exclusivamente de la existencia de accesos al mismo en alguna interacción.

5.5.3 Operaciones

Una operación *op* se agrega a una clase C como consecuencia de la existencia de un mensaje de llegada m a una instancia de C en alguna de las interacciones. La operación *op* será agregada a C sólo si no existe ya o en casos particulares que se discuten a continuación. El criterio de igualdad para operaciones es la coincidencia tanto del nombre, como de la cantidad y tipo de los parámetros. Los parámetros de una operación son una secuencia por lo que el orden de comparación debe ser posición a posición.

De esta forma se da soporte a la sobrecarga de operaciones ya que no basta que los nombres de dos operaciones coincidan para ser consideradas iguales. También se da soporte a la redefinición de operaciones. En un Diagrama de Clases de UML, la redefinición de una operación es expresada por la repetición de la operación en una subclase.

Nombre El nombre de *op* es tomado del nombre de m.

Presencia de método El valor de isAbstract de *op* es falso, excepto cuando el valor de locallyResolved de m es falso y no existe ningún mensaje anidado en m. Eso quiere decir que m no puede ser resuelto completamente por el objeto destino, y además ese objeto no delega a ningún otro la resolución de al menos una parte de m (es decir, no existe un mensaje anidado en m). Esto indica que el destino no puede tener un método para resolver m. Como observación, un mensaje *m'* es anidado respecto a m, si el origen de *m'* coincide con el destino de m, y además el número de secuencia de m está totalmente contenido en el inicio del número de secuencia de *m'*. Por ejemplo, el mensaje getPrice() desde el objeto sl de tipo SalesLineItem hacia una instancia de ProductCatalog está anidado respecto al mensaje calculateSubTotal() en la interacción correspondiente a endSale() (ver figura 9); el número de secuencia del primero es 2.2.1 y el del segundo es 2.2, y el origen de getPrice() es el objeto sl.

Parámetros Los parámetros de *op* se obtienen a partir de los argumentos de m; se copia el nombre, y a partir del tipo del argumento se busca el clasificador correspondiente y se lo

asigna como tipo del parámetro. Los parámetros de *op* se corresponden uno a uno con los de m y en el mismo orden.

Tipo de retorno El tipo de retorno de *op* se determina buscando el clasificador cuyo nombre coincida con el del tipo de retorno de m (si existe).

En [11] se discute la utilidad de especificar para una clase en un Diagrama de Clases de Diseño las operaciones de creación de objetos o de acceso a sus atributos. En particular, una clase con n atributos podría llegar a tener $2n$ operaciones de acceso, lo cual podría comprometer la legibilidad del diagrama resultante. Por tal razón, se tomó el criterio de no generar operaciones para los constructores así como tampoco para las operaciones de acceso. Una alternativa a esto podría ser que su generación fuese opcional. Otra, podría ser la anotación de los atributos con información que permita conocer la existencia de operaciones de lectura y escritura sobre cada uno de ellos. Las propiedades readOnly y changeable de UML aplican a atributos aún respecto a los métodos de la propia clase, lo cual hace que no sea apropiado su uso en este tipo de casos, ya que lo que se desea especificar es su acceso pero desde objetos de otras clases. Una posibilidad es agregar a cada atributo las propiedades booleanas isReadable e isWritable para indicar la presencia de una operación get y una operación set respectivamente. Nuevamente, esto requiere la modificación del metamodelo.

Un mensaje a un multiobjeto de clase C no genera una operación en la clase C. Los multiobjetos representan una colección, y se desea tratar a las colecciones de manera uniforme, esto es, que todas ellas ofrezcan en forma genérica operaciones típicas de colecciones (usualmente búsquedas por clave e iteraciones), y sin operaciones particulares que dependan del tipo de objetos contenidos en ella. Según [11], los Diagramas de Clases de Diseño no modelan explícitamente las colecciones como una entidad de diseño de por sí, sino que quedan implícitamente especificadas por la multiplicidad en el extremo correspondiente. Es tarea del implementador incluirlas en la implementación. De esta forma, los mensajes a las colecciones son siempre los correspondientes a las interfaces de diccionario e iterador, no aportando nueva información. De acuerdo con este criterio, los mensajes a los multiobjetos simplemente son ignorados por la transformación.

5.5.4 Asociaciones

Una asociación a entre las clases C_1 y C_2 es creada como consecuencia de la existencia de al menos un mensaje m desde una instancia de C_1 hacia una de C_2 con visibilidad por asociación. Antes de crear una nueva asociación es necesario verificar que la asociación no exista ya. Dado que el nombre de una asociación es opcional, el criterio de comparación de asociaciones es por el nombre de las clases que ésta relaciona. En particular, existen dos combinaciones posibles dado que cada asociación tiene dos extremos; que C_1 sea el primer extremo y C_2 el segundo, o viceversa. Esto impide que se consideren múltiples asociaciones entre un mismo par de clases. Si el nombre de una asociación fuese obligatorio, podría modificarse el criterio contando como iguales dos asociaciones con el mismo nombre y con el mismo conjunto (para ignorar el orden) de clases relacionadas. Es importante observar que los nombres de asociación no son únicos en general, sin embargo parece razonable exigir que sí lo sean cuando las clases relacionadas coinciden. Esto no podría ser impuesto a nivel del metamodelo, sino que debería ser verificado en cada modelo construido.

Nombre El nombre de la asociación será "C_1–C_2". La excepción se da cuando en el Modelo de

Dominio existe una asociación entre C_1 y C_2, en cuyo caso se toma como nombre el nombre de dicha asociación.

Clases relacionadas La clase en end1 es C_1 y la clase en end2 es C_2 (la clase del objeto que origina el mensaje corresponde al primer extremo).

Multiplicidades Cuando a es una asociación nueva, el extremo correspondiente a C_1 no es navegable, por lo que podría considerarse que su multiplicidad es irrelevante. Esto es cierto solamente cuando se cuenta con la seguridad de que dicho extremo nunca pasará a ser navegable como parte de un procesamiento posterior. Como en general no se sabe si aparecerá un mensaje m' que lo vuelva navegable, es necesario determinar esa multiplicidad de la mejor forma posible. Para determinar la multiplicidad del lado de C_1 se busca inspiración en el Modelo de Dominio. Si existe un valor correspondiente, se lo considera como el mejor candidato. En caso de que no exista, se determina "0..1" como el valor menos restrictivo, ya que permite aumentar tanto el mínimo a "1" o el máximo a "*" cuando sea conveniente. Esto se discute a continuación. Cuando a ya existe en el Diagrama de Clases de Diseño, esta multiplicidad en a no es afectada.

