Definición y ejecución de métricas en el contexto de ADM

Javier Luis Cánovas Izquierdo, Belén Cruz Zapata, Jesús García Molina
Universidad de Murcia
{jlcanovas, b.cruzzapata, jmolina}@um.es

Resumen

La *Modernización de Software Dirigida por Modelos* ha surgido como una nueva disciplina centrada en la utilización de técnicas de *Desarrollo de Software Dirigido por Modelos* en los procesos de evolución de software. La iniciativa *Architecture Driven Modernization* (ADM) de OMG define un conjunto de metamodelos estándar para representar como modelos la información involucrada en las tareas de una modernización o reingeniería de software.

Para la representación de métricas, ADM incluye el metamodelo *Software Metrics Metamodel* (SMM). Sin embargo, la definición de SMM solamente proporciona la sintaxis abstracta, pero no una notación. En este artículo se dota a SMM de una sintaxis concreta textual y se define un motor de ejecución que permite ejecutar métricas y calcular mediciones sobre modelos Ecore.

1. Introducción

Las técnicas del *Desarrollo de Software Dirigido por Modelos* (DSDM) no sólo son útiles para la creación de nuevos sistemas software, sino que también pueden ser aplicadas en procesos de modernización o reingeniería de software. Para ello, es necesaria una etapa inicial que convierta los artefactos software del sistema existente en modelos que los representen. Sin duda, la iniciativa más ambiciosa sobre modernización dirigida por modelos es *Architecture Driven Modernization* (ADM) [1] lanzada por OMG en 2003.

El propósito de ADM es favorecer la interoperabilidad entre herramientas de modernización de software mediante la definición de un conjunto de especificaciones estándares de metamodelos que representan la información gestionada, normalmente, en las tareas de modernización, como análisis estático o *refactoring*. Del total de siete especificaciones previstas, en la actualidad sólo se han publicado tres de ellas: *Abstract Syntax Tree Metamodel* (ASTM), que permite representar el código fuente como árboles de sintaxis abstracta; *Knowledge Discovery Metamodel* (KDM), destinado a representar el código en diferentes vistas arquitecturales y que es la base para la interoperabilidad; y *Software Metrics Metamodel* (SMM), que permite representar tanto métricas como mediciones. El resto de metamodelos están relacionados con el análisis de programas, las pruebas, el *refactoring*, la visualización y las transformaciones.

En el desarrollo de software, como sucede en otras disciplinas de ingeniería, las mediciones de ciertas propiedades o características de un sistema o proceso de software son esenciales para conocer tanto la calidad del producto o del trabajo realizado, como para controlar el proceso y establecer planes de mejora. En un proceso de modernización de software las métricas ayudan en actividades como el análisis del código existente, la evaluación del resultado de la evolución o el control del proceso aplicado [2].

SMM fue publicado en mayo de 2009 y sólo incluye el metamodelo de la sintaxis abstracta. En este trabajo se dota a este metamodelo de una sintaxis concreta textual para definir completamente un lenguaje específico de dominio (DSL), denominado Medea, destinado a la definición y ejecución de métricas. Las métricas son expresadas textualmente y se pueden aplicar sobre cualquier artefacto software representado mediante un modelo conforme a un metamodelo MOF o Ecore, como por ejemplo KDM. Se ha construido un motor que ejecuta especificaciones Medea y obtiene modelos con las mediciones que resultan de aplicar las métricas. La aportación de este trabajo abre la posibilidad de experimentar con el metamodelo SMM mediante la definición de métricas ejecutables para KDM u otros

metamodelos. Por lo que sabemos se trataría del primer motor de ejecución de SMM disponible.

La organización de este documento es la siguiente. La Sección 2 describe el metamodelo SMM y comenta sus principales elementos. La Sección 3 presenta la sintaxis textual propuesta, mientras que la Sección 4 describe el motor de ejecución implementado. La Sección 5 muestra unos ejemplos de definición de medidas con Medea, los cuales han sido utilizados en un caso de estudio real. La Sección 6 presenta el trabajo relacionado y la Sección 6 finaliza el artículo con las conclusiones y trabajo futuro.

2. El metamodelo SMM

Como hemos indicado, SMM es un metamodelo que proporciona un formato estándar para representar tanto las métricas de software como las mediciones con dichas métricas. Aunque se ha definido en el contexto de ADM, SMM permite representar métricas de cualquier tipo referidas a todo tipo de artefacto software representado mediante elementos de un modelo conforme a un metamodelo (MOF según la especificación).

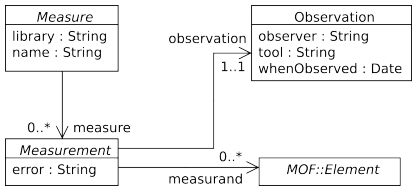

Figura 1. Elementos principales de SMM

La Figura 1 muestra los principales elementos de SMM. Los dos conceptos principales del metamodelo son *medida* (*measure*) y *medición* (*measurement*). Mientras que una medida define un proceso de evaluación (esto es, una métrica), una medición es el resultado de la aplicación de una medida. Cada medición tiene asociada (a través de la referencia *measurand*) el conjunto de elementos que han sido medidos (en nuestro caso elementos de modelos Ecore). Además, las mediciones también tienen asociada información (*observation*) acerca del momento y la herramienta que ha sido utilizada para calcularla.

SMM incluye una jerarquía de medidas que representan distintas formas de evaluar características de un elemento. La Figura 2a muestra una parte de esta jerarquía y en ella se observa que las medidas se clasifican en dos categorías: las que asignan un valor numérico de acuerdo a una unidad de medida (*DimensionalMeasure*) y las que retornan un símbolo que indica que la característica considerada tiene un valor en cierto rango (*Ranking*).

Existen varios tipos de *DimensionalMeasure,* como las medidas simples o de aplicación directa (*DirectMeasure*), por ejemplo la medida que comprueba si un elemento satisface una propiedad retornando 0 o 1 (*Counting*); y las medidas compuestas que representan operaciones que involucran a varias medidas. Hay dos tipos de medidas compuestas: las que realizan el cálculo entre dos elementos (*BinaryMeasure*) y las que son aplicadas a un conjunto de elementos (*CollectiveMeasure*). *RatioMeasure* es una subclase de *BinaryMeasure* que representa una relación entre dos medidas bases (por ejemplo, el número medio de líneas por módulo) y *AdditiveMeasure* y *MaximalMeasure* son tipos de *CollectiveMeasure* que representan una suma de mediciones y la obtención de la medición máxima, respectivamente. Ambos tipos de medida compuesta especifican las medidas en las que se basan. Así, *BinaryMeasure* especifica sus operandos con las referencias *baseMeasure1* y *baseMeasure2*, mientras que *CollectiveMeasure* utiliza la referencia *baseMeasure*.

Para aquellas medidas que puedan ser descritas por su nombre, ya que son bien conocidas, como la complejidad ciclomática de McCabe, SMM ofrece la metaclase *NamedMeasure*.

Como se observa en la Figura 2a, todas las medidas en SMM tienen asociado el ámbito de aplicación (*scope*) y la característica que se está midiendo (*characteristic*). El ámbito permite especificar el dominio de la medida, esto es, sobre qué tipos de elementos va a aplicarse. La característica es una cadena de texto que describe la propiedad que se mide, por ejemplo, líneas de código o longitud de un fichero.

De forma paralela a la jerarquía de medidas, SMM incluye una jerarquía de mediciones. Esta jerarquía permite reproducir la estructura de una medida pero con el objetivo de representar el resultado de la medición. Por ejemplo, para las medidas de tipo *CollectiveMeasure* existe la medición *CollectiveMeasurement*. La Figura 2b muestra una parte de la jerarquía de mediciones.

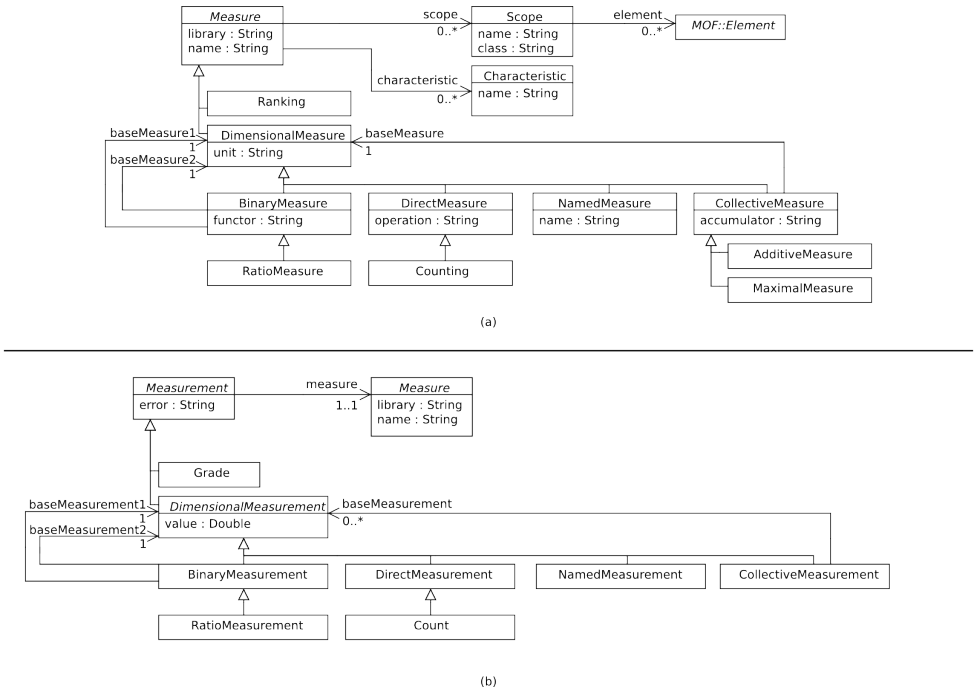

Figura 2. Jerarquías principales de SMM. (a) Jerarquía de medidas. (b) Jerarquía de mediciones

SMM también incluye conceptos básicos para organizar las medidas y mediciones en librerías y categorías.

La Figura 3 (basada en uno de los ejemplos usados en la especificación de SMM) muestra un modelo SMM que representa el resultado de contar el número de módulos existentes en un modelo KDM. Esta figura muestra cómo los modelos SMM incluyen elementos relacionados con las medidas, las mediciones y los elementos evaluados.

La Figura 3a muestra los elementos necesarios para la definición de la medida, estos son, una medida compuesta de tipo *AdditiveMeasure* que suma las mediciones de una medida directa de tipo *Counting* que devuelve 0 o 1 según el elemento sea o no de tipo *code::Module*. La medida compuesta define como ámbito los elementos del paquete *Code* de KDM de tipo *CodeModel* y especifica el atributo *accumulator* a *sum* para indicar que la operación a realizar con los resultados de las medidas que la componen es la adición. Por otro lado, la medida *Counting* es una medida simple que define como ámbito los elementos del paquete *Code* de KDM de tipo *AbstractCodeElement* y especifica en el atributo *operation* la operación OCL a realizar para comprobar que un elemento es del tipo indicado en el *Scope*. Conviene destacar que SMM utiliza OCL para expresar las operaciones asociadas a cualquier medida directa.

La Figura 3b muestra los elementos de tipo medición asociados a las medidas anteriores, indicando los valores numéricos calculados para cada una de ellas. En este ejemplo, la medición se ha realizado para un modelo KDM que sólo contiene un módulo. Obsérvese que las mediciones calculadas hacen referencia a los elementos del modelo que son objeto del cálculo de la medida, en el ejemplo las mediciones referencian a los elementos *CodeModel* y *Module* del modelo KDM (Figura 3c).

3. Definición de la sintaxis concreta

Dado que la especificación de SMM solamente proporciona el metamodelo de la sintaxis abstracta, decidimos dotarla de una notación o sintaxis concreta para facilitar la tarea de crear

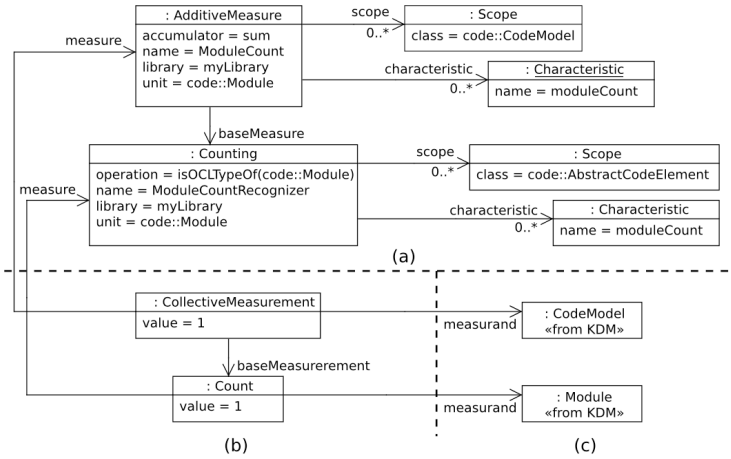

Figura 3. Ejemplo de modelo de sintaxis abstracta en SMM extraída del documento de especificación. Las medidas definidas cuentan el número de módulos en los modelos de código de un modelo KDM

modelos que expresen medidas SMM. De este modo, los usuarios dispondrían de un DSL completo, en vez de tener que definir las medidas como instancias de los elementos de la sintaxis abstracta con algún editor. La ejecución de las especificaciones de medidas expresadas con dicho lenguaje permitiría realizar las mediciones sobre artefactos software representados como modelos Ecore y obtener los correspondientes modelos SMM. A continuación presentaremos la sintaxis del DSL Medea y comentaremos el proceso de definición de dicha sintaxis, y en la siguiente sección describiremos su motor de ejecución.

Consideramos que una sintaxis concreta textual sería más apropiada que una gráfica, por la naturaleza de las métricas, el tipo de usuario, y sobre todo porque según nuestra experiencia es conveniente comenzar antes por un DSL textual que por uno gráfico, como también se indica en [3].

Es importante destacar que la definición de la sintaxis concreta textual solamente afecta a la parte de SMM encargada de la definición de medidas, que es la parte que debe ser definida manualmente. La parte de SMM dedicada a la definición de mediciones se obtiene como resultado de la ejecución de las medidas, por lo que no es necesaria su especificación por parte del usuario.

Existen diversas herramientas para crear DSLs textuales basados en un metamodelo, las cuales se pueden clasificar en dos categorías: i) las que parten de la gramática para generar el metamodelo y el *parser* que extrae modelos de la especificación textual, como por ejemplo xText [4]; y ii) las que generan el parser a partir del metamodelo, por ejemplo TCS [5] y EMFText [6].

Nosotros hemos usado el lenguaje Gra2MoL, creado especialmente para extraer modelos a partir de código fuente de un lenguaje de programación [7, 8], ya que realmente posibilita la extracción a partir de cualquier especificación textual conforme a cualquier gramática libre de contexto. Por tanto se puede ver como un tercer enfoque para crear DSLs: dado un metamodelo destino y una gramática origen se establece la correspondencia entre ellos mediante una transformación cuya ejecución produce el modelo a partir de una especificación conforme a la gramática, como expresa el esquema de la Figura 4 para el caso de Medea y SMM.

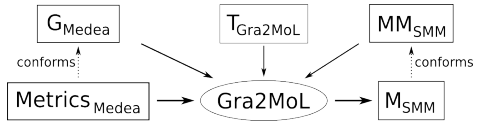

Figura 4. Proceso de extracción de un modelo SMM que representa medidas

Gra2MoL incorpora un lenguaje de consultas especialmente creado para resolver las referencias entre elementos del código fuente, lo que lo hace especialmente apropiado para utilizarlo en este

caso como veremos más adelante. Esta característica es una ventaja frente a aproximaciones como TCS o EMFText, donde es necesario proporcionar un mecanismo para dicha resolución de referencias.

Una definición de transformación en Gra2MoL consiste en un conjunto de reglas que especifican las relaciones entre los elementos de la gramática y los elementos del metamodelo. Cada una de estas reglas expresa cómo crear un elemento del metamodelo a partir de un elemento de la gramática como se describe en [7]. Como muestra la Figura 4, el motor de ejecución de Gra2MoL recibiría como entrada cuatro elementos: (1) la gramática G_{Medea} definida para SMM, (2) el metamodelo MM_{SMM} de la sintaxis abstracta de SMM, (3) la definición de la transformación Gra2MoL $T_{Gra2MoL}$ y (4) la especificación Medea conforme a la gramática. Como resultado de la transformación, se obtiene un modelo conforme al metamodelo destino, en este caso SMM.

La creación de un DSL con Gra2MoL requiere como primer paso la definición de una gramática en formato ANTLR. Esta gramática libre de contexto permite expresar tanto la sintaxis abstracta como la concreta. Como es bien conocido, metamodelos y gramáticas son dos formalismos para representar la sintaxis abstracta de un lenguaje y existen algoritmos para convertir gramáticas en metamodelos y viceversa [9]. En general, dado un metamodelo, cada clase corresponde a un símbolo no-terminal y para cada uno de ellos hay una producción o regla gramatical cuya parte izquierda es dicho símbolo no-terminal y la derecha está formada por una concatenación deducida de sus atributos y asociaciones, uno por cada atributo y clase destino de cada asociación contenedora y referencia, aunque es posible aplicar diferentes estrategias para simplificar la gramática o mejorar la legibilidad. Por ejemplo, una jerarquía de herencia puede ser representada como una producción con una alternativa por cada subclase, en la que cada alternativa está formada por el símbolo no terminal que representa a la subclase, por otro lado, también pueden incluirse caracteres delimitadores, como por ejemplo las llaves, para definir bloques en el texto y facilitar la legibilidad.

A continuación mostramos un fragmento de la gramática definida, el cual nos permitirá describir la estructura de una especificación Medea para el ejemplo de la Figura 3a. El símbolo inicial es *smm_model* y la primera producción expresa que un modelo de medidas SMM está formado por un conjunto de elementos, así como que una especificación Medea comenzará por la palabra clave *smm_model* seguida del identificador que denota el nombre del modelo y de la definición de los elementos encerrada entre llaves. Según la segunda producción cada elemento puede ser de los diferentes tipos considerados en SMM (librería, ámbito, característica, etc.). La tercera establece que una librería contiene cero o más medidas y se expresa con la palabra clave *libraries* seguida del nombre de la librería y las definiciones de las medidas entre llaves. La cuarta establece que hay dos tipos de medidas: *ranking* y *dimensional*. La quinta y la sexta definen la estructura de una medida dimensional que puede ser de dos tipos: directa y compuesta.

```
smm_model:
 'smm_model' ID '{' elements* '}'
;

elements :
 libraries | scopes | charac |...
;

libraries :
 'library' ID '{' measures+ '}'
;

measures :
 ranking | dimensional
;

dimensional :
 'dimensionalMeasure' name=ID '{'
   'scope' sc=ID
   ('trait'tr=ID)?
   'unit' un=ID
   'type' type
 '}'
;

type :
   'direct' '{' 'operation' OP }
 | 'collective' '{'
      'accumulator' accumulator
      'baseMeasure' bm=ID
   '}'
 | ...
;
```

El ejemplo de la Figura 3a podría ser expresado en Medea del siguiente modo:

```
smm_model myModel {
 characteristic ModuleCount
 scope codeModel{
  class code::CodeModel
 }
```

```
scope AbstractCodeElement{
 class code::AbstractCodeElement
}
library myLibrary{
 dimensionalMeasure ModuleCount{
  scope codeModel
  trait ModuleCount
  unit code::Module
  type collective{
   accumulator sum
   baseMeasure ModuleCountRecognizer
  }
 }
 dimensionalMeasure ModuleCountRecognizer{
  scope AbstractCodeElement
  trait ModuleCount
  unit code::Module
  type direct{
   operation "oclIsTypeOf(code::Module)"
  }
 }
}
```

La especificación del modelo de medidas SMM se ha llamado *myModel* y contiene un conjunto de declaraciones de medidas, ámbitos y características. Primero se define la característica a medir `ModuleCount` y los ámbitos *codeModel* y *AbstractCodeElement* utilizados por las medidas. Luego se define la librería *myLibrary* que engloba las definiciones de la medida compleja *ModuleCount* y de la medida simple *ModuleCountRecognizer*. La primera de ellas es una medida que suma el resultado de la segunda. En el ejemplo se han subrayado aquellos identificadores que actúan como referencias a elementos definidos en otro ámbito de la especificación. Por ejemplo, cuando se define la medida *ModuleCount* se referencia a la medida *ModuleCountRecognizer*. Dichas referencias se pueden comprobar en el modelo de ejemplo de la Figura 3. Tal y como se ha comentado anteriormente, la resolución de estas referencias para la obtención del modelo se ha visto facilitada por el uso del lenguaje de consultas integrado en Gra2MoL [7].

4. Motor de ejecución

El motor de ejecución creado es capaz de interpretar una especificación de medidas expresada en Medea, ejecutarla y finalmente crear un modelo que contenga las mediciones asociadas a las medidas.

La Figura 5 muestra el esquema del proceso de interpretación y ejecución de los modelos de medidas SMM. Las entradas del intérprete son: (1) la especificación Medea, denotada como $Metrics_{Medea}$ en la figura (2) el metamodelo al que conforman los modelos sobre los que aplicar las medidas, MM_{KDM} en la figura y (3) el modelo sobre el que aplicar las medidas, M_{KDM} en la figura, que conformará con el metamodelo MM_{KDM}. El primer paso es generar un modelo SMM a partir de la especificación Medea, para lo cual el intérprete interactúa con el motor de ejecución de Gra2MoL (pasos 1, 2 y 3 en la figura)

Conforme son interpretadas, en un segundo paso las medidas del modelo generado son ejecutadas sobre el modelo a evaluar M_{KDM}. Este proceso de ejecución utiliza un motor de ejecución de expresiones OCL (*OCLManager*) y un gestor de modelos (*ModelManager*). El primero permite ejecutar las expresiones OCL con las que se expresan las operaciones a ejecutar, las cuales son interpretadas por el framework *EMF Validation* de la plataforma Eclipse; y el segundo se encarga de recorrer el modelo de entrada. Conforme se van ejecutando las medidas, se va construyendo un modelo SMM que contiene los elementos de medición resultantes (M_{SMM} en la figura).

La mayor parte de las operaciones relacionadas con la navegación y consulta del modelo sobre el cual calcular las métricas son realizadas por el componente *ModelManager* (formado por una única clase Java). Sin embargo, dado que se utiliza el API generada por EMF para el metamodelo KDM, se ha creado el componente adicional *KDMModelManager*, el cual especializa el comportamiento del manejador de modelos para hacer uso de los elementos de dicho API. El componente *KDMModelManager* se genera automáticamente junto con el API de EMF y del mismo modo se podrían crear especializaciones para otros metamodelos.

Como resultado de la ejecución del proceso, se obtiene un modelo conforme al metamodelo SMM que contiene las mediciones resultado de haber aplicado las medidas, junto con las referencias al modelo evaluado.

5. Ejemplo

Con el propósito de presentar un proceso para la creación de herramientas de modernización basadas en ADM, en [10] se utiliza un caso de estudio de medición de métricas relacionadas con la complejidad de migrar aplicaciones Oracle

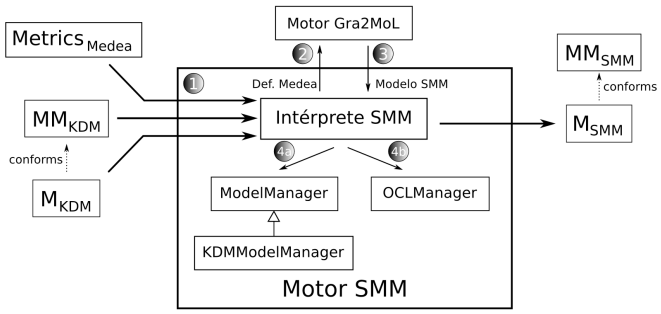

Figura 5. Proceso definido para el motor de ejecución de métricas SMM.

Forms a la plataforma Java. En dicho proceso, primero se obtienen modelos KDM del código PL/SQL de una aplicación Oracle Forms (Figura 6a), utilizando ASTM para crear un modelo intermedio que representa el código como un árbol de sintaxis abstracta. A continuación, por medio de una trasformación modelo-a-modelo, se calculan una serie de métricas que permiten medir el nivel de acoplamiento del código PL/SQL con la interfaz de usuario. El objetivo de calcular dichas métricas es disponer de información que ayude a cuantificar el esfuerzo para llevar a cabo la migración del código PL/SQL. De esta forma, cuanto mayor es el nivel de acoplamiento, mayor es el esfuerzo para efectuar la migración. En aquel contexto, las mediciones se representaban por medio de un metamodelo *ad-hoc*, dado que la especificación SMM todavía no había sido publicada.

La transformación modelo-a-modelo tomaba como entrada el modelo KDM generado y obtenía el modelo de mediciones a partir de las operaciones de medición expresadas en las reglas de la transformación, esto es, no se manejaba un modelo de medidas, solamente de mediciones.

Ahora hemos modificado el proceso anterior para utilizar el motor SMM y el metamodelo SMM como formato de representación de las medidas y mediciones. La Figura 6b muestra la incorporación del motor de métricas en el proceso sustituyendo a la transformación modelo-modelo mencionada.

Según la forma de acceder a la interfaz gráfica, se definieron tres niveles de acoplamiento: reflexivo (por ejemplo, mediante el uso de las funciones `NAME_IN` o `COPY`), declarativo (por ejemplo, utilizando sentencias `select`) e imperativo (por ejemplo, mediante el uso de sentencias de asignación). El acoplamiento reflexivo es el más complicado de migrar debido a que es necesario estudiar el contexto de ejecución del código.

Las métricas definidas están basadas en la localización y conteo de los diferentes tipos de sentencias que hacen uso de la interfaz gráfica en un modelo KDM. Los modelos KDM representan a los *triggers* PL/SQL como elementos *CallableUnit*, los cuales están formados por un conjunto de *ActionElement*s que describen las sentencias que contienen. Los elementos *ActionElement* contienen elementos *AbstractActionRelationship* que describen de forma atómica el comportamiento de la sentencia. Por ejemplo, los elementos *Reads* y *Writes* describen la lectura y escritura de variables, respectivamente. El atributo *kind* de los elementos *ActionElement* permite identificar el tipo de sentencia, esto es, declarativo o imperativo. Por otra parte, la identificación de sentencias reflexivas se realiza por medio de la localización de elementos de tipo *Call*, los cuales representan una llamada a un método, en este caso, debe comprobarse si el método llamado es reflexivo.

Por tanto, se ha definido una especificación Medea con tres medidas compuestas basadas en tres directas, cuyo objetivo es localizar y contar el número de sentencias asociadas a cada nivel de acoplamiento. La medida compuesta *impCount* basada en la medida directa *impCountRecognizer* se encarga contar el número de sentencias imperativas que interactúan con la interfaz gráfica en un *trigger*. Su definición textual sería la siguiente:

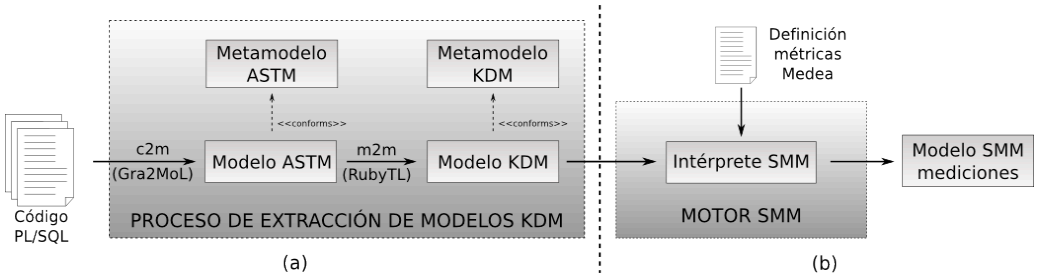

Figura 6. (a) Proceso de extracción de modelos KDM a partir de código PL/SQL. (b) Uso del motor SMM para el cálculo de métricas

```
characteristic imperativeCount
scope CallableUnit{
 class code::CallableUnit
}
scope AbstractActionRelationship {
 class action::AbstractActionRelationship
}
dimensionalMeasure impCount{
 scope CallableUnit
 trait imperativeCount
 unit action::AbstractActionRelationship
 type collective {
  accumulator sum
  baseMeasure impCountRecognizer
 }
}
dimensionalMeasure impCountRecognizer{
 scope AbstractActionRelationship
 trait imperativeCount
 unit action::AbstractActionRelationship
 type direct {
  operation
  "if(self.oclIsTypeOf(action::Reads))
   then
    let rVar : action::Reads =
     self.oclAsType(action::Reads) in
      rVar.from->
       exists(e| e.kind <> 'select' or
                 e.kind <> 'insert' or
                 e.kind <> 'update') and
      rVar.to.
       oclIsKindOf(code::StorableUnit) and
      rVar.to.name.substring(1, 1) = ':
   else
    if(self.oclIsTypeOf(action::Writes))
    then
     let wVar : action::Writes =
      self.oclAsType(action::Writes) in
       wVar.from->
        exists(e| e.kind <> 'select' or
                  e.kind <> 'insert' or
                  e.kind <> 'update') and
       wVar.to.
        oclIsKindOf(code::StorableUnit) and
       wVar.to.name.substring(1, 1) = ':'
    else false
    endif
   endif"
 }
}
```

La estrategia seguida para definir esta medida es parecida a la mostrada en el ejemplo de la Sección 3 para calcular el número de módulos en un modelo. El objetivo genérico es contar el número de elementos de un modelo que cumplen una determinada condición. La forma de describir este comportamiento en SMM es utilizando una medida compleja que sume el resultado de una medida simple, la cual comprueba si el elemento cumple la condición. De esta forma, en la definición anterior, la medida *impCount* es una medida de tipo *CollectiveMeasure* que se encarga de sumar el resultado de la medida directa *impCountRecognizer*. Esta última medida aplica la expresión OCL especificada para comprobar que una sentencia es de tipo imperativa y comienza con el carácter '`:`', que es el formato para representar variables de la interfaz gráfica. Además, se define la característica *imperativeCount* que representa a las medidas, así como los ámbitos principales, los cuales también serán utilizados en las siguientes medidas. El primer ámbito, llamado *CallableUnit*, establece el elemento *CallableUnit*, que representa a los *triggers* en el modelo KDM, como ámbito de la medida *impCount*. El segundo ámbito, llamado *AbstractActionRelationship*, establece el elemento *AbstractActionRelationship*, que representa a las sentencias del *trigger,* como ámbito de la medida *impCountRecognizer*.

La segunda métrica compuesta, llamada *declCount*, está basada en la medida directa *declCountRecognizer* y se encarga de contar el número de sentencias declarativas que interactúan con la interfaz gráfica en un *trigger*. Su definición es similar a la anterior exceptuando la expresión OCL utilizada, la cual comprueba que el atributo *kind* sea igual a alguno de los valores que identifican a una sentencia declarativa (`select`,

`insert`, `update`, en el ejemplo). Por cuestiones de espacio, no incluimos la definición de esta medida, la cual puede ser descargada desde la web de la herramienta.

Finalmente, la métrica compuesta refCount basada en la medida directa *refCountRecognizer* se encarga de contar el uso de sentencias reflexivas que interactúan con la interfaz gráfica.

```
characteristic reflectiveCount
scope AbstractActionRelationship{
 class action::Calls
}
dimensionalMeasure refCount{
 scope CallableUnit
 trait reflectiveCount
 unit  action::ActionElement
 type collective{
  accumulator sum
  baseMeasure refCountRecognizer
 }
}
dimensionalMeasure refCountRecognizer{
 scope AbstractActionRelationship
 trait reflectiveCount
 unit  action::Calls
 type direct{
  operation
    "self.to.
      oclIsKindOf(code::MethodUnit) and
     (self.to.name == 'name_in' or
      self.to.name == 'copy')"
 }
}
```

La definición de medida directa anterior difiere de las dos anteriores en que el ámbito es *AbstractActionRelationship* en vez de *AbstractCodeElement.* En este caso la medida simple se encarga de comprobar si un elemento es de tipo *Calls* y llama a un método reflexivo.

Las métricas fueron aplicadas al código PL/SQL de una aplicación Oracle Forms utilizada como parte del Sistema de Gestión Académica de alumnos de la Universidad de Murcia. Del modelo de mediciones obtenido, se aplicó una transformación modelo a código para obtener un fichero de valores separados por comas para poder representarlos gráficamente. La Figura 7 muestra el nivel de acoplamiento detectado en los *triggers* de tres de los formularios de la aplicación. La información visualizada ayuda a comprender la dificultad del proceso de migración para cada formulario. Considerando el acoplamiento de la interfaz de usuario, en este caso el formulario *EUProjects* sería más difícil de migrar que los otros dos, sin embargo, también debería tenerse en cuenta otros aspectos como el tamaño y complejidad del código.

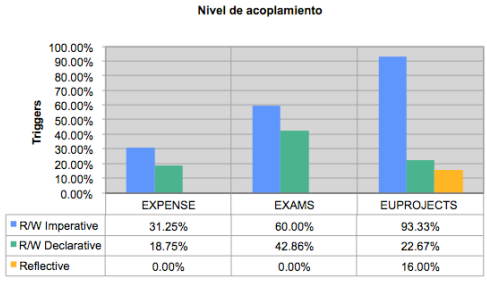

Figura 7. Nivel de acoplamiento de tres formularios Oracle Forms en un sistema de gestión de alumnos. Cada barra indica la proporción de *triggers* que contiene un tipo concreto de acoplamiento. Un *trigger* puede contener más de un tipo de acoplamiento.

6. Trabajo relacionado

En la actualidad, dada la reciente aparición de la especificación SMM, no existe un gran número de trabajos relacionados con este metamodelo. Precisamente, en [11, 12] se presentan aproximaciones generativas dirigidas por modelos para la construcción de software de cálculo de métricas, sin embargo, no utilizan los metamodelos de ADM.

En [13] se presenta una aproximación con el mismo propósito que Medea. En ella los modelos de medidas SMM se generan a partir de reglas que capturan patrones de métricas expresados como operaciones OCL. Los diferentes tipos de medidas de un modelo SMM se derivan de las tuplas que resultan al ejecutar el código OCL de la regla y un motor de ejecución SMM produce el modelo de medición. La principal diferencia con nuestra propuesta es que no se proporciona un DSL para expresar medidas sino que el usuario debe usar las reglas existentes o crear nuevas reglas. Aunque se comenta la implementación de un prototipo de motor SMM, la herramienta aún no está disponible y no hemos podido evaluarla.

MoDisco [14] forma parte del proyecto GMT de Eclipse y es un framework creado para obtener modelos en el contexto de modernización dirigida por modelos. Ofrece una implementación para el metamodelo KDM y actualmente acaba de publicar una implementación del metamodelo SMM. Sin embargo, no dispone todavía de herramientas para la definición o ejecución de modelos SMM.

Existen diversas propuestas para definir métricas basadas en OCL pero no son genéricas y se definen para modelos UML [15, 16].

7. Conclusiones

En este artículo se ha presentado Medea, que según nuestro conocimiento es el primer lenguaje de definición de medidas SMM ejecutables. Un motor de ejecución permite ejecutar las medidas y obtener modelos de medición SMM. Se ha utilizado Gra2MoL para dotar de una sintaxis concreta textual al metamodelo de sintaxis abstracta de SMM.

Hemos mostrado la aplicación de Medea a un caso de estudio real que anteriormente se había abordado mediante un metamodelo de métricas ad-hoc. Con el uso de SMM nos ahorramos la definición de los metamodelos de métricas y mediciones y ganamos en interoperabilidad ya que todos los metamodelos empleados son estándar: KDM y SMM.

Con el motor de ejecución de SMM implementado será posible que la comunidad experta en métricas pueda probar y validar el metamodelo SMM y valorar su utilidad real. El motor de ejecución puede descargarse desde la dirección `http://modelum.es/medea`.

Como trabajo futuro, creemos interesante estudiar cómo mejorar la sintaxis concreta para elevar su potencial y dotarlo de constructores más complejos que puedan permitirnos definir métricas más eficientemente, como por ejemplo, sentencias para definir directamente medidas de conteo de elementos sin tener que utilizar una medida compuesta que sume el resultado de otra simple. Otra tarea a abordar es la integración de la herramienta en la plataforma AGE [16], ofreciendo un editor específico para el lenguaje definido. Finalmente, como objetivo a largo plazo, sería la identificación de métricas software y su organización en librerías, permitiendo disponer de un repositorio de métricas expresado en Medea.

Agradecimientos

Este artículo ha sido parcialmente financiado por las ayudas Seneca 08797/PI/08 y 129/2009 de la Consejería de Universidades e Investigación (Murcia). Javier Luis Cánovas Izquierdo dispone de una beca FPI de la Fundación Séneca

Referencias

[1] ADM. http://adm.omg.org
[2] Tom Mens, Serge Demeyer, “Future Trends in Software Evolution Metrics”. In: Proc. Int’l Workshop on Principles of Software Evolution (IWPSE 2001), pp. 83-86 ACM Press.
[3] Markus Völter: MD* Best Practices. Journal of Object Technology, (6): 79-102 (2009)
[4] Xtext. http://www.eclipse.org/Xtext/
[5] F. Jouault, J. Bèzivin, and I. Kurtev, “TCS: a dsl for the specification of textual concrete syntaxes in model engineering", in GPCE, pp. 249-254 (2006).
[6] Emftext. http://www.emftext.org
[7] J. Cánovas Izquierdo, J. G. Molina. “A domain specific language for extracting models in software modernization” en ECMDA 2009, LNCS 5562, pp. 82-97, 2009
[8] J. Cánovas Izquierdo, O. Sánchez, J. Sánchez Cuadrado, J. G. Molina. “DSLs para la extracción de modelos en modernización”, V Taller DSDM 2008, Gijón (España).
[9] Jack Greenfield, *Software Factories* (capítulo 8), John Wiley & Sons, 2004.
[10] J. Cánovas Izquierdo, J. García Molina. “An Architecture-Driven Modernization Tool for Calculating Metrics”, IEEE Software, Software Evolution SI, Julio/Agosto 2009.
[11] M. Montperrus, J. M. Jézéquel, B. Baudry, J. Champeau, B. Hoeltzener. “Model-driven generative development of measurement software”. Software and Systems Modeling (SoSyM), 2010. *To be published*.
[12] SMF Tool. http://alarcos.esi.uclm.es/smf
[13] M. Engelhardt, et al. “Generation of Formal Model Metrics for MOF based Domain Specific Languages”, OCL Workshop, MODELS, 2009.
[14] MoDisco Eclipse Tool website. http://www.eclipse.org/gmt/modisco
[15] M. Clavel, M. Egea, V. Torres da Silva. “Model Metrication in MOVA: A Metamodel-based Approach using OCL”. (2007)
[16] A.L. Baroni, et al.., “Using OCL to Formalize Object Oriented Design Metrics Definitions”. In: Proc. of QAOOSE Málaga, Spain (2002)
[17] J. S. Cuadrado, J. G. Molina. “Building Domain-Specific Languages for Model-Driven Development”, IEEE Software 24(5): 48-55 (2007)