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

Reporte Técnico RT 10-07

Estado del arte de verificación de transformación de modelos

Horacio López, Fernando Varesi, Marcelo Viñolo,
Daniel Calegari, Carlos Luna

2010

Estado del arte de verificación de transformación de modelos

Horacio López, Fernando Varesi, Marcelo Viñolo, Daniel Calegari, Carlos Luna

ISSN 0797-6410
Reporte Técnico RT 10-07
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, 2010

Estado del Arte de Verificación de Transformación de Modelos

Horacio López, Fernando Varesi, Marcelo Viñolo, Daniel Calegari, Carlos Luna
Instituto de Computación, Facultad de Ingeniería, UDeLaR

Abril 2010

Resumen

El Desarrollo de Software Guiado por Modelos (Model-Driven Development, MDD) es un enfoque de ingeniería de software basado en el modelado de un sistema como la principal actividad del desarrollo y la construcción del mismo guiada por transformaciones de dichos modelos. Su éxito depende fuertemente de la disponibilidad de lenguajes y herramientas apropiados para realizar las transformaciones y validar su corrección. En relación a este último punto, este documento presenta un relevamiento del estado del arte de los diferentes enfoques y técnicas de verificación de transformaciones de modelos empleados para MDD. Se analizan las principales características de los enfoques existentes, a saber: basado en casos de prueba, model checking y métodos deductivos. Así mismo se estudian las diferentes técnicas existentes para cada enfoque y se presentan las herramientas utilizadas en la bibliografía, ejemplificando su uso.

Índice

1. Introducción

El Desarrollo de Software Guiado por Modelos (Model-Driven Development, MDD [44]) es un enfoque de ingeniería de software basado en el modelado de un sistema como la principal actividad del desarrollo y la construcción del mismo guiada por transformaciones de dichos modelos.

Como se describe en [43] y se resume en la Figura 1, una transformación de modelos toma como entrada un modelo `Ma` que conforma con cierto metamodelo de origen `MMa` y produce como salida un modelo `Mb` que conforma con un metamodelo de destino `MMb`. La transformación es también definida como un modelo `Mt` que a su vez conforma con un metamodelo `MMt`. Todos los metamodelos deben conformar a su vez con un metametamodelo `MMM`. Las transformaciones pueden ser definidas mediante reglas con un enfoque relacional u operacional, como transformaciones entre grafos, entre otros enfoques, pero todos siguiendo básicamente el mismo esquema aquí propuesto. Como se describe en [40] existen diferentes lenguajes que soportan este enfoque, como KM3 [42] y MOF [51] para la representación de los metamodelos, OCL [52] para la especificación de restricciones sobre ellos, y ATL [43], QVT [53] y Tefkat [37] para la especificación de las transformaciones.

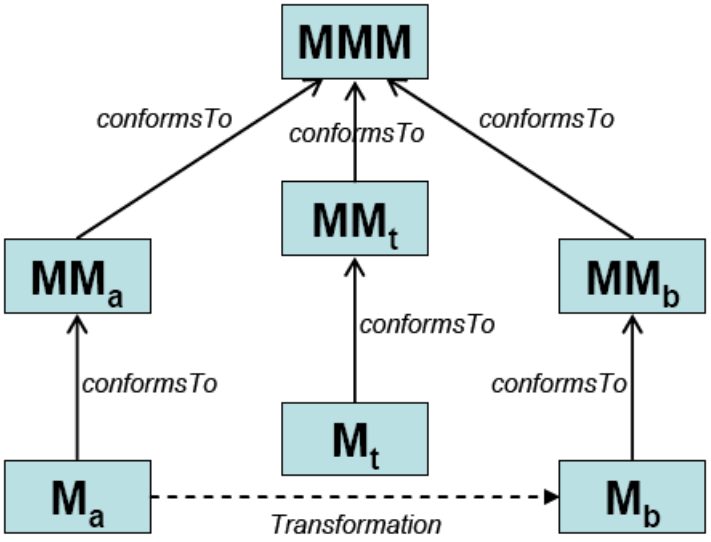

Figura 1: Esquema general de la transformación de modelos

El éxito del paradigma depende fuertemente de la disponibilidad de lenguajes y herramientas apropiados para realizar las transformaciones y validar su corrección. Con respecto a esto último, la validación de una transformación incluye la verificación de la corrección sintáctica de la misma y de los modelos producidos, así como la corrección semántica de la transformación, es decir, la preservación de la corrección del modelo resultante con respecto al correspondiente modelo original. Muy pocas propuestas se ocupan de este último punto. Adicionalmente, la mayoría de las técnicas de transformación se enfocan en la transformación de la estructura, dejando de lado su comportamiento.

