PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 11-01

Estado del arte de testing de transformaciones de modelos

Leonardo López, Leonardo Pintos, Daniel Calegari, Carlos Luna

2011

Estado del arte de testing de transformaciones de modelos

Leonardo López, Leonardo Pintos, Daniel Calegari, Carlos Luna.

ISSN 0797-6410
Reporte Técnico RT 11-01
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, febrero de 2011

Estado del Arte de Testing de Transformaciones de Modelos

Leonardo López, Leonardo Pintos, Daniel Calegari, Carlos Luna.

Montevideo, Febrero 2011

Instituto de Computación, Facultad de Ingeniería, UdelaR

Resumen

La Ingeniería Dirigida por Modelos es un enfoque de Ingeniería de Software que se basa en el modelado de un sistema como la principal actividad para la construcción del mismo. La viabilidad de este enfoque está dada por la existencia de un proceso de construcción (semi) automático guiado por transformaciones de modelos. El proceso comienza con modelos abstractos del sistema a construir que son transformados hasta generar un modelo ejecutable. La calidad global del proceso depende fuertemente de la calidad de las transformaciones, por lo que es vital su verificación. Este reporte presenta el estado del arte de las técnicas de diseño, ejecución y validación de casos de prueba (i.e. testing) aplicadas a transformaciones de modelos. Se analizan barreras y desafíos existentes, y se describen lenguajes y herramientas utilizados.

1. Introducción

El enfoque de Ingeniería Dirigida por Modelos (Model Driven Engineering, MDE) tiene dos componentes principales: los modelos y las transformaciones. Un modelo es una especificación formal de la función, estructura o comportamiento de una aplicación o sistema [1]. En el contexto de MDE, 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.

La construcción del sistema es llevada a cabo mediante un proceso (semi) automático de transformación de dichos modelos, partiendo de modelos abstractos hasta obtener modelos ejecutables. Una transformación de modelos es entonces la generación automática de un modelo destino a partir de un modelo origen, de acuerdo a una especificación basada en un conjunto de reglas de transformación. Estas reglas describen cómo una o más construcciones del modelo origen (especificadas en el metamodelo) son transformadas en una o más construcciones

del modelo destino, y son ejecutadas por un motor de transformaciones. En la Figura 1 se puede apreciar este esquema:

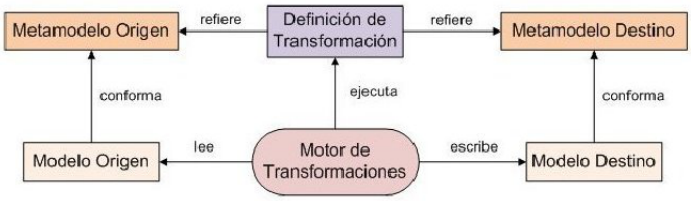


Figura 1: Elementos de una transformación de modelos.[2]

La factibilidad del enfoque MDE depende de la posibilidad de (semi) automatizar el proceso de construcción en base al esquema de transformaciones de modelos presentado anteriormente. En consecuencia, la calidad global del proceso dependerá fuertemente de la calidad de las transformaciones, por lo que resulta vital la verificación de las mismas.

Existe una gran variedad de técnicas para verificar transformaciones de modelos, las cuales se pueden agrupar en tres enfoques diferentes [3] 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. Ambos tipos de verificación mencionados presentan varias dificultades. La verificación utilizando métodos deductivos requiere conocimientos de los formalismos utilizados y las pruebas comúnmente se deben realizar de forma manual o semi-automática. En tanto la verificación de modelos es aplicable únicamente sobre grafos y la complejidad de las pruebas crece exponencialmente con respecto al tamaño del grafo. 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. La corrección en este caso se prueba para un conjunto representativo del dominio con lo que se generan ciertos niveles de confianza en la corrección esperada para el resto de los modelos a los cuales se aplicará la transformación.

El objetivo de este reporte es estudiar las técnicas de diseño, ejecución y validación de casos de prueba (de ahora en más testing) aplicadas a transformaciones de modelos.

El resto del documento se organiza de la siguiente manera. La sección 2 presenta las características del testing de aplicaciones y sus técnicas más conocidas de forma tal de relacionarlas luego con el testing de transformaciones de modelos. La sección 3 analiza las dificultades y los obstáculos que se presentan al abordar el testing de transformaciones de modelos. La sección 4 describe

las técnicas aplicadas al testing de transformaciones de modelos encontradas en la bibliografía científica, en tanto la sección 5 introduce las herramientas y los lenguajes utilizados en esta área. Finalmente, la sección 6 presenta las conclusiones del trabajo.

2. Testing de Aplicaciones

En [4, 5, 6] se define el testing de aplicaciones como el proceso de ejecución de un programa con la intención de verificar la corrección de los requerimientos, identificar diferencias entre el comportamiento real y esperado, medir calidad, proporcionar confianza y detectar errores. El testing de aplicaciones es un proceso en el cual se define un conjunto de pruebas y posteriormente se ejecutan y validan las mismas. El conjunto de pruebas es definido teniendo en cuenta la técnica de diseño de casos de prueba a aplicar, en la mayoría de los casos técnicas de caja negra o técnicas de caja blanca. Cada prueba debe tener definido un objetivo de prueba, casos de prueba diseñados y codificados para luego ejecutarlos y comparar los resultados obtenidos con los resultados esperados. La comparación de resultados comúnmente es realizada de forma manual por los testeadores. Este software llamado oráculo, realiza la comparación entre los resultados obtenidos de la ejecución del caso de prueba y los resultados esperados, retornando un veredicto (paso/fallo).

2.1. Niveles de Prueba.

Existen diferentes niveles de prueba que pueden ser utilizados en la verificación de una aplicación, dependiendo este nivel del desarrollo de la aplicación. La figura siguiente describe el modelo V de pruebas, el cual muestra la conexión entre las actividades del desarrollo y testing para los diferentes niveles de desarrollo.

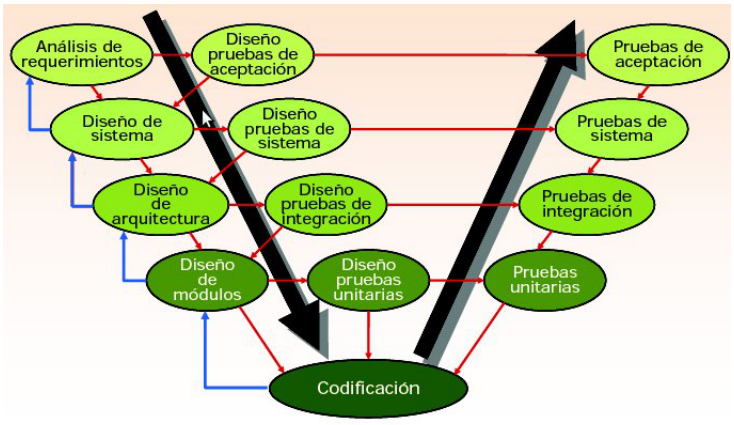


Figura 2: Modelo V [7]

A continuación describiremos los principales niveles de pruebas:

Pruebas Unitarias : Las pruebas unitarias son realizadas sobre distintos componentes de la aplicación para chequear el correcto funcionamiento de los mismos antes de realizar una integración. Dentro de las pruebas unitarias encontramos técnicas estáticas y técnicas dinámicas. Las técnicas estáticas consisten en examinar el código de la aplicación buscando errores. Las técnicas dinámicas se basan en experimentar con el comportamiento de un módulo o componente de la aplicación para determinar su correcto funcionamiento.

Pruebas de Integración: Pruebas que se realizan para descubrir errores que pueden ocurrir durante el proceso de ensamblado de componentes de la aplicación.

Pruebas Funcionales: Las pruebas funcionales se realizan para comprobar que el sistema efectivamente cumple los requerimientos. Los requerimientos son una descripción precisa del comportamiento esperado del sistema según la perspectiva del usuario final. Estas pruebas en general se basan en los casos de uso para generar casos de prueba.

Pruebas de Sistema: El propósito de las pruebas de sistema es demostrar que el producto es consistente con los objetivos planteados originalmente. No debe confundirse el concepto de objetivo con el de requerimiento (visto en la definición de Pruebas funcionales). Los objetivos establecen que es lo que el sistema debería hacer y que tan bien debería hacerlo, pero no establecen la representación de las funciones del sistema.

Finalizando con los niveles de prueba, la aplicación a transformaciones de modelos depende de la estructura de la transformación. Si las funcionalidades de la transformación se encuentran en un solo componente, se aplicarían pruebas unitarias. En caso que estuviera diseñada en varios componentes, podríamos aplicar los cuatro niveles de prueba vistos.

2.2. Técnicas de diseño de casos prueba.

A continuación definiremos las técnicas de diseño de casos de prueba mas utilizadas al aplicar los niveles de pruebas vistos en 2.1.

- 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.[4]

- 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.[4]

En [6] se recomienda realizar una combinación, que logra buenos resultados, de ambos métodos. Esta combinación implica realizar casos de prueba usando

métodos de caja-negra y luego desarrollar tantos casos de prueba suplementarios como sea necesario con los métodos de caja-blanca.

Es necesario mencionar que el testing de transformaciones de modelos tiene sus particularidades, por lo tanto hay que enfrentar determinados obstáculos que analizaremos en la sección 3.

2.3. Testing de programas declarativos.

En el contexto de las transformaciones de modelos existe una variada gama de lenguajes que sirven para especificar las mismas, algunos de ellos utilizan el enfoque declarativo. En este apéndice se detallan algunos trabajos que tratan sobre el testing en este tipo de lenguajes.

Los lenguajes de programación declarativos ofrecen un grado de abstracción alto que facilita el desarrollo de sistemas complejos. Ayudan a escribir código más legible y reusable [8], sin embargo todavía puede contener errores. Realizar pruebas formales en sistemas complejos se vuelve difícil y costoso en términos de tiempo, por lo tanto se han propuesto aproximaciones basadas en heurísticas para exponer los errores. En [9] se utiliza el lenguaje Curry (lógico-funcional) que tiene una sintaxis similar a Haskell pero usa un diferente mecanismo de evaluación para mostrar una herramienta que permite la generación sistemática de casos de prueba para programas lógico-funcionales. El enfoque utilizado es el de caja blanca argumentando que es preferible esta técnica de testing para lenguajes declarativos debido a que los programas declarativos típicamente consisten en una secuencia de definiciones de funciones que tienen una estructura de complejidad algorítmica, lo que hace que sea difícil deducir todos los posibles comportamientos tomando en cuenta solo la especificación del programa.

Los criterios de cubrimiento son los mismos que se utilizan para los programas imperativos,el cubrimiento del grafo de control de flujo y el cubrimiento de las cadenas definición-uso [10]. Sin embargo los lenguajes declarativos con evaluación perezosa como Curry tienen un grafo de control de flujo complicado y son necesarias adaptaciones de los criterios anteriores. Los autores presentan dos criterios de cubrimiento diferentes: Global Branch Coverage y Function Coverage ambos correspondientes a variantes del cubrimiento del grafo de control y también presentan un prototipo de una herramienta que utiliza cualquiera de estos dos criterios para generar sistemáticamente casos de prueba.

En un trabajo posterior [11] estos mismos autores extienden la herramienta para que también pueda lograr el cubrimiento del flujo de datos. La información sobre el cubrimiento se obtiene mediante un programa que recolecta esta información en tiempo de ejecución .

Otros trabajos relacionados con el área del testing de programas declarativos se basan en la técnica de caja negra, en particular Random Testing. Estos trabajos al no tomar en cuenta la implementación no pueden asegurar que todas las partes del programa fueron ejecutadas, por lo tanto errores en partes no cubiertas pueden pasar inadvertidos. En esta línea se puede ver la herramienta QuickCheck [12] para el lenguaje Haskell.

3. Barreras y desafíos del testing de transformaciones 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, la heterogeneidad de los lenguajes de transformación y la escasa cantidad de herramientas de gestión de modelos hacen del testing de transformaciones de modelos un duro trabajo para los testeadores. En esta sección se analizaran problemas a enfrentar para realizar el testing de una transformación de modelos y se comentan sugerencias sobre como encaminar la solución de alguno de ellos. El desarrollo de esta sección esta basado completamente en conocimientos brindados en [13] y [14].

3.1. Problemas que impactan al testing de transformaciones de modelos

3.1.1. Datos de entrada/salida complejos.

Los modelos de entrada y salida de las transformaciones conforman con sus respectivos meta-modelos, los cuales suelen ser complejos tanto por su estructura como por las restricciones no estructurales que poseen. A diferencia de una aplicación común, una transformación de modelos tiene como parámetro de entrada uno o mas modelos.

La complejidad de los meta-modelos hace que la generación de modelos de prueba sea una tarea complicada debido a su tamaño, restricciones, multiplicidades y una cantidad de propiedades a tener en cuenta a la hora de generarlos. Las técnicas manuales y automáticas de generación de modelos de prueba se dificultan por la gran cantidad de instancias del meta-modelo que se deben considerar.

Con respecto a los modelos de salida de la transformación, su complejidad dificulta la creación de oráculos. Cuando se tienen resultados esperados, el oráculo compara la salida de la transformación con los resultados esperados, siendo esta comparación un problema NP-Completo. Si el oráculo es construído a partir de propiedades extraídas de los modelos de salida, su construcción también es dificultosa por la complejidad del meta-modelo de salida que describe a los modelos de salida.

3.1.2. Ambientes que manejan modelos.

El paradigma MDE se encuentra en una fase de investigación y estudio, por lo tanto no abundan los entornos de desarrollo capaces de manipular modelos.

La verificación de una transformación de modelos necesita de estos entornos para construir, editar, visualizar y analizar modelos. El manejo de modelos debería ser realizado por programas que generen instancias de un meta-modelo o por editores de modelos que permitan la creación y edición manual de modelos y sus propiedades. Los editores pueden ser generales y no específicos para un determinado lenguaje. En consecuencia puede que no se cuente con iconos o

cuadros de diálogo para establecer valores a atributos o propiedades a los modelos que se especifican. Se debe tener en cuenta que la especificación manual de modelos es una tarea dificultosa y propensa a errores.

Otro punto a mejorar en los editores gráficos es la visualización de los modelos de salida. Se debe contar con una funcionalidad que permita analizar la salida de la transformación. No contar con una visualización correcta dificulta el análisis, así como la comparación entre modelos. Esta tarea es muy utilizada en pruebas de regresión, donde el modelo de salida de una prueba se compara con el modelo de la salida anterior.

3.1.3. Heterogeneidad de lenguajes y herramientas.

La OMG (Object Management Group) [15] ha definido un estándar para implementar transformaciones de modelo, llamado QVT. Sin embargo, existen un gran número de lenguajes y herramientas [3]. Las transformaciones pueden ser implementadas en lenguajes de programación de propósito general o usando lenguajes dedicados específicamente a transformaciones de modelos.

La diversidad de lenguajes y técnicas existentes es grande, y es complicado saber si existe una que se adapte a todas las necesidades requeridas por las transformaciones de modelos. Esta diversidad tiene un impacto importante a la hora de realizar la verificación de transformaciones, porque no se puede elegir un lenguaje de referencia, lo cual lleva a implementar criterios de prueba generales.

3.1.4. Especificación de requerimientos.

Se deben desarrollar técnicas para obtener una precisa definición de los requerimientos de la transformación. En la actualidad la especificación de requerimientos tiende a ser informal, en consecuencia no pueden ser procesados por herramientas para generar automáticamente el resultado esperado. [14]

3.2. Enfoques y sugerencias para superar los problemas.

3.2.1. Datos de entrada/salida complejos.

El principal desafío a enfrentar para la generación de modelos de pruebas es la complejidad de las restricciones del dominio de entrada, las cuales pueden llegar a ser complejas y reducen considerablemente el espacio de posibles modelos válidos. La estrategia mas común para la construcción de modelos válidos es generar el modelo y luego verificar las propiedades, descartando aquellos que no las cumplan. En [16] y [17] se pueden observar enfoques que permiten realizar esto, de todos modos en ambos trabajos se tiene como limitación la satisfactibilidad del conjunto completo de restricciones.

Para la definición de oráculos se tienen varias estrategias, una primera estrategia cuando la transformación genera un modelo ejecutable, es probar directamente el modelo de salida. Otra posibilidad es generar un oráculo parcial que corrobore la corrección de determinadas propiedades del modelo de salida. Estos enfoques son estudiados por Solberg en [18] y Mottu en [19], entre otros.

Solberg propone patrones para expresar pre y post condiciones de la transformación. Estos patrones son plantillas que describen características esperadas en los modelos de entrada y salida, y pueden ser vistos como una especialización de los meta-modelos de entrada y salida.

Mottu propone la utilización de contratos para construir la transformación de modelos y describe como los contratos pueden ser utilizados como oráculos para testear la transformación de modelos. El estudio sobre oráculos se profundiza en la sección 4.4.

3.2.2. Ambientes que manejan modelos.

Trabajos enfocados en la gestión de modelos pueden ser adaptados para pruebas de transformaciones de modelos. Mediante la comparación de modelos se podrían generar oráculos que brinden un veredicto entre el resultado de la transformación y resultados esperados. Un ejemplo de esto es la herramienta EMFCompare [20], basada en el algoritmo propuesto por Xing en [21], que esta disponible para el entorno de trabajo Eclipse. Esta herramienta detecta coincidencias y diferencias entre dos modelos.

El versionado de modelos también puede ayudar de gran manera al testing de transformaciones. Puede ser utilizado para definir un oráculo que compare modelos y detecte conflictos. Existen herramientas que manejan versionado tales como CVS [22] y SVN [23]. Un mecanismo de detección de conflictos puede ser el presentado en [24].

3.2.3. Heterogeneidad de lenguajes y herramientas.

La heterogeneidad de los lenguajes es un desafío para la definición del criterio de test que se va a utilizar. Se sugieren dos posibles estrategias para su resolución. La primer estrategia es tener criterios y técnicas de test específicos para cada lenguaje particular. La segunda es utilizar técnicas de caja negra que ignoren el lenguaje utilizado en la transformación.

Un ejemplo de las estrategias descritas anteriormente son los trabajos de Fleurey en [25] y Kuster en [26]. El primero propone el criterio de caja negra para evaluar la calidad de los modelos usados como casos de prueba. 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 define el concepto de plantillas para generar modelos basados en la reglas de la transformación. 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. Además, un enfoque de caja blanca debe cubrir cada paso de la transformación, esto hace que el testeador deba generar casos de prueba para cada uno de ellos. En consecuencia, los criterios de caja blanca son más detallados pero más eficaces que los de caja negra.

3.2.4. Especificación de requerimientos.

Se están estudiando técnicas para extraer formulas lógicas de los lenguajes naturales. Otra solución sería que se modelara el comportamiento esperado como las reglas interpretadas por los oráculos.[14]

4. Testing de transformaciones de modelos

El testing de transformaciones de modelos se basa principalmente en la adaptación de las técnicas clásicas a este enfoque. En la secciones 4.1 y 4.2 se detalla la generación de casos de prueba y el testing de las transformaciones aplicando las técnicas de diseño de casos de prueba de caja negra y caja blanca, respectivamente.

En la sección 4.3 se describe el análisis de mutación que permite medir la calidad del conjunto de casos de prueba generados. Por último, en la sección 4.4 se trata el problema de la generación de oráculos en el contexto de transformaciones de modelos.

4.1. Generación de Casos de Prueba - Caja Negra

En la técnica de caja negra se utiliza el meta-modelo de entrada de la transformación, que define completamente el conjunto de los posibles modelos de entrada. Esta aproximación tiene como ventaja que es independiente del lenguaje en el cual está programada la transformación.

La generación de casos de prueba propuesta en [27] tiene como objetivo cubrir la mayor parte de este meta-modelo de entrada. Los autores notan que el meta-modelo de entrada contiene una gran cantidad de datos que no son relevantes para el testing de la transformación. Definen entonces previo a la generación de los casos de prueba, el meta-modelo que es relevante para la transformación, al que llaman *meta-modelo efectivo*.

El *meta-modelo efectivo* se construye utilizando los datos de las pre y post condiciones así como también los elementos referenciados en la especificación de la transformación. Utilizando este meta-modelo efectivo se verán dos técnicas [28] que se adaptan al testing de transformaciones de modelos y que pueden utilizarse de forma conjunta o separada. El objetivo es seleccionar combinaciones de propiedades del *meta-modelo efectivo* que nos permitan acotar aún más la cantidad de casos de prueba a generar y así lograr un mayor cubrimiento del mismo.

4.1.1. Particiones en clases de equivalencia

La primera técnica se conoce como particiones en categorías o particiones en clases de equivalencia [29]. La idea básica de esta estrategia es dividir el dominio de entrada 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.

Un ejemplo que se presenta en el artículo es el de una partición de un tipo de datos String. Luego de aplicar la técnica el resultado consiste de una partición con dos rangos: el rango que contiene la cadena vacía y el rango que contiene todas las cadenas no vacías, representado con una expresión regular.

En el contexto de las transformaciones de modelos [28] los autores proponen primero identificar todas las propiedades del meta-modelo efectivo, para luego identificar en cada propiedad los valores representativos, lográndo así la división del dominio de esa propiedad en un conjunto de rangos que forman una partición.

Para determinar estas particiones proponen utilizar dos métodos: el primero llamado particionado por defecto y el segundo llamado particionado basado en el conocimiento [27].

El particionado por defecto podría ser aplicable cuando no se tiene información sobre valores representativos y consiste en definir una partición basada en la estructura o el tipo de los datos. Por ejemplo, se selecciona un valor mínimo, un máximo y un valor que no sea límite.

Por el contrario, el método de 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 definidas las particiones y los rangos se generan los casos de prueba tomando los valores de cada uno de los rangos.

Los autores expresan que la efectividad de esta estrategia reside en la calidad de las particiones a ser usadas. Es por esto que proponen identificar claramente los valores límites y los valores que tienen un significado especial, y asignarlos en rangos específicos para que sean tomados en cuenta al momento de generar los casos de prueba.

Fragmentos de modelo y Fragmentos de objeto

Cubrir los valores y multiplicidades de cada propiedad del meta-modelo de entrada independientemente no es suficiente para generar casos de prueba relevantes [30] por lo tanto se propone combinar valores de los distintos rangos para formar los casos de prueba.

En el artículo referido se evalúa una primera aproximación que 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 la aproximación anterior no es suficiente, 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[28]. Se expone lo anterior mediante un ejemplo.

La Figura 3 muestra el meta-modelo correspondiente a un lenguaje para modelar un diagrama de clases simple.

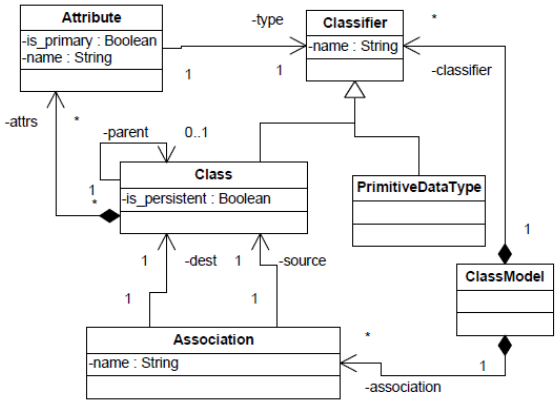


Figura 3 - Meta-modelo del Diagrama de Clases UML simple [28].

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

Luego de aplicar particionado por defecto sobre el meta-modelo de la Figura 3 se obtienen las particiones (lado izquierdo de la tabla) y los rangos(lado derecho de la tabla) que se muestran en la Figura 4. 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 4 - Particiones para el meta-modelo del diagrama de clases simple [28].

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 meta-modelo.

Como alternativa se generan fragmentos de objetos que son instancias de clases del meta-modelo 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 objetos.

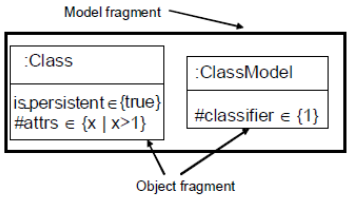


Figura 5 - Ejemplo de fragmentos de objeto y de modelo [28].

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 modelos 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 meta-modelo 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 meta-modelo debe de 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 y el número de fragmentos de objetos de cada fragmento de modelo da una medida del tamaño de los casos 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 meta-modelo de entrada.

4.1.2. Testing de Interacción Combinatoria

La técnica llamada Testing de Interacción Combinatoria [31],[32] selecciona un subconjunto de todas las posibles combinaciones de las variables a testear (de todas las posibles combinaciones de los rangos de todas las particiones).

Se basa en la observación de que las mayorías de las fallas son disparadas por interacciones entre un número pequeño de variables. Esto llevó a la definición del 2-way testing [33], 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 [34].

4.1.3. Método de Clasificación de Árbol

Otra técnica utilizada es el Método de Clasificación de Árbol [35]. 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. A continuación introducimos el ejemplo presentado en [36].

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 6.

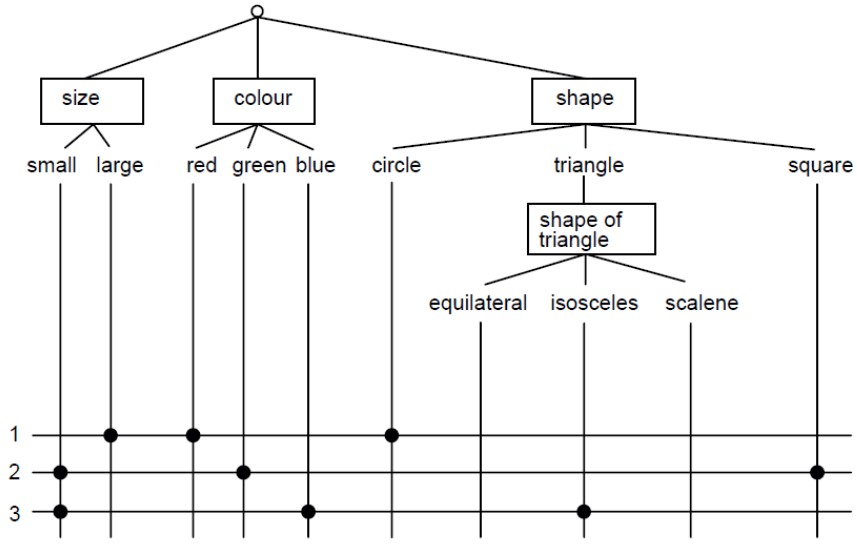


Figura 6 - Árbol de Clasificación [36].

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

4.2. Generación de Casos de Prueba - Caja Blanca

En esta sección se describe el testing de caja blanca para transformaciones de modelos según [26], que toma en cuenta el diseño y la implementación de la transformación para construir casos de prueba.

El autor define diseño como el diseño de alto nivel de la transformación de modelos que tiene como objetivo producir una descripción semi-formal de la transformación, abstrayéndose de los detalles. Este diseño de alto nivel es una descripción incompleta y no ejecutable de la transformación y es especificado con un conjunto de *reglas conceptuales de transformación* r: L->R, donde el lado izquierdo L muestra un subconjunto de elementos del modelo de entrada y el lado derecho R muestra un subconjunto de elementos del modelo de salida.

La implementación es la especificación de la transformación en un lenguaje de alto nivel. Podría ser un conjunto de reglas en un lenguaje de transformación como ATL o instrucciones de un programa Java.

La principal desventaja de las técnicas de diseño de casos de prueba de caja blanca es que están fuertemente acopladas al lenguaje de transformación y deben ser adaptadas o completamente redefinidas si el lenguaje cambia. Los criterios de caja blanca suelen ser más detallados y efectivos que los de caja negra, debido a que se basan específicamente en los pasos para realizar la transformación.

Se introduce la definición de testing de corto alcance, que refiere al testing que consiste en testear cada regla de la transformación, y testing de largo alcance, que refiere al testing que consiste en testear cada transformación realizada.

El artículo se centra principalmente en generar casos de prueba a partir de la especificación de la transformación, tomando como base el diseño de alto nivel de la transformación y en particular las *reglas conceptuales de transformación*.

Para empezar el autor expone algunos errores que pueden cometerse en la definición de las *reglas conceptuales de la transformación*:

- Cubrimiento de meta-modelo: la regla puede haber sido definida sin cubrir completamente los elementos del meta-modelo, por lo tanto puede ocurrir que algunos modelos no puedan ser transformados.
- Creación de modelos sintácticamente incorrectos: La parte derecha (R) de 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 meta-modelo 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.

El artículo describe tres técnicas que pueden ayudar a depurar los errores mencionados anteriormente, la primera y la tercera de ellas corresponderían a testing de corto alcance, la segunda técnica correspondería a testing de largo alcance.

4.2.1. *Testing de cubrimiento del meta-modelo.*

En esta aproximación una *regla conceptual de transformación* se transforma en una plantilla de meta-modelo, facilitando así la creación automática de casos de prueba mediante instancias de plantillas. La plantilla es un modelo abstracto con ciertos parámetros que representa la parte izquierda de la regla de transformación y que puede instanciarse asignando a cada parámetro un conjunto posible de valores.

A modo de ejemplo presentamos en la Figura 7 una regla conceptual, su plantilla de meta-modelo y un par de posibles instancias que se pueden generar a partir de esta plantilla.

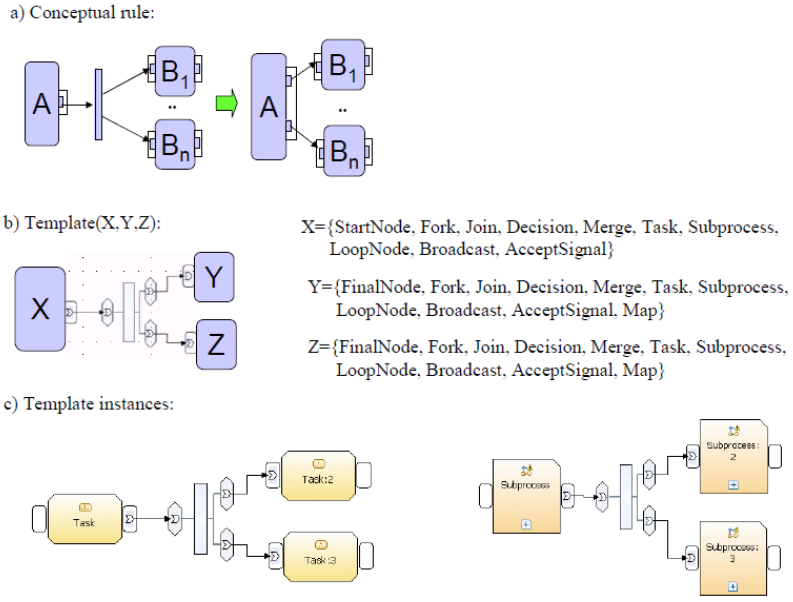


Figura 7 - Regla conceptual, plantilla de meta-modelo y posibles instancias [26].

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 7 b) se deriva de la regla de la Figura 7 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 meta-modelo del lenguaje de modelado de procesos de negocio.

Por último en la Figura 7 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 meta-modelo de entrada al cual aplica. Con el conjunto de plantillas de cada regla se puede asegurar un alto grado de cubrimiento del meta-modelo 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.

4.2.2. *Uso de restricciones para construir casos de prueba.*

Las restricciones de integridad definidas en la especificación de una transformación pueden ser violadas por la aplicación de una o varias reglas de transformación. El objetivo es corroborar el cumplimiento de estas restricciones de integridad luego de aplicada la transformación.

Los pasos a seguir para construir los casos de prueba a partir de las restricciones son los siguientes:

- Identificar los elementos que fueron modificados luego de la transformación.
- Identificar las restricciones que se vieron afectadas por algun 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 7 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 8.

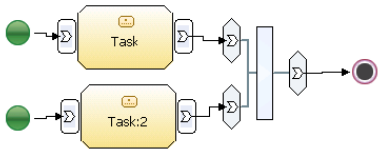


Figura 8 - Caso de prueba para restricciones[26].

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 meta-modelo descrito anteriormente.

4.2.3. *Uso de pares de reglas.*

Con esta aproximación se intenta detectar los errores de confluencia. La propiedad de confluencia determina que la ejecución de reglas en diferente orden en modelos equivalentes debe dar el mismo resultado.

Para la generación de los casos de prueba se toman dos reglas 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.

Veamos un ejemplo: Se considera la regla de la Figura 9.

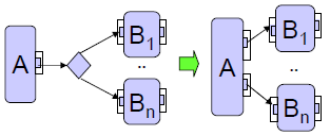
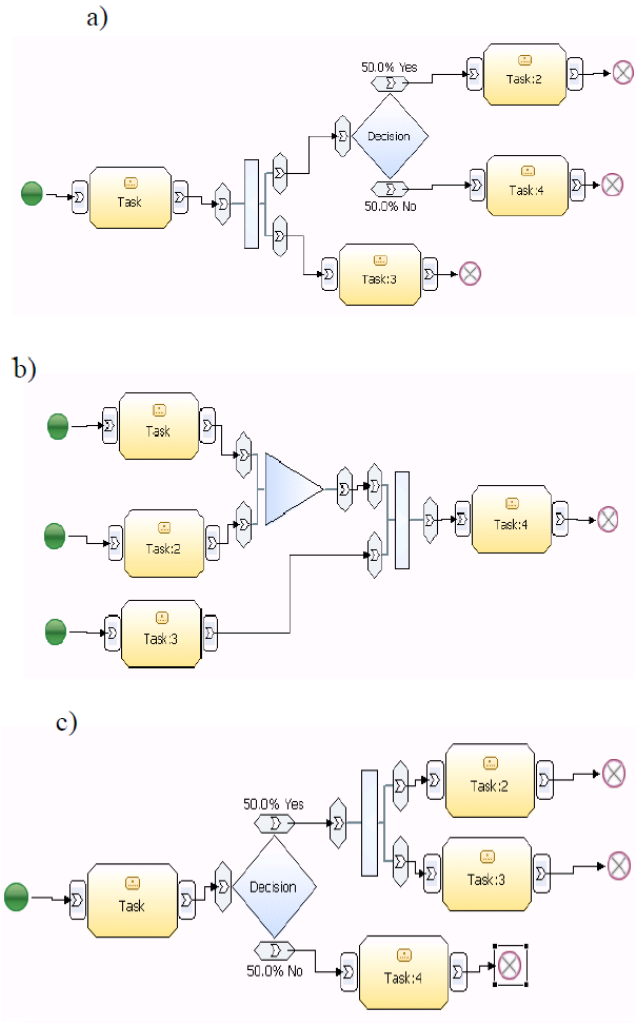


Figura 9 - Regla de eliminación de la decisión[26].

Los casos de prueba que se generan tomando en cuenta la regla de la Figura 7 a) y la regla de la Figura 9 se describen en la Figura 10.



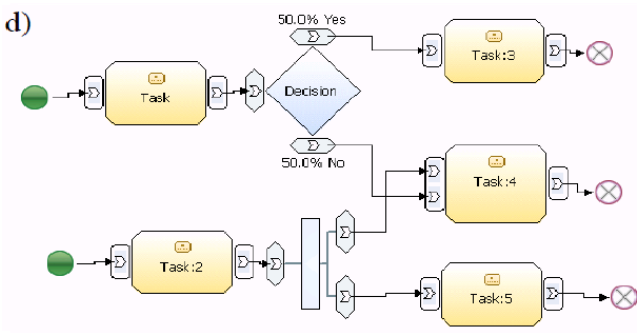


Figura 10 - Casos de prueba para confluencia[26].

Si se observa el caso de prueba de la Figura 10 c) da un error de confluencia, porque si se aplica primero la regla de la Figura 7 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 9 entonces el caso de prueba no se transformaría en un modelo inválido.

4.3. Validación de Casos de Prueba

La validación de los casos de prueba la trataremos en dos partes. La primera, que se aborda en el punto 4.3.1, consiste en verificar la corrección de los datos de entrada a la transformación. La segunda, que se trata en el punto 4.3.2, consiste en evaluar la efectividad de los casos de prueba generados.

4.3.1. Corrección de los datos de entrada.

Los datos de entrada con los cuales realizaremos el testing de la transformación son modelos y por lo tanto la corrección de estos datos implica que deben conformar con el meta-modelo correspondiente al dominio de entrada de la transformación. Además deben cumplir con las restricciones impuestas sobre las cardinalidades del meta-modelo y deben respetar las restricciones formuladas en OCL, que eventualmente pueden existir.

4.3.2. Análisis de Mutación

Según [37] el análisis de mutación consiste en crear versiones de un programa con defectos, estos programas serán llamados *mutantes*, y consiste también en medir que tan eficiente es el conjunto de casos de prueba en encontrar los defectos en estos *mutantes*.

El autor busca que los *mutantes* contengan defectos realistas para que sean efectivos y propone como manera de crear estos mutantes la aplicación de un operador de mutación al programa original. En el análisis desarrollado en el artículo se muestra que bajo el paradigma MDD los defectos que se deben inyectar son aquellos que típicamente pueden ocurrir al desarrollar transformaciones de modelos, ya que la aproximación clásica del análisis de mutación no es apropiada en

este contexto. Entonces, para definir los operadores de mutación nos debemos centrar en las fallas semánticas que pueden ocurrir en la implementación de una transformación. El autor identifica 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 [38], ver Figura 11.

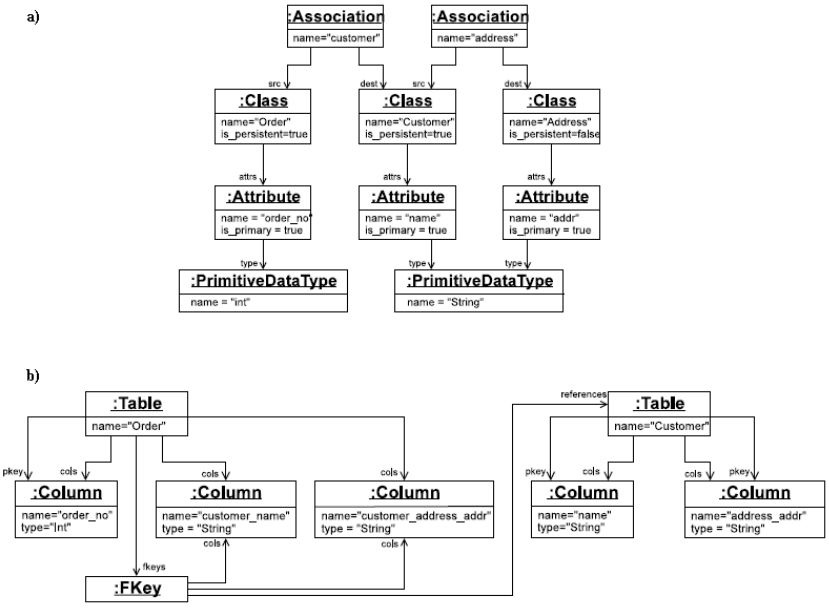


Figura 11 - Transformación de UML a RDBMS[38].

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.

El proceso de análisis de mutación consiste en ejecutar los casos de prueba en el programa original (P) y en los mutantes, luego una función oráculo (que no se describe en el trabajo de los autores pero puede verse en la sección 4.4.2) compara ambas salidas y si los resultados son diferentes entonces el caso de prueba pudo detectar la falla introducida en el mutante. El mutante se considera muerto, si los resultados no difieren. Entonces tenemos dos posibilidades: que el mutante sea equivalente al programa original con lo cual se descarta, o, que el caso de prueba actual no detecte la falla introducida, con lo cual se mantiene el mutante para pruebas posteriores.

En la Figura 12 se muestra a grandes rasgos en que consiste el proceso de análisis de mutación antes descrito.

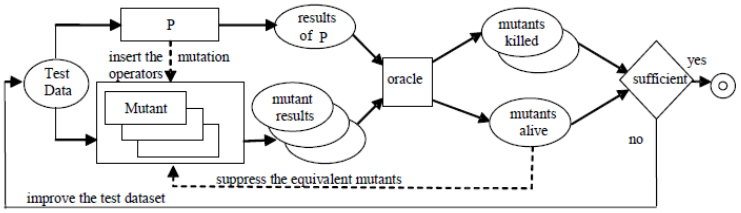


Figura 12 . Proceso del análisis de mutación[37].

El poder de descubrimiento de fallas del conjunto de casos de prueba se puede medir calculando el cociente entre los programas erróneos o mutantes que son detectados y el número total de mutantes no equivalentes.

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.

4.4. Validación de Resultados

La validación de los resultados la trataremos en dos partes. La primera, que se aborda en el punto 4.4.1, consiste en verificar la corrección de los datos de salida de la transformación. La segunda parte, que se trata en el punto 4.4.2, consiste en validar la semántica de la transformación, es decir, si lo transformado tiene sentido con respecto a lo que se quiso transformar.

4.4.1. Corrección de los resultados

Los resultados son modelos y como tales deben conformar con el metamodelo correspondiente al dominio de salida de la transformación. Además deben cumplir con las restricciones impuestas sobre las cardinalidades del metamodelo y eventualmente deben respetar las restricciones formuladas en OCL que puedan estar presentes.

4.4.2. Generación de Oráculo

Validar la corrección de un modelo de salida requiere chequear un gran número de propiedades sobre la estructura y la semántica del mismo, es por ello que es necesaria la generación de un oráculo apropiado al contexto de las transformaciones de modelos. El oráculo chequea la validez de un modelo de salida retornado por la transformación, lo analiza y retorna el veredicto para el caso de prueba. Se puede consultar [39] para ver la discusión sobre los problemas que pueden presentarse al definir un oráculo.

En estudios anteriores reflejados en [40] y [41] 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 [39] 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.

En el siguiente capítulo se establecen 3 técnicas que manipulan y analizan modelos y que pueden ser usadas para implementar la función oráculo mencionada anteriormente y luego 6 funciones oráculo de ejemplo que utilizan estas técnicas mencionadas en el artículo [39], así como también ejemplos de su aplicación en casos concretos.

4.4.3. Técnicas para implementar la función oráculo

Comparación de Modelos

Es común que en la actualidad los modelos se manipulen y representen mediante grafos de objetos. La comparación de modelos en estos casos es equivalente a encontrar soluciones al problema de 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 [40] y [42]. En [20] se analiza la herramienta EMFCompare que implementa la comparación de modelos.

Contratos

Bajo la suposición que disponemos de contratos [43] que especifican las condiciones que se tienen que cumplir antes y después de la ejecución de un método de la transformación entonces 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 [19] se puede ver el proceso de especificación e implementación de un oráculo para transformaciones de modelos basado en contratos expresados en OCL.

Los contratos pueden ser expresados en OCL o en otros lenguajes como por ejemplo Kermeta o ATOM3 [44].

Reconocimiento de patrones ("Pattern Matching")

El reconocimiento de patrones 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 que los objetos 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 meta-modelo. 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. Se verá un ejemplo de "snippet" en la próxima sección de Funciones Oráculo.

4.4.4. Funciones Oráculo u Oráculos

En esta sección veremos las técnicas descritas anteriormente aplicadas a la construcción de oráculos para transformaciones de modelos.

- 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.

Introducimos un poco de notación presente en [39] para ejemplificar.

Llamaremos `mt` al caso de prueba y `mtout` al modelo de salida que resulta de aplicar la transformación a `mt`.El segundo parámetro de la función oráculo será la transformación de referencia a la que llamaremos `R`.

Entonces la función oráculo O_1 queda de la siguiente forma:

```
O1(mtout , (R,mt)) : Boolean is
result := compare(mtout,R(mt))
end.
```

La transformación de referencia podría ser una implementación de la transformación en otro lenguaje.

El autor señala que este oráculo tiene problemas al ser reusado, 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 (`I`) pero hay que tener en cuenta que la transformación debe ser una función inyectiva para que esto sea posible.

Entonces la función oráculo O_2 queda de la siguiente forma:

```
O2(mtout , (I,mt)) : Boolean is
result := compare(mt,I(mtout))
end.
```

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 (`mtexpected`) provisto por el tester.

Entonces la función oráculo O_3 queda de la siguiente forma:

```
O3(mtout , mtexpected) : Boolean is
result := compare(mtout,mtexpected)
end.
```

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(`Cg`) 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.

Entonces la función oráculo O_4 queda de la siguiente forma:

```
O4(mtout , (Cg,mt)) : Boolean is
result := (mt,mtout).satisfies(Cg)
end.
```

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 si 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 [45] 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 (`Cd`). La aserción OCL (`Cd`) 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.

Entonces la función oráculo O_5 queda de la siguiente forma:

```
O5(mtout , Cd) : Boolean is
result := (mtout).satisfies(Cd)
end.
```

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"*(`ms`) . 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(`n`) y un operador lógico(`op`).

Entonces la función oráculo O_6 queda de la siguiente forma:

```
O6(mtout , list{(ms,n,op)}) : Boolean is
result := list.forAll(
compare(nb_match((mtout,ms),n,op)) //compara 2 números dependiendo de un operador lógico op y retorna un booleano.
end.
```

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.

5. Herramientas de testing sobre lenguajes

Luego de ver las distintas técnicas y estrategias de testing que pueden utilizarse para validar las transformaciones de modelos veremos en esta sección una recopilación de algunas herramientas que sirven como soporte para el testing de transformaciones de modelos y que pueden servir de base para la ejecución de tests automáticos. Se da una descripción breve y se enumeran las ventajas de la utilización de las mismas.

5.1. Framework para pruebas de Transformaciones de Modelos.

Un framework que realiza transformaciones de modelos [46] usualmente tiene las reglas de la transformación y las estrategias de aplicación escritas en un lenguaje especial llamado: especificación de la transformación, el cual puede ser gráfico o textual. Este trabajo [46] se enfoca en probar la corrección de la especificación de la transformación.

Los modelos fuente y la especificación de la transformación son interpretados por el motor de la transformación para generar los modelos destino. Luego de transformar un modelo, se tiene que poder realizar lo siguiente:

- Comparar modelos : Se debe comparar el resultado obtenido con el resultado esperado.
- Visualizar las diferencias entre los modelos en una forma gráfica.
- Debug de las especificaciones de la transformación. Un debugger de la transformación de modelos debe entender la representación del modelo así como también tener la habilidad de pasar entre líneas individuales de la especificación de la transformación.

El framework que se presenta, el cual se puede observar en la siguiente figura, realiza la generación de pruebas, la ejecución de pruebas y la documentación de las mismas. Esta compuesto por tres componentes principales: el generador de casos de pruebas, el motor de pruebas y el analizador de pruebas. Como motor de la transformación utiliza C-SAW (Constraint Specification Aspect Weaver) [47]. Este motor esta integrado en GME (Generic Modeling Environment) [48]. Dentro de C-SAW el lenguaje usado para expresar las reglas y las estrategias de la transformación es el ECL (Embedded Constraint Language).

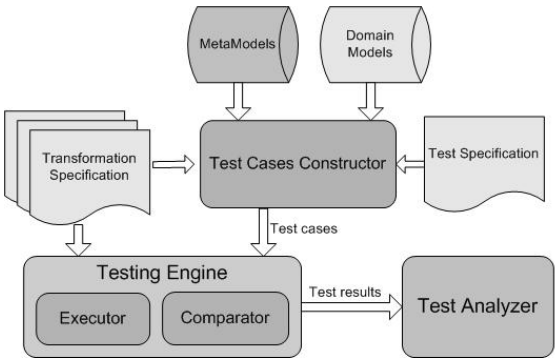


Figura 13. Framework para transformaciones de modelos.[46]

5.1.1. Generador de casos de prueba.

Para este framework la planificación de las pruebas es realizada por el usuario que brinda una especificación textual de la misma. Una especificación de casos de prueba simple define: una especificación de la transformación a probar (archivo ECL), modelo origen, modelo esperado y criterios para determinar si el modelo de pruebas tuvo éxito (comparación entre resultado esperado y resultado obtenido). El constructor de casos de prueba toma la información brindada por la especificación y el modelo origen, y genera casos de prueba ejecutables.

5.1.2. Motor de la transformación.

Para la ejecución del caso de prueba, la especificación de la transformación y el modelo origen son las entradas del motor de ejecución. Este realiza la transformación y genera el modelo destino. El comparador toma este modelo y el modelo esperado y los compara. El resultado de esta comparación pasa al analizador para ser visualizado. Si los modelos son iguales, la prueba es exitosa; de lo contrario se buscan las diferencias entre los modelos.

El meta-modelo es usado para brindar la estructura y las restricciones requeridas para la ejecución de los casos de prueba y la comparación de modelos.

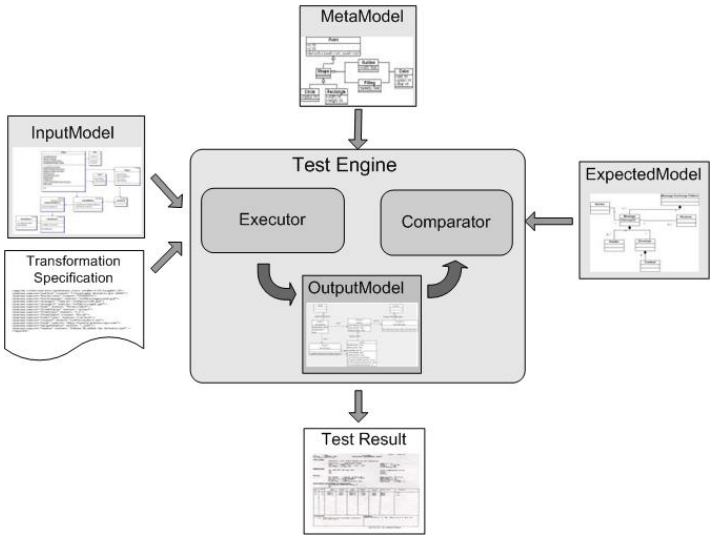


Figura 14.Ejecución de caso de pruebas.[46]

5.1.3. Analizador de pruebas.

El analizador de pruebas provee una interfaz intuitiva para examinar las diferencias entre dos modelos.

Se despliega en un panel el modelo destino junto a sus atributos y en otro panel el modelo esperado junto a sus atributos. Se puede navegar entre ambos paneles observando las diferencias entre los modelos.

5.1.4. Comparación de modelos.

La comparación entre dos modelos siempre tendrá como resultado dos conjuntos: un conjunto de mapeo que contiene todos los pares de elementos que se mapean uno a otro entre los dos modelo y un conjunto de diferencias que es un poco mas complicado de definir.

Las diferencias entre modelo esperado(M1) y el resultante(M2) de la transformación pueden ser, las siguientes:

- Elementos que estén en M1 pero no en M2 . (estructural)
- Elementos que estén en M2 pero no en M1 . (estructural)
- Un valor particular de un atributo es distinto.

Se puede realizar la comparación de los modelos sobre XML, pero reflejar las diferencias entre dos modelos utilizando esta representación no es sencillo. Otra alternativa es utilizar algoritmos de comparación de grafos, pero comúnmente esto es computacionalmente costoso.

Se puede profundizar sobre comparación de modelos en la sección 3 de [46] y en la sección 4 de [49]. En la sección 5 de [46] podemos observar un ejemplo de aplicación del framework.

5.2. Alloy

Alloy [50] es un lenguaje de especificación declarativo y textual basado en lógica relacional de primer orden. Utilizando lógica de primer orden Alloy es capaz de traducir la especificación en una gran expresión booleana que luego es resuelta por un solucionador SAT. Además dada una fórmula lógica en Alloy, el Alloy Analyzer intenta luego encontrar un modelo que satisfaga esta expresión.

El Alloy Analyzer provee dos funcionalidades: Simulación y chequeo de Asserts(Aserciones). La Simulación produce una instancia aleatoria del modelo que conforma a la especificación, esto asegura que el modelo desarrollado es consistente.

Para analizar la transformación el primer paso sería convertir la especificación y las reglas de la transformación de modelos (MT) en una especificación y en reglas equivalentes expresadas en lenguaje Alloy. Esto involucra la conversión del meta-modelo origen y destino y sus respectivas restricciones así como también la conversión de las reglas que definen la transformación entre el meta-modelo origen y destino.

El segundo paso sería utilizar el Alloy Analyzer para detectar fallas en la especificación de la transformación de modelos. Si el Analyzer no puede producir una instancia de la transformación entonces hay una inconsistencia en la definición de las reglas de la transformación. La ventaja es que en el caso de encontrarse inconsistencias estas pueden resolverse de manera directa usando el Alloy Analyzer, ya que la herramienta provee una funcionalidad UnSat Core que resalta las sentencias que llevan a inconsistencias lógicas.

También pueden usarse aserciones para saber si un modelo generado por las reglas de transformación satisface ciertas propiedades. Si la propiedad no es satisfecha el Alloy Analyzer presenta un contraejemplo que es una instancia del modelo destino que viola la propiedad. Luego, dicho contraejemplo puede ser inspeccionado para deducir el error en la definición de la transformación.

Por último veremos algunas desventajas encontradas por los autores del artículo con respecto a la utilización de Alloy:

- No se conocen las posibilidades de escalabilidad de la herramienta y su utilidad frente a grandes meta-modelos y reglas de transformación complejas.
- Puede haber alguna dificultad al momento de traducir de forma directa una transformación de modelos escrita en un lenguaje imperativo o híbrido hacia el lenguaje Alloy, que es un lenguaje Declarativo.
- Los modelos en Alloy son estáticos y definen una instancia de un sistema donde las restricciones son satisfechas. Lo cual quiere decir que no posee las habilidades de una máquina de estados y como

consecuencia nos permite analizar solamente sobre las propiedades estáticas de la transformación. Por ejemplo, no es posible analizar si aplicar la regla 1 y luego la regla 2 dará el mismo resultado que aplicar la regla 2 y luego la regla 1 (propiedad de confluencia).

5.3. Cartier

En el artículo [51] se presenta la herramienta Cartier. El objetivo de la misma es combinar las restricciones que definen los modelos válidos de entrada para la transformación y las restricciones que apuntan a seleccionar casos de prueba para luego guiar la selección de los casos de prueba de forma automática

A continuación se explican las diferentes entradas que maneja Cartier.

El primer parámetro sería el meta-modelo de entrada expresado como un modelo Ecore con restricciones en lenguaje OCL.

El segundo parámetro serían las pre-condiciones de la transformación expresadas en lenguaje OCL.

También se ingresan a Cartier las particiones del meta-modelo que son un conjunto de objetos con propiedades, expresados en el formato de Fragmentos de Modelo, visto en la sección 4.1.1. Estos fragmentos pueden obtenerse utilizando la herramienta MMCC[52] desarrollada por los mismos autores.

Por último se ingresan los objetivos de los casos de prueba (que ingresa el tester de la transformación). Estos objetivos se expresan en lenguaje Alloy y se basan en lo que el tester considera que se debe probar, basándose en los requerimientos de la transformación.

Cartier transforma todas las restricciones antes mencionadas al lenguaje de modelado Alloy. Este programa de lógica relacional implementado en Alloy es transformado en una fórmula Booleana, que se resuelve mediante un SAT solver. En la Figura 15 se resume el proceso de selección automática de los casos de prueba. El cuadro superior de la figura es el marco de trabajo de Cartier, la elipse de mayor tamaño representa el conjunto de todos los modelos de entrada posibles y la elipse de menor tamaño representa la selección de modelos que se obtienen como salida luego de utilizar Cartier.

Luego de ejecutar la transformación se debe validar que los modelos de salida, subconjunto de todos los posibles modelos de salida, cumplan con las post-condiciones de la transformación.

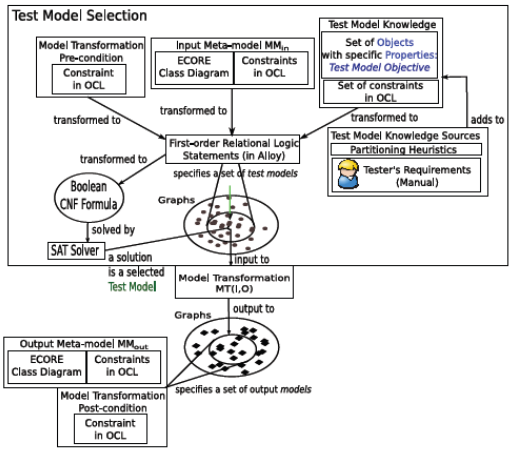


Figura 15 - Vista de Diseño de Cartier. [51]

La transformación de restricciones OCL al lenguaje Alloy presenta ciertas particularidades que pueden verse en [53]. Según los autores, no es posible asegurar que siempre se puedan transformar las restricciones OCL a Alloy.

Resumiendo entonces Cartier invoca Alloy para generar una fórmula booleana en CNF (forma normal conjuntiva) y la resuelve mediante un SAT solver. Luego la herramienta transforma estas soluciones nuevamente a instancias del meta-modelo de entrada en formato XMI. Con esto se logra obtener modelos de test para realizar el testing de la transformación.

Sin embargo en el artículo [54] los autores exponen que los modelos generados por Cartier son triviales ya que no son guiados por ninguna estrategia y por lo tanto analizan y comparan estrategias que sirvan para optimizar la generación automática de los modelos de test.

Los autores analizan la performance de diferentes estrategias utilizadas conjuntamente con la herramienta Cartier. La primer estrategia utilizada fue la generación aleatoria de modelos que se comparó con la generación guiada por fragmentos de modelos obtenidos luego del particionamiento del meta-modelo de entrada.

La métrica la realizaron en base a un análisis de mutación que sirvió como oráculo para determinar cual de los conjuntos de modelos de test generados siguiendo las distintas estrategias fue el que más fallas detectó.

Los resultados que obtuvieron muestran que la técnica de particionar el dominio de entrada es mejor para detectar bugs (87 %) con respecto a una estrategia aleatoria (72 % de detección de errores).

Un resumen de la metodología se describe a continuación y se ilustra en la figura 16 obtenida de [54]: (1) Las entradas a Cartier son un meta-modelo Ecore, los Alloy Facts en el modelo Ecore, Alloy Predicates para las pre-condiciones de la transformación. (2) Cartier genera un modelo Alloy desde el Ecore usando las diferentes entradas que comentamos al inicio de esta sección descritas en [51].

En particular, se agregan también al modelo Alloy comandos run asociados a los predicados de los fragmentos de modelo. Esta tarea la realiza el Transformation Engine que se muestra en la Figura 16. (3) Cartier ejecuta cada uno de los comandos run de Alloy, para obtener los fragmentos de modelo consistentes (4) Cartier invoca los comandos run que usan el motor KodKod [55] para transformar el modelo Alloy a la fórmula CNF, seguido luego de la invocación al SAT solver (ZChaff) para resolver esta fórmula y generar soluciones en Alloy XML.

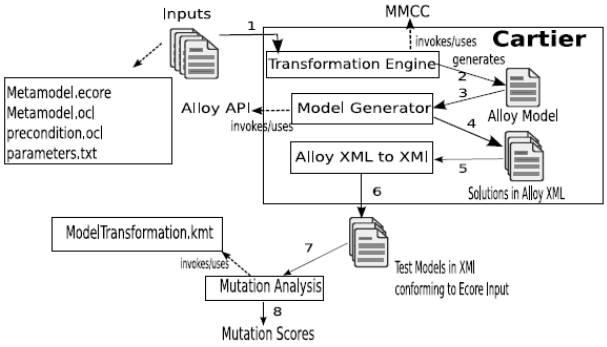


Figura 16. Metodología para medir la calidad de los casos de prueba generados por ambas estrategias [54].

(5,6) Cartier transforma las instancias de XML Alloy en XMI y estos XMI conforman con el meta-modelo Ecore de entrada (7) Obtenemos modelos expresados en XMI a los cuales se les realiza un análisis de mutación, ejecutando la transformación y (8) se da una puntuación.

5.4. Maude - Moment

Maude [56, 57, 58] es un lenguaje de especificación algebraica basado en lógica ecuacional y lógica de reescritura. En el desarrollo del lenguaje se ha intentado potenciar la expresividad, la sencillez y el rendimiento. Este lenguaje tiene otras cualidades que se detallan a continuación, que lo distingue de los demás lenguajes algebraicos: la posibilidad de ejecutar las especificaciones, posee soporte para programación orientada a objetos (OO), permite la creación de módulos parametrizados, brinda la posibilidad del uso de Reflection.

Además, Maude es un lenguaje que permite la ejecución de las especificaciones que hemos creado, lo que permitirá animar los modelos y crear prototipos para comprobar el funcionamiento del sistema.

Moment [59, 60] es una herramienta de gestión de modelos, basada en una aproximación híbrida entre una herramienta formal y un entorno industrial de modelado, Maude y Eclipse Modelling Framework (EMF) [61]. Esta herramienta permite definir modelos como especificaciones algebraicas de una teoría expresada en lógica ecuacional condicional. Proporciona un mecanismo para transformar modelos, basándose en el soporte formal y automático que proporcionan los

sistemas de reescritura de términos, como Maude. Este mecanismo de transformación permite automatizar gran cantidad de procesos software, que permiten entre otras cosas generar modelos específicos de plataforma partiendo de modelos independientes de plataforma.

Moment aprovecha la base teorica brindada por Maude y los conocimientos en el campo aplicado a la ingeniería de software de EMF.

5.5. Jemtte

Jemtte [62] es una extensión de JUnit, una herramienta que simplifica la construcción, organización y ejecución automática de casos de prueba escritos en Java. Un caso de prueba típico en JUnit contiene un número de métodos a probar, previo a la ejecución del caso de prueba, el código a probar se establece en un estado conocido, y al finalizar la ejecución del caso de prueba la salida se compara con valores esperados utilizando aserciones.

Jemtte extiende JUnit con aserciones específicas para poder verificar los modelos de salida de una transformación. Cada una de estas nuevas aserciones recibe un modelo como parámetro de entrada asi como también una o mas XPath Expressions que identifican características que debe cumplir dicho modelo. El lenguaje XPath permite realizar consultas sobre archivos en formato XML de forma de obtener todos los elementos que coincidan con un patrón dado. Jemtte permite utilizar el lenguaje XPath para convertir modelos al formato XML según la especificación XMI de la OMG y luego validar las aserciones.

Jemtte brinda tres nuevos tipos de aserciones: `assertExists(mdl, expr)`, `assertEqual(mdl, val, expr)` y `assertEquivalent(mdl, expr1, expr2)`.

Daremos un ejemplo de la primera: `assertExists(mdl, expr)` - Las aserciones de este tipo se usan para validar la presencia o ausencia de propiedades particulares en un modelo de salida. Cuando se evalúa la aserción, la expresión XPath `(expr)` se evalúa contra la representación en XMI del modelo de salida de la transformación `(mdl)`. Si no hay nodos en el XML que coincidan con la expresión entonces la aserción falla.

Jemtte tiene como ventajas la reducción del esfuerzo y la reducción en la cantidad de código necesario para testear una transformación, se puede usar con cualquier plataforma de modelado que soporte la serialización de modelos a formato XMI y se puede utilizar para realizar el testing de transformaciones implementadas en diversos lenguajes. En particular, el ejemplo referido en el artículo es sobre el lenguaje ATLAS Transformation Languaje (ATL) [63].

En la página del autor no se pudo encontrar información actualizada del proyecto.

5.6. Otras Herramientas

Durante la investigación realizada las herramientas antes mencionadas sobresalieron en mayor o menor medida por su enfoque sobre Testing de Transformaciones de modelos. Actualmente se puede encontrar un número importante de herramientas y estudios aplicables a lenguajes de transformaciones de modelos.

A continuación nombramos alguno de ellos que pueden utilizarse para probar una transformación o validar parte de la misma.

5.6.1. Maudeling

Maudeling es un ambiente de desarrollo de eclipse [64], basado en el lenguaje Maude, que permite realizar operaciones sobre modelos y meta-modelos. Por más información dirigirse a [65].

5.6.2. UMLAUT

Umlaut es un framework genérico de transformaciones que permite realizar manipulaciones complejas sobre modelos UML. Estos modelos se convierten en un árbol de sintaxis abstracta. El motor de transformaciones de Umlaut realiza las mismas en base a exploraciones en el árbol. Por más información en [66].

6. Conclusiones.

Para realizar este reporte se estudiaron y relevaron investigaciones sobre transformaciones de modelos. El paradigma MDE que se estudia desde hace una década aproximadamente contiene áreas como el testing que se encuentran en plena investigación. En particular, la complejidad 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.

Con respecto a las técnicas de testing utilizadas, se observó que la mayoría de las investigaciones se realizan en base a una adaptación de las técnicas de testing tradicionales, técnicas de caja negra y caja blanca, al paradigma de transformaciones de modelos. En nuestro trabajo identificamos tres puntos principales para el testing de transformaciones de modelos: la generación de datos de prueba, la validación de esos datos de prueba y la construcción de oráculos para los datos de prueba. Con respecto al primer punto se examinaron trabajos correspondientes a las técnicas de caja negra y caja blanca aplicados a este paradigma. Aquí se constató que el interés principal de los investigadores reside en las técnicas de caja negra, dado que no existe un lenguaje de implementación de transformaciones de modelos que haya emergido como estándar aunque se están realizando esfuerzos para lograrlo [67] . Para la validación de los casos de prueba se examinaron trabajos basados en la técnica de análisis de mutación. En la sección referida a la construcción de oráculos se vieron posibles implementaciones de los mismos, así como ventajas y desventajas que nos permiten optar entre uno u otro según las características de la transformación en la cual estemos trabajando.

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).

Según la bibliografía de testing de transformaciones de modelos, la técnica de caja negra consiste en observar el meta-modelo 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 de la transformación, con lo cual la técnica de caja blanca en este contexto es una técnica de caja gris. Este nivel sería como 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 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 de QVT por ejemplo.

Existe un número importante de herramientas y lenguajes que dan soporte al desarrollo y validación de las transformaciones de modelos. En particular encontramos que es muy utilizado el lenguaje Alloy, basado en lógica relacional de primer orden, que se utiliza para generar instancias de modelos y verificar si existen inconsistencias en la definición de las reglas de la transformación. Otra herramienta es Moment que se utiliza para gestionar modelos y que hace uso del lenguaje Maude que permite ejecutar la especificación realizada para comprobar el funcionamiento del sistema.

Como puntos abiertos de investigación encontramos que UML Testing Profile no ha sido aplicado en la verificación de transformaciones de modelos. Otro punto que nos pareció importante y podría ser un área a explorar en el futuro es la aplicación de técnicas de testing de programas declarativos al testing de transformaciones de modelos, creadas con lenguajes declarativos.

Apéndice 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.

Referencias

[1] OMG, "Mda guide version 1.0.1," Object Management Group, Tech. Rep., 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] 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.

[4] 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.

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

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

[7] A. G. Miretti, "Testing de software," http://www.inti.gov.ar/software/pdf/ . Último acceso Diciembre 2010.

[8] J. Hughes, "Why functional programming matters," *Computer Journal*, vol. 32, no. 2, pp. 98–107, 1989.

[9] S. Fischer and H. Kuchen, "Systematic generation of glass-box test cases for functional logic programs," in *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, ser. PPDP '07. New York, NY, USA: ACM, 2007, pp. 63–74.

[10] R. S. Pressman, *Software engineering: a practitioner's approach (2nd ed.).* New York, NY, USA: McGraw-Hill, Inc., 1986.

[11] S. Fischer and H. Kuchen, "Data-flow testing of declarative programs," *SIGPLAN Not.*, vol. 43, pp. 201–212, September 2008.

[12] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," *SIGPLAN Not.*, vol. 35, pp. 268–279, September 2000.

[13] 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.

[14] 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.

[15] "Object Management Group (OMG)." http://www.omg.org/. Último acceso Diciembre 2010.

[16] K. Ehrig, J. M. Küster, G. Taentzer, and J. Winkelmann, "Generating Instance Models from Meta Models," in *Formal Methods for Open Object-based Distributed Systems (FMOODS)*, 2006, pp. 156–170.

[17] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon, "Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool," pp. 85–94, 2006.

[18] 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.

[19] 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.

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

[21] Z. Xing and E. Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering ASE '05*. ACM, 2005, pp. 54–65.

[22] "Concurrent Versions System (CVS)," http://es.wikipedia.org/wiki/CVS. Último acceso Diciembre 2010.

[23] "Subversion (SVN)," http://es.wikipedia.org/wiki/Subversion Último acceso Diciembre 2010.

[24] T. Reiter, K. Altmanninger, A. Bergmayr, W. Schwinger, and G. Kotsis, "Models in conflict - detection of semantic conflicts in model-based development." *In Proc. of the 3rd Int. Workshop on Model-Driven Enterprise Information Systems (MDEIS 2007) in conjunction with ICEIS, Funchal, Portugal, June 2007.*, 2007.

[25] 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.

[26] 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.

[27] 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.

[28] B. Baudry, "Testing Model Transformations: A case for Test Generation from Input Domain Models," in *Model Driven Engineering for Distributed Real-time Embedded Systems*, J.-P. Babau, M. Blay Fornarino, J. Champeau, S. Gèrard, S. Robert, and A. Sabetta, Eds. ISTE, 2009.

[29] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Commun. ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

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

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

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

[39] 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.

[40] 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).

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

[42] M. Alanen and I. Porres, "Difference and Union of Models," in *UML*, 2003, pp. 2–17.

[43] 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.

[44] J. de Lara and H. Vangheluwe, "AToM3: A Tool for Multi-formalism and Meta-modelling," *FASE*, vol. 2306, pp. 174–188, 2002.

[45] 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.

[46] Y. Lin, J. Zhang, and J. Gray, "A Testing Framework for Model Transformations," *in Model-Driven Software Development - Research and Practice in Software Engineering. 2005. Springer*, pp. 219–236, 2005.

[47] "Constraint Specification Aspect Weaver (C-SAW)," http://www.gray-area.org/Research/C-SAW/. Último acceso Diciembre 2010.

[48] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments." vol. IEEE Computer, pp. 44–51, 2001.

[49] Y. Lin, J. Zhang, and J. Gray, "Model comparison: A key challenge for transformation testing and version control in model driven software development," pp. 219–236, 2004.

[50] K. Anastasakis, B. Bordbar, and J. M. Küster, "Analysis of Model Transformations via Alloy," *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pp. 47–56, 2007.

[51] S. Sen, B. Baudry, and J.-M. Mottu, "On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing," *in Proceedings of International Conference on Software Testing, Verification, and Validation, 2008*, vol. 0, pp. 328–337, 2008.

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

[53] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "Uml2alloy: A challenging model transformation," in *MoDELS*, 2007, pp. 436–450.

[54] S. Sen, B. Baudry, and J.-M. Mottu, "Automatic Model Generation Strategies for Model Transformation Testing," in *ICMT*, 2009, pp. 148–164.

[55] E. Torlak and D. Jackson, "Kodkod: A relational model finder," *In Tools and Algorithms for Construction and Analysis of Systems (TACAS*, pp. 632–647, 2007.

[56] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott, "Maude 2.1.1 manual. versión 1.0." 2003.

[57] F. J. Lucas, A. Boronat, J. L. Fernandez, J. A. Carsi, A. Toval, and I. Ramos, "Maude como soporte formal para una herramienta de gestion de modelos," *II Jornadas de Trabajo DYNAMICA, junto a JISBD 2004*, 2004.

[58] P. Queralt, L. Hoyos, A. Boronat, J. A.Carsi, and I. Ramos, "Un motor de transformación de modelos con soporte para el lenguaje QVT Relations," *Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06). Sitges, Spain.*, 2006.

[59] I. R. Artur Boronat, Jose A. Carsi, "Una plataforma para la Gestión Formal de Modelos," *Informe técnico. Ref. No: DSIC-II/4/04. Pag: 244. Julio 2004.*, 2004.

[60] A. Boronat, J. A. Carsi, and I. Ramos, "An algebraic baseline for automatic transformations in mda." *Software Evolution Through Transformations: Model-based vs. Implementation-level Solutions Workshop (SETra'04), Second International Conference on Graph Transformation (ICGT2004), Electronic Notes in Theoretical Computer Scienc.*, 2005.

[61] "Eclipse Modelling Framework (EMF)." http://www.eclipse.org/emf/ Último acceso Diciembre 2010.

[62] M. J. Mcgill and B. H. C. Cheng, "Test-Driven Development of a Model Transformation with Jemtte," 2007.

[63] F. Jouault and I. Kurtev, "Transforming Models with ATL," in *In Proceedings of MoDELS Satellite Events*, 2005, pp. 128–138.

[64] "Eclipse," http://www.eclipse.org/ Último acceso Diciembre 2010.

[65] "Maudeling," http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Maudeling. Último acceso Diciembre 2010.

[66] "Unified Modeling Language All pUrposes Transformer (UMLAUT)," http://www.irisa.fr/UMLAUT/. Último acceso Diciembre 2010.

[67] OMG, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, Object Modeling Group, 2011.