El máximo de la multiplicidad del lado de C_2 se determina dependiendo de si el objeto destino de m es un multiobjeto o no. Si se trata de un multiobjeto, en todos los casos el máximo de la multiplicidad será "*", si se trata de un objeto simple, el máximo de la multiplicidad será "1". Para el mínimo de dicha multiplicidad se busca una correspondencia en el Modelo de Dominio; si existe un valor correspondiente, ese será el mínimo, en caso de que no exista el mínimo será "0" (por la misa razón que arriba). En caso de que a ya exista en el Diagrama de Clases de Diseño, el mínimo de la multiplicidad no es afectado, y el máximo es aumentado a "*" en caso de que el destino de m sea un multiobjeto. Este mecanismo explica la razón de haber elegido a "1" como el valor por defecto del máximo de una multiplicidad.

Lo descrito arriba aplica a la determinación de las multiplicidades de a utilizando únicamente información relativa a m. Dado el criterio de "aumento" presentado para los máximos, se considera que una vez determinadas todas las multiplicidades, éstas presentan en su valor máximo final. Sin embargo, se puede observar en la figura 14 que en el resultado de la transformación en pocos casos las multiplicidades conservan en su mínimo su valor por defecto (recuérdese que dicho valor es "0"). Eso ocurre porque es posible refinar los valores mínimos de las multiplicidades aplicando el siguiente criterio. Si por ejemplo la multiplicidad en el extremo correspondiente a C_1 tiene mínimo "0", significa que para un objeto de clase C_2 la conexión con un objeto de clase C no es obligatoria, mientras que si el mínimo es "1", sí lo es. Es necesario poder detectar esta opcionalidad. Si en algún punto de alguna interacción una instancia de C_2 crea, recibe como parámetro, o recibe como resultado de un mensaje a una instancia de C_1, se podría asumir que a partir de ello se estableció una conexión que antes no existía. Esto permitiría inferir con cierta confianza que ante esta situación la conexión es opcional. Por lo tanto, en caso de no detectarse tal caso, la conexión sería obligatoria y el mínimo de la multiplicidad debería aumentarse a "1". Una aplicación de esto se puede apreciar en el caso de la asociación logs-completed entre Store y Sale; en las interacciones correspondientes a makeCashPayment() y makeCheckPayment() una instancia de Store recibe en addSale() una instancia de Sale, lo que permite suponer que en algún momento dicha instancia de Store no se encontraba conectada con ninguna de Sale. En cambio, en la asociación Register-ProductCatalog (que no existe en el Modelo de Dominio),

no ocurre ninguna de las tres condiciones mencionadas, por lo que el mínimo por defecto de "0" fue aumentado a "1". Este es el funcionamiento de la operación refineMultiplicities() de la clase Transformation.

Navegabilidades Si la asociación a no existe, la navegabilidad en el extremo de C_1 es falsa y la navegabilidad en el extremo de C_2 es verdadera. Si la asociación ya existía, la navegabilidad en el extremo de C_2 pasa ser verdadera, mientras que la otra permanece incambiada. Esto permite por lo menos mensajes desde objetos de clase C_1 hacia objetos de clase C_2, como m.

5.5.5 Dependencias

La dependencia es una relación asimétrica entre clases donde el elemento independiente se denomina *proveedor* y al dependiente se lo denomina *cliente*. La relación es tal que un cambio en el proveedor puede afectar al cliente. Las asociaciones navegables y las generalizaciones se consideran una forma más fuerte de dependencia, por lo que de existir alguna de éstas la dependencia se asume existente y no se representa explícitamente. Por ejemplo, si la clase C_1 tiene una asociación navegable hacia C_2, se dice que C_1 depende de C_2; similarmente, si la clase C_1 es subclase de C_2 también se dice que C_1 depende de C_2. Por lo tanto, en un Diagrama de Clases de Diseño se agregará una dependencia de C_1 a C_2 si se produce una forma pura de dependencia entre dichas clases.

Estas formas de dependencia que serán capturadas en un Diagrama de Clases de Diseño son las siguientes:

- Una instancia de clase C_1 envía un mensaje m a una instancia de clase C_2 mediante visibilidad de parámetro. Eso significa que la instancia de clase C_1 recibió a la instancia de clase C_2 como parámetro en el mensaje que contiene inmediatamente a m.

- Una instancia de clase C_1 envía un mensaje m a una instancia de clase C_2 mediante visibilidad local. Eso significa que la instancia de clase C_1 o bien creó a la instancia de clase C_2 durante la ejecución del mensaje que inmediatamente contiene a m, o bien la recibió como resultado de un mensaje enviado antes de m.

- Los casos de envío de mensaje mediante los tipos de visibilidad mencionados no son los únicos en los cuales se puede producir una dependencia. No es obligatorio que un objeto le envíe un mensaje a otro recibido como parámetro, u obtenido como resultado del envío de otro mensaje. Aún así se establece una relación de dependencia.

  Como ejemplo de esto, en la interacción correspondiente a enterItem() (ver figura 8), una instancia de Register recibe como resultado del envío del mensaje findSpecification() una instancia de clase ProductSpecification; esto crea la dependencia desde Register hacia ProductSpecification. Inmediatamente después en la misma interacción, esa misma instancia de ProductSpecfication (llamada spec) le es pasada como argumento a una instancia de Sale en el mensaje makeLineItem(), produciendo la dependencia desde Sale hacia ProductSpecification.

En todos los casos de ocurrencia de dependencia, aún cuando se sepa que no existe una relación de generalización entre la clase C_1 y la clase C_2, aún es posible que otro mensaje previamente procesado haya provocado la creación de una asociación navegable desde C_1 hacia C_2. Por tal razón, cuando se detecta la presencia de una dependencia es preciso verificar que no exista tal

generalización o asociación, en cuyo caso la dependencia no es creada. A manera de ejemplo, en la interacción correspondiente a endSale() (ver figura 9), el mensaje calculateSubTotal() desde una instancia de Sale a la instancia sl de SalesLineItem, por tener asociada visibilidad local y en virtud del segundo punto de arriba, debería provocar una dependencia desde Sale hacia SalesLineItem. Sin embargo, el mensaje next() desde la misma instancia de Sale hacia el multiobjeto de tipo SalesLineItem pudo haber sido procesado antes, y por tener visibilidad de tipo asociación, ya habría determinado una asociación navegable desde Sale hacia SalesLineItem, dejando sin efecto la dependencia recién detectada.

5.5.6 Factorización

Una aspecto que fue omitido, o dejado implícito, en el método de la sección 5.1 es el de la factorización de propiedades comunes a varias clases. Esto es, la generalización de atributos, operaciones o asociaciones comunes a distintas clases. En una aplicación manual del método sería posible apelar a la intuición y, comprendiendo la semántica de las diferentes clases y sus propiedades, realizar tal proceso de abstracción.

En la presente transformación no es posible contar con la semántica de los elementos mencionados, ni tampoco con su intuición. Sin embargo se definió un criterio para detectar situaciones en las cuales es posible abstraer propiedades. Cuando se tiene una clase C que es clase base de las clases C_i y se detecta que todas las clases C_i poseen una misma propiedad, entonces es posible abstraerla, eliminándola de todas las clases C_i y agregándola en la clase C. El criterio para determinar esta situación depende del tipo de propiedad:

Atributos En todas las clases C_i debe existir un atributo con el mismo nombre y el mismo tipo. En ese caso se crea un atributo en la clase C con ese nombre y tipo, y se elimina de todas sus subclases.