El objetivo de este documento es resumir el conocimiento adquirido sobre el estado del arte de las técnicas de verificación de transformación de modelos. Éstas abarcan técnicas de verificación sintáctica y/o semántica de una transformación.

Las técnicas relevadas pertenecen a tres diferentes enfoques de verificación: basado en casos de prueba, verificación de modelos (model-checking) y métodos deductivos. El primer enfoque 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 correctitud en este caso se prueba para un conjunto representativo del dominio con lo que se generan ciertos niveles de confianza en la correctitud esperada para el resto de los modelos a los cuales se aplicará la transformación. El segundo enfoque permite probar en forma automática propiedades sobre los modelos origen y destino, y sobre la transformación en caso de que esta pueda ser representada con un grafo. Las principales ventajas de este enfoque radican en su facilidad de uso y en el hecho de que no se requiere la asistencia del usuario para realizar las pruebas. La verificación formal de transformaciones bajo este enfoque nos permite asegurar la corrección de las mismas utilizando, en general, herramientas semi-automáticas. El último enfoque centra la verificación sobre los metamodelos en lugar de los modelos. Las técnicas dentro de este enfoque utilizan definiciones formales de los metamodelos y expresan la transformación como un conjunto de assertions (ej: fórmulas lógicas del tipo $\forall\exists$), construyendo de esta forma un framework que permite realizar razonamientos formales. De esta forma, la prueba de propiedades a este nivel garantiza que éstas se cumplan para todos los modelos que conformen con dicho metamodelo. Las técnicas de verificación sobre modelos utilizan un menor nivel de abstracción al de las técnicas de verificación formal sobre metamodelos, por lo que el rango de propiedades que se pueden validar es mayor que en el segundo caso, pero no pueden asegurar corrección en términos absolutos [21].

El documento está organizado de la siguiente manera. En la sección 2 de este documento se presentan las técnicas de verificación basadas en casos de prueba. Se exponen las principales dificultades de este enfoque y se presentan las adaptaciones de técnicas conocidas como caja blanca, caja negra y mutation testing. En la sección 3 se presenta el enfoque basado en model-checking. Las técnicas aquí referidas tienen puntos de contacto con las de otras categorías pero resulta de interés distinguir el uso de las herramientas de model-checking. La sección 4 agrupa técnicas basadas en la utilización de métodos deductivos. Este enfoque requiere el uso estricto de lenguajes formales de especificación y de asistentes de prueba para probar en forma absoluta propiedades de una transformación. Por último, la sección 5 expone las conclusiones sobre el relevamiento realizado y la sección 6 presenta una tabla de referencias con los principales temas presentes en este relevameinto y con los trabajos existentes en la bibliografía.

2. Verificación utilizando Casos de Prueba

Las tareas de testing en la actualidad son de vital importancia para la calidad final del producto desarrollado. Para los enfoques tradicionales de desarrollo de software, como el paradigma de orientación a objetos, se han definido y estudiado una amplia variedad de técnicas de testing. Es posible adaptar estas técnicas a MDD con las complejidades propias de este contexto, por ejemplo definiendo los conceptos de testing de caja blanca y negra como la condición de utilizar o no las reglas de transformación para la definición de los casos de prueba. En esencia, las técnicas son similares a las técnicas tradicionales, basándose en la generación de casos de prueba, su validación y su posterior ejecución para verificar transformaciones.

En esta sección se describen en primer lugar dificultades generales de aplicar el enfoque a la verificación de transformaciones de modelos. Posteriormente se presentan diferentes técnicas de generación de casos de prueba y finalmente de validación de los mismos.

2.1. Dificultades generales

En esta sección se describirán en líneas generales los principales problemas que se deben resolver al realizar verificación de transformaciones. Los conceptos fueron tomados de [9] y [68]. Se menciona brevemente la relación de estos conceptos con trabajos relevados en este documento.

Complejidad de los datos

En general, los modelos de entrada y salida de las transformaciones suelen ser grafos de objetos de gran tamaño, cuyas estructuras se encuentran restringidas por sus respectivos metamodelos. A su vez, los metamodelos son estructuras complejas que pueden presentar restricciones en lenguajes (ej: OCL). Esta complejidad de los datos afecta la generación de casos de prueba en términos de memoria y tiempo de ejecución para explorar el espacio de los modelos [9].

Una posible solución para este problema es un enfoque constructivo donde los modelos son construidos primero y las restricciones son verificadas posteriormente. Un ejemplo de este enfoque es aplicado en [59] y se detalla en la subsección 3.2 de este documento. El principal problema de este enfoque es que un gran número de modelos generados no satisfacen la totalidad de las restricciones y por lo tanto no son útiles para los casos de prueba. Si bien existen solucionadores SAT[1]que pueden producir instancias válidas a partir de un conjunto grande de restricciones, presentan limitaciones que suelen resolverse empleando heurísticas para la búsqueda en el espacio de soluciones [33].

La complejidad de datos repercute además negativamente en la verificación basada en oráculos: es difícil crear en forma manual o automática el resultado esperado de una transformación. Una vez generado el modelo destino de la transformación, este debe ser comparado con el resultado esperado. Si los modelos involucrados se basan en grafos, esto equivale a resolver un problema de isomorfismo de grafos (problema de clase NP-completo).

Una forma de solucionar este problema para transformaciones que producen modelos destino ejecutables es ejecutar directamente dichos modelos para comprobar su correctitud. Otra alternativa posible es contar con un oráculo parcial que verifique sólo propiedades específicas del modelo destino en lugar de verificar todo el modelo. Para construir el oráculo parcial se pueden definir patrones que especifiquen las precondiciones y postcondiciones de la transformación. Estos patrones se expresan en templates que describen características esperadas en los modelos origen y destino, que pueden verse como especializaciones de los respectivos metamodelos. Este último enfoque es el empleado en [19] y se detalla en la subsección 2.2.3 de este documento.

[1] Un solucionador SAT determina si las variables de una fórmula booleana pueden ser asignadas de forma tal que la expresión sea evaluada a TRUE

Un último aspecto a considerar es la importancia de definir en forma precisa y procesable los requerimientos de una transformación de modelos. A menudo, los requerimientos tienden a ser informales y no pueden ser procesados por una herramienta que genere los resultados esperados automáticamente. Investigaciones actuales en el campo de la ingeniería de requerimientos exploran técnicas automáticas para extraer fórmulas lógicas a partir de lenguaje natural. Otra solución es solicitar a los desarrolladores que expresen el comportamiento esperado como reglas que puedan ser interpretadas por un oráculo. Este último enfoque es empleado por [59], donde a partir de una especificación en OCL se infieren las operaciones que permiten verificar la consistencia semántica.

Entornos de administración de modelos

MDD se encuentra actualmente en etapa de investigación, por lo que los entornos de desarrollo aún no cuentan con soporte totalmente adecuado para la manipulación de los modelos [24]. Tanto la definición como la visualización de los modelos resulta una tarea difícil que frecuentemente conduce a errores y dificulta el análisis.

La existencia de herramientas sofisticadas para la comparación de modelos es un requerimiento importante para poder realizar pruebas de regresión. De esta forma, es posible verificar que para dos versiones diferentes de una transformación, el resultado de transformar un mismo caso de prueba es idéntico.

Existen varios trabajos en la administración de modelos que pueden incorporarse para facilitar la verificación de modelos. La herramienta EMFCompare [20] para el IDE Eclipse permite realizar comparación de modelos. Esta comparación detecta igualdades y diferencias basadas en la similitud de tipos, nombres, valores de atributos y relaciones. Otra herramienta importante mencionada en este trabajo es CVS Model. Esta herramienta es de código abierto y permite realizar versionado de modelos. No encontramos información actual sobre el estado de este último proyecto.

Heterogeneidad en técnicas y lenguajes de transformación de modelos

Existe una gran diversidad de técnicas y lenguajes de transformación de modelos que abarcan desde técnicas de manipulación directa (ej: transformaciones desarrolladas en Java) hasta lenguajes dedicados como son las implementaciones basadas en el estándar QVT. Esta heterogeneidad se manifiesta también en las herramientas brindadas para especificar y ejecutar las transformaciones. El enfoque empleado es dependiente de la naturaleza del problema a resolver y no existe un único enfoque que contemple todas las posibles necesidades. Esto tiene un fuerte impacto en la definición y selección de técnicas efectivas de testing de caja blanca, que no deben ser dependientes de un enfoque específico [9].

Una forma de solucionar este problema es utilizar técnicas de testing de caja negra para evaluar la calidad de los modelos utilizados en los casos de prueba. La ventaja de esta aproximación es que puede utilizarse con cualquier enfoque de transformación de modelos. Estas técnicas se basan únicamente en el cubrimiento del espacio de modelos definido por el metamodelo origen. Se sugiere además emplear técnicas que consideren la intención de la transformación que se intenta verificar. Una aplicación de la técnica de caja negra puede observarse en [23] y se detalla en la subsección 2.3.1 de este documento.

La principal desventaja de las técnicas de caja blanca es que estan fuertemente acopladas al lenguaje de transformación y deben ser adaptadas o completamente redefinidas si el lenguaje cambia. Sin embargo, 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 crear la transformación. Se puede observar una aplicación de la técnica de caja blanca en [34]. Esta se describe sucintamente en la subsección 2.2.1.

Cubrimiento de metamodelos

Para determinar el valor de un conjunto de casos de pruebas se debe considerar el cubrimiento que éstos realizan del espacio de modelos generados por el metamodelo origen. Resulta imposible hacer un

cubrimiento exhaustivo de este espacio por lo que se requieren técnicas que permitan generar un conjunto representativo de dicho espacio. Además, este conjunto debe considerar la intención de la transformación: algunas transformaciones manejan conceptos puramente estructurales (ej: transformar un modelo de clases UML a un modelo relacional) mientras que otras incorporan conceptos relacionados con su comportamiento (ej: refinamiento).

En [68] se definen en forma incremental las propiedades de cubrimiento de metamodelos basándose en especificaciones MOF. De esta forma es posible definir el cubrimiento utilizando las construcciones estructurales de MOF: clase, atributo, herencia y asociación.

- *Cubrimiento de atributo:* se dice que una regla de transformación cubre un atributo de un elemento del modelo, si la regla referencia ese atributo en el elemento del modelo desde el metamodelo origen al metamodelo destino.
- *Cubrimiento de herencia:* se dice que una transformación de modelos cubre la herencia entre una subclase y una superclase si todos los atributos heredados de la superclase en la subclase son cubiertos.
- *Cubrimiento de asociación:* se dice que una transformación de modelos cubre los extremos navegables de las asociaciones, si todos los atributos de estos extremos son cubiertos. Este cubrimiento requiere que se referencie la multiplicidad del extremo de la asociación.
- *Cubrimiento de elementos:* se dice que una transformación de modelos cubre un elemento del metamodelo, si todos sus atributos (incluyendo los heredados) y extremos navegables asociados son cubiertos.
- *Cubrimiento de metamodelo:* se dice que una transformación de modelos cubre un metamodelo, si todos sus elementos son cubiertos.

En [68] se definen algoritmos generales para verificar estas definiciones de cubrimiento y se presenta un caso de estudio en donde se aplican dichos algoritmos usando Tefkat. El caso de estudio es el de transformación de un modelo de clases UML a un modelo relacional (ver detalles de este caso de estudio en [40]).

2.2. Generación de casos de prueba

La verificación de transformaciones utilizando casos de prueba requiere de modelos origen que respeten el metamodelo y las meta-restricciones definidas. La especificación manual de dichos modelos es una tarea tediosa y compleja, ya que los mismos deben respetar simultáneamente una variedad de restricciones. Las técnicas detalladas en esta sección tienen como objetivo verificar tranformaciones de modelos utilizando conjuntos de casos de prueba generados de forma automática. No todos los trabajos mencionan una herramienta para realizar dicha generación automática, pero presentan las ideas para lograrla.

2.2.1. Verificación de caja blanca

En [34] se presentan experiencias en pruebas de transformación de modelos utilizando técnicas de testing de caja blanca. Se proponen tres técnicas para construir casos de prueba, que se describen a continuación:

- *Testing de cubrimiento del metamodelo*: El testing de cubrimiento de metamodelos es un caso donde la técnica de testing de caja blanca resulta de gran utilidad. En esta aproximación una regla de transformación puede ser transformada en un template de metamodelo. Estos templates de metamodelo permiten crear automáticamente casos de prueba como instancias de los mismos. Por cada regla, un

número de templates pueden ser derivados y juntos pueden asegurar un alto grado de cubrimiento del metamodelo. Si por cada regla se obtiene el cubrimiento del fragmento del metamodelo que involucra, se puede deducir el cubrimiento del metamodelo para toda la transformación.

- *Uso de restricciones para construir casos de prueba*: Típicamente, los metamodelos especifican restricciones de integridad que pueden ser escritas en lenguaje natural u OCL. Estas restricciones pueden ser violadas por la interacción de varias reglas de transformación. Estas violaciones al conjunto de restricciones pueden ser omitidas por las técnicas de testing de cubrimiento del metamodelo. Como la transformación modifica elementos del modelo, es necesario verificar que todas las restricciones que pueden ser violadas debido al cambio, se cumplan luego de aplicar la transformación. Se propone utilizar las restricciones de los metamodelos para construir casos de pruebas cuyo objetivo es descubrir errores que provoquen violación de restricciones de integridad. Este enfoque, que se puede apreciar en [59], será descrito en la sección 3.2.

- *Uso de pares de reglas*: El desafío es construir casos de prueba que lleven a la detección de errores de confluencia. Por confluencia se entiende que la aplicación de reglas en un mismo modelo, o en uno equivalente, brinde el mismo resultado al ser ejecutadas las reglas en diferente orden. La idea básica es tomar dos reglas de transformación y calcular posibles modelos que satisfagan la condición del lado izquierdo de esas reglas. A continuación, se utilizan los elementos de estos modelos que son afectados por las reglas para construir un nuevo modelo. Si este nuevo modelo es sintácticamente incorrecto, se descarta. Si en cambio es sintácticamente correcto, el modelo es utilizado como caso de prueba.

2.2.2. Verificación de caja negra

En [64] el estudio se enfoca en los desafíos para validar transformaciones utilizando técnicas de testing de caja negra. La generación de modelos para testing de caja negra consiste en encontrar modelos origen válidos del conjunto de todos los posibles. Estos modelos deben satisfacer restricciones que incrementen la confiabilidad en la calidad de ellos como casos de prueba, y de esta forma aumentar la capacidad para detectar errores en una transformación.

En dicho trabajo se propone la generación automática de modelos (casos de prueba) que satisfagan las restricciones definidas utilizando la herramienta Cartier [63]. Cartier transforma la especificación del dominio de entrada de una transformación en una especificación expresada en un lenguaje de restricciones llamado Alloy [30][2]. Con la especificación escrita en Alloy, Cartier obtiene fórmulas booleanas en FNC (Forma Normal Conjuntiva), e invoca un solucionador SAT (como por ejemplo zChaff [41]) para generar modelos que se adecuan al dominio de entrada de la transformación. Es decir, el solucionador SAT determina todas las posibles combinaciones de variables que hacen verdaderas las fórmulas booleanas, y a partir de los resultados se obtienen los modelos que satisfacen las restricciones del dominio de entrada de la transformación.

Debido a la infinita cantidad de casos de prueba que pueden existir, se toman distintas estrategias de generación de modelos. Se define como estrategia al proceso que genera predicados Alloy que son restricciones agregadas al modelo Alloy obtenido por Cartier. El modelo Alloy combinado es resuelto y las soluciones son transformadas a modelos origen que satisfacen los predicados.

Las estrategias utilizadas son:

- Estrategia aleatoria: Es la forma más básica, sólo el modelo Alloy obtenido desde la transformación y el metamodelo son usados para generar los modelos. No se suministra ningún conocimiento extra al solucionador para generar los modelos.

[2]Cartier toma como entrada un metamodelo expresado en formato Ecore de Eclipse Modelling Framework. Las restricciones expresadas en lenguaje natural u OCL sobre el metamodelo deben ser transformadas manualmente al lenguaje Alloy.

- Estrategias basadas en el particionamiento del dominio de entrada: Se generan los modelos utilizