Informe Final del Proyecto de Grado
Testing de Transformaciones de Modelos

Estudiantes: Leonardo López, Leonardo Pintos
Tutores: Daniel Calegari, Carlos Luna.

Montevideo, Mayo 2011.

Instituto de Computación, Facultad de Ingeniería, UdelaR.

Resumen

En el contexto del Desarrollo de Software Guiado por Modelos (Model-Driven Development, MDD) las transformaciones de modelos constituyen la actividad principal en el desarrollo de los sistemas. Las transformaciones de modelos pueden ser vistas como programas que toman como entrada un modelo y generan otro en base a un conjunto de reglas previamente definidas. Estas transformaciones deben ser verificadas y validadas rigurosamente para lograr el éxito del paradigma MDD . El proyecto de grado documentado en este informe expone un relevamiento sobre las técnicas de testing tradicionales que han sido adaptadas para transformaciones de modelos y brinda un panorama sobre las barreras existentes y los desafíos a superar en el futuro. A los efectos de poner en práctica los conceptos teóricos relevados se realiza un caso de estudio. El caso de estudio consiste en la verificación de una transformación definida en un proyecto anterior, en base a la generación de casos de prueba y la aplicación de una técnica de verificación existente en la literatura.

Palabras Clave: Desarrollo de Software Guiado por Modelos, Transformación de modelos, Testing de transformaciones de modelos, Caja Negra, Caja Blanca

Índice

1. Introducción.

El Desarrollo de Software Dirigido por Modelos (Model-Driven Development, MDD [1]) es un enfoque que propone un proceso (semi) automático de transformación de los modelos que parte de la especificación de requerimientos y alcanza el código ejecutable. En el contexto de MDD, un modelo es una abstracción de cierto aspecto del sistema que se desea construir, que permite comprender mejor el sistema evitando la complejidad intrínseca de la realidad. Estos modelos permiten obtener diferentes visiones del sistema en construcción a diferentes niveles de abstracción. Todo modelo conforma con su metamodelo, que especifica la sintaxis abstracta de un modelo.

Los elementos básicos de toda transformación de modelos se pueden apreciar en la Figura 1.1. Aquí se puede ver como una transformación se define en base a metamodelos origen y destino. Se dice entonces que estos modelos conforman a este metamodelo. Esta definición de transformación es tomada por el motor de transformaciones el cual genera un modelo destino a partir del modelo origen.

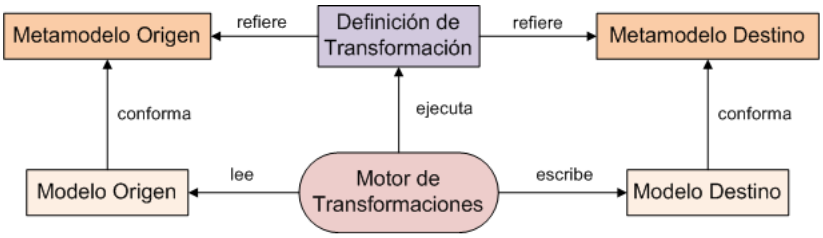


Figura 1.1: Elementos de una transformación de modelos.

Escribir estas transformaciones de modelos es una tarea propensa a errores por la complejidad de las mismas y se requieren técnicas de validación y verificación que nos permitan lograr transformaciones de modelos confiables.

Existe una gran variedad de técnicas para verificar transformaciones de modelos, las cuales se pueden agrupar en tres enfoques diferentes [2]: verificación de modelos, métodos deductivos, y basados en casos de prueba. La verificación de modelos (i.e. model-checking) permite probar en forma automática características de los modelos involucrados en una transformación, e incluso propiedades sobre la transformación en caso de que esta pueda ser representada con un grafo. Por su parte, los métodos deductivos utilizan lenguajes formales y herramientas matemáticas para realizar demostraciones que permiten certificar transformaciones como correctas. Finalmente, la verificación basada en casos de prueba (i.e. testing) centra la verificación de la transformación a nivel de los modelos particulares que son transformados, trabajando sobre un conjunto de modelos origen y sus correspondientes modelos destino.

El objetivo principal de este proyecto de grado fue realizar un estudio del estado del arte sobre testing de transformaciones de modelos(verificación basada en casos de prueba [3, 4]). Este relevamiento fue documentado en el reporte técnico publicado y adjunto a este informe: "Estado del Arte de Testing de

Transformaciones de Modelos"[5]. Se realizó también un caso de estudio que consistió en realizar el testing de una transformación de modelos de interés para el proyecto de investigación general. Para el caso de estudio se utilizó la técnica de generación de casos de prueba tomando como base el algoritmo propuesto en [6] y luego se verificó el cubrimiento que tenía el conjunto de casos de prueba con respecto al metamodelo de entrada de la transformación, mediante la utilización de la herramienta MMCC [7] producto del trabajo en [8].

El informe se organiza de la siguiente manera. En el capítulo 2 se presentarán las barreras existentes y desafíos a superar para el testing de las transformaciones de modelos, se explicará también como lograr la adaptación de las técnicas tradicionales de testing a este enfoque y se muestran estrategias tanto para la validación de los casos de prueba como para la validación de los resultados obtenidos luego de la transformación. Posteriormente en el capítulo 3 se presentará la transformación de modelos que servirá como caso de estudio, la generación de casos de prueba y la utilización de MMCC para medir el cubrimiento del conjunto de casos de prueba. Finalmente en el capítulo 4 se presentan las conclusiones del trabajo y una guía para el trabajo futuro. El anexo A muestra las diferentes etapas del proyecto y su duración. A continuación, el anexo B describe los detalles técnicos de la herramienta MMCC. Por último, los anexos C y D muestran los casos de prueba y el código fuente de los mismos respectivamente.

2. Testing de Transformaciones de Modelos.

Este capítulo resume el relevamiento realizado en el reporte técnico[5] sobre el estado del arte de testing de transformaciones de modelos. El testing realizado sobre transformaciones de modelos se basa principalmente en la adaptación de técnicas utilizadas en aplicaciones de informática en general[9, 10, 11] a este paradigma. Se detallarán las barreras y desafíos que se deben afrontar, así como la generación de casos de prueba, validación de casos de prueba y validación de resultados.

Este capítulo está organizado de la siguiente manera. En la subsección 2.1 se comentan las barreras y desafíos que se tienen al realizar el testing de una transformación de modelos, asi como posibles caminos o sugerencias que se pueden tomar para afrontarlas. En la subsección 2.2 se describe la generación de casos de prueba de caja negra. En la subsección 2.3 se presenta la generación de casos de prueba de caja blanca. En la subsección 2.4 se describen técnicas para validar los casos de pruebas. Por último en le subsección 2.5 se describen técnicas para validar resultados.

2.1. Barreras y Desafíos del Testing de Transformación de Modelos.

Las transformaciones de modelos forman parte de una clase de programas en los cuales la complejidad de los datos de entrada y salida y la heterogeneidad de los lenguajes de transformación hacen del testing de transformaciones de modelos un duro trabajo para los testeadores. Analizaremos los problemas a enfrentar y comentaremos sugerencias sobre como encaminar la solución de alguno de ellos. El desarrollo de la subsección 2.1 esta basado completamente en conocimientos brindados en [12] y [13].

2.1.1. Complejidad de datos de entrada/salida.

La complejidad que tienen los metamodelos debido a su tamaño, restricciones, multiplicidades, lleva a que la generación de modelos no sea una tarea sencilla. Las técnicas de generación tanto manuales como automáticas se dificultan por la gran cantidad de instancias del metamodelo que se deben considerar. Una de las estrategias mas utilizadas para la construcción de modelos válidos es construir el modelo y luego verificar restricciones, descartando los modelos que no las cumplen. De todas formas se tiene en cuenta la limitante de satisfacer el conjunto completo de restricciones, que reducen considerablemente el espacio de modelos posibles y válidos.

Con respecto a los modelos de salida de la transformación, su complejidad dificulta la creación de oráculos. El oráculo es un programa que se encarga de validar la salida de un programa. Puede ser construido a partir de los resultados esperados, comparando los mismos con la salida de la transformación, siendo esta comparación un problema NP-Completo. Si el oráculo es construido a partir de propiedades extraídas de los modelos de salida, su construcción también es

dificultosa por la complejidad del metamodelo destino que describe a los modelos de salida.

Existen otras alternativas para la generación de oráculos. Una de ellas plantea la generación de un oráculo parcial que corrobore la corrección de determinadas propiedades del modelo de salida. Esta estrategia es estudiada en [14] y en [15]. Otra posibilidad, cuando la trasformación genera un modelo ejecutable, es probar directamente el modelo de salida. En la sección 2.5 se profundiza el estudio sobre oráculos.

2.1.2. Heterogeneidad de lenguajes y herramientas.

Existe un gran número de lenguajes y herramientas [16], en las cuales las transformaciones pueden ser implementadas. Estos lenguajes y herramientas pueden ser de propósito general o específicos para transformaciones de modelos. La diversidad existente es grande, y es complicado saber si existe un lenguaje que se adapte a todas las necesidades requeridas por las transformaciones de modelos. En consecuencia a la hora de realizar la verificación de transformaciones no se puede elegir un lenguaje de referencia, lo cual lleva a implementar criterios de prueba generales.

A la hora de definir criterios de pruebas se sugieren dos posibles estrategias. La primera es tener criterios y técnicas específicos para cada lenguaje particular. La segunda propone utilizar técnicas de caja negra donde se ignore el lenguaje utilizado en la transformación. Ejemplos de las estrategias descritas anteriormente se encuentran en [17] y en [4]. El primero propone el criterio de caja negra para evaluar el cubrimiento que logran los modelos usados como casos de prueba con respecto al metamodelo origen, para esto se utiliza lo planteado en 2.2 y en 3.5. El beneficio de este enfoque es que se puede usar con cualquier lenguaje de transformación. El segundo trabajo utiliza un método de caja blanca, el cual se describe en 2.3. El inconveniente de este enfoque es que se encuentra fuertemente acoplado al lenguaje de transformación y tendría que ser adaptado o completamente redefinido para otro lenguaje de transformación.

2.2. Generación de Casos de Prueba - Caja Negra.

La generación de casos de prueba utilizando la técnica de caja negra se basa en el metamodelo origen de la transformación (el cual define completamente el conjunto de los posibles modelos de entrada). Esta técnica tiene como una de sus principales ventajas que es independiente del lenguaje en el cual se formule la transformación.

La aproximación propuesta en [3] tiene como objetivo cubrir la mayor parte posible del metamodelo de entrada, aunque los autores notan que existen datos irrelevantes para el testing de la transformación. En consecuencia, previo a la generación de casos de prueba, se define un metamodelo relevante para las pruebas de la transformación llamado metamodelo efectivo.

El metamodelo efectivo se construye utilizando las pre y post condiciones y los elementos referenciados en la especificación de la transformación. Utilizando

este metamodelo efectivo se estudiará la aplicación de técnicas que se adaptan a transformaciones de modelos. El objetivo de las mismas es encontrar combinaciones de propiedades del metamodelo efectivo que permitan acotar aun mas el número de casos de prueba generados y brindar datos relevantes para el cubrimiento del metamodelo.

A continuación detallaremos la técnica mas estudiada y utilizada dentro de la generación de casos de pruebas de caja negra. La misma es conocida con el nombre de Particiones en clases de equivalencia y fue la utilizada en el caso de estudio realizado para este proyecto.

2.2.1. Particiones en clases de equivalencia.

La técnica de partición en clases de equivalencias consiste en dividir el dominio de cada propiedad en sub dominios, a estos sub dominios se les llama rangos. La división se realiza en base al conocimiento que se tenga del dominio de entrada, y consiste en identificar aquellos subconjuntos de valores del dominio de entrada para los cuales el sistema bajo test se comporta de la misma manera. Los rangos definen una partición del dominio de entrada, por lo que no deben solaparse.

En [18] los autores proponen en primer medida encontrar el conjunto de propiedades del metamodelo efectivo y luego buscar el conjunto de valores representativos para cada una de ellas. De esta forma se divide el dominio de cada propiedad en un conjunto de rangos que forman una partición.

Para determinar las particiones se pueden utilizar dos métodos. El primero, llamado particionado por defecto, se aplica cuando no se tiene conocimiento de los valores representativos y consiste en definir una partición basada en la estructura o el tipo de datos. El segundo método, llamado particionado basado en el conocimiento consiste en extraer valores representativos de la transformación. Estos valores podrían ser provistos por el testeador o ser extraídos de la especificación de la transformación. A modo de ejemplo, los autores mencionan que las pre y post condiciones de los métodos nos permiten identificar valores relevantes para atributos y posiblemente multiplicidades para las asociaciones.

Luego de definir las particiones y los rangos se generan los casos de prueba tomando valores de cada uno de los rangos.

La efectividad de esta técnica deriva de la calidad de la definición de las particiones. Por este motivo se deben identificar claramente los valores relevantes y asignarlos a rangos específicos para que sean tomados en cuenta a la hora de generar los casos de prueba.

Fragmentos de modelo y Fragmentos de objeto

En [8] el autor señala que el cubrimiento total de rangos, para cada propiedad, de forma independiente no es suficiente para la generación de casos de prueba relevantes. En consecuencia describe técnicas para generar casos de prueba que combinen valores de los distintos rangos que se tienen.

La primera aproximación que se comenta consiste en generar el producto cartesiano de todas las particiones para todas las propiedades, encontrándose

los siguientes problemas:

1. Explosión combinatoria, el número de combinaciones generadas se vuelve inmanejable.

2. Generación de casos de prueba que no son relevantes para el testing de la transformación.

3. Pérdida de combinaciones relevantes.

Como podemos observar, la aplicación de la aproximación anterior puede tener varios problemas, por lo tanto los autores introducen los conceptos de fragmento de modelo y de fragmento de objeto. Los fragmentos de objeto y de modelo definen restricciones sobre los objetos y los modelos que deben estar presentes en un conjunto de casos de prueba para que sean relevantes para el testing de la transformación[18]. Se expone lo anterior mediante un ejemplo.

La Figura 2.1 muestra el metamodelo correspondiente a un lenguaje para modelar un diagrama de clases simple. Un diagrama está compuesto por clases, tipos primitivos y asociaciones. Las clases pueden tener un conjunto de atributos con un nombre y tipo que debe ser un tipo primitivo y pueden a su vez ser la generalización de otra clase (relación parent). Además, las clases pueden estar asociadas entre sí.

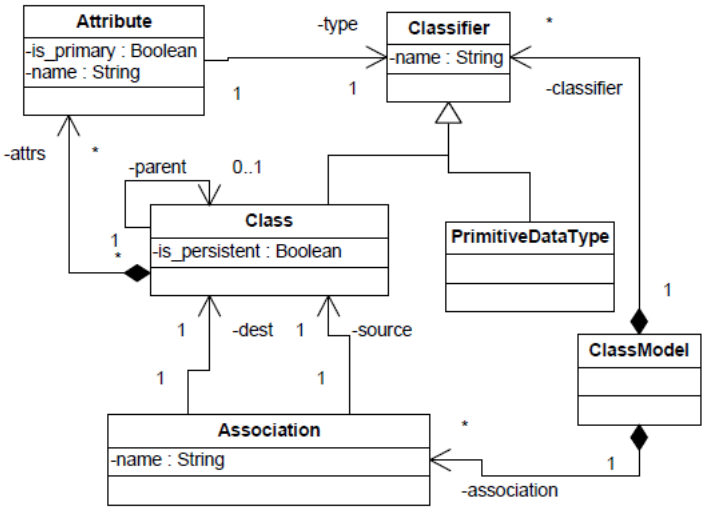


Figura 2.1: Metamodelo del Diagrama de Clases UML simple [18].

Este metamodelo define todos los modelos posibles de entrada para cualquier transformación que manipule diagramas de clases simples.

Luego de aplicar particionado por defecto sobre el metamodelo de la Figura 2.1 se obtienen las particiones (lado izquierdo de la tabla) y los rangos(lado derecho de la tabla) que se muestran en la Figura 2.2. Por ejemplo la partición

sobre el atributo booleano is_persistent de la clase Class contiene dos rangos para los dos valores posibles {true} y {false}.

Attribute::is_primary	{true}, {false}
Attribute::name	{« »}, {.+}
Attribute::#type	{1}
Class::is_persistent	{true}, {false}
Class::#parent	{0}, {1}
Class::#attrs	{0}, {1}, {x | x>1}
Association::name	{« »}, {.+}
Association::#dest	{1}
Association::#source	{1}
ClassModel::#association	{0}, {1}, {x | x>1}
ClassModel::#classifier	{0}, {1}, {x | x>1}

Figura 2.2: Particiones para el metamodelo del diagrama de clases simple [18].

Si se emplea la estrategia de requerir un caso de prueba por cada combinación de rangos posible (producto cartesiano), el total de casos de prueba que se tendría que generar asciende a 1296, lo cual representa un número elevado si consideramos la baja cantidad de conceptos que hay en el metamodelo.

Como alternativa se generan fragmentos de objeto que son instancias de clases del metamodelo que contienen valores restringidos dentro de algún rango válido. Estos fragmentos de objeto conforman fragmentos de modelo que establecen condiciones que deben ser satisfechas por al menos un caso de prueba del conjunto de casos de prueba.

En la siguiente figura se muestra un ejemplo de un fragmento de modelo que contiene dos fragmentos de objeto.

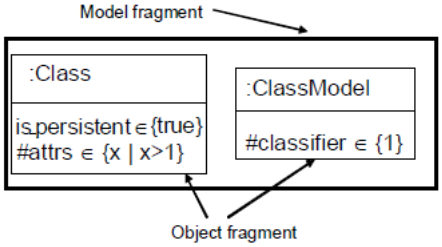


Figura 2.3: Ejemplo de fragmentos de objeto y de modelo [18].

El fragmento de objeto de la izquierda especifica que debe haber una instancia de "Class" en el caso de prueba tal que la propiedad "is_persistent" tome el valor "true" y que la propiedad "attrs" tenga una multiplicidad mayor que 1. El fragmento de objeto de la derecha especifica que en el caso de prueba anterior también debe haber una instancia de "ClassModel" tal que el número de "classifier" sea 1.

Usando los fragmentos de modelo las combinaciones particulares que deberían ser cubiertas por los casos de prueba se pueden representar fácilmente.

El conjunto de fragmentos de modelo para lograr el cubrimiento del metamodelo debe cumplir con las siguientes reglas:

Regla 1 - (Cubrimiento de clase): cada clase concreta debe ser instanciada en al menos un fragmento de modelo.

Regla 2 - (Cubrimiento de rango): cada rango de cada partición para todas las propiedades que tiene el metamodelo debe ser usado en al menos un fragmento de modelo.