Asociaciones Existe una clase C' tal que tiene una asociación con cada una de las C_i. En ese caso se eliminan todas esas asociaciones y se crea una nueva entre C y C'. En este tipo de situaciones sería de esperarse que las multiplicidades y las navegabilidades coincidieran posición a posición en todas las asociaciones eliminadas, en cuyo caso la nueva asociación tomaría sus valores directamente de ellas. Sin embargo, dado que en la transformación, por particularidades de los mensajes, es posible que no todas las asociaciones (salvo en el extremo correspondiente a C_i) coincidan (en multiplicidades y navegabilidades), se debe buscan un mecanismo para realizar una "conjunción" de asociaciones. Cuando todos los valores coinciden, se trata del caso trivial ya mencionado. Cuando existen diferencias, la navegabilidad en un extremo de la nueva asociación se define como la *conjunción lógica* de las navegabilidades de todas las asociaciones en el extremo "correspondiente". El caso de las mutiplicidades es similar; el mínimo es el mínimo de todas las multiplicidades, y el máximo es el máximo de todas ellas. De esta forma, se entiende que se podría llegar a debilitar alguna de las restricciones estructurales mencionadas, pero no sería posible fortalecer ninguna de ellas. De ocurrir esta discrepancia, la transformación podría emitir un reporte informando la situación.

Operaciones El criterio para determinar la coincidencia de operaciones ya fue descrito antes; coincidencia en el nombre de la operación, y coincidencia tanto el la cantidad de parámetros como en el tipo de los mismos, posición a posición.

La operación factorize() de la clase Trasnformation tiene como propósito detectar alguna de estas situaciones y modificar el Diagrama de Clases de Diseño en forma acorde a lo descrito arriba.

En el caso de estudio efectivamente ocurre un caso de factorización, en particular, factorización de asociaciones. Previo a la factorización, existe una asociación Sale-CashPayment y otra asociación Sale-CheckPayment, correspondientes al interés de que una venta mantenga una asociación con el pago correspondiente. Se está pues en la situación de que todas las subclases de Payment (la clase C en la explicación de arriba) tienen una asociación con la clase Sale (la clase C'). Aparece como razonable modificar el Diagrama de Clases de Diseño de forma tal que una venta esté *estáticamente* asociada con un pago, y en tiempo de ejecución se asocie con la variante específica de pago que corresponda. De esta forma, la operación factorize() detecta la situación y elimina ambas asociaciones, creando una asociación entre Sale y Payment de acuerdo a lo discutido. Una asociación tal ya existe en el Modelo de Dominio, por lo que se nombra a la nueva asociación con su nombre, es decir, paid-by. Los valores de paid-by se determinan en este caso en forma simple, puesto que para las antiguas asociaciones las multiplicidades eran "1" en ambos extremos, y navegables únicamente desde la venta hacia los pagos particulares. Por lo tanto paid-by también tiene multiplicidad "1" en ambos extremos, siendo el extremo correspondiente a Sale no navegable, mientras que el correspondiente a Payment sí.

5.6 Análisis de la Transformación

Para concluir la presentación de la transformación realizada en este trabajo, en esta sección se analiza la transformación en sí y se estudian aspectos relativos tanto a la correctitud como a la completitud de un resultado entregado por la misma. Esto involucra analizar la sintaxis y la semántica del resultado, y el impacto de la calidad de los datos de entrada en él.

Los resultados entregados por la transformación son sintácticamente correctos, puesto que la misma construye una instancia válida del metamodelo Metamodels::ClassDiagrams presentado en la sección 3.2. La validez de la instancia construida respecto a dicho metamodelo es verificada en forma estática por el *typechecker* de Kermeta. Por su parte, la correctitud semántica del resultado depende de la lógica de la transformación, pero depende también de la calidad de los datos de entrada. Respecto a lo primero, la transformación fue diseñada para que el modelo resultante y las interacciones utilizadas para su construcción sean consistentes, de acuerdo a lo definido en la sección 3.3 y a lo presentado en la sección anterior. Según lo discutido en la sección 5.5.2, los atributos generados para una clase pueden diferir de los esperados. Eso afecta la calidad del resultado dado que su objetivo es ser un Diagrama de Clases de Diseño. Sin embargo este aspecto no afecta su consistencia respecto a las interacciones, ya que la noción de consistencia manejada no incluye a los atributos. Respecto a lo segundo, la transformación asume la correctitud tanto sintáctica como semántica de los modelos de entrada, no resulta interesante discutir el impacto de un error en los modelos de entrada sobre el modelo de salida. Sí es de interés conocer el impacto de falta de información en la entrada sobre el resultado. Esto se discute a continuación.

Como fuera discutido en la Sección 3, los valores de algunas de las propiedades en los modelos de entrada son opcionales. Por ejemplo, el tipo de un atributo en un Modelo de Dominio, o el nombre de un objeto en un Diagrama de Comunicación pueden ser omitidos. En lo sucesivo se analiza el efecto sobre el Diagrama de Clases de Diseño resultante de la transformación de la omisión de los valores opcionales en los modelos de entrada de la misma. En particular interesa estudiar si la falta de tal información causaría que el Diagrama de Clases de Diseño resulte incompleto (con

omisiones) o incorrecto (con algunas piezas de información con un valor diferente al esperado).

5.6.1 Modelo de Dominio

El Modelo de Dominio es utilizado en la transformación para extraer diferente tipo de información para la generación del Diagrama de Clases de Diseño. En particular, provee los atributos para algunas de las clases, relaciones de generalización, nombres de asociaciones e incluso multiplicidades. A continuación se discute para cada construcción de un Modelo de Dominio el impacto de la falta de información.

Clases En un Modelo de Dominio el nombre de una clase no es opcional, de forma que la información asociada es siempre utilizable si se requiere.

El dato de si una clase (si es utilizada) es abstracta o no influye en su correspondiente en Diagrama de Clases de Diseño. Salvo en el caso en que una clase sea determinada abstracta por poseer una operación abstracta, no especificar este dato en el Modelo de Dominio causa que ninguna clase del Diagrama de Clases de Diseño resulte abstracta.

Los nombres de los atributos no son opcionales. Los tipos de los atributos sí lo son; que falte la información de tipo causa que lo mismo ocurra en el modelo resultante.

No especificar la clase base de otra probablemente compromete la correctitud (o más bien la utilidad) del Modelo de Dominio. Como se expresó antes, dicho modelo se asume refinado y a un nivel de abstracción adecuado, por lo que omisiones intencionales u accidentales como esta no se consideran.

Asociaciones El nombre de una asociación es opcional. En caso de ser omitido y de ser necesario, la consecuencia que tiene en el modelo resultante es que la asociación se nombrará como la concatenación de los nombres de las clases que relaciona.

La falta de una multiplicidad no es un problema grave ya que la transformación asume valores por defecto los cuales ella misma refina. En abstracto, tanto la elección de los valores por defecto como el mecanismo de refinamiento parecen razonables, y los resultados de su aplicación al caso de estudio fueron satisfactorios. Sin embargo, aplicar el refinamiento sobre valores dados por un modelador en lugar de sobre valores inferidos puede ser preferible.

5.6.2 Diagramas de Comunicación

Los Diagramas de Comunicación son la entrada principal de la transformación; de ellos se extraen las clases del Diagrama de Clases de Diseño, así como la mayoría de sus relaciones y operaciones. A continuación se discute para cada construcción de un Diagrama de Comunicación el impacto de la falta de información.

Objetos En la mayoría de los casos el nombre de un objeto es opcional. Cuando un objeto debe ser utilizado, por ejemplo como argumento en un mensaje, su nombre no es opcional. Omisiones en este caso compromete la correctitud del diagrama de entrada mismo. En los demás casos su ausencia no afecta el resultado de la transformación. El tipo de un objeto sí es obligatorio puesto que de él se extrae el nombre de la clase a partir de la cual será creado. De esta forma, se crea exactamente las clases de objetos necesarias para producir

los participantes de todas las interacciones. Notar que al crear una clase en un Diagrama de Clases de Diseño puede estarse creando efectivamente más de una. Esto ocurre cuando se detecta una correspondencia de la nueva clase con el Modelo de Dominio, donde la clase correspondiente es subclase de otra. Esto quiere decir que el *full descriptor* necesario para crear un objeto participante de una interacción puede estar compuesto por los *segment descriptors* de varias clases. En otras palabras, la cantidad de tipos de objetos en la unión de todas las interacciones puede ser menor a la cantidad de nombres de clases del modelo resultado.

Mensajes El nombre de un mensaje es obligatorio puesto que de él se extrae el nombre de la operación que será invocada en la interacción (a excepción de los accesos a atributos y constructores de objetos, como ya fuera discutido). De esta forma, se crean todas las operaciones necesarias para soportar el envío de mensajes en cada interacción de entrada.

Los argumentos de los mensajes son obligatorios. Por su parte, los tipos de los argumentos son opcionales. En caso de no encontrarse especificados, la consecuencia directa es que faltarán (algunos de) los tipos de los parámetros en la operación correspondiente. Otra consecuencia de la falta de información de tipos es la pérdida de dependencias; cuando un objeto recibe un mensaje, su clase queda dependiente de las clases que sean tipo de alguno de sus argumentos. Sin embargo, cuando un argumento se llame igual que un objeto dentro de una misma interacción, se asume que el argumento es ese mismo objeto y por lo tanto se puede inferir el tipo del parámetro correspondiente puesto que la clase del objeto es conocida. Este mecanismo permite además no perder algunas de las dependencias.

Al igual que el nombre de un objeto, el nombre del valor retornado por un mensaje es opcional, pero en ocasiones es necesario especificarlo. Si se trata de un data value, es decir, de una instancia de un data type, su omisión no trae consecuencias al modelo resultante. Si se trata en cambio de un objeto, podría ser necesaria su especificación puesto que de no contar con la información del tipo de retorno ni con información del tipo visibilidad en mensajes hacia él se podría llegar a perder dependencias.

De no contar con el tipo de retorno de un mensaje, la operación correspondiente podría llegar a aparecer como un procedimiento, siendo en realidad una función. Además, también en este caso podrían perderse dependencias. De la misma forma en que se puede llegar a inferir el tipo de un argumento, también se puede llegar a inferir el tipo de retorno de un mensaje, aunque para ello sí es necesario contar al menos con el nombre del valor retornado. Respecto al resultado de un mensaje, lo ideal es contar con el nombre del valor retornado y su tipo (si es que el mensaje devuelve un valor). No contar con ambos tipos de información causa que se pierdan dependencias y que la operación asociada parezca no devolver valor alguno. Al igual que en el caso de los argumentos, contando con uno de los dos datos sería posible no incurrir en omisiones en el Diagrama de Clases de Diseño.

Si la propiedad de visibilidad de un mensaje no se encuentra especificada es necesario inferirla. Esto afecta en forma directa el tipo de relación a establecer entre un par de clases; una asociación o una dependencia (ver sección 5.5.5). En un principio, saber si un objeto tiene visibilidad de parámetro sobre otro en el envío de un mensaje es relativamente simple; tal cual se hace en la operación addDependencies() de la clase Transformation, basta con determinar si el objeto que envía el mensaje recibió al destino del mismo como argumento. Esto debe ocurrir en el mensaje que anida inmediatamente al mensaje para el cual se desea

inferir la visibilidad. Algo similar ocurre con la visibilidad local; si el objeto destino fue creado por el objeto origen del mensaje, o si el objeto destino fue recibido como resultado de otro mensaje, la visibilidad es local. Sin embargo, considérese la siguiente situación. Sean o_1 y o_2 dos objetos y supóngase que se desea conectarlos de forma tal que o_1 tenga visibilidad por asociación con o_2 (eso significa que para ello deba existir una asociación navegable desde la clase de o_1 hacia la clase de o_2). Naturalmente, ambos objetos no pueden ser creados a la vez y ya conectados entre sí de la forma deseada. Eso significa que en algún momento es necesario establecer el link entre o_1 y o_2 de alguna forma (típicamente asignando una referencia perteneciente a o_1 hacia o_2). Las formas posibles de hacer esto son: que o_1 reciba a o_2 como argumento en un mensaje, que o_1 cree a o_2, o que o_1 obtenga a o_2 como resultado de un mensaje. Estas son las formas de ganar visibilidad de parámetro o local discutidas antes. Por lo tanto, una visibilidad como las anteriores puede convertirse en visibilidad por asociación dependiendo de lo que haga o_1 con o_2. Es decir, si durante el mensaje en que o_1 recibe como argumento, crea, u obtiene como resultado a o_2, o_1 no establece un link estable con o_2, entonces la visibilidad de o_1 sobre o_2 será por parámetro o local, según corresponda. Pero si durante dicho mensaje, o_1 guarda una referencia a o_2, entonces la visibilidad se convierte en visibilidad por asociación. En otras palabras, que un objeto reciba a otro como argumento (lo cree o lo obtenga como resultado de un mensaje), es condición necesaria, pero no suficiente para que exista visibilidad por parámetro (local). Lamentablemente, a partir de una inspección de la interacción no es posible determinar si el objeto destino de un mensaje será mantenido por el objeto origen terminado el procesamiento del mensaje. Por otra parte, si no ocurre ninguno de estos tres casos, se está ante la situación de que el objeto origen envía un mensaje al destino pero no es posible determinar cómo obtuvo visibilidad sobre él. Esto es, si el objeto destino no fue creado por el origen, no fue recibido como argumento, ni tampoco obtenido como resultado de un mensaje previo, es razonable pensar que ambos objetos estuvieran de antemano conectados por un link. Esta situación se entiende que es una condición necesaria y suficiente para una visibilidad por asociación. En resumen, en muchos casos las visibilidades por asociación se pueden inferir, sin embargo la condición para visibilidad por parámetro o local no es suficiente para realizar la inferencia. La implicancia que esto tiene para la transformación es que faltando información de visibilidad es factible que no sea posible decidir si crear una asociación o una dependencia. En términos concretos, la problemática puede plantearse de la siguiente forma. Retomando la situación hipotética planteada más arriba sobre los objetos o_1 y o_2, una vez que o_1 se hace con una referencia a o_2 puede ocurrir lo siguiente: (a) o_1 no establece un link hacia o_2 y a continuación en forma opcional le envía un mensaje m; (b1) o_1 establece un link hacia o_2, y a continuación en forma opcional le envía un mensaje m, y además le enviará más adelante un mensaje m' fuera del alcance de la operación en la cual el link fue establecido, idealmente en otra interacción; o (b2) o_1 establece un link hacia o_2, y a continuación en forma opcional le envía un mensaje m, pero jamás enviará un mensaje con las mismas características de m'. Está claro que en el caso (a) la visibilidad para m, si es que ocurre, es por parámetro o local, según por qué vía haya o_1 obtenido la referencia de o_2, provocando en todo caso una dependencia. En el caso (b1) la visibilidad deberá ser por asociación tanto para m, si ocurre, como para m', provocando una asociación. Finalmente en el caso (b2) la visibilidad para m, si ocurre, es también por asociación, provocando una vez más una asociación. El problema radica en que cuando un objeto como o_1 obtiene una referencia sobre otro como o_2, localmente no es posible decidir cuál de los tres casos vale. Sin embargo, si se logra

encontrar un mensaje con las características del m' significa que se está ante el caso (b1), por lo que en definitiva debe crearse una asociación. Complementariamente, si tal mensaje no ocurre, podría estarse tanto ante el caso (a) como ante el caso (b2); como se dijo antes, se está ante la duda de si crear una asociación o una dependencia entre las clases de o_1 y o_2. Pensando en casos particulares sería posible el desarrollo de heurísticas que permitan establecer un criterio para distinguir entre ambas situaciones. Por ejemplo, dado que se está en conflicto, el mensaje m' se sabe que no ocurre. Supóngase además que m tampoco ocurre y que la forma en que o_1 obtuvo su referencia a o_2 fue mediante su creación. En este caso, se podría concluir que el caso que aplica es (b2). La justificación es que o_1 creó a o_2 y jamás le envió un mensaje posterior. Resulta poco intuitivo pensar que en tal situación no se haya establecido un link entre o_1 y o_2, porque de otra forma o_2 habría sido creado inútilmente. En el caso de estudio, sobre un total de *veinticuatro* mensajes enviados, solamente *once* de ellos pueden ser clasificados en alguno de los tres casos recién discutidos (es decir, son los mensajes en que un objeto obtiene una referencia a otro por alguna de las tres vías mencionadas). En base a la información de visibilidad especificada en los modelos, de estos once mensajes, *siete* corresponden al caso (b1), mientras que *dos* corresponden a (a) y los *dos* restantes a (b2). Los correspondientes al caso (a) son los mensajes findSpecification() y makeLineItem() en la interacción correspondiente a enterItem(). Estos mensajes, por ser de caso (a), son precisamente los que provocaron las dependencias desde Register hacia ProductSpecification, y desde Sale hacia ProductSpecification respectivamente. Por su parte, los mensajes create() en las interacciones correspondientes a makeCashPayment() y makeCheckPayment() son los correspondientes al caso (b2). Estos mensajes provocaron las asociaciones entre Sale y CashPayment, y entre Sale y CheckPayment respectivamente (las cuales fueron generalizadas para producir la asociación final paid-by). Notar que aplicando la heurística comentada arriba, hubiese sido posible clasificar correctamente estos dos casos, ya que la venta crea un pago (obteniendo así una referencia hacia él), y nunca le envía otro mensaje posterior al create() dentro del alcance de makeCashPayment() o de makeCheckPayment() (no ocurre el mensaje m). Esto permite afirmar que de haber faltado en el caso de estudio la totalidad de la información respecto a las visibililidades, hubiese sido posible generar en su totalidad las asociaciones y dependencias correctas. Sin embargo, no es posible generalizar y afirmar que lo aquí discutido solucione el problema en cualquier caso de estudio. Otras alternativas que pueden ser de ayuda para decidir entre una asociación y una dependencia es el uso del mensaje *destroy*, el cual elimina o destruye el objeto destino. Si en el mismo nivel de anidamiento del create se encuentra posteriormente un mensaje destroy al mismo objeto, necesariamente la visibilidad en todos los casos ha de haber sido local. Sin embargo, de la misma forma en que un mensaje create se asocia con los constructores de los lenguajes de programación, el mensaje destroy se asocia con los destructores. La presencia de un garbage collector en la mayoría de los ambientes de ejecución de los lenguajes de programación de la actualidad ha eliminado la noción de destructor como construcción del lenguaje, y por consiguiente su uso ha quedado desalentado en lenguajes de modelado como UML. Por esto, la dependencia sobre su uso no parece ser una buena idea. Como alternativa a anotar cada mensaje con su visibilidad correspondiente, se podría anotar el objeto destino. Tal anotación especificaría si el objeto creado es local al mensaje o si será finalmente mantenido (permitiendo esto distinguir el caso (a) de los otros dos). Esta alternativa requiere el mismo conocimiento que las anotaciones de visibilidad propuestas originalmente, y su especificación requiere prácticamente el mismo esfuerzo por parte del modelador, lo que indicaría que son

equivalentes tanto en utilidad como en la probabilidad de que su información no se encuentre disponible. A partir de todo esto se concluye que, ya sea dada o inferida, la información de visibilidad es crucial para la correctitud del Diagrama de Clases de Diseño.

La información de si un mensaje es resuelto localmente por el objeto destino es opcional. Independientemente de este dato, es posible determinar si en una interacción existe un mensaje anidado en otro. Cuando tal mensaje existe, es posible determinar que la operación asociada al mensaje original tendrá un método en la clase del objeto destino. Pero cuando no existe ningún mensaje anidado, y el valor de locallyResolved no fue especificado, entonces no es posible decidir si el mensaje puede o no ser resuelto por el destino. Eso tiene como consecuencia que no sea posible detectar operaciones abstractas, por lo que todas las operaciones en el Diagrama de Clases de Diseño en esta situación aparecerían especificadas como concretas.

En resumen, se puede afirmar que la transformación produce un modelo de clases que es consistente con las interacciones en las cuales está basado. Sin embargo, existen aspectos que escapan a dicha consistencia y que pueden afectar la calidad del modelo resultante por tratarse de un Diagrama de Clases de Diseño; a saber, la pertinencia de los atributos generados, y la precisión del mínimo en las multiplicidades. Estos aspectos dependen de la calidad del Modelo de Dominio y de las particularidades de la realidad del problema al cual se aplica la transformación.

Por otra parte, la incompletitud en los modelos de entrada, de acuerdo a lo permitido por sus metamodelos respectivos, puede afectar tanto la completitud como la correctitud del modelo generado, y por lo tanto su utilidad. En particular, es posible que:

- Se genere una asociación entre dos clases cuando debió generarse una dependencia entre ellas, o viceversa.
- Alguna operación abstracta no sea identificada como tal.
- Alguna clase abstracta no sea identificada como tal.
- No se disponga del tipo de algún atributo.
- No se disponga del tipo de algún parámetro de operación.
- No se disponga del tipo de retorno de alguna operación.
- Se pierdan algunas dependencias.

Con seguridad, el más grave de los problemas enumerados arriba sea el primero de ellos, ya que no se trata de falta de información sino de información *equivocada*, la cual podría ser más propensa a errores posteriores y más difícil de corregir. Sin embargo, todos estos inconvenientes pueden ser evitados aplicando un esfuerzo mayor a la actividad de especificación de las interacciones. Un aspecto importante es que metodológicamente la generación del Diagrama de Clases de Diseño se realiza en forma contigua en el tiempo, e incluso solapada, con el diseño de las interacciones. Esto permite que el resultado de la transformación pueda retroalimentar esta última actividad las veces que sea necesario hasta obtener un resultado de conformidad. Por otra parte, el esfuerzo requerido para producir la información opcional en los modelos de entrada es comparable al esfuerzo requerido para completar el modelo resultante. Por ejemplo, en el caso del tipo de un argumento,

tiene el mismo costo determinar su tipo al momento de utilizar como entrada de la transformación la interacción en el que ocurre, que determinar el tipo del parámetro correspondiente en el modelo resultante. De esta forma, y dada la proximidad en el tiempo con que los artefactos de diseño son producidos, se entiende que especificar en forma más completa las interacciones puede verse como adelantar trabajo que de todas formas será necesario realizar. Asimismo se entiende que no se genera a raíz de esto un sobrecosto de importancia, principalmente por la naturaleza automática de la transformación.

6 Conclusiones

En este trabajo se propuso una transformación de modelos en la cual se genera, a partir del Modelo de Dominio y de los Diagramas de Comunicación que especifican el diseño de las operaciones del sistema para un caso de uso, el Diagrama de Clases de Diseño que describe la estructura de clases necesaria para que las interacciones dadas puedan ocurrir, es decir, a partir del cual se puede producir la colaboración que realiza el caso de uso. La transformación es un refinamiento de la propuesta de alto nivel presentada por Larman. Para ello se definieron metamodelos específicos para tanto para los Diagramas de Comunicación, como para el Modelo de Dominio y Diagrama de Clases de Diseño. Para el desarrollo de la transformación se siguió el enfoque basado en herramientas de metamodelado y se utilizó el ambiente Kermeta, el cual es plenamente compatible con el enfoque elegido. El funcionamiento de la transformación fue demostrado a través de su aplicación a un caso de estudio conocido en la bibliografía.

El principal aporte de este trabajo es la presentación del proceso completo que fue seguido para el desarrollo de la transformación, aplicado a un problema, el de generación de un Diagrama de Clases de Diseño, que resulta de una complejidad mayor a la de aquellos ejemplos encontrados usualmente en la bibliografía. El presente reporte documenta el esquema de definición de transformaciones basadas en el enfoque de herramientas de metamodelado, así como un detalle de cada uno de los elementos necesarios para su desarrollo, y su posicionamiento en dicho esquema. Esto puede resultar de utilidad a quien desee conocer los lineamientos básicos de este enfoque de definición de transformaciones, y un ejemplo completo y no trivial de su puesta en práctica.

En el enfoque basado en metamodelado, la transformación y los metamodelos de los modelos de entrada y salida se encuentran al mismo nivel. Esto significa que la definición de clases realizada para la transformación se ve "aumentada" con todas las clases de dichos metamodelos. Esto produce una definición de clases integrada y uniforme, en la cual los modelos de entrada, el modelo de salida y todos sus componentes son objetos que se encuentran verdaderamente al mismo nivel que los objetos que realizan la transformación, lo cual resultó adecuado para el problema atacado. En el ambiente de Kermeta estas clases son generadas automáticamente y quedan accesibles como en una suerte de biblioteca, ya que dispone de soporte nativo para la carga y manipulación de metamodelos especificados en Ecore y de instancias de los mismos especificadas en XMI. Asimismo, el soporte de Kermeta a OCL, que incluye colecciones genéricas con operaciones como *select*, *collect* y *exists* (las cuales reciben λ-expresiones como argumentos) y navegación, hace que la manipulación de modelos sea muy sencilla y compacta, posicionando en efecto a Kermeta como un DSL para la ingeniería de metamodelos.

Los resultados obtenidos de la aplicación de la transformación a la realidad del caso de estudio fueron los esperados, incluso detectando errores en el Diagrama de Clases de Diseño construido en

la versión original del caso de estudio. Esto permite concluir que la transformación fue desarrollada en forma satisfactoria. Los Diagramas de Clases de Diseño tienen un rol central en el Modelo de Diseño y en la documentación de una aplicación, y son un insumo básico para la generación (tanto automática como manual) de código. Esto hace que sea factible que una versión refinada de la transformación propuesta en este trabajo pueda utilizarse para dar apoyo a algunas actividades en procesos de desarrollo basados en RUP. Por todo esto, la transformación en sí constituye un ejemplo práctico de la pertinencia de la transformación de modelos y la factibilidad de su aplicación en el desarrollo de software.

El desarrollo de transformaciones tiene un impacto sobre el desarrollo de software, comparable al que tiene un software (por ejemplo un sistema de información) sobre la organización para la cual fue producido. Primeramente la organización se ve favorecida porque el software automatiza el manejo de la información y acelera su gestión, permitiéndole reducir costos y tornándola más competitiva en su sector. Lo mismo puede producir el incorporar transformaciones al desarrollo de software. Pero fundamentalmente un sistema computacional obliga a la organización a comprender y *formalizar* sus procesos. Ese fue uno de los principales desafíos que planteó el desarrollo de la transformación durante este trabajo. En ese sentido, para que un enfoque como MDE pueda ser aplicable, y en particular el de transformaciones de modelos, es necesario que en la comunidad de desarrolladores se valore el rol del modelamiento de software y que la actividad de modelar deje de entenderse como que es dibujar-para-documentar y de un aporte discutible. Como se pudo apreciar en el análisis de la sección 5.6, la calidad y la utilidad de un modelo generado depende fuertemente de la calidad de los modelos de entrada. Por ello, si el modelamiento es considerado una pérdida de tiempo y por esa misma razón no se realiza en forma sistemática y parte del conocimiento no es incorporado a los modelos, la información que sí esté contenida en los modelos no podrá ser aprovechada realmente. Cabe notar que en muchas ocasiones la incompletitud de los modelos no viene dada por la carencia de información, sino que es intencional. En la medida de que la creación de modelos sea simplificada (tal vez por la vía de mejores herramientas), y de que existan transformaciones que disminuyan la cantidad de modelos a construir, la transición a un enfoque realmente basado en MDE será posible.

\+ Trabajos futuros

En el enfoque de herramientas de metamodelado, una transformación es un modelo (ejecutable) de un programa orientado a objetos que transforma programas. Por lo tanto todas las técnicas conocidas para el desarrollo orientado a objetos son aplicables a las transformaciones. Sin embargo, no se conocen pautas específicas, por ejemplo de análisis o diseño, que puedan ser desarrolladas para ser aplicadas a cualquier transformación. El diseño presentado en la figura 12 de la sección 5.3.1 es una alternativa posible, pero otras soluciones con mejores propiedades podrían ser encontradas. Incluso, sería de mucha utilidad caracterizar la estructura y comportamiento general de transformaciones mediante algún patrón de arquitectura conocido, o uno nuevo en caso de que ninguno de ellos aplique. La transformación propuesta en este trabajo por ejemplo, a pesar de que su estructura lógica no necesariamente lo muestra de esa forma, se presenta conceptualmente como una suerte de pipes & filters donde el Diagrama de Clases de Diseño es poblado de elementos en forma secuencial.

En esa misma línea, y dado que las transformaciones son modelos, podría resultar de utilidad el desarrollo de transformaciones de transformaciones. Por ejemplo, se podría desarrollar una transformación abstracta o genérica que no sea aplicable en forma directa y que represente cierto conocimiento parcial del proceso de desarrollo, a la cual se le podría aplicar una transformación

más sencilla y donde el resultado sí fuese aplicable a un contexto concreto.

Una de las mejoras posibles a realizar a la transformación propuesta refiere a la implementación de operaciones auxiliares que trabajan sobre las clases que Kermeta genera automáticamente a partir de los metamodelos. Un ejemplo de ello es la operación matchMessage() que es utilizada en la operación factorize() de la clase Transformation que fuera presentada en la sección 5.5.6. La operación matchMessage(), que también es una operación privada de la clase Transformation, tiene como propósito determinar si dos mensajes son iguales. Recordar que la lógica de tal operación no es trivial y consiste en comparar no sólo el nombre del mensaje, sino además los tipos de los argumentos posición a posición (en esta situación la carencia de información de tipos es también un problema). Conceptualmente, dicha operación debería ser una operación equals() de la clase Message, pero dicha clase es generada a partir de una especificación Ecore (la cual no contiene comportamiento) y no está disponible para ser modificada. Si los metamodelos en cambio, fuesen especificados en el propio lenguaje de Kermeta (el cual puede contener comportamiento), sí sería posible una modificación de las clases como la mencionada en el ejemplo anterior. De esta forma el diseño de la transformación resulta más adecuado, por ocurrir que cada operación es propiedad de la clase que corresponde. La modificación mencionada puede realizarse de dos formas. La primera es realizar la especificación de los metamodelos en el lenguaje de Kermeta, lo cual implica crearlos desde cero. Si se dispone de una especificación en Ecore de los metamodelos, la otra alternativa es utilizar la transformación Ecore2Kermeta, incluida en el ambiente de Kermeta, para producir el metamodelo en Kermeta a partir de la especificación en Ecore.

Una versión completa de la transformación propuesta necesitaría contemplar aquellos aspectos dejados de lado en la sección 5.2. Para considerar operaciones y atributos de clases, sería necesario incorporar un nuevo tipo de receptor de mensajes; una clase. Esto puede hacerse creando en el metamodelo Metamodels::Interactions una nueva especialización de Lifeline que represente justamente una clase. Existe un inconveniente con esto y es que todo lifeline tiene un nombre y un tipo, donde este último es obligatorio. Para una clase sólo interesa el nombre, por lo cual debería ser necesario modificar la jerarquía existente para separar estos dos atributos y así no obligar a esta nueva clase a tener un tipo asociado. De esta forma, uno de estos nuevos elementos se corresponde con una clase del Diagrama de Clases de Diseño, y un mensaje recibido provoca una operación de clase. La visibilidad en estos casos es global, por lo que sería además necesario modificar el enumerado VisibilityKind, aunque inferir la visibilidad en estos casos es muy sencillo. Para los atributos de clase persiste el mismo problema que ya fuera discutido para los atributos de instancia; no es posible determinar a partir de las interacciones, excepto cuando existe un mensaje que los acceda, qué atributos son utilizados en la resolución de cada mensaje. Los mensajes de acceso son fácilmente identificables ya que sus nombres siguen el mismo patrón que en el caso de los atributos de instancia. Por otra parte, las operaciones de clase pueden acceder únicamente a atributos de clase. Para los calificadores de acceso a las propiedades de una clase no existe un estándar general. Usualmente los atributos son privados y las operaciones son públicas. No es común encontrar atributos públicos, sin embargo sí es posible encontrar operaciones privadas. Un criterio podría ser que las operaciones generadas a partir de automensajes sean privadas. El caso en que se considera cualquier valor posible para las multiplicidades es algo complejo. Como ya fuera discutido, no es posible determinar con certeza valores concretos para los límites de una multiplicidad, como por ejemplo “3..11”. La práctica indica que los valores considerados son suficientes para la mayoría de los casos, ya que es común que un máximo mayor que 1 sea tomado como “*”. Sin embargo sería posible dedicar esfuerzo a desarrollar técnicas que permitan detectar

valores concretos con mayor precisión. La consideración de automensajes no requiere una modificación considerable de la transformación; simplemente contemplar el caso en que el origen y el destino del mensaje coinciden, para el cual no se deben generar ni asociaciones ni dependencias. Las autoasociaciones pueden permitirse introduciendo nombres de roles en los extremos de asociación, haciéndolos obligatorios cuando exista ambigüedad. Asimismo, el uso de roles en los extremos de asociación habilita múltiples asociaciones entre un mismo par de clases; el nombre del rol correspondiente a la asociación a través de la cual es enviado el mensaje debería ser utilizado en las interacciones. Esta información puede ser especificada en el mensaje, con lo cual se debería modificar el metamodelo, o podría ser el nombre del receptor del mensaje, con lo cual el metamodelo quedaría incambiado a costa de introducir una mayor complejidad en la transformación. Por su parte, las visibilidades de tipo "global" deberían considerarse como consecuencia de las operaciones de clase. Finalmente, con un uso adecuado de la propiedad locallyResolved de los mensajes no es complejo permitir la especificación de la resolución de un mensaje en una interacción separada.

Algunas características que sería posible incorporar a la transformación para mejorar su usabilidad y tornarla más flexible pueden ser las siguientes. Por un lado sería interesante disponer de un archivo de configuración en el que se pueda especificar qué elementos del Diagrama de Clases de Diseño se desea generar. Por ejemplo en una primera generación del diagrama podría ser útil omitir la información de tipo en atributos y parámetros de operaciones, así como las dependencias, logrando así un diagrama más fácilmente legible, con el objetivo de realizar un inspección manual ágil en el contexto de una retroalimentación de los modelos de entrada, como fuera discutido en la sección 5.6.2. Podría resultar de utilidad además contar con un cierto registro que pueda ser consultado en forma posterior a la finalización de la ejecución de la transformación, en el que se indiquen posibles problemas causados por falta de información en los modelos de entrada y que sugiera la revisión manual de partes concretas del modelo resultante. En su forma actual, la transformación cuando es llamada a ejecutar lo hace en forma completa y sin asistencia alguna por parte del usuario. Una versión interactiva de la transformación, en la cual se pueda consultar al usuario la confirmación de determinadas decisiones, o alguna pieza de información faltante en los modelos de entrada, podría mejorar la calidad del resultado. Este mecanismo no debería reemplazar los criterios de inferencia actualmente implementados, sino complementarlos, ya que es posible que el usuario ignore parte de la información solicitada por la transformación.

Como fuera expresado en la introducción de este documento, adicionalmente a la implementación en Kermeta de la transformación, se cuenta además con una implementación en C# de la misma. Si bien no es el objetivo de este trabajo reportar la experiencia dejada por esta implementación, ni tampoco realizar una comparación profunda entre ellas, se considera oportuno realizar algunas observaciones al respecto. La implementación en C# corresponde al enfoque de *lenguajes de programación de propósito general*, el cual difiere del implementado por Kermeta. La implementación en C# consistió en la definición de un conjunto de clases de acuerdo a los metamodelos presentados antes, y una clase para la transformación, en forma similar a la clase Transformation. Podría decirse que el diseño realizado es prácticamente idéntico al presentado en la sección 5.3, realizándose los mismos pasos en la modificación del Diagrama de Clases de Diseño. Incluso, la lógica de cada uno de los pasos es la misma, resultando sin embargo más compacta la versión en Kermeta debido a las facilidades de navegación que provee. Esta es una de las principales diferencias que se distinguen entre ambas implementaciones. Adicionalmente, Kermeta provee una implementación en forma automática de las clases que componen los metamodelos;

en la implementación en C# dichas clases necesitaron ser generadas manualmente. Asimismo, el ambiente de Kermeta provee editores para la generación de instancias de los metamodelos (es decir, los modelos de entrada del caso de estudio), su validación sintáctica, y su almacenamiento y recuperación de memoria secundaria en formato XMI. Los modelos de entrada en la otra implementación debieron construirse programáticamente. La implementación en C# también puede ser accedida en `http://www.dcc.uchile.cl/~avignaga/transformation`.

Finalmente, como complemento a la transformación propuesta, podría ser de interés el desarrollo de otra transformación la cual generaría la colaboración que realiza el caso de uso que involucra las operaciones del sistema tratadas en las interacciones utilizadas como entrada en este caso. Para ello sería necesario generar un Diagrama de Colaboración de UML2 (no confundirlo con los discontinuados Diagramas de Colaboración de UML1.x), en el cual se especifiquen los objetos, junto con sus conexiones, que deben estar presentes en el sistema en tiempo de ejecución para que las interacciones puedan ser llevadas a cabo. Este tipo modelos complementan a los Diagramas de Clases de Diseño en los Modelos de Diseño, y son referenciados en los Modelos de Caso de Uso. Adicionalmente a esto, resultan de utilidad al momento de realizar el testing del caso de uso asociado.

Bibliografía

[1] Bézivin, J., Kurtev, I.: Model Driven Engineering: Foundations, Standards, Tools & Applications. Project Modeling Solution for Software Systems (MODELWARE). Proyecto IST 511731. Internet: `http://www.modelware-ist.org`, 2005.

[2] Chen, P. P.: The Entity-Relationship Model - Toward a Unified View of Data. ACM Transactions in Database Systems. Volume 1, number 1, pages 9-36, 1976.

[3] Eclipse Foundation: Eclipse SDK 3.2. Internet: `http://www.ecplipse.com`, 2006.

[4] European Computer Manufacturers Association: Standard ECMA-334. C# Language Specification. Fourth edition. Internet: `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`, 2006.

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

[6] Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A: Transformation: The Missing Link of MDA. In Procedding of 1^{st} Internactional Conference on Graph Trasformation (ICGT'2002), 2002.

[7] IBM: IBM Rational Unified Process. Internet: `http://www-306.ibm.com/software/awdtools/rup/`, 2006.

[8] INRIA: Kermeta, Triskell Metamodeling Kernel. Internet: `http://www.kermeta.org`, 2005.

[9] Jézéquel, J. M.: Model Transformation Techniques. ModelWare, INRIA. Internet: `http://modelware.inria.fr/rubrique.php3?id_rubrique=21`, 2005.

[10] Kent, S.: Model Driven Engineering. Invited Presentation at the 3^{rd} International Conference on Integrated Formal Methods (IFM2002), 2002.

[11] Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design. Prentice Hall, primera edición, 1997.

[12] MetaCase: MetaEdit+® DSM Environment. Internet: `http://www.metacase.com/products.html`, 2006.

[13] Object Management Group, Inc., Needham, Mass., Internet: `www.omg.org`, 2006.

[14] Object Management Group, Inc., Model Driven Architecture Guide, Version 1.0.1, omg/03-06-01, 2001.

[15] Object Management Group, Inc., Meta Object Facility (MOF) Core v2.0, formal/06-01-01, 2006.

[16] Object Management Group, Inc., Object Constraint Language (OCL) Specification v2.0, formal/06-05-01, 2006.

[17] Object Management Group, Inc., MOF 2.0 Query/Views/Transformations RFP, ad/02-04-10, 2002.

[18] Object Management Group, Inc., Revised Submission for MOF 2.0 Query/View/Transformation RFP (ad/2002-04-10) Version 2.0, ad/05-03-02, 2005.

[19] Object Management Group, Inc., UML Superstructure Specification, v2.0, formal/05-07-04, 2005.

[20] Object Management Group, Inc., XML Metadata Interchange (XMI), v2.1, formal/05-09-01, 2005.

[21] Rumbaugh, J., Jacobson, I., Booch, G.,: The Unified Modeling Language Reference Manual, Second Edition. Addison-Wesley, 2005.

[22] Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[23] Xactium: XMF-Mosaic 1.0. Internet: `http://albini.xactium.com/web/index.php?option=com_content&task=blogcategory&id=27&Itemid=46`, 2006.