Un conjunto de casos de prueba es satisfactorio si los casos de prueba cubren todos los fragmentos de modelo definidos. En caso que un fragmento de modelo no fuera cubierto se debe generar un caso de prueba manualmente y agregarlo al conjunto.

El número de fragmentos de modelo nos da entonces una pauta de la cantidad de casos de prueba. El número de fragmentos de objeto de un fragmento de modelo da una medida del tamaño del caso de prueba.

Al concluir el artículo los autores presentan que estos criterios pueden usarse en la práctica para crear un conjunto inicial de fragmentos pero en la mayoría de los casos este conjunto deberá ser mejorado, ya sea por el testeador o usando un criterio más fuerte. Asimismo establecen que la estrategia presentada es genérica y puede ser aplicada a cualquier transformación de modelo, ya que se basa en el conocimiento de la estructura del metamodelo de entrada.

2.2.2. Otras técnicas.

Además de la técnica de particionado por clases de equivalencias, la cual observamos es la mas utilizada y estudiada por los líderes en el área del testing de transformaciones de modelos, existen otras técnicas que buscan de diferente forma, el cubrimiento del metamodelo y la generación de datos relevantes para definir casos de prueba. A continuación describiremos algunas de ellas.

La técnica llamada Testing de Interacción Combinatoria [19, 20], selecciona un subconjunto de todas las posibles combinaciones de las variables a testear. Está fundamentada en la observación de que la mayoría de la fallas son disparadas por interacciones entre un número pequeño de variables. En consecuencia se definió el 2-way testing [21], que consiste en armar un conjunto de casos de prueba tal que estén todos los posibles pares de valores de las variables.

A modo de ejemplo consideremos un sistema S que tiene 3 variables de entrada X, Y y Z. Asumamos que se tiene un conjunto de valores D que pueden tomar cada una de las variables, tales que D(X)={1,2}; D(Y)={Q,R}; y D(Z)={5,6}.

El número de casos de prueba totales que podemos armar es 2x2x2=8. Luego de aplicar 2-way testing el conjunto de casos de prueba se reduce a la mitad y los casos de prueba quedarían:

CP1: [1,Q,5]; CP2:[1,R,6]; CP3:[2,Q,6]; y CP4:[2,R,5], siendo los componentes de los vectores los valores correspondientes a las respectivas entradas X, Y y Z. Por más información ver [22].

Otra técnica utilizada es el Método de Clasificación de Árbol [23]. Esta técnica utiliza y mejora las ideas planteadas en la técnica de Particiones en Clases de Equivalencia y requiere de los testeadores para definir las clasificaciones mediante las cuales se dividirá el dominio de entrada. Se toma el dominio de entrada de la transformación y se identifican los aspectos relevantes para el test. Por cada aspecto se forman clasificaciones completas y disjuntas que llamaremos clases. La partición del dominio de entrada mediante estas clasificaciones se representa gráficamente en forma de árbol. Por último, se generan los casos de prueba combinando clases de diferentes clasificaciones. La fuente más importante de información para el testeador es la especificación funcional de la transformación. Una gran ventaja del método de clasificación de árbol es que convierte el diseño de casos de prueba en un proceso que comprende varias partes estructuradas y sistematizadas, el cual es fácil de manejar, comprensible y documentable.

A continuación introducimos el ejemplo presentado en [24].

Se cuenta con un sistema visor que debe determinar ciertas características de objetos que se le pasan como entrada. Los aspectos en este caso serían el tamaño, el color y la forma del objeto de entrada.

La clasificación basada en el aspecto "color" lleva a la partición del dominio en objetos de color rojo, verde y azul. Asimismo, la clasificación basada en el aspecto "forma" lleva a la partición del dominio en objetos de forma triangular, circular o cuadrada. Se agrega en el ejemplo un aspecto adicional sobre la clase triángulo: el tipo de triángulo. Las clasificaciones y las clases se muestran en la Figura 2.4.

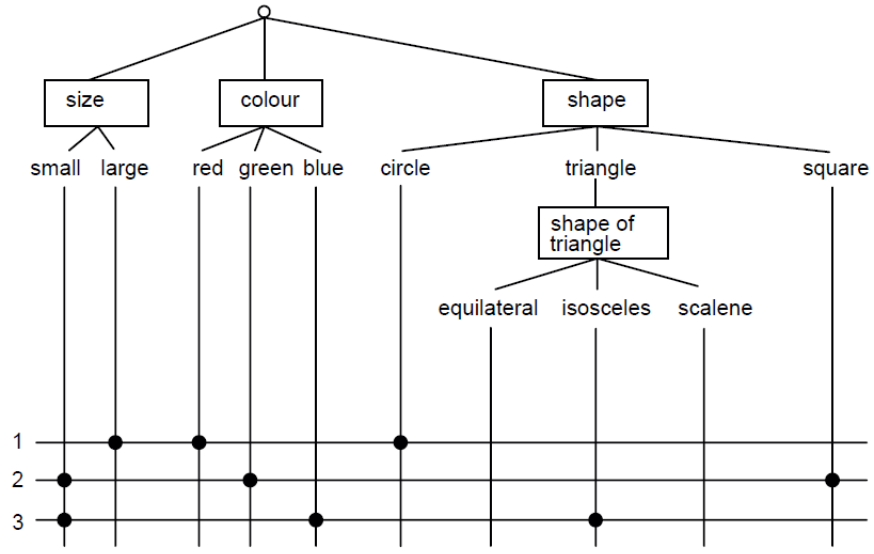


Figura 2.4 - Árbol de Clasificación [24].

En la parte inferior de la figura se muestra la tabla de combinaciones asociadas con tres posibles casos de prueba de ejemplo. Por mas información consultar

[24].

2.3. Generación de Casos de Prueba - Caja Blanca.

Esta subsección se basa completamente en los conocimientos brindados en [4]. Las técnicas de caja blanca utilizan el diseño y la implementación de la transformación para la generación de casos de prueba. El diseño es definido mediante un conjunto de reglas conceptuales, las cuales toman como entrada un subconjunto de elementos del modelo de entrada y retornan un subconjunto de elementos del modelo de salida. La implementación de la transformación consiste en plasmar estas reglas en un lenguaje de alto nivel como QVT [25] que es específico para transformaciones o en un lenguaje general como Java. Una desventaja importante de estas técnicas es el acoplamiento al lenguaje en el cual se implementa la transformación. En consecuencia, se deben redefinir o adaptar las reglas si el lenguaje cambia. Por el contrario, estas técnicas son mas detalladas y efectivas que las técnicas basadas en caja negra, debido a que se basan en los pasos para realizar la transformación. Algunos errores que se pueden cometer en la definición de las reglas conceptuales de la transformación son los siguientes:

- Cubrimiento de metamodelo: la regla puede haber sido definida sin cubrir completamente los elementos del metamodelo, por lo tanto puede ocurrir que algunos modelos no puedan ser transformados.
- Creación de modelos sintácticamente incorrectos: La regla de transformación no está correctamente implementada, esto lleva a que los modelos resultantes de la transformación no conformen o violen las restricciones especificadas en el metamodelo destino.
- Creación de modelos semánticamente incorrectos: La regla de transformación ha sido aplicada a un modelo de entrada para el cual no corresponde, o sea el modelo resultado es sintácticamente correcto pero no es una transformación semánticamente correcta del modelo origen.
- Confluencia: La transformación produce diferentes salidas para el mismo modelo de entrada, porque la transformación no es confluente.
- Corrección de la semántica de la transformación: La transformación no preserva una propiedad que fue especificada para la transformación (por ejemplo: ausencia de deadlocks).
- Errores debido a codificación incorrecta.

En el artículo se describen técnicas para depurar los incidentes comentados.

La primera de ellas, es la técnica de cubrimiento del metamodelo. Esta técnica consiste en generar plantillas por cada regla conceptual de la transformación. Estas plantillas son un modelo abstracto con parámetros que pueden instanciarse asignando a cada parámetro un conjunto de valores. Con el conjunto de

plantillas para cada regla se puede asegurar un alto grado de cubrimiento del metamodelo origen.

Mediante esta técnica se puede facilitar también la generación automática de casos de prueba como instancias de las plantillas de cada regla. A modo de ejemplo presentamos en la Figura 2.5 una regla conceptual, su plantilla de metamodelo y un par de posibles instancias que se pueden generar a partir de esta plantilla.

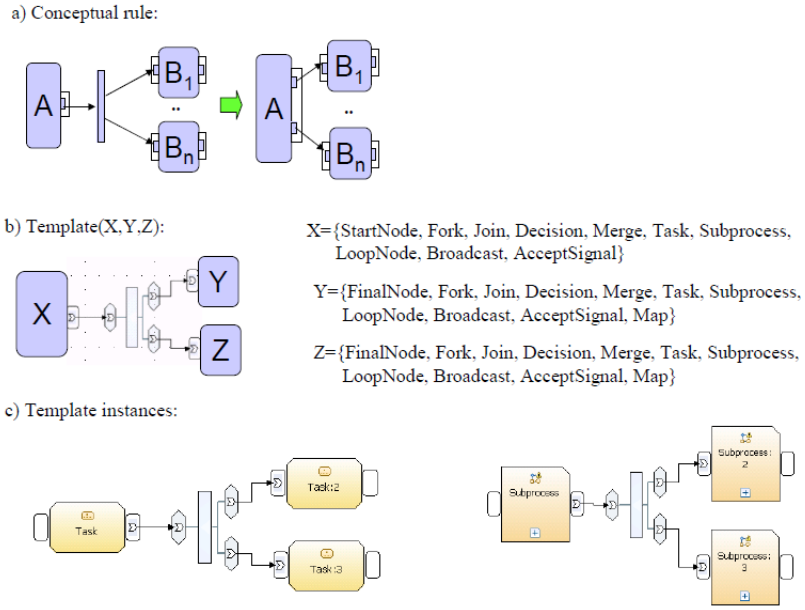


Figura 2.5 - Regla conceptual, plantilla de metamodelo y posibles instancias.

El ejemplo muestra la regla conceptual que tiene como resultado remover el "fork" del modelo de proceso de negocio expresado en CANF (Control Action Normal Form) para transformarlo en un modelo de proceso de negocio expresado en PNF (Pinset Normal Form). La plantilla que se muestra en la Figura 2.5 b) se deriva de la regla de la Figura 2.5 a) fijando la cantidad de nodos B_i a 2; X, Y y Z denotan los parámetros y se muestran también los valores que puede tomar cada uno. Estos valores se obtienen del metamodelo del lenguaje de modelado de procesos de negocio.

Por último en la Figura 2.5 c) se pueden ver dos ejemplos de instancias de plantillas que se derivan de b).

Es importante notar que por cada regla se pueden construir varias plantillas que cubran una parte del metamodelo de entrada al cual aplica. Con el conjunto de plantillas de cada regla se puede asegurar un alto grado de cubrimiento del metamodelo de entrada.

Una posible extensión a este enfoque es utilizar las pre-condiciones de la transformación y solamente generar los casos de prueba que cumplen con dichas

pre-condiciones. Sin embargo, si hay una pre-condición que involucra varios elementos del modelo y estos elementos forman parte de más de una regla entonces no se podrán generar casos de prueba válidos.

La siguiente técnica se basa en el uso de restricciones para construir los casos de prueba. El objetivo de la misma es corroborar el cumplimiento de las restricciones de integridad luego de aplicada la transformación. A continuación comentamos los pasos a seguir para la construcción de casos de prueba a partir de estas restricciones :

- Identificar los elementos que fueron modificados luego de la transformación.
- Identificar las restricciones que se vieron afectadas por algún cambio en un elemento del modelo.
- Para cada restricción identificada construir un caso de prueba que valide que se cumpla la restricción luego de la transformación.

A modo de ejemplo y analizando la regla conceptual de la Figura 2.5 a) se observa que los elementos que modifica esta regla son el conjunto de pins de A, porque se agrega un pin adicional, y las aristas, porque se modifican los nodos de origen o nodos destino de las mismas.

Consideremos en el modelo de proceso de negocio una restricción CA1: Un nodo final tiene una sola arista de entrada. Entonces se construye un caso de prueba tal que luego de la transformación de como resultado un nodo final con dos aristas de entrada. El caso se muestra en la siguiente Figura 2.6.

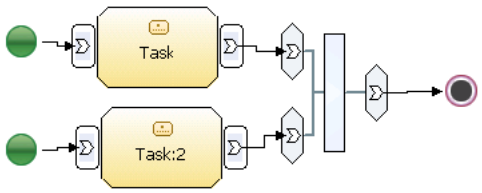


Figura 2.6 - Caso de prueba para restricciones.

Una implementación incorrecta trataría de reconectar las dos aristas de entrada al nodo final. Este chequeo de restricciones puede servir como complemento del Testing de cubrimiento del metamodelo descrito anteriormente.

Por último se menciona la técnica llamada “pares de reglas”. Esta técnica consiste en generar casos de prueba tomando dos reglas de transformación diferentes, basándonos en el lado izquierdo de cada una de estas reglas, el objetivo es detectar los elementos del modelo de una regla que se puedan reemplazar por los elementos del modelo de la otra. Luego se construyen modelos del solapamiento de los dos elementos de modelos utilizados. Si este nuevo modelo es sintácticamente correcto se utiliza como caso de prueba, sino se descarta. Esta técnica es utilizada para intentar detectar errores de confluencia. La propiedad

de confluencia determina que la ejecución de reglas en diferente orden en modelos equivalentes debe dar el mismo resultado.

Veamos un ejemplo: Se considera la regla de la Figura 2.7.

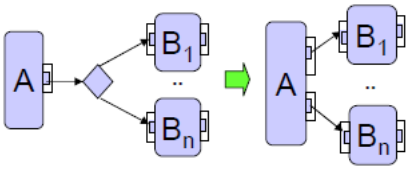


Figura 2.7 - Regla de eliminación de la decisión[4].

Los casos de prueba que se generan tomando en cuenta la regla de la Figura 2.5 a) y la regla de la Figura 2.7 se describen en la Figura 2.8.

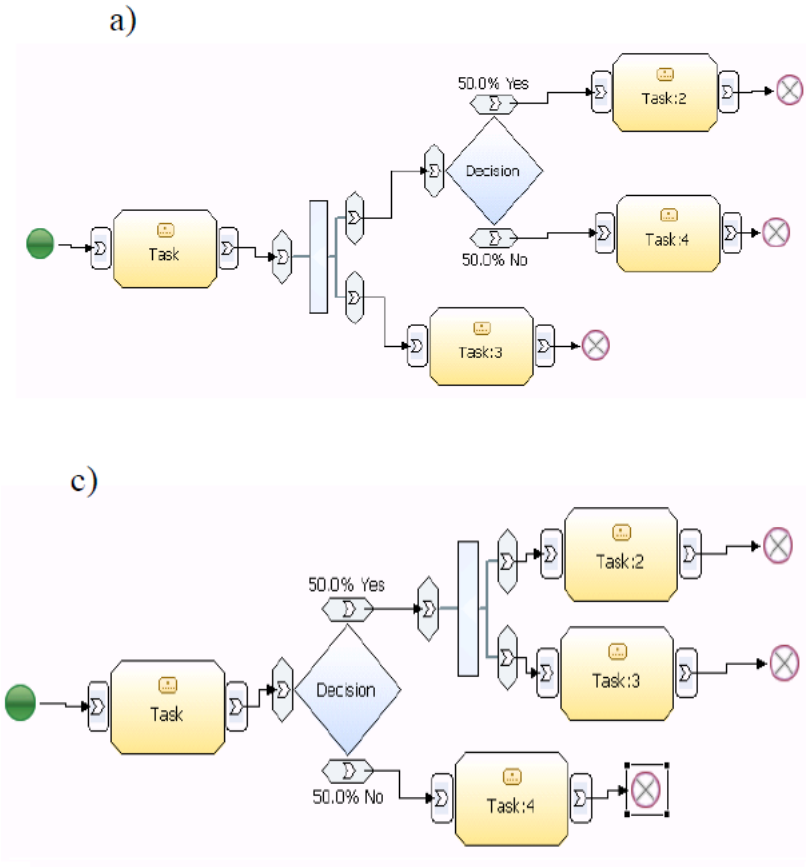


Figura 2.8 - Casos de prueba para confluencia.

Si se observa el caso de prueba de la Figura 2.8 c) da un error de confluencia, porque si se aplica primero la regla de la Figura 2.5 a) nos quedarían dos pins en la decisión, lo cual representa un modelo de proceso de negocios inválido. Sin embargo, si se aplica primero la regla de la Figura 2.7 entonces el caso de prueba no se transformaría en un modelo inválido.

2.4. Validación de Casos de Prueba

La validación de los casos de prueba se realizará en dos partes. La primera consiste en verificar los datos de entrada de la transformación. Esto implica que se debe corroborar que los modelos de prueba conformen con el metamodelo origen y cumplan con las restricciones tanto estructurales como no estructurales que puedan existir.

La segunda parte evalúa la efectividad de los casos de prueba generados. Esto se realiza mediante un análisis de mutación. Para llevar a cabo este análisis se generan versiones de la transformación que contienen defectos, llamados mutantes, y se ejecuta el conjunto de casos de pruebas sobre los mutantes verificando el porcentaje de defectos encontrados [26]. En el artículo se busca que los defectos inyectados sean realistas y efectivos por lo tanto se plantea la utilización de un operador de mutación para crear los programas mutantes. Para la generación del operador de mutación el autor se basa en fallas semánticas que pueden ocurrir en la implementación de una transformación. Se identifican cuatro operaciones abstractas que describen las actividades de un proceso de transformación de modelos:

1. Navegación o Recorrida del modelo
2. Filtrado de los elementos del modelo
3. Creación del modelo de salida
4. Modificación del modelo de entrada: Cuando el modelo de salida de la transformación es una modificación del modelo de entrada.

Para ejemplificar se considera la transformación UML a RDBMS, expuesta en [27], ver Figura 2.9.

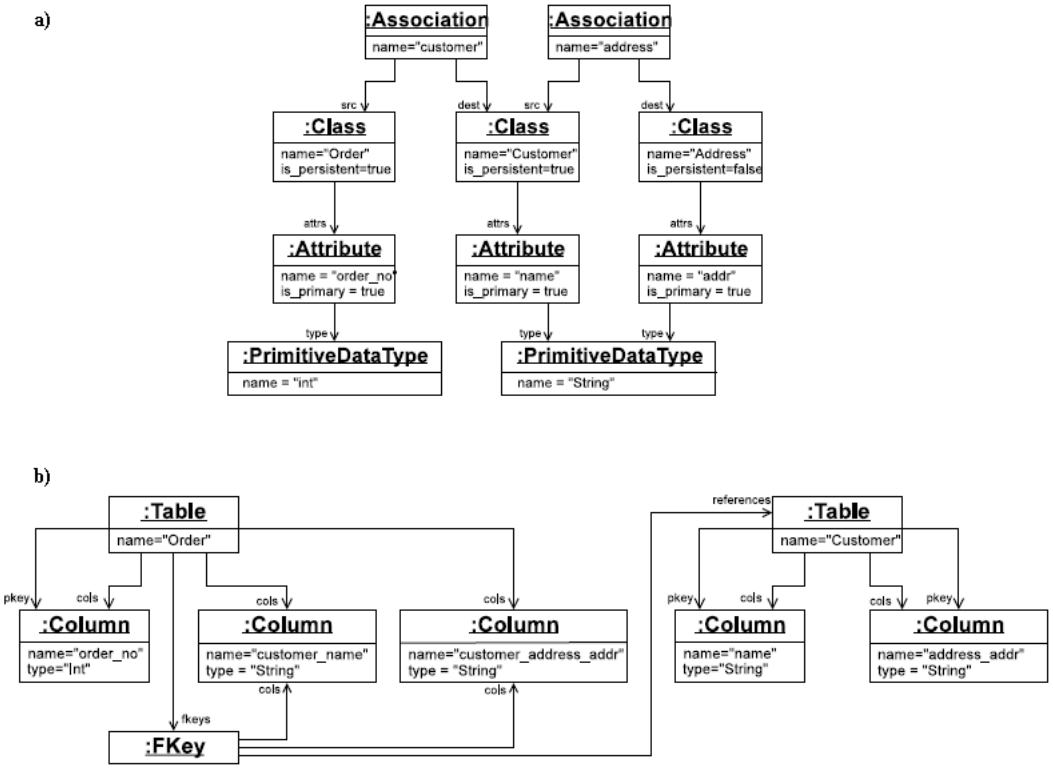


Figura 2.9 - Transformación de UML a RDBMS[27].

Esta transformación tiene como dato de entrada un diagrama de clases UML y transforma las clases persistentes en tablas con el mismo nombre que conformarán la base de datos. Primero se recorre el modelo de entrada para encontrar las clases, que luego se filtran para seleccionar solamente las persistentes. Se crea una tabla por cada clase persistente obtenida.

En base a estas operaciones se definen operadores de mutación que servirán para crear los mutantes. Estos operadores de mutación deben ser implementados en el mismo lenguaje que este implementada la transformación.

Luego de ejecutados los casos de prueba en el programa original (P en la figura 2.10) y en los mutantes se comparan las salidas. Si los resultados son diferentes entonces el caso de prueba pudo detectar la falla introducida en el mutante. Por el contrario, si las salidas son iguales se tienen dos posibilidades. La primera que el mutante sea equivalente al programa original y por lo tanto sea descartado y la segunda que el caso de prueba actual no detecte la falla introducida, con lo cual se mantiene el mutante para pruebas posteriores.

En la Figura 2.10 se muestra a grandes rasgos en que consiste el proceso de análisis de mutación antes descrito.

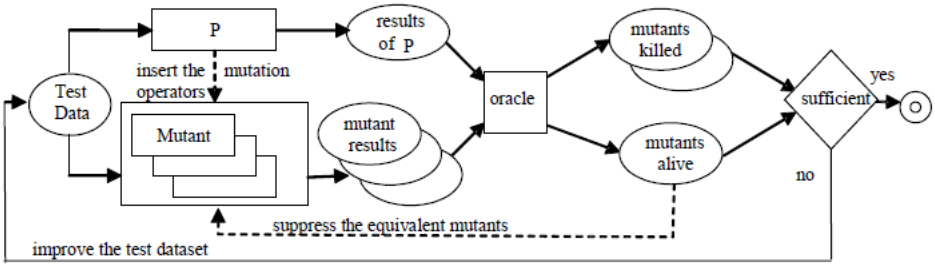


Figura 2.10: Proceso del análisis de mutación[26].

El valor del análisis de mutación se basa en asumir que si un conjunto de casos de prueba logra diferenciar al programa original de todos los mutantes entonces será capaz de detectar errores involuntarios.

2.5. Validación de Resultados

La validación de resultados se realizará en dos partes. La primera parte trata de verificar la corrección de los datos de salida de la transformación. La segunda parte analiza la semántica de la transformación verificando la coherencia entre modelos de entrada y salida.

Los datos de salida de la transformación son modelos por lo tanto estos deben conformar con el metamodelo destino de la transformación, además deben cumplir con las restricciones tanto estructurales como no estructurales.

Para verificar la semántica de la transformación se deben corroborar en los modelos de salida un gran número de propiedades, es por esto que es necesaria la generación de un oráculo adecuado a la transformación. El oráculo analiza la validez de un modelo de salida retornando un veredicto. Se puede consultar [28] para ver la discusión sobre los problemas que pueden presentarse al definir un oráculo.

En estudios anteriores reflejados en [29] y [30] se supone conocido el modelo esperado para un caso particular de ejecución de la transformación, por lo tanto el problema de la definición del oráculo pasa a ser el problema de la comparación de dos modelos: el modelo esperado con respecto al modelo de salida de la transformación.

El autor de [28] opina que esta suposición es restrictiva ya que el modelo esperado puede no ser fácil de obtener, por eso plantea la definición del oráculo como una función (función oráculo) con dos parámetros en donde el primero de ellos es el modelo de salida que resulta de aplicar la transformación al caso de prueba y el segundo parámetro que el autor llama "oracle data" es provisto por la persona que realiza el test de la transformación. Eventualmente este último podría ser el modelo esperado para el caso de prueba o el propio caso de prueba ya que podría ser necesario extraer información del mismo para verificar el modelo de salida.

A continuación describiremos tres técnicas que manipulan y analizan modelos, estas pueden ser utilizadas para implementar la función oráculo mencionada anteriormente. La primer técnica tiene el nombre de comparación de modelos, es común que los modelos sean representados mediantes grafos de objetos, en estos casos la técnica de comparación de modelos es equivalente al isomorfismo de grafos que es NP-Completo. Sin embargo, se han propuestos varias aproximaciones que tienen un costo computacional menor y se pueden ver en detalle en los trabajos [31] y [30]. En [32] se analiza la herramienta EMFCompare que implementa la comparación de modelos.

La segunda técnica consiste en la utilización de contratos [33] que especifican las condiciones que se tienen que cumplir antes y después de la ejecución de un método de la transformación. Las pre-condiciones del contrato restringen los modelos válidos de entrada y las post-condiciones declaran un conjunto de propiedades que se espera que se cumplan en el modelo de salida. Se puede entonces tomar esas post-condiciones para implementar una función oráculo, en [14] se puede ver el proceso de especificación e implementación de un oráculo para transformaciones de modelos basado en contratos expresados en OCL [34].

Por último se describe la técnica de reconocimiento de patrones, esta consiste en identificar la presencia de un patrón en un modelo. Se define un patrón como un conjunto de elementos de un modelo. Es utilizado en los oráculos para determinar que ciertos elementos se encuentren en el modelo de salida. El autor presenta dos maneras de expresar estos patrones: con aserciones, o con *"model snippets"*. Las aserciones las expresa en el lenguaje OCL, indicando los objetos que deben estar presentes y las condiciones que se deben cumplir en el modelo de salida para el caso de prueba específico.

Un *"model snippet"* o *"snippet"* es un subconjunto de elementos de un modelo donde cada elemento es instancia de una metaclase definida en el metamodelo. Para el oráculo los patrones expresan restricciones sobre el modelo de salida y podrían considerarse como post-condiciones pero con la salvedad de que los patrones se enfocan en un modelo específico de salida, o sea son restricciones sobre un modelo de salida para un caso de prueba en particular y no para toda la transformación.

2.5.1. Funciones Oráculo u Oráculos

En esta subsección analizaremos la creación de oráculos estudiada en [28] utilizando las técnicas mencionadas anteriormente.

- Oráculo usando una transformación de modelos de referencia.

Luego de aplicar la transformación que queremos probar y la transformación de referencia al caso de prueba, se utiliza la técnica de comparación de modelos para verificar que los resultados obtenidos por ambas transformaciones sean iguales. El autor señala que este oráculo tiene problemas al ser reutilizado, ya que cuando se realizan cambios en la especificación de la transformación se debe impactar también la transformación de referencia, lo cual puede llegar a ser una tarea muy compleja para el desarrollador.

- Oráculo usando una transformación inversa.

Consiste en realizar la comparación entre el caso de prueba ingresado a la transformación y el modelo obtenido luego de aplicar dos transformaciones: la primera vez se aplica la transformación que estamos probando y luego al resultado se aplica la transformación inversa.

El tester deberá proveer esta transformación inversa pero hay que tener en cuenta que la transformación debe ser una función inyectiva para que esto sea posible. El autor menciona que según la experiencia en general las transformaciones de modelos no son inyectivas por lo tanto esta aproximación no tendría buena aplicación. Además, en el caso de ser posible la creación de una transformación inversa, asegurar la confiabilidad de la misma es un problema más a solucionar.

- Oráculo usando el modelo de salida esperado.

Consiste en comparar el modelo salida de la transformación con el modelo de salida esperado provisto por el tester. Este tipo de oráculo implica un mayor esfuerzo cuando se cambia la especificación de la transformación ya que tendremos que actualizar los modelos esperados de salida por cada versión nueva de la transformación.

- Oráculo usando un contrato genérico

En este contexto se denomina contrato genérico a una post-condición de la transformación que restringe los modelos de salida posibles de acuerdo al caso de prueba provisto. El contrato genérico es capaz de analizar tanto el caso de prueba como su correspondiente modelo de salida, y validarlo.

El autor señala que cuando los contratos se vuelven muy complejos se vuelven difíciles de mantener y de expresar. Esto puede introducir errores en la especificación de los mismos, por lo tanto no recomienda la utilización de este tipo de oráculo cuando las transformaciones son muy complejas. No obstante, resulta útil cuando es necesaria una gran cantidad de casos de prueba, ya que utilizando un contrato genérico ahorraríamos esfuerzo y reutilizaríamos el mismo oráculo para diferentes casos de prueba.

En [35] se pueden ver algunas limitaciones de OCL como lenguaje para expresar contratos, como por ejemplo la complejidad para especificar restricciones sobre relaciones entre el modelo origen y el modelo destino.

- Oráculo usando una aserción OCL

Consiste en expresar las propiedades que debe contener el modelo de salida. Se chequea entonces si el modelo de salida satisface estas aserciones OCL. La aserción OCL es capaz de analizar sólo el modelo de salida. Los autores no recomiendan especificar en la aserción todas las propiedades a cumplir por el modelo de salida ya que eso se haría más fácilmente con el oráculo de modelo de salida esperado.

Este oráculo tiene la ventaja de poder reutilizarse fácilmente si cambia la especificación de la transformación. Pero como se aplica para validar modelos

de salida puntuales cuando se necesita un gran número de casos de prueba para realizar el testing entonces esta aproximación puede requerir mucho esfuerzo y riesgo de errores si la comparamos con técnicas como la del oráculo usando un contrato genérico.

- Oráculo usando *"model snippets"*

El oráculo chequea si el modelo de salida contiene una cantidad n de *"snippets"*. En este caso el tester debe proveer un patrón o lista de *"model snippets"* en la cual cada uno de estos *"model snippets"* tiene asociado una cardinalidad y un operador lógico.

Los autores plantean que estos *"snippets"* y sus oráculos son simples de escribir y modularizar así como también se consideran de fácil reuso ante eventuales cambios en la especificación de la transformación. Son apropiados para transformaciones complejas ya que no consideran todos los requerimientos de la transformación a la vez sino que se crean en base al resultado esperado de la aplicación de cada regla de la transformación.

3. Caso de Estudio.

Como ejemplo práctico se llevó a cabo la verificación de una transformación de modelos. La verificación consistió en la generación de casos de prueba siguiendo el procedimiento propuesto en [6] y se realizó la verificación teniendo en cuenta el cubrimiento del metamodelo de origen según los criterios propuestos en [8] utilizando la herramienta MetaModel Coverage Checker (MMCC) [7]. La transformación a verificar consiste en la traducción de metamodelos expresados en el lenguaje de metamodelado KM3[36] a una representación en COQ[37] que fue el caso de estudio desarrollado en [2]. KM3 es un lenguaje textual especializado para la especificación de metamodelos mientras que COQ es un asistente de pruebas que permite expresar proposiciones matemáticas, verificar la corrección de pruebas para estas proposiciones y asistir a la generación de pruebas formales. Además permite generar programas correctos a partir de una demostración constructiva de la especificación formal del programa.

La transformación es declarativa y fue escrita utilizando el lenguaje híbrido ATL[38], según lo visto en el capítulo 1 toma como modelo origen un metamodelo en KM3 que conforma con un metametamodelo KM3 y devuelve un modelo destino que consiste en un metamodelo expresado en COQ que conforma con un metametamodelo COQ.

Nuestro objetivo se centró entonces en realizar el testing de dicha transformación. La técnica empleada fue la de Caja Negra descrita en el punto 2.2 de este informe y se tomó como base para la generación de los casos de prueba el dominio de entrada de la transformación. El caso de estudio se concentró en mostrar y seguir paso a paso una metodología de trabajo que permitiera la construcción de los casos de prueba así como también la evaluación de los mismos según criterios previamente establecidos y la posterior verificación de la transformación y del resultado.

Como resultado de probar la transformación con el conjunto de casos de prueba generado se detectaron errores en la transformación y se puede asegurar que bajo los criterios establecidos se logra un 100 % de cubrimiento del dominio de entrada.

Este capítulo se organiza de la siguiente manera. En la sección 3.1 se brinda una descripción alto nivel del proceso de construcción de casos de prueba, en las secciones 3.2 y 3.3 se muestra como se aplicó el proceso para el caso concreto utilizando el algoritmo presente en el trabajo[6].

Las secciones restantes del capítulo muestran la construcción de casos de prueba, la herramienta MMCC utilizada para verificar el cubrimiento de los casos de prueba generados con respecto al metamodelo de entrada y los errores encontrados en la transformación.

3.1. Metodología de construcción de los casos de prueba

La metodología utilizada para la creación del conjunto de los casos de prueba fue la propuesta en [6]. Lo que nos llevó a la utilización de este algoritmo es que ya contábamos con el metamodelo de origen de la transformación claramente

especificado en [2] y este metamodelo es precisamente el principal parámetro de entrada del algoritmo. En la Figura 3.1 se puede observar el metamodelo KM3 de origen, en el recuadro se indican los elementos que fueron considerados para la transformación.

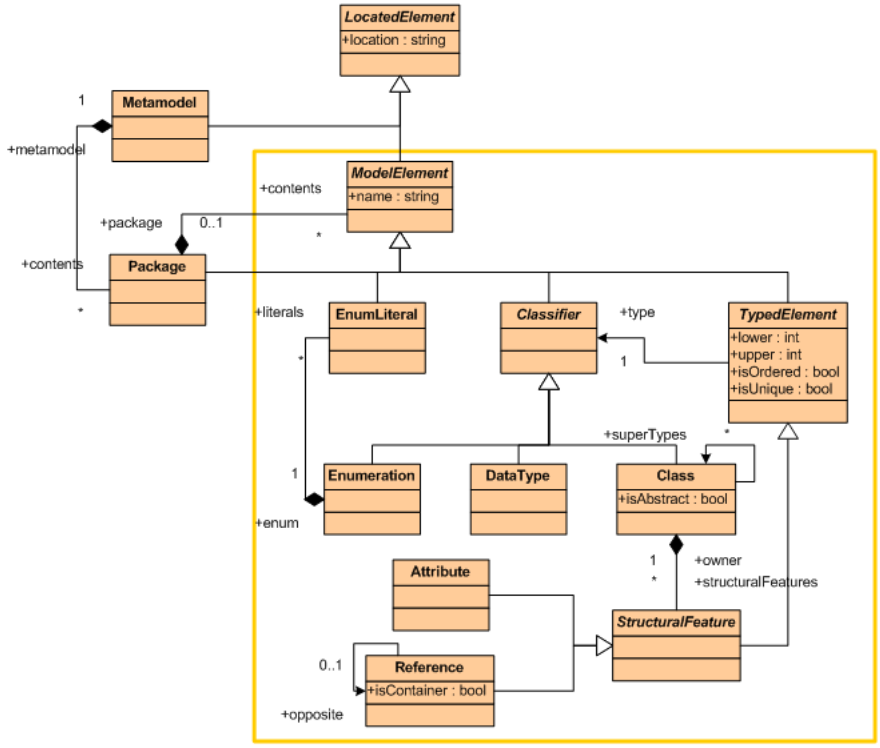


Figura 3.1: Metamodelo KM3 [2]

En [36] se puede ver la descripción del metamodelo KM3 así como también las restricciones no estructurales del mismo que nos serán de utilidad más adelante. En primer lugar mencionamos a grandes rasgos los 3 pasos del proceso de generación de casos de prueba. El primer paso consiste en descomponer en clases de equivalencia los dominios de todos los atributos simples (Boolean, Integer, String) y multiplicidades de asociaciones que aparecen en las clases del metamodelo de origen de la transformación, por más detalles se puede consultar el punto 2.2.1 de este documento.

El segundo paso consiste en construir fragmentos de modelos en base a las particiones obtenidas en el primer paso, para ver la definición de fragmento de modelo se puede referir al punto 2.2.1 de este documento.

El último paso del proceso consiste en generar modelos a partir de los fragmentos de modelos en base a estrategias preestablecidas.

En la siguiente sección se muestra la aplicación de cada uno de los pasos del proceso a la transformación en cuestión.

3.2. Aplicación del proceso de generación de casos de prueba

El resultado de hallar las particiones del metamodelo de origen KM3 según la técnica de particionado por defecto se muestra en el siguiente cuadro:

ModelElement::Name	{{""},{"N"},{*}}
EnumLiteral::enum	{1}
Enumeration::#literals	{{0},{1},{>1}}
TypedElement::type	{1}
TypedElement::lower	{{0},{1},{>1}}
TypedElement::upper	{{0},{1},{>1}}
TypedElement::isOrdered	{{true},{false}}
TypedElement::isUnique	{{true},{false}}
Class::isAbstract	{{true},{false}}
Class::#superTypes	{{0},{1},{>1}}
Class::#structuralFeatures	{{0},{1},{>1}}
StructuralFeature::owner	{1}
Reference::isContainer	{{true},{false}}
Reference::#opposite	{{0},{1}}

Figura 3.2: Particiones del Metamodelo KM3

En base a estas particiones especificamos tres conjuntos de cinco fragmentos de modelo cada uno que nos sirvieran como base para el último paso del proceso. En una situación ideal los fragmentos de modelo deberían ser deducidos automáticamente del metamodelo de origen tomando como base las particiones y un criterio de test dado, pero para esta aproximación los autores asumen que los fragmentos de modelo son seleccionados por el tester.

Dado que nuestro objetivo es lograr el cubrimiento del dominio de entrada de la transformación según ciertos criterios de test y que para lograrlo se utilizará la herramienta MMCC que se explica en el siguiente capítulo, podríamos elegir distintas configuraciones iniciales de fragmentos de modelo llegando en todos los casos a lograr el cubrimiento total del dominio de entrada. Sin embargo la cantidad de fragmentos de modelo que componen los conjuntos y la cantidad de fragmentos de objeto en cada fragmento de modelo son dos factores que influyen en el tamaño de los casos de prueba generados y en la complejidad de construcción de los mismos.

Una posible forma de medir que tan eficiente es nuestra configuración inicial sería tener información estadística sobre el cubrimiento de nuestro dominio de entrada con los conjuntos elegidos, así como también del tiempo invertido en la creación de los casos de prueba. Otra posible forma sería aplicar análisis de mutación sobre la transformación siguiendo lo propuesto en [26] para medir la calidad de los modelos generados con este conjunto de fragmentos de modelo como se propone en [6]. Esta evaluación no fue realizada, en primera instancia

porque no se contaba con información estadística pasada sobre otros conjuntos de casos de prueba para la transformación en cuestión y en segunda instancia porque quedaba fuera del alcance del objetivo propuesto inicialmente sobre lograr el cubrimiento del dominio de entrada.

A continuación mostramos los conjuntos de fragmentos de modelo elegidos especificados utilizando las particiones previamente definidas:

Conjunto 1	MF1:	{{Class::(isAbstract= true)}}
	MF2:	{{Enumeration::(#literals= 1)}}
	MF3:	{{TypedElement::(type=1)}}
	MF4:	{{Class::(#structuralFeatures= 2)},{ModelElement::(Name= "M1")}}
	MF5:	{{Reference::(isContainer=true)}}
Conjunto 2	MF1:	{Class::(isAbstract = false, #superTypes = 2)}
	MF2:	{Enumeration::(#literals = 3)}
	MF3:	{TypedElement::(lower=0, upper=300, isUnique=true)}
	MF4:	{StructuralFeature::(owner = 1)}
	MF5:	{Reference::(isContainer = false, #opposite=1)}
Conjunto 3	MF1:	{ModelElement::(name="N")}
	MF2:	{EnumLiteral::(enum=1)}
	MF3:	{Class::(#structuralFeatures=1)}
	MF4:	{Class::(#superTypes=1)}
	MF5:	{Reference::(#opposite=0)}

Figura 3.3: Conjunto de fragmentos de modelo.

Recordamos que un fragmento de modelo consiste en un conjunto de fragmentos de objeto y que un fragmento de objeto especifica una instancia parcial de una clase. A modo de ejemplo la siguiente figura muestra el diagrama de objetos que conforma con los fragmentos de modelo del Conjunto 3:

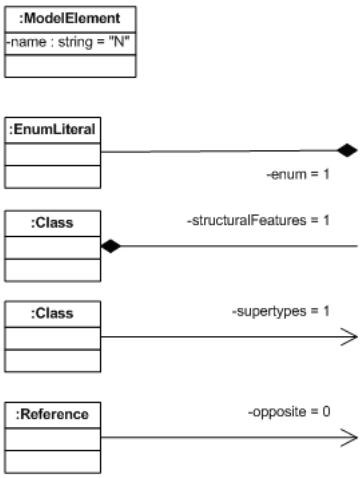


Figura 3.4: Objetos correspondientes al Conjunto 3.

La finalidad del algoritmo de generación de modelos es completar esta vista parcial de un modelo (Figura 3.4) para que conforme con el metamodelo de origen de KM3. A continuación presentamos el algoritmo utilizado junto con los puntos de variación y estrategias posibles, luego se muestran las estrategias utilizadas para la generación de los casos de prueba y se expone la creación de los casos de prueba correspondientes al conjunto 1 de fragmentos de modelo.

3.3. Algoritmo de generación de modelos

La porción de pseudo código que se muestra a continuación corresponde al algoritmo para la generación de modelos [6].

```
1:entrada: conjunto de fragmentos de modelo S, meta-modelo MM.
2:salida: conjunto de modelos L que conforman con MM
3:mientras haya fragmentos de modelo sin cubrir en S hacer
4:{ crear un modelo vacío M
5:  mientras el límite para el tamaño de M no se alcance(1) y M todavía pueda crecer hacer
6:  { elegir un fragmento de modelo MF sin cubrir en S
7:    para cada fragmento de objeto OF en MF hacer
8:    { encontrar un objeto O que sea instancia de la clase parcialmente especificada por OF(3)
9:      para cada restricción R definida en OF sobre la propiedad P hacer
10:     { si P es un atributo entonces
11:         seleccionar un valor y setear P en O (5)
12:       sino (P es multiplicidad)
13:       { seleccionar una cardinalidad N de acuerdo a R (5)
14:         si el tipo de P es una clase entonces
15:           encontrar N objetos con tipo P y setearlos a P en O(3)
16:         sino encontrar N valores en la partición P y setearlos a P en O(5)
17:       }}
18:       agregar O a M
19:         completar M hasta que conforme con MM (2,4)
20:   }}
21:   marcar MF como cubierto
22:   agregar el modelo M a L}
```

Como se puede observar el algoritmo propuesto por los autores presenta algunos puntos que se representan mediante un número encerrado entre paréntesis que permiten optar por distintas estrategias al momento de la construcción de los modelos. Estos puntos de variación tienen fuertes efectos en los modelos resultantes como se verá a continuación.

	Puntos de Variación	Estrategias Provistas
1	Tamaño de los modelos	máximo número de objetos mínimo número de fragmentos de modelo
2	Completar el modelo	sin criterio camino
3	Elección de objetos (durante el cubrimiento del fragmento de modelo)	crear siempre reusar siempre que sea posible nuevo objeto por cada OF
4	Elección de objetos (mientras se completa el modelo)	crear siempre reusar siempre que sea posible
5	Elección de clase de equivalencia en partición y elección del valor dentro de la clase de equivalencia	por defecto límites cíclica

Figura 3.5: Puntos de variación y estrategias posibles [6]

El primer punto de variación permite tener control sobre el tamaño de los modelos que se generan, en cada iteración del algoritmo se incluye un nuevo fragmento de modelo y el modelo(caso de prueba) crece hasta que llega a un límite específico o hasta que no pueda hacerse más grande. La estrategia de "máximo número de objetos" establece una cota superior para la cantidad de instancias de clases que puede contener el modelo. Por otro lado la estrategia de "mínimo número de fragmentos de modelo" expresa una cota inferior para la cantidad de fragmentos de modelo que se deben cubrir. Después de cubrir cada fragmento de objeto del fragmento de modelo se aplica un proceso mediante el cual el modelo debe completarse para conformar con el metamodelo MM, las estrategias posibles son las del punto 2.

Cuando el algoritmo tiene que elegir valores para las propiedades que se especifican en el fragmento de objeto (líneas 9 a 18 del algoritmo) se debe elegir un valor que pertenezca a la clase de equivalencia en cuestión, para ello se pueden utilizar cualquiera de las estrategias del punto 5. La estrategia "por defecto" elige un valor aleatorio dentro de la clase de equivalencia. La estrategia "límites" selecciona los valores de los límites de la clase de equivalencia y la estrategia "cíclica" selecciona una a una las clases de equivalencia y luego elige un valor aleatorio dentro de la misma.

Por último los puntos de variación 3 y 4 muestran estrategias para determinar si las instancias que se necesitan para cubrir los fragmentos de modelo o para completar el modelo deben ser creadas o se deben reutilizar las ya existentes. En la sección siguiente veremos la aplicación del algoritmo para la construcción de un caso de prueba y se explicarán las decisiones tomadas y las estrategias utilizadas en cada caso.

3.4. Construcción de casos de prueba

Para poder ver claramente las diferencias entre la aplicación de ciertas estrategias en los puntos de variación del cuadro 3.5 se optó por variar las estrategias de los puntos 3 y 4 que son las que tienen mayor impacto en el tamaño de los modelos construidos. Para el resto de los puntos de variación se estableció la siguiente configuración:

- Para el punto de variación 1 se eligió la estrategia "mínimo número de fragmentos de modelo" y se estableció el valor en 5.
- Para el punto de variación 2 se eligió la estrategia "sin criterio predeterminado".
- Para el punto de variación 5 se eligió la estrategia "por defecto".

En el ejemplo que se expone a continuación se trabajó con el primer conjunto de fragmentos de modelo del cuadro 3.3, en cada una de las tres subsecciones siguientes se aplican las estrategias "crear siempre", "reutilizar siempre que sea posible" y "nuevo objeto por cada OF" respectivamente. Para los restantes conjuntos de fragmentos del cuadro 3.3 se aplicaron las mismas estrategias que para el conjunto 1 y se puede consultar el resto de los casos de prueba construidos en el Anexo C.

3.4.1. Estrategia "crear siempre"

En la figura 3.6 se ilustra el resultado de la aplicación del algoritmo con los puntos de variación 3 y 4 aplicando la estrategia "crear siempre" lo cual significa que tanto para cubrir los fragmentos de modelo del conjunto como para completar el modelo, no se pueden reutilizar objetos ya existentes. Se espera como resultado un modelo con mayor cantidad de objetos y menos complejo con respecto a los modelos correspondientes a las restantes estrategias.

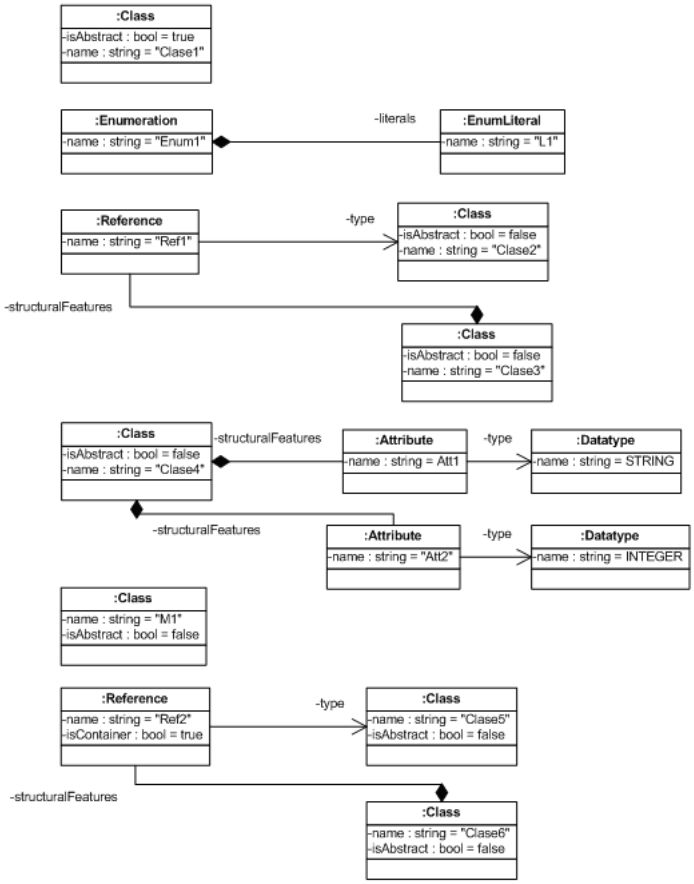


Figura 3.6: Estrategia "crear siempre", fragmentos del conjunto 1.

En la figura 3.6 se diferencian claramente las estructuras necesarias para cubrir los fragmentos de modelo del conjunto 1, explicaremos detalladamente como se logran cubrir los dos primeros fragmentos de modelo.

El primer fragmento de modelo (MF1) del conjunto 1 contiene un solo fragmento de objeto que condiciona a que exista una instancia de *Class* con la propiedad *isAbstract* tomando el valor *true*. Se debe agregar el nombre de la clase para que conforme con el metamodelo.

El fragmento de modelo MF2 contiene un solo fragmento de objeto que condiciona a que exista una *asociación* con nombre *literals* desde una instancia de la clase *Enumeration*. El algoritmo indica que se debe encontrar un objeto O que sea instancia de la clase parcialmente especificada por el fragmento de objeto y nuestra estrategia elegida implica que se debe crear siempre la instancia, por lo tanto se crea la clase *Enumeration* y la *asociación* de nombre *literals* con cardinalidad 1. Luego se debe incluir una instancia de la clase *EnumLiteral* para completar la asociación.

Se repite este mismo procedimiento con los restantes fragmentos de modelo del conjunto hasta que el caso de prueba cubre al menos 5 de ellos, entonces se puede dar por terminada la iteración.

3.4.2. Estrategia "reutilizar siempre que sea posible"

En la figura 3.7 se ilustra el resultado de la aplicación del algoritmo con los puntos de variación 3 y 4 aplicando la estrategia "reutilizar siempre que sea posible" lo cual significa que tanto para cubrir los fragmentos de modelo del conjunto como para completar el modelo se deben reutilizar objetos ya creados. Se espera como resultado un modelo con muy pocos objetos y más complejo con respecto a los modelos correspondientes a las restantes estrategias.

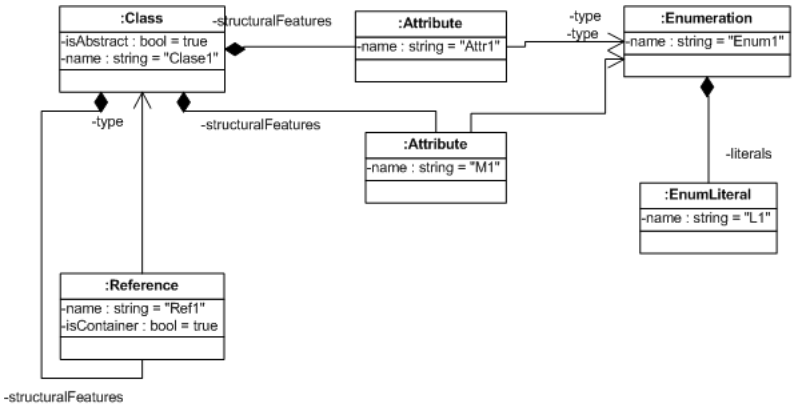


Figura 3.7: Estrategia "reutilizar siempre que sea posible", fragmentos del conjunto 1.

Una clara diferencia que se observa con la aplicación de esta estrategia se muestra en los objetos necesarios para cubrir los fragmentos de modelo MF3 y MF5.

El MF3 puede ser una instancia de *Attribute* o de *Reference*, en esta ocasión se optó por *Attribute*.

Para poder conformar con el metamodelo la instancia recién creada de *Attribute* debe tener un nombre, un tipo (*type*) y debe pertenecer a una clase.

A causa de la estrategia de "reutilizar siempre que sea posible", la instancia de *Attribute* debe pertenecer a la *Class* "Clase1" que se creó en la primer iteración y debe tener un tipo *Enumeration* "Enum1" que se creó en la segunda iteración.

La última iteración determina la creación de una instancia de *Reference* con el atributo *isContainer* en *true*. Se procede de igual forma que en el caso anterior reutilizando la *Class* "Clase1" como origen y destino de la referencia.

La iteración se termina cuando se cubren 5 fragmentos de modelo.

3.4.3. Estrategia "nuevo objeto por cada fragmento de objeto(OF)"

En la figura 3.8 se ilustra el resultado de la aplicación del algoritmo con el punto de variación 3 aplicando la estrategia "nuevo objeto por cada OF" y con el punto 4 aplicando la estrategia "reutilizar siempre que sea posible". Esta estrategia está entre medio de las dos estrategias previamente presentadas.

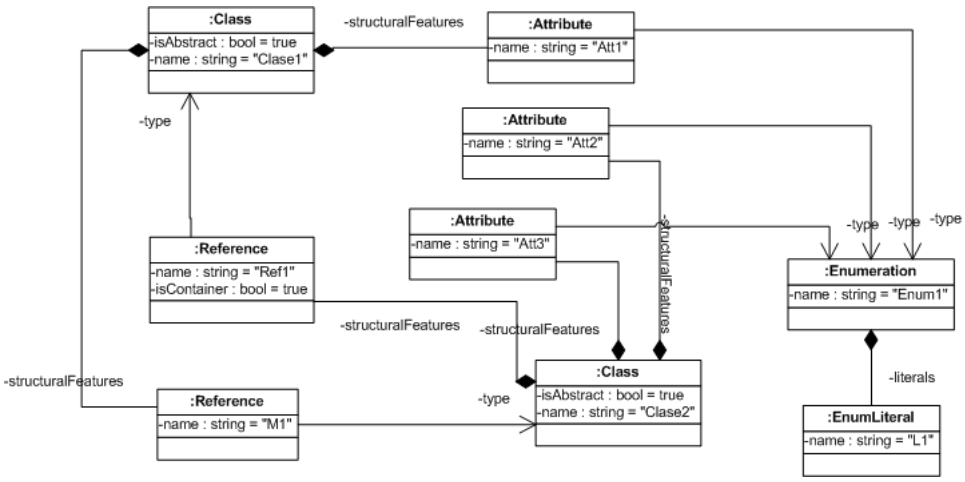


Figura 3.8: Estrategia "nuevo objeto por cada OF", fragmentos del conjunto 1.

En este caso explicaremos la iteración que corresponde al fragmento de modelo MF4 que contiene dos fragmentos de objeto.

La iteración determina que según el primer fragmento de objeto nuestro modelo debe contener una clase con dos *StructuralFeatures*, debido a la estrategia de "nuevo objeto por cada fragmento de objeto" creamos una nueva instancia de *Class* de nombre "Clase2" y dos nuevas instancias de *Attribute* de nombre "Att2" y "Att3" respectivamente. Para conformar con el metamodelo se debe asignar un tipo a los atributos, las dos posibilidades serían la clase "Clase1" y el enumerado "Enum1", en este caso optamos por el enumerado para ambos atributos.

Para el segundo fragmento de objeto de MF4 se optó por la creación de una *Reference* con nombre "M1" y para completar el modelo se reutilizaron las dos clases existentes una como propietaria de la referencia y la otra como tipo de la referencia.

La iteración al igual que en los casos anteriores se termina cuando se cubren 5 fragmentos de modelo.

3.5. MMCC (Metamodel Coverage Checker)

Luego de generar el conjunto de casos de prueba nos planteamos la necesidad de verificar el cubrimiento de los mismos con respecto al metamodelo de origen de la transformación y también poder identificar las partes del metamodelo que nos quedaron sin cubrir. Para ello nos basamos en el trabajo [8] que plantea el uso de la herramienta MMCC (Meta-Model Coverage Checker, [7]) con esa finalidad. Cabe mencionar que el poder medir que tan efectivo es un conjunto de casos de prueba es un importante desafío en el contexto de MDD como se señala en [12].

Lo primero que los autores establecen es el criterio para poder clasificar el conjunto de casos de prueba bajo la técnica de "caja negra", argumentando que el metamodelo de origen de la transformación es una especificación completa del conjunto de posibles modelos de entrada para la transformación. El criterio consiste en evaluar el conjunto de casos de prueba con respecto al cubrimiento del metamodelo de origen.

Los autores proponen que un conjunto de casos de prueba debe al menos cumplir con los siguientes tres puntos:

Cubrimiento de clases: Cada clase concreta del metamodelo debe ser instanciada en al menos un caso de prueba.

Cubrimiento de atributos: Cada atributo del metamodelo debe instanciarse con un conjunto de valores que cubran el dominio de dicho atributo.

Cubrimiento de asociaciones: Cada asociación del metamodelo debe instanciarse con un conjunto representativo de multiplicidades.

El proceso propuesto por los autores se ilustra en la figura 3.9 y tiene dos entradas: el metamodelo de entrada de la transformación y un conjunto de casos de prueba previamente generado.

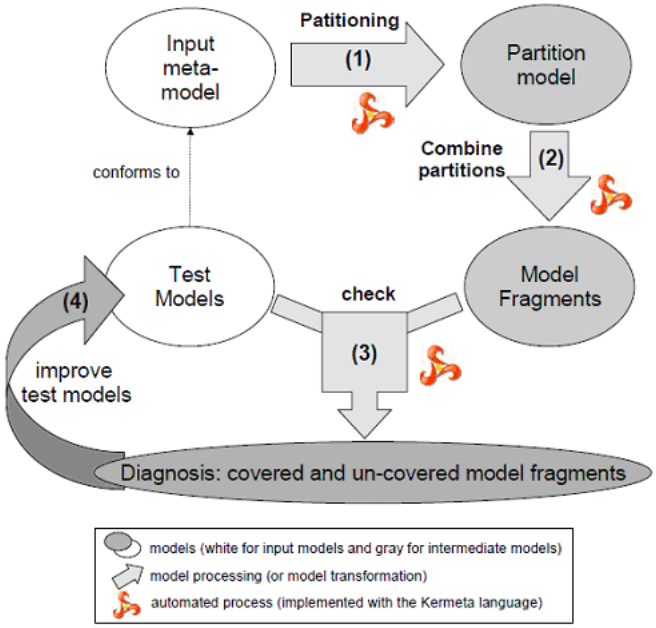


Figura 3.9: Proceso propuesto por MMCC para validar y mejorar la calidad de los casos de prueba [8]

En las siguientes subsecciones se explicarán detalladamente cada uno de los pasos del proceso.

3.5.1. Generación de particiones (1)

El primer paso del proceso consiste en generar las particiones por defecto para todos los atributos y asociaciones presentes en el metamodelo de origen.

Previo a la utilización de la herramienta se tuvo que realizar el pasaje del metamodelo KM3 (metamodelo origen de la transformación) al formato del meta-metamodelo Ecore [39]para que conforme con este.

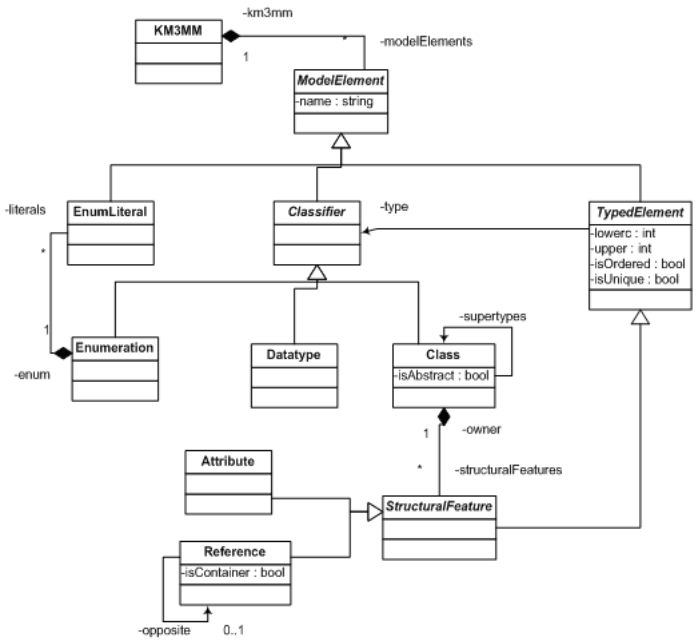


Figura 3.10: Metamodelo KM3 que conforma con el metametamodelo Ecore.

En esta sección ilustramos el uso de MMCC para mejorar nuestro conjunto de casos de prueba inicial que construimos en la sección 3.4. Como MMCC está implementado en EMF todas las clases del metamodelo KM3 deben estar contenidas en una clase "raíz", en este caso como se ve en la figura 3.10 se agregó la clase KM3MM que no estaba presente en el metamodelo original de la figura 3.1.

3.5.2. Generación de fragmentos de modelo (2)

El segundo paso toma como parámetro de entrada un **criterio de test** que determina como se deben formar los fragmentos de modelo y genera los fragmentos de modelo que deben cubrirse.

Los criterios de test utilizados fueron *AllRanges* y *AllPartitions* que se obtuvieron de [8] y ya se encuentran implementados en la herramienta MMCC.

Recordemos de la sección 2 que una partición para una propiedad de tipo primitivo (integer, string, boolean) del metamodelo se define como un conjunto de rangos que no se solapan y que especifican los posibles valores para esa propiedad.

- El criterio de test *AllRanges* implica que cada uno de los rangos de cada una de las propiedades del metamodelo sea usado en un fragmento de modelo diferente.

- El criterio de test *AllPartitions* implica que todos los rangos de una partición estén presentes en un mismo fragmento de modelo, esto se aplica para todas las propiedades.

3.5.3. Evaluación de casos de prueba (3)

El tercer paso verifica que haya al menos un caso de prueba que cubra cada fragmento de modelo generado en el paso anterior. Si no hay fragmentos de modelo sin cubrir entonces el conjunto de casos de prueba satisface el criterio de test elegido, de lo contrario se deben examinar los fragmentos de modelo que reporta la herramienta y es responsabilidad del tester mejorar el conjunto de casos de prueba inicial para lograr cubrir los fragmentos faltantes o justificar porque no pueden cubrirse, por ejemplo debido a restricciones estructurales.

En conclusión el proceso no solo permite medir la calidad de los casos de prueba sino que brinda información valiosa para poder mejorar el conjunto de casos de prueba.

Para ilustrar la utilización de MMCC en la siguiente sección mostraremos los resultados obtenidos para nuestro caso de estudio.

3.5.4. Utilización de MMCC para la transformación KM3 – COQ

Comenzamos con el criterio de test más débil *AllRanges*, cuando se utiliza MMCC con el metamodelo de la figura 3.10 y el criterio *AllRanges* la herramienta genera 33 fragmentos de modelo. Luego se pasaron los 9 casos de prueba construidos y la herramienta MMCC se encarga de evaluar el cubrimiento de los 33 fragmentos anteriores.

El resultado obtenido fue de 25 fragmentos de modelo cubiertos.

La herramienta también muestra los 8 fragmentos que quedaron sin cubrir:

a) MF{KM3MM::(#modelElements = 1)}
b) MF{Enumeration::(#literals = 0)}
c) MF{TypedElement::(isOrdered = false)}
d) MF{TypedElement::(isOrdered = true)}
e) MF{TypedElement::(isUnique = false)}
f) MF{ModelElement::(name = “”)}
g) MF{TypedElement::(upper = 0)}
h) MF{KM3MM::(#modelElements = 0)}

Para satisfacer el criterio de test *AllRanges* es necesario agregar nuevos casos de prueba al conjunto inicial para que cubran estos fragmentos.

Para cubrir el fragmento de modelo indicado en el punto a) se agregó un modelo con un único elemento, ver figura 3.11. Los fragmentos indicados en los puntos b) a e) se cubrieron agregando un modelo que los incluyera a todos como puede verse en la figura 3.12.

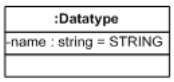


Figura 3.11: Modelo creado para cubrir el punto a).

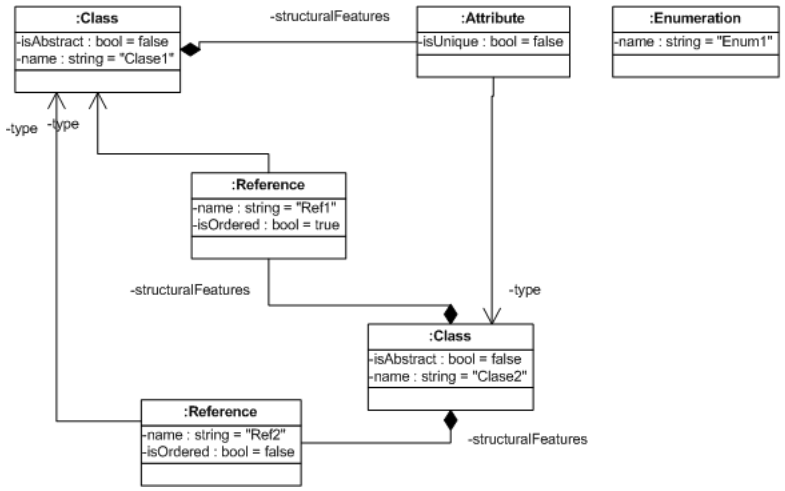


Figura 3.12: Modelo creado para cubrir los puntos b) a e).

Los fragmentos indicados en los puntos f) y g) no pueden cubrirse debido a las restricciones estructurales que posee KM3, no se puede definir un elemento del metamodelo que no tenga nombre y el atributo *upper* debe ser mayor o igual a 1, ver Manual KM3[36]. Finalmente el fragmento del punto h) indica el modelo vacío. Luego de agregar estos tres casos de prueba al conjunto inicial (los dos que se presentan en las figuras 3.11 y 3.12 más el modelo vacío) el criterio se considera satisfecho.

Al ejecutar MMCC con el criterio *AllPartitions*, para el metamodelo KM3 de la figura 3.10 la herramienta genera 15 fragmentos de modelo. Luego se tomaron los 9 casos de prueba iniciales más los 3 que se agregaron para satisfacer el criterio de test *AllRanges*. El resultado da que de los 15 fragmentos de modelo se cubrieron 8. La herramienta también expone los 7 fragmentos sin cubrir:

a) MF{Enumeration::(#literals = 0, #literals = 1, #literals > 1)}

b) MF{TypedElement::(isUnique = false, isUnique = true)}

c) MF{Reference::(isContainer = false, isContainer = true)}

d) MF{Class::(#supertypes = 0, #supertypes = 1, #supertypes > 1)}

e) MF{ModelElement::(name = “”, name = .+)}

f) MF{TypedElement::(upper = 0, upper = 1, upper > 1)}

g) MF{KM3MM::(#modelElements = 0, #modelElements = 1, #modelElements > 1)}

De igual manera a como se procedió para satisfacer el criterio de test de *AllRanges* debemos agregar al conjunto nuevos modelos que cubran los fragmentos faltantes.

Se optó por construir un solo modelo que debía contener los fragmentos especificados en los puntos a) a d) que se muestra en la figura 3.13.

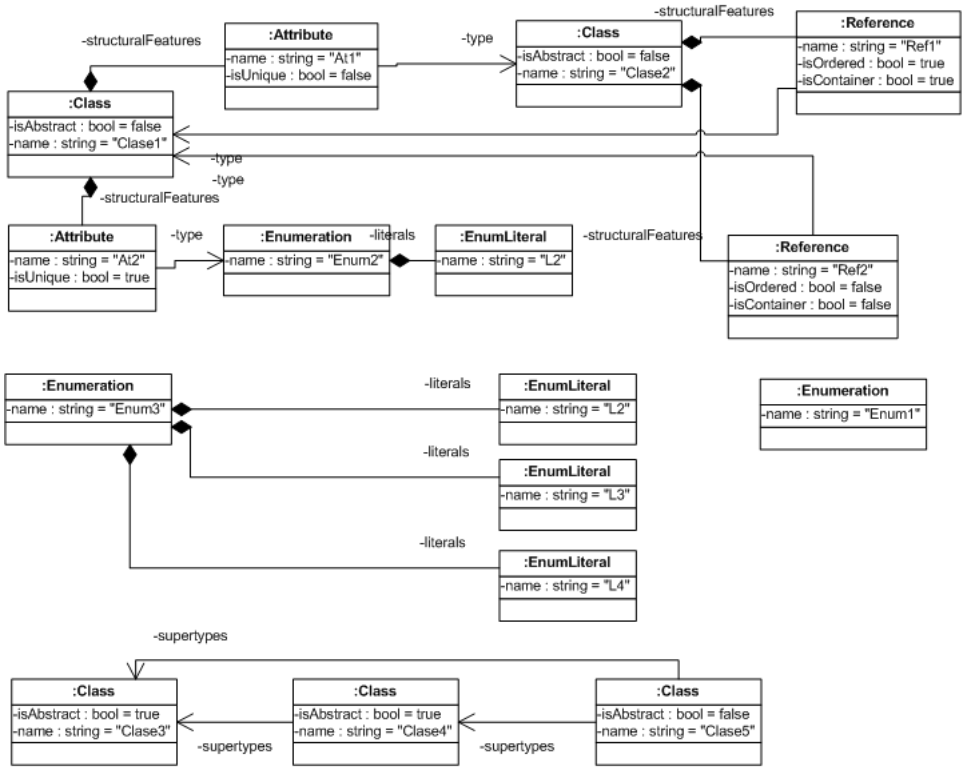


Figura 3.13: Modelo creado para cubrir los puntos a) a d).

Con respecto a los fragmentos de modelo de los puntos e) y f) cabe la misma explicación que para el caso anterior, por restricciones estructurales de KM3 no se puede construir el modelo.

El fragmento de modelo especificado en el punto g) claramente no se puede construir ya que implicaría tener un modelo vacío, un modelo que tenga solamente un elemento y un modelo que tenga más de un elemento, todas estas condiciones en un mismo modelo. Finalmente al agregar el modelo de la figura 3.13 al conjunto se logra satisfacer el criterio de test *AllPartitions* quedando los 3 fragmentos anteriores sin cubrir.

Como se explicó para el fragmento de modelo g) y como señalan los autores en [8] existen fragmentos de modelo que es imposible cubrirlos y es una limitante presente en la mayoría de los criterios de test: se generan restricciones que no pueden ser satisfechas lo cual no impide que sean útiles para mejorar los conjuntos de casos de prueba iniciales.

3.6. Errores encontrados en la transformación.

En esta sección se comentan los errores encontrados al ejecutar la transformación de modelos KM3 a COQ estudiada en [2].

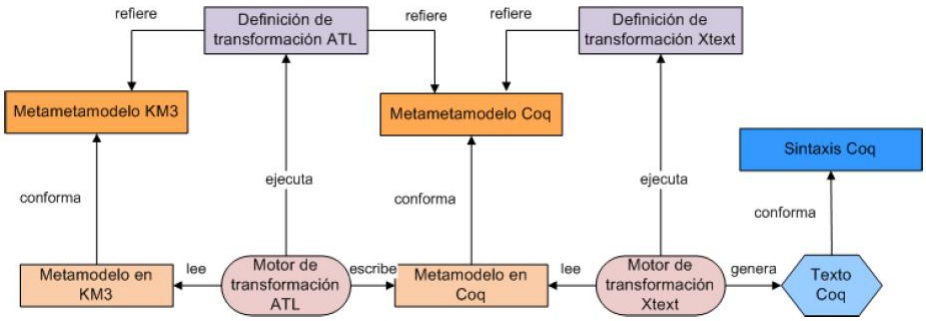


Figura 3.14: Transformación Km3Coq.

En el artículo se observa que la transformación KM3 a COQ se realiza en dos partes. La primer parte, que de ahora en adelante llamaremos transformación uno, realiza una transformación modelo-modelo, esta transformación escrita en lenguaje ATL [38] toma como entrada metamodelos escritos en lenguaje KM3 (modelos de entrada) y retorna metamodelos expresados con constructores COQ (modelos de salida).

La segunda parte, que de ahora en adelante llamaremos transformación dos, consiste en una transformación modelo-texto, escrita en lenguaje Xtext [40], que toma como modelos de entrada los modelos resultantes de la primer transformación y retorna una representación textual que concuerda con la sintaxis COQ.

Los errores encontrados fueron los siguientes:

- Error en la generación de clases abstractas sin atributos ni asociaciones.
- Error en la generación de procedimientos para comparar instancias de clases abstractas.
- Error en procedimientos "isTypeOfClase" cuando se tienen dos superclases para una clase.
- Errores en la importación de librerías.

A continuación detallaremos los errores mencionados anteriormente. El primer error se observó al ejecutar un modelo el cual contenía una clase abstracta sin atributos ni asociaciones.

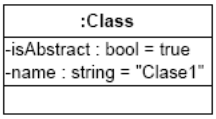


Figura 3.15: Clase Abstracta.

Cada clase expresada en KM3 se transforma a un tipo inductivo de COQ al realizar la transformación uno. Al definir estos tipos inductivos se declaran componentes para representar atributos y asociaciones. Según el metamodelo COQ los tipos inductivos pueden tener cero o más componentes.

La representación de la clase mencionada luego de la transformación uno es la siguiente :

<coq:InductiveType xmi:id="a1" name="Clase1" isAbstract="true"/>

Al ejecutar la transformación dos con el modelo de entrada que incluye esta línea se produjo el error. Al analizar el problema vimos que la transformación dos utiliza una gramática para generar un parser y un metamodelo origen, utilizando ambos para la generación de la representación textual en COQ. La definición de tipo inductivo en la gramática se encontraba como se muestra a continuación:

```
InductiveType:
    '<coq:InductiveType xmi:id=' name=ID 'name=' typeName=ID ('isAbstract='isAbstract='true')?
    'hasComponents=' (components+=[Component])+ ('subTypes=' (subtypes+=[InductiveType])+)?
    ('superTypes=' (supertypes+=[InductiveType])+)? '/>'
;
```

Figura 3.16: Gramática con error encontrado.

Como se observa la cantidad de componentes “hasComponents” referida en la gramática de la figura anterior es uno o mas.

La figura siguiente toma en cuenta que la cantidad de componentes “has-Components” pueda ser cero. Se modificó la gramática de la siguiente forma:

```
InductiveType:
    '<coq:InductiveType xmi:id=' name=ID 'name=' typeName=ID ('isAbstract='isAbstract='true')?
     ('hasComponents=' (components+=[Component])+)? ('subTypes=' (subtypes+=[InductiveType])+)?
      ('superTypes=' (supertypes+=[InductiveType])+)? '/>'
;
```

Figura 3.17: Gramática corregida.

Utilizando esta definición de la gramática la transformación dos resultó sin problemas tanto para los modelos conflictivos como para los modelos que anteriormente funcionaban de forma correcta.

Otro error que se encontró fue la generación de procedimientos para comparar instancias de una clase abstracta. La comparación de instancias de una clase se realiza utilizando el componente "oid" de tipo *Integer* que se genera para cada clase no abstracta. Al chequear la sintaxis en COQ y encontrar procedimientos que quieren comparar instancias de clases abstractas, las cuales no tienen definido el componente "oid" por no ser instanciables, se retorna un error. A continuación se muestra lo descrito anteriormente, la definición de la función que compara instancias de una clase abstracta, lo cual no es correcto de acuerdo a la especificación de las reglas de la transformación para clases abstractas. Esta definición no debe generarse en la salida COQ.

```
Definition beq_Clase1 (o1 : Clase1) (o2 : Clase1) : bool :=
        beq_nat (Clase1_oid o1) (Clase1_oid o2).
```

Figura 3.18: Procedimiento de comparación de instancias de clases abstractas.

Continuando con la validación de los modelos COQ de salida de la transformación dos encontramos un incidente que se obtuvo al tener en el modelo de entrada una clase con dos superclases. Al definir los métodos "isTypeOfClaseX", siendo ClaseX la subclase que extiende las dos superclases, se verificó que se definen con el mismo nombre, en consecuencia el compilador COQ devuelve un error.

Una opción posible para resolver este problema consiste en cambiar la firma de las funciones de la siguiente forma : "isTypeOfClaseX_1" y "isTypeOfClaseX_2", para cada una de las superclases. A continuación podemos observar la definición de ambos procedimientos nombrados de igual forma.

```
Definition isTypeOfClasel (a1 : Clase2) : bool :=
  match (Clase2_subClasel a1) with
    | None => false
    | Some a => true
  end.

Definition asTypeClasel (a1 : Clase2) : option Clasel := Clase2_subClasel a1.


Definition isTypeOfClasel (a2 : Clase3) : bool :=
  match (Clase3_subClasel a2) with
    | None => false
    | Some a => true
  end.

Definition asTypeClasel (a2 : Clase3) : option Clasel := Clase3_subClasel a2.
```

Figura 3.19: Error en definición de procedimientos al tener dos superclases.

Posteriormente se observaron todos los modelos transformados y verificados en COQ, y se encontró que la importación de las librerías se encuentra en una sola línea. En consecuencia se retorna un error por parte del compilador COQ, el mismo describe que no encuentra la librería String. La corrección de este error

es simple, se debe colocar cada importación en una nueva línea. A continuación se muestra una imagen con el error y una imagen con la corrección sobre el archivo de salida.

```
Require Import String.Require Import List.Require Import Arith.
```

Figura 3.20: Error en importación de librerías.

```
Require Import String.
Require Import List.
Require Import Arith.
```

Figura 3.21: Corrección de librerías importadas.

Por último se ejecutó el modelo de entrada que involucra todo el metamodelo KM3 [2], como complemento al caso de estudio, para observar el comportamiento de un ejemplo de mayor tamaño. Se observó que el chequeo en Coq contenía errores. Para las clases abstractas como "LocatedElement" y "ModelElement" se encontraron los problemas vistos anteriormente con la comparación de instancias de clases abstractas.

4. Conclusiones y Trabajo a Futuro.

El paradigma MDD(model-driven development) que se estudia desde hace una década aproximadamente tiene a las transformaciones de modelos como parte medular del mismo y contiene áreas como el testing que se encuentran en plena investigación. En particular, la complejidad de las transformaciones y de los modelos manejados, la diversidad de lenguajes de transformación y la falta de herramientas de gestión de modelos son aspectos que influyen negativamente en el testing de las transformaciones.

Diversos investigadores han realizado aportes que en su mayoría se basan en la adaptación de las técnicas de testing tradicionales, técnicas de caja negra y caja blanca, a las transformaciones de modelos. Durante el desarrollo del proyecto se elaboró un reporte técnico que brinda una visión inicial de las barreras y desafíos existentes en el área de testing de transformaciones. En dicho trabajo también se analizan los tres puntos principales para el testing de transformaciones de modelos: la generación de casos de prueba, la validación de casos de prueba y la validación de resultados.

Como aporte importante del reporte se constata que el interés principal de los investigadores está centrado en las técnicas de caja negra, las cuales resultan mas propicias debido a que existen diversos lenguajes de propósito general o específicos mediante los cuales pueden ser implementadas las transformaciones de modelos.

Nos interesa también resaltar que, en particular en el testing tradicional, la técnica de caja negra consiste en ver la especificación de un programa y la técnica de caja blanca consiste en ver la implementación (en cierto lenguaje). De acuerdo con la bibliografía de testing de transformaciones de modelos, la técnica de caja negra consiste en observar el metamodelo y sus restricciones (sería como ver solamente los datos y no la especificación en el testing tradicional). En tanto la técnica de caja blanca consiste en observar la especificación (es similar a la técnica de caja negra del testing tradicional). Podemos concluir que existe un nivel más de testing relacionado al motor que ejecuta la transformación, con lo cual la técnica de caja blanca en este contexto es una técnica de caja gris. Este nivel sí correspondería a la técnica de caja blanca tradicional en donde se considera la implementación de la transformación. Esto implicaría considerar aspectos del lenguaje de implementación tales como si es declarativo, operacional, basado en grafos, etc. A su vez sería necesario considerar características de la ejecución de esos lenguajes (sobre el motor de la transformación en el caso de lenguajes declarativos) ya que no es lo mismo la parte declarativa de ATL que la de QVT por ejemplo.

La última etapa del proyecto consistió en poner en práctica la técnica de testing de caja negra para la construcción de casos de prueba. La transformación objetivo seleccionada fue la implementada en el proyecto de grado [2] que toma como entrada un metamodelo en formato KM3 y devuelve el correspondiente metamodelo especificado en COQ.

En primera instancia se intentaron generar los casos de prueba de forma automática mediante la herramienta AGG[41]. Esta aproximación no resultó

debido a que el archivo que generaba las reglas de la gramática obtenido del sitio http://user.cs.tu-berlin.de/~gragra/agg/MM2GraGra/no funcionaba correctamente.

La aproximación tomada finalmente consistió en seguir paso a paso el algoritmo establecido en el trabajo [6] que implicó la construcción manual de un conjunto de casos de prueba con la finalidad de lograr cubrir el metamodelo de entrada , algo que pensamos que podria ser un criterio fuerte para validar una transformacíón.
Para asegurarnos que el conjunto de casos de prueba a utilizar fuera relevante lo evaluamos con la herramienta MMCC[7] descrita en [8] que además de verificar el cubrimiento de las propiedades del metamodelo de entrada nos brindó información adicional sobre los elementos sin cubrir necesarios para mejorar nuestro conjunto inicial de casos de prueba.

La idea de los criterios de test utilizados por MMCC, *AllRanges* y *AllPartitions* es poder construir casos de prueba más complejos y con ellos mejorar la calidad del testing, como trabajo a futuro se podrían utilizar criterios de test más exigentes como los propuestos en [8] que se basan en combinar de distintas formas los rangos de cada una de las propiedades del metamodelo de entrada.

Una limitación importante para automatizar el proceso de generacíón de casos de prueba es poder generar modelos que cumplan todas las restricciones impuestas sobre el metamodelo de origen tales como las reglas de "buena formación" y las precondiciones de la transformación.

Existen herramientas tales como OMOGEN [6], propiedad de Télecom Fránce que lo implementan, una línea de trabajo a futuro es lograr una herramienta similar a OMOGEN y que se complemente con el MMCC para lograr automatizar el proceso lo mayor posible.
La ejecución del conjunto de casos de prueba fue satisfactoria, desde el punto de vista del testing, revelando errores tanto en la primer transformación del prototipo como en el resultado esperado final.

Como trabajo a futuro se plantea la utilización de técnicas de caja blanca como puede ser verificar el cubrimiento de las reglas de la transformación como complemento de lo realizado, el estudio de técnicas de testing tradicionales que aún no se hayan aplicado al testing de transformaciones de modelos y explorar la aplicación de técnicas de testing de programas declarativos al testing de transformaciones de modelos, creadas con lenguajes declarativos.

Anexo A

Glosario

Modelo: Descripción o especificación de un sistema realizado con un lenguaje determinado. Abstracción semánticamente completa de un sistema. Representación abstracta de un sistema.
Meta-modelo: Definición precisa de los constructores y reglas necesarios para definir la semántica de los modelos.Define un lenguaje de modelado para modelos.
Meta-meta-modelo: Definición de un lenguaje de modelado para meta-modelos.
Meta-modelo efectivo: Conjunto o subconjunto de elementos del meta-modelo relevantes para realizar pruebas. Se construye utilizando los datos de las pre y post condiciones así como también los elementos referenciados en la especificación de la transformación.
MDA (Model Driven Architecture): La arquitectura dirigida por modelos es un acercamiento al diseño de software, propuesto y patrocinado por el Object Management Group. MDA se ha concebido para dar soporte a la ingeniería dirigida a modelos de los sistemas de software. MDA es una arquitectura que proporciona un conjunto de guías para estructurar especificaciones expresadas como modelos.
MDD (Model Driven Development): El Desarrollo de Software Guiado por Modelos es un enfoque de ingeniería de software que se basa en el modelado de un sistema como la principal actividad del desarrollo y construcción del mismo. MDD implica una generación semi-automática de la implementación a partir de los modelos.
Modelo de plataforma específica (PSM – Platform Specific Model): Dista del sistema desde un punto de vista específico de la plataforma.
Modelo independiente del cómputo (CIM – Computation Independent Model): Vista del sistema desde un punto de vista independiente del cómputo. No muestra detalles de la estructura del sistema.
Modelo independiente de la plataforma (PIM – Platform Independent Model): Vista del sistema desde un punto de vista independiente de la plataforma.
OMG (Object Management Group): Es un consorcio dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos. Es una organización sin ánimo de lucro que promueve el uso de tecnología orientada a objetos mediante guías y especificaciones para las mismas.
Transformación de modelos: Proceso que convierte un modelo en otro del mismo sistema, mediante un conjunto de reglas.
Verificación: Busca comprobar que el sistema cumple con los requerimientos especificados (funcionales y no funcionales). Todas las actividades que son llevadas a cabo para averiguar si el software cumple con sus objetivos.

Casos de Prueba: Conjunto de condiciones o variables bajo las cuáles se determinará si el requisito de una aplicación es parcial o completamente satisfactorio.
Testing: Verificación Dinámica de la adecuación del sistema a los requerimientos. Procesos que permiten verificar e identificar diferencias entre el comportamiento real y el comportamiento esperado de un sistema.
Regla de transformación: Una regla de transformación es una descripción de como una o mas construcciones en el lenguaje fuente pueden ser transformadas en una o mas construcciones en el lenguaje destino.
Motor de transformación: Software que realiza la ejecución del código de la transformación, transformando un modelo de entrada que conforma con el metamodelo de entrada en un modelo de salida que conforma con el metamodelo de salida.
Model checker: Técnicas que buscan comprobar en forma automática, bajo ciertas restricciones, características de los modelos de entrada y salida. Aplicable únicamente sobre grafos.
Métodos deductivos: La verificación mediante métodos deductivos se basa en la especificación formal de la transformación y de los metamodelos origen y destino. Se trabaja sobre las especificaciones mencionadas utilizando lenguajes formales y asistentes de prueba. Por lo general, este tipo de verificación se realiza en forma manual o semi-automática.
Oráculo: Software que realiza la comparación entre los resultados obtenidos de la ejecución del caso de prueba y los resultados esperados, retornando un veredicto (paso/fallo).
Caja Negra: La aplicación a probar se visualiza como una caja-negra de la que no se conoce o considera el contenido. La estructura o diseño de la aplicación no son tomados en cuenta para generar los casos de prueba. Los mismos son elaborados a partir de los requerimientos y/o especificación de la aplicación.
Caja Blanca: La base para las técnicas de caja blanca es el código de la aplicación a probar. Es un enfoque que necesita disponer del código y que tiene en cuenta las características de la implementación. Por este motivo, son llamadas pruebas de análisis basadas en código o técnicas de pruebas estructurales.
Clases de equivalencia: La relación de equivalencia ~ define subconjuntos disjuntos en K llamados clases de equivalencia de la siguiente manera: Dado un elemento z que pertenece a K, al conjunto formado por todos los elementos relacionado con z por la relacion de equivalencia ~, se le llama clase de equivalencia asociada al elemento z.
OCL (Object Constraint Language) : Lenguaje declarativo que define reglas aplicadas a modelos UML.
QVT (Query/View/Transformation) : Lenguaje creado por la OMG para definir transformaciones de modelos.

Anexo B

Avances del Proyecto

La figura C.1 muestra cada uno de los hitos del proyecto así como las tareas asociadas a los mismos.

En la figura C.2 se expone el diagrama de Gantt correspondiente.

Id	Nombre de tarea	Comienzo	Fin
1	Primera reunión con los tutores y entrega de documentos para leer	mar 23/03/10	mar 23/03/10
2	Introducción al tema testing de transformaciones de modelos	mar 23/03/10	jue 10/06/10
3	Se entregan nuevos documentos para lectura	mar 11/05/10	mar 11/05/10
4	Búsqueda de documentos sobre el tema testing de software	jue 10/06/10	vie 25/06/10
5	Se entrega primer resumen de los documentos leídos	vie 25/06/10	vie 25/06/10
6	Estado del arte de Testing de Transformación de Modelos	vie 25/06/10	jue 17/02/11
7	Búsqueda de documentos sobre UML Testing Profile y TTCN3	jue 01/07/10	jue 01/07/10
8	Lectura de documentos sobre testing en líneas de productos de software y revisión sistemática	mar 13/07/10	mar 13/07/10
9	Entrega de primera versión del estado del arte de Testing de Transformación de Modelos	mar 10/08/10	mar 10/08/10
10	Entrega de 2a versión del estado del arte de Testing de Transformación de Modelos	lun 30/08/10	lun 30/08/10
11	Devolución de los tutores	mié 01/09/10	mié 01/09/10
12	Reunión con tutores	jue 02/09/10	jue 02/09/10
13	Devolución de los tutores	jue 16/09/10	jue 16/09/10
14	Tutores entregan documentación para estudio de diferentes técnicas de testing de software	jue 16/09/10	jue 16/09/10
15	Entrega de 3a versión del estado del arte de Testing de Transformación de Modelos	jue 21/10/10	jue 21/10/10
16	Presentación de estrategias de testing de caja negra y caja blanca	mar 02/11/10	lun 22/11/10
17	Entrega de 4a versión del estado del arte de Testing de Transformación de Modelos	lun 29/11/10	lun 29/11/10
18	Entrega de 5a versión del estado del arte de Testing de Transformación de Modelos	lun 20/12/10	lun 20/12/10
19	Entrega de 6a versión del estado del arte de Testing de Transformación de Modelos	lun 10/01/11	lun 10/01/11
20	Versión Final del Estado del Arte de Testing de Transformaciones de Modelos	jue 17/02/11	jue 17/02/11
21	Caso de Estudio	jue 17/02/11	mié 20/04/11
22	Tutores plantean objetivo del caso de estudio y transformación a testear	jue 17/02/11	jue 17/02/11
23	Problemas técnicos con herramienta AGG para generación de instancias de metamodelos	sáb 19/02/11	sáb 19/02/11
24	Puesta en práctica de algoritmo para generación de casos de prueba en caso de estudio	lun 14/03/11	lun 14/03/11
25	Utilización de herramienta MMCC para validar casos de prueba	mié 23/03/11	mié 23/03/11
26	Informe Final	jue 31/03/11	

Figura C.1: Avance del Proyecto - Tareas

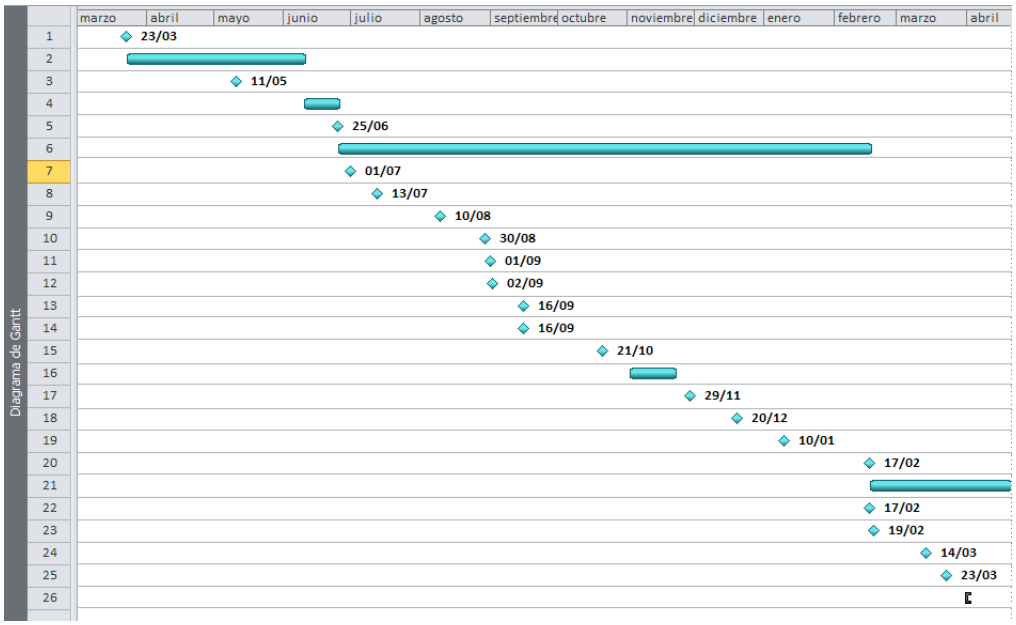


Figura C.2: Avance del proyecto - Diagrama de Gantt

Anexo C

Detalles de configuración del MMCC

En este Anexo se describe la forma de utilizar la herramienta MMCC (Meta-Model Coverage Checker) vista en la sección 3.5.

La herramienta MMCC puede ser obtenida de forma gratuita del sitio http://www.irisa.fr/triskell/Softwares/protos/MMCC/ y está desarrollada con el lenguaje Kermeta en el entorno Eclipse.

Se cuenta con un video tutorial explicativo que puede accederse en la dirección [7] en el se describen los componentes de la herramienta y los pasos a seguir para lograr el cubrimiento del metamodelo de un caso de ejemplo.

Para nuestro caso de estudio se tuvo que generar el metamodelo de KM3 en formato ecore, se puede ver el esqueleto del mismo en la siguiente figura:

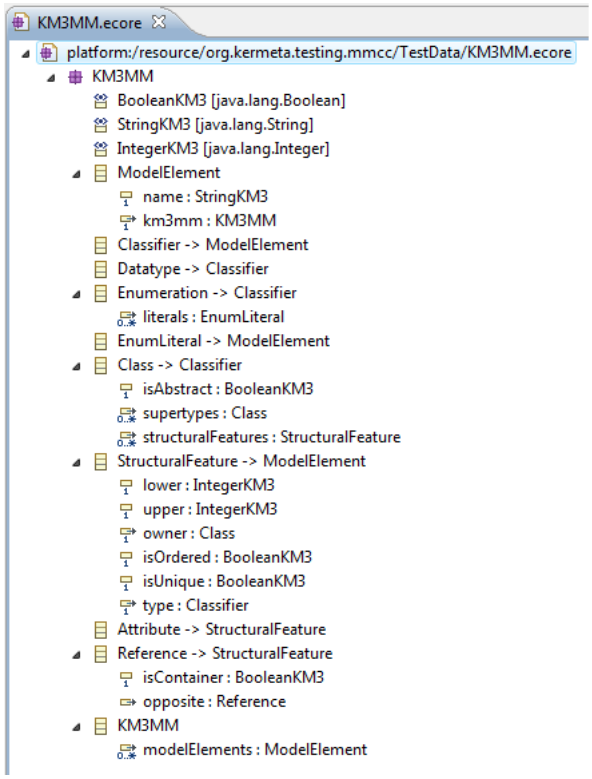


Figura B.1: Metamodelo KM3 de la figura 3.13.[1]

Como primer paso se deben registrar en el ambiente EMF tanto el metamodelo de la herramienta (*ciMM.ecore*) como el metamodelo KM3 (*KM3MM.ecore*) que es al que conforman nuestros casos de prueba como se indica en la siguiente figura:

[1] Por simplicidad se optó por no incluir la clase TypedElement, en cambio se ubicaron los atributos correspondientes en la clase StructuralFeature.

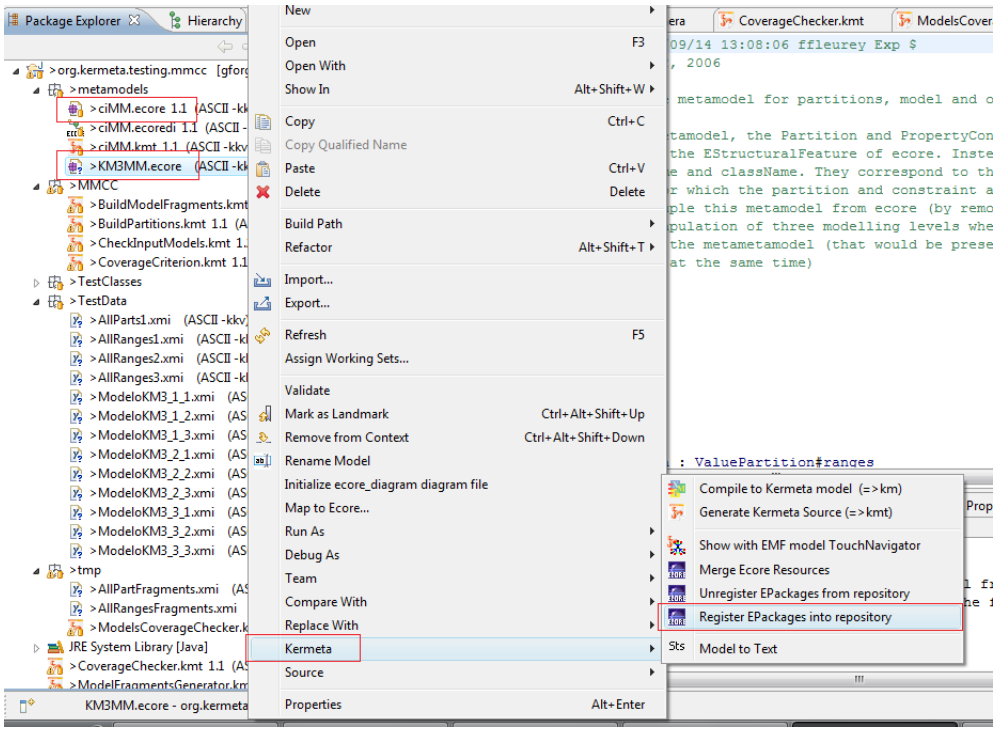


Figura B.2: Registro de metamodelos en formato ecore en el Eclipse Modelling Framework.

Los casos de prueba iniciales se almacenaron bajo el directorio > **TestData**, luego se agregaron bajo este mismo directorio los restantes casos de prueba para cumplir con los criterios de test establecidos según se vio en la sección 3.5.4.

El archivo *ModelFragmentsGenerator.kmt* nos permite definir el criterio de test a utilizar y tiene como función generar los fragmentos de modelo que deberán ser cubiertos por nuestros casos de prueba. Los fragmentos de modelo se guardan en el directorio temporal > **tmp** como se muestra en la siguiente figura.

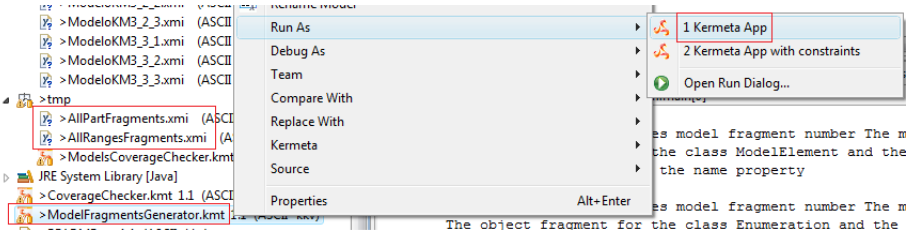


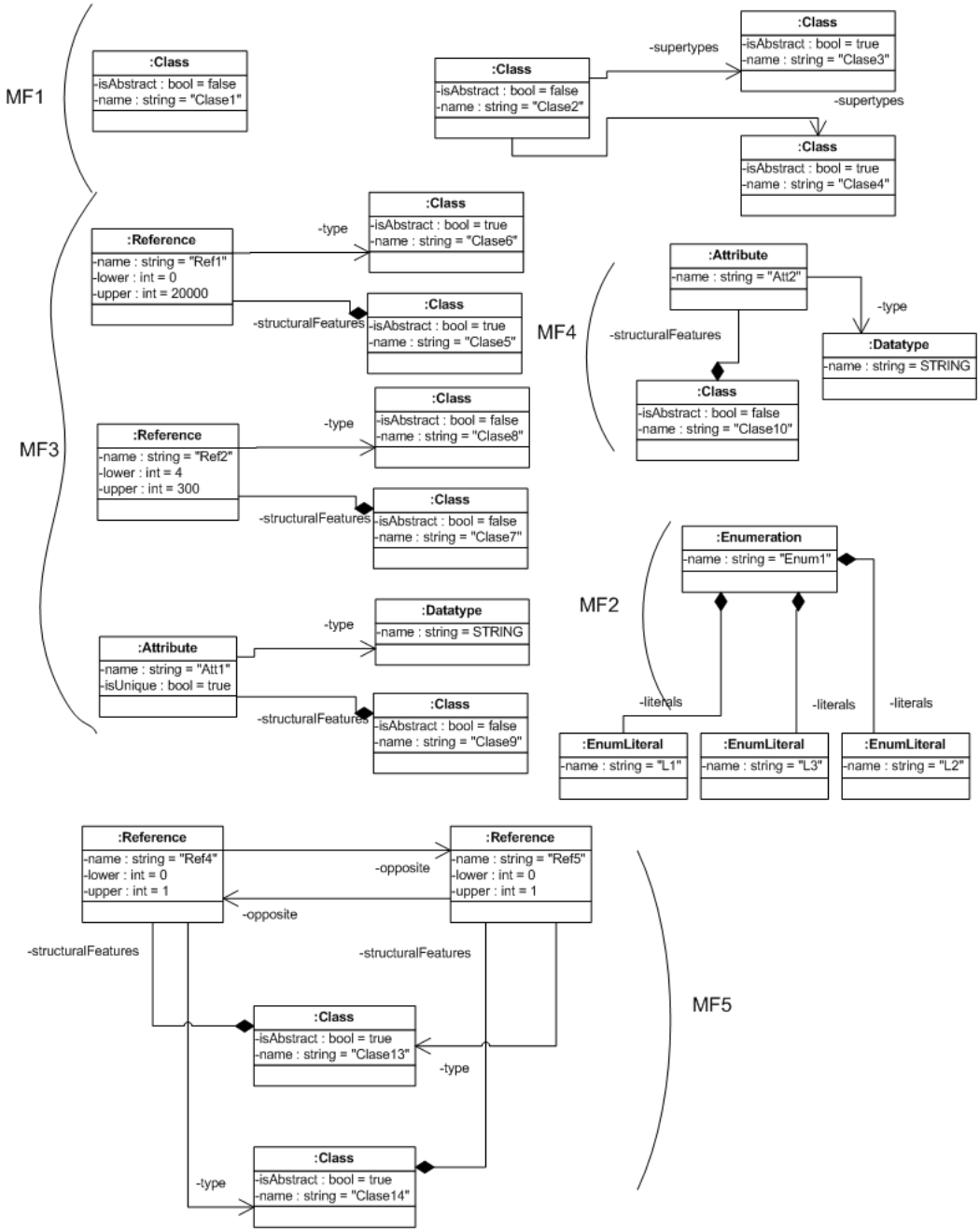
Figura B.3: Ejecución de *ModelFragmentsGenerator.kmt* y resultados temporales

Por último el resultado del cubrimiento y el reporte con los fragmentos de modelo restantes se obtiene ejecutando el archivo *CoverageChecker.kmt*, de igual forma que el archivo anterior.

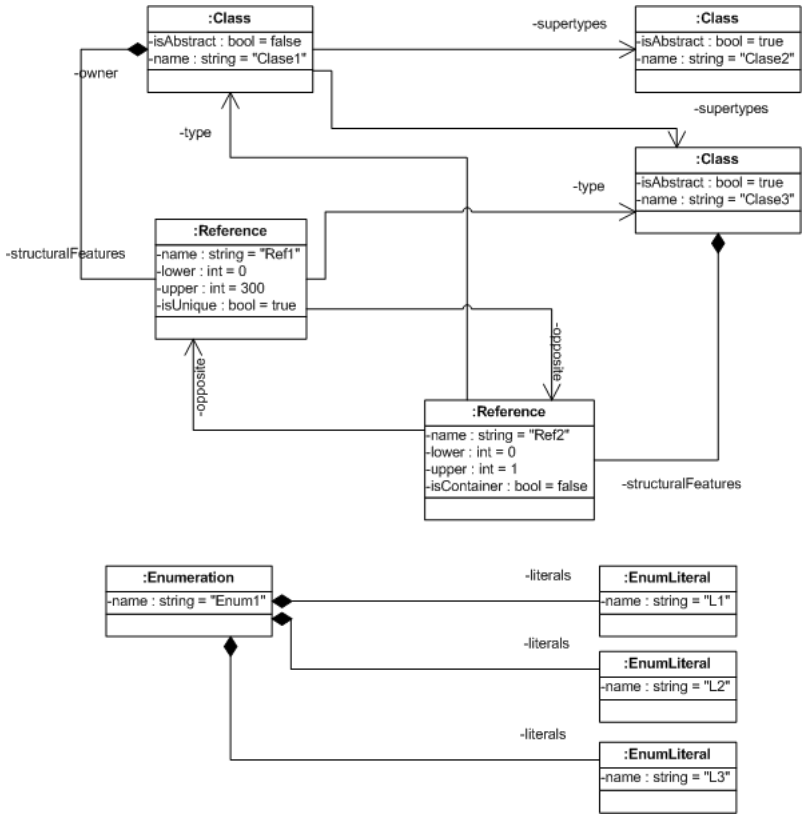
Anexo D

Casos de prueba

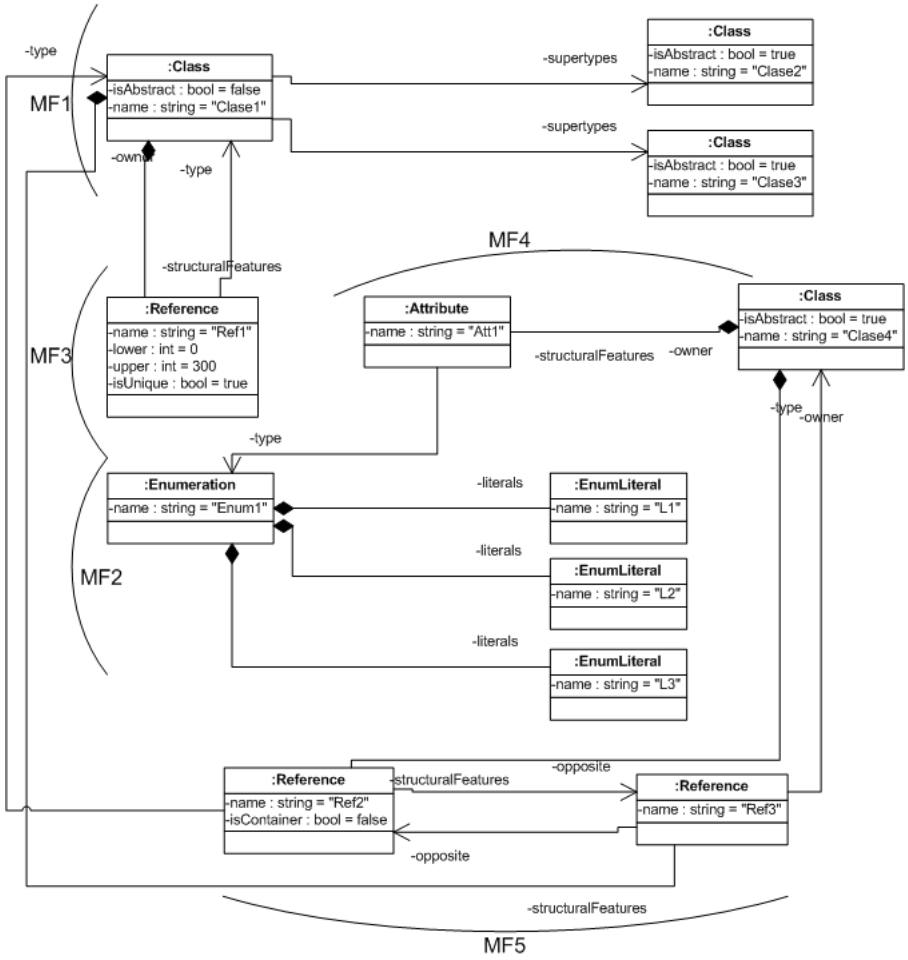
Se muestran los restantes casos de prueba correspondientes a la sección 3.4. Correspondientes a los fragmentos de modelo del **conjunto 2** del cuadro 3.4. Estrategia "crear siempre":


MF1
:Class
-isAbstract : bool = false
-name : string = "Clase1"
:Class
-isAbstract : bool = false
-name : string = "Clase2"
-supertypes
:Class
-isAbstract : bool = true
-name : string = "Clase3"
-supertypes
:Class
-isAbstract : bool = true
-name : string = "Clase4"
MF3
:Reference
-name : string = "Ref1"
-lower : int = 0
-upper : int = 20000
-type
:Class
-isAbstract : bool = true
-name : string = "Clase6"
-structuralFeatures
:Class
-isAbstract : bool = true
-name : string = "Clase5"
:Reference
-name : string = "Ref2"
-lower : int = 4
-upper : int = 300
-type
:Class
-isAbstract : bool = false
-name : string = "Clase8"
-structuralFeatures
:Class
-isAbstract : bool = false
-name : string = "Clase7"
:Attribute
-name : string = "Att1"
-isUnique : bool = true
-type
:Datatype
-name : string = STRING
-structuralFeatures
:Class
-isAbstract : bool = false
-name : string = "Clase9"
MF4
:Attribute
-name : string = "Att2"
-type
:Datatype
-name : string = STRING
-structuralFeatures
:Class
-isAbstract : bool = false
-name : string = "Clase10"
MF2
:Enumeration
-name : string = "Enum1"
-literals
-literals
-literals
:EnumLiteral
-name : string = "L1"
:EnumLiteral
-name : string = "L3"
:EnumLiteral
-name : string = "L2"
MF5
:Reference
-name : string = "Ref4"
-lower : int = 0
-upper : int = 1
-opposite
:Reference
-name : string = "Ref5"
-lower : int = 0
-upper : int = 1
-opposite
-structuralFeatures
-structuralFeatures
:Class
-isAbstract : bool = true
-name : string = "Clase13"
-type
:Class
-isAbstract : bool = true
-name : string = "Clase14"
-type

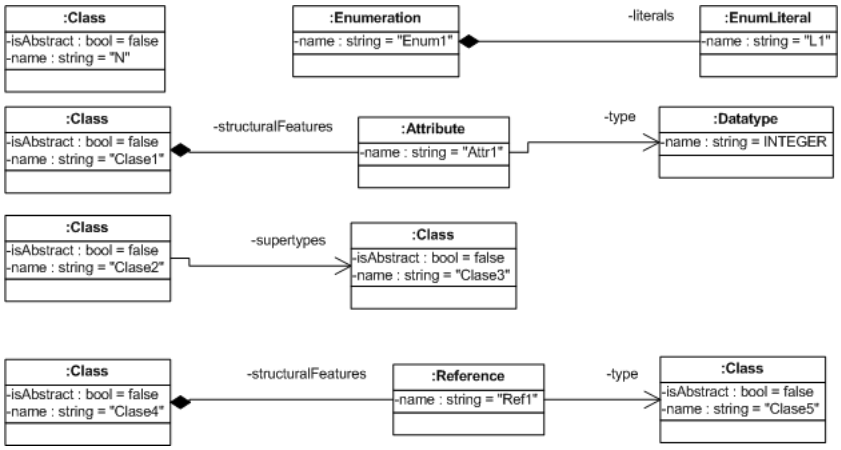
Estrategia “reutilizar siempre que sea posible”:



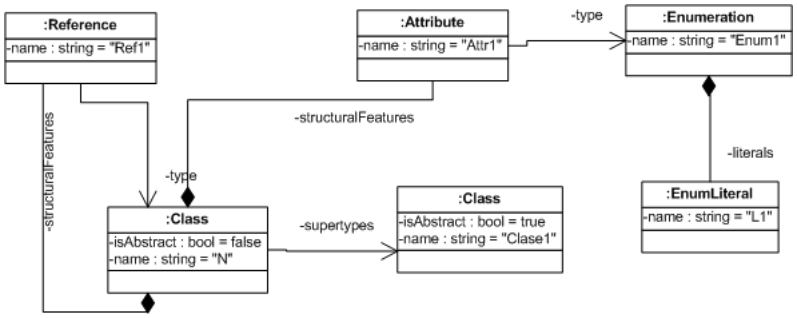
Estrategia “nuevo objeto por cada fragmento de objeto(OF)”:



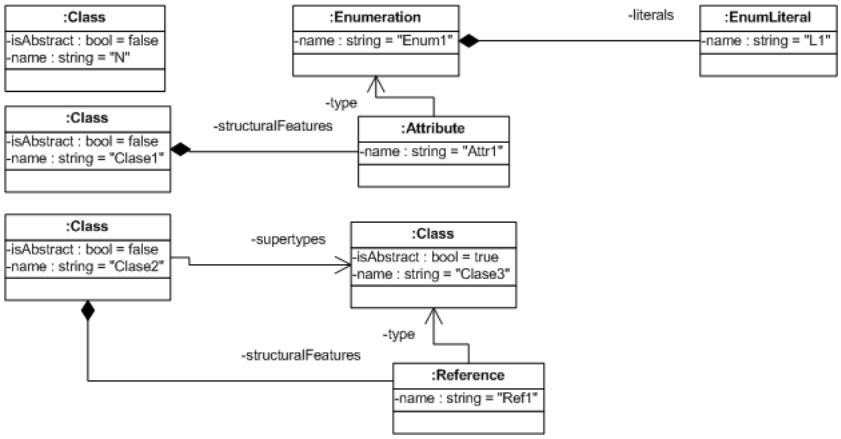
Correspondientes a los fragmentos de modelo del **conjunto 3** del cuadro 3.4.
Estrategia "crear siempre":



Estrategia "reutilizar siempre que sea posible":



Estrategia "nuevo objeto por cada fragmento de objeto(OF)":



Anexo E

Código Fuente de los casos de prueba

ModeloKM3_1_1.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Class" isAbstract="true" name="Clase1">
<structuralFeatures xsi:type="Attribute" name="Attr1" type="//@modelElements.1"></structuralFeatures>
<structuralFeatures xsi:type="Attribute" name="M1"></structuralFeatures>
<structuralFeatures xsi:type="Reference" name="Ref1" isContainer="true" type="//@modelElements.0"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Enumeration" name="Enum1">
<literals xsi:type="EnumLiteral" name="L1"></literals>
</modelElements>

</KM3MM>
```

ModeloKM3_1_2.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Class" isAbstract="true" name="Clase1"> </modelElements>
<modelElements xsi:type="Enumeration" name="Enum1">
<literals xsi:type="EnumLiteral" name="L1"></literals>
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase3">
<structuralFeatures name="Ref1" xsi:type="Reference" type="//@modelElements.3"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase2"></modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase4">
<structuralFeatures xsi:type="Attribute" name="Att1" type="//@modelElements.5"></structuralFeatures>
<structuralFeatures xsi:type="Attribute" name="Att2" type="//@modelElements.6"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Datatype" name="StringKM3"></modelElements>
<modelElements xsi:type="Datatype" name="IntegerKM3"></modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="M1"></modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase6">
<structuralFeatures xsi:type="Reference" name="Ref2" type="//@modelElements.9"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase5"></modelElements>

</KM3MM>
```

ModeloKM3_1_3.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Class" isAbstract="true" name="Clase1">
<structuralFeatures xsi:type="Attribute" name="Att1" type="//@modelElements.2"></structuralFeatures>
<structuralFeatures xsi:type="Reference" name="M1" type="//@modelElements.1"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase2">
<structuralFeatures xsi:type="Reference" name="Ref1" type="//@modelElements.0"
isContainer="true"></structuralFeatures>
<structuralFeatures xsi:type="Attribute" name="Att2" type="//@modelElements.2"></structuralFeatures>
<structuralFeatures xsi:type="Attribute" name="Att3" type="//@modelElements.2"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Enumeration" name="Enum1">
<literals xsi:type="EnumLiteral" name="L1"></literals>
</modelElements>
```

```
</KM3MM>
```

ModeloKM3_2_1.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Class" isAbstract="false" name="Clase1"
supertypes="//@modelElements.1 //@modelElements.2">
<structuralFeatures xsi:type="Reference" name="Ref1" type="//@modelElements.2"
lower="0" upper="300" isUnique="true" opposite="//@modelElements.2/@structuralFeatures.0">
</structuralFeatures>
</modelElements> <modelElements xsi:type="Class" isAbstract="true" name="Clase2"> </modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase3">
<structuralFeatures xsi:type="Reference" name="Ref2" type="//@modelElements.0"
lower="0" upper="1" isContainer="false" opposite="//@modelElements.0/@structuralFeatures.0">
</structuralFeatures>
</modelElements>
<modelElements xsi:type="Enumeration" name="Enum1">
<literals xsi:type="EnumLiteral" name="L1"></literals>
<literals xsi:type="EnumLiteral" name="L2"></literals>
<literals xsi:type="EnumLiteral" name="L3"></literals>
</modelElements>

</KM3MM>
```

ModeloKM3_2_2.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Class" isAbstract="false" name="Clase1"> </modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase2"
supertypes="//@modelElements.2 //@modelElements.3"> </modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase3"> </modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase4"></modelElements>
<modelElements xsi:type="Enumeration" name="Enum1">
<literals xsi:type="EnumLiteral" name="L1"></literals>
<literals xsi:type="EnumLiteral" name="L2"></literals>
<literals xsi:type="EnumLiteral" name="L3"></literals>
</modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase5">
<structuralFeatures xsi:type="Reference" name="Ref1" type="//@modelElements.6" lower="0"
upper="20000"> </structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase6"> </modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase7">
<structuralFeatures xsi:type="Reference" name="Ref2" type="//@modelElements.8" lower="4"
upper="300"> </structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase8"> </modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase9">
<structuralFeatures xsi:type="Attribute" name="Att1" type="//@modelElements.10" isUnique="true">
</structuralFeatures>
</modelElements>
<modelElements xsi:type="Datatype" name="StringKM3"> </modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase10">
<structuralFeatures xsi:type="Attribute" name="Att2" type="//@modelElements.10"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase11">
<structuralFeatures xsi:type="Reference" name="Ref3" type="//@modelElements.13"
lower="1" upper="1" isContainer="false"> </structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase12"> </modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase13">
<structuralFeatures xsi:type="Reference" name="Ref4" type="//@modelElements.15"
lower="0" upper="1" opposite="//@modelElements.15/@structuralFeatures.0"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase14">
<structuralFeatures xsi:type="Reference" name="Ref5" type="//@modelElements.14"
opposite="//@modelElements.14/@structuralFeatures.0"></structuralFeatures>
</modelElements>
```

```
</KM3MM>
```

ModeloKM3_2_3.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Class" isAbstract="false" name="Clase1"
supertypes="//@modelElements.1 //@modelElements.2">
<structuralFeatures xsi:type="Reference" name="Ref1" type="//@modelElements.0" lower="0"
upper="300" isUnique="true" > </structuralFeatures>
<structuralFeatures xsi:type="Reference" name="Ref3" type="//@modelElements.4"
opposite="//@modelElements.4/@structuralFeatures.0" > </structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase2" > </modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase3">
<structuralFeatures xsi:type="Attribute" name="Att1" type="//@modelElements.3"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Enumeration" name="Enum1">
<literals xsi:type="EnumLiteral" name="L1"></literals>
<literals xsi:type="EnumLiteral" name="L2"></literals>
<literals xsi:type="EnumLiteral" name="L3"></literals>
</modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase4">
<structuralFeatures xsi:type="Reference" name="Ref2" type="//@modelElements.0"
isContainer="false" opposite="//@modelElements.0/@structuralFeatures.1"> </structuralFeatures>
</modelElements>

</KM3MM>
```

ModeloKM3_3_1.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Class" isAbstract="false" name="A" supertypes="//@modelElements.2">
<structuralFeatures xsi:type="Attribute" name="At1" type="//@modelElements.1"></structuralFeatures>
<structuralFeatures xsi:type="Reference" name="Ref1" type="//@modelElements.0"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Enumeration" name="Enum1">
<literals xsi:type="EnumLiteral" name="L1"></literals>
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase1"></modelElements>

</KM3MM>
```

ModeloKM3_3_2.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Class" isAbstract="false" name="N"> </modelElements>
<modelElements xsi:type="Enumeration" name="Enum1">
<literals xsi:type="EnumLiteral" name="L1"></literals>
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase1">
<structuralFeatures xsi:type="Attribute" name="Att1" type="//@modelElements.3"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Datatype" name="IntegerKM3"> </modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase2" supertypes="//@modelElements.5">
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase3">
<structuralFeatures xsi:type="Reference" name="Ref1" type="//@modelElements.6"></structuralFeatures>
</modelElements> <modelElements xsi:type="Class" isAbstract="false" name="Clase4"></modelElements>

</KM3MM>
```

ModeloKM3_3_3.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Class" isAbstract="false" name="N"></modelElements>
<modelElements xsi:type="Enumeration" name="Enum1">
<literals xsi:type="EnumLiteral" name="L1"></literals>
</modelElements>
```

```
<modelElements xsi:type="Class" isAbstract="false" name="Clase1">
<structuralFeatures xsi:type="Attribute" name="At1" type="//@modelElements.1"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase2" supertypes="//@modelElements.4">
<structuralFeatures xsi:type="Reference" name="Ref1" type="//@modelElements.4"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase3"></modelElements>

</KM3MM>
```

AllRanges1.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Datatype" name="StringKM3"></modelElements>

</KM3MM>
```

AllRanges2.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Class" isAbstract="false" name="Clase1">
<structuralFeatures xsi:type="Attribute" name="Att1" isUnique="false" type="//@modelElements.1">
</structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase2">
<structuralFeatures xsi:type="Reference" name="Ref1" type="//@modelElements.0"
isOrdered="true"></structuralFeatures>
<structuralFeatures xsi:type="Reference" name="Ref2" type="//@modelElements.0"
isOrdered="false"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Enumeration" name="Enum1"></modelElements>

</KM3MM>
```

AllRanges3.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">

</KM3MM>
```

AllParts1.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<KM3MM xmi:version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI" xmlns="http://www.kermeta.org/testing/mmcc/KM3MM.ecore">
<modelElements xsi:type="Class" isAbstract="false" name="Clase1">
<structuralFeatures xsi:type="Attribute" name="At1" isUnique="false" type="//@modelElements.1">
</structuralFeatures>
<structuralFeatures xsi:type="Attribute" name="At2" isUnique="true" type="//@modelElements.6">
</structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase2">
<structuralFeatures xsi:type="Reference" name="Ref1" type="//@modelElements.0"
isOrdered="true" isContainer="true"></structuralFeatures>
<structuralFeatures xsi:type="Reference" name="Ref2" type="//@modelElements.0"
isOrdered="false" isContainer="false"></structuralFeatures>
</modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase3"></modelElements>
<modelElements xsi:type="Class" isAbstract="true" name="Clase4" supertypes="//@modelElements.2">
</modelElements>
<modelElements xsi:type="Class" isAbstract="false" name="Clase5"
supertypes="//@modelElements.2 //@modelElements.3"></modelElements>
<modelElements xsi:type="Enumeration" name="Enum1"></modelElements>
<modelElements xsi:type="Enumeration" name="Enum2">
<literals xsi:type="EnumLiteral" name="L1"></literals>
</modelElements>
<modelElements xsi:type="Enumeration" name="Enum3">
<literals xsi:type="EnumLiteral" name="L2"></literals>
<literals xsi:type="EnumLiteral" name="L3"></literals>
<literals xsi:type="EnumLiteral" name="L4"></literals>
```

```
</modelElements>
</KM3MM>
```

Referencias

[1] S. J. Mellor, A. N. Clark, and T. Futagami, "Guest editors' introduction: Model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 14–18, 2003.

[2] F. Varesi, H. Lopez, M. Vinolo, D. Calegari, and C. Luna, "Especificación y verificación de transformaciones de modelos." *Technical Report RT10-07, InCo-PEDECIBA, Uruguay*, 2010.

[3] Baudry, Fleurey, and Steel, "Validation in Model-Driven Engineering Testing Model Transformations." *In Proceedings. 2004 1st International Workshop on Model, Design and Validation. SIVOES - MoDeVa 2004 (IEEE Cat. No.04EX984)*, 2004.

[4] J. M. Küster and M. Abd-El-Razik, "Validation of Model Transformations - First Experiences Using a White Box Approach," in *Proceedings of MoDELS Workshops*, 2006, pp. 193–204.

[5] L. Lopez, L. Pintos, D. Calegari, and C. Luna, "Estado del arte de testing de transformaciones de modelos," *Technical Report. Instituto de Computacion, Facultad de Ingenieria, UdelaR.*, Febrero 2011.

[6] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon, "Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool," *In Proceedings of the ISSRE'06*, pp. 85–94, 2006.

[7] "Metamodel Coverage Checker (MMCC)," http://www.irisa.fr/triskell/Softwares/protos/MMCC/ Último acceso Diciembre 2010.

[8] F. Fleurey, B. Baudry, P.-A. Muller, and Y. L. Traon, "Qualifying input test data for model transformations," *Journal of Software and System Modeling*, vol. 8, no. 2, pp. 185–203, 2009.

[9] R. M. Hierons, "Software testing foundations: A study guide for the certified tester exam. dpunkt.verlag, heidelberg, germany, 2006,pp 266," *Softw. Test., Verif. Reliab.*, vol. 16, no. 4, pp. 289–290, 2006.

[10] IEEE, "Standard for Software Test Documentation," 1983, http://standards.ieee.org/findstds/standard/829-1998.html. Último acceso Diciembre 2010.

[11] G. J. Myers, *The Art of Software Testing.* New York: Wiley, 1979.

[12] B. Baudry, T. Dinh-trong, J. marie Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. L. Traon, "Model Transformation Testing Challenges," in *In Proceedings of IMDT workshop in conjunction with ECMDA06*, 2006.

[13] B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Commun. ACM*, vol. 53, no. 6, pp. 139–143, 2010.

[14] J.-M. Mottu, B. Baudry, and Y. L. Traon, "Reusable MDA Components: A Testing-for-Trust Approach," in *Proceedings of MoDELS*, 2006, pp. 589–603.

[15] A. Solberg, R. Reddy, D. Simmonds, R. France, and S. Ghosh., "Developing Service Oriented Systems Using an Aspect-Oriented Model Driven Framework," *In the International Journal of Cooperative Information Systems (IJCIS), Volume 15, No 4*, 2006.

[16] H. Lopez, F. Varesi, M. Vinolo, D. Calegari, and C. Luna, "Estado del arte de verificación de transformaciones de modelos," *Technical Report RT10-07, InCo-PEDECIBA, Uruguay*, 2010.

[17] F. Fleurey, B. Baudry, P. A. Muller, and Y. Le Traon, "Towards Dependable Model Transformations: Qualifying Input Test Data," *Journal of Software and Systems Modeling (SoSyM)*, 2007.

[18] B. Baudry, "Testing Model Transformations: A case for Test Generation from Input Domain Models," in *Chapter in Model Driven Engineering for Distributed Real-time Embedded Systems.* Hermes, 2009.

[19] D. Cohen, I. C. Society, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, pp. 437–444, 1997.

[20] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE Software*, vol. 13, pp. 83–88, 1996.

[21] D. R. Kuhn, S. Member, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing." *IEEE Trans. Software Engineering*, vol. 30, p. 2004, 2004.

[22] J. Bach and P. J. Schroeder, "Pairwise testing: A best practice that isn't," *In Proceedings of the 22nd Annual Pacific Northwest Software Quality Conference*, pp. 180–196, 2004.

[23] M. Lamari, "Towards an automated test generation for the verification of model transformations," in *Proceedings of the Symposium on Applied Computing*, 2007, pp. 998–1005.

[24] M. Grochtmann and D. benz Ag, "Test Case Design Using Classification Trees," *In Proceedings of STAR, 1994*, 1994.

[25] "Query/View/Transformation (QVT)," http://www.omg.org/spec/QVT/ Último acceso Mayo 2011.

[26] J.-M. Mottu, B. Baudry, and Y. L. Traon, "Mutation Analysis Testing for Model Transformations," in *Proceedings of ECMDA-FA*, 2006, pp. 376–390.

[27] J. Bézivin, B. Rumpe, and L. Tratt, "Model Transformation in Practice Workshop Announcement," *MoDELS'05*, 2005.

[28] J.-M. Mottu, B. Baudry, and Y. L. Traon, "Model transformation testing: oracle issue," *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pp. 105–112, 2008.

[29] R. Heckel and M. Lohmann, "Towards model-driven testing," *Journal of Electr. Notes Theor. Comput. Sci.*, vol. 82, no. 6, 2003.

[30] Y. Lin, J. Gray, and F. Jouault, "DSMDiff: A Differentiation Tool for Domain-Specific Models," *European Journal of Information Systems*, vol. 16, no. 4, pp. 349–361, August 2007, (Special Issue on Model-Driven Systems Development).

[31] M. Alanen and I. Porres, "Difference and Union of Models," in *Proceedings of the 6th joint meeting of the European Software Engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, 2003, pp. 2–17.

[32] A. Toulme, "The EMF Compare Utility," http://www.eclipse.org/modeling/emft/ Último acceso 30/12/2010.

[33] Y. L. Traon, B. Baudry, and J.-M. Jézéquel, "Design by Contract to Improve Software Vigilance," *IEEE Trans. Software Eng.*, vol. 32, no. 8, pp. 571–586, 2006.

[34] "Object Constraint Language (OCL)," http://st.inf.tu-dresden.de/oclportal/index.php. Último acceso Diciembre 2010.

[35] E. Cariou, R. Marvie, L. Seinturier, and L. Duchien, "Ocl for the specification of model transformation contracts," in *In Proceedings of Workshop OCL and Model Driven Engineering*, 2004.

[36] ATLAS Group, *KM3: Kernel MetaMetaModel. LINA & INRIA. Manual v0.3 (2005).*

[37] T. C. D. Team, *The Coq Proof Assistant:Reference Manual, 2010, version 8.3. [Online].*, http://coq.inria.fr. [Online]. Available: http://coq.inria.fr

[38] Grupo ATLAS, *ATL User Manual*, 2006, version 0.7.

[39] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Boston, MA: Addison-Wesley, 2009.

[40] "Eclipse. xtext user guide." http://www.eclipse.org/Xtext/documentation/1_0_0/xtext.html. Último acceso Mayo 2011.

[41] G. Taentzer, "Agg: A graph transformation environment for modeling and validation of software," in *Applications of Graph Transformations with Industrial Relevance*, 2003, pp. 446–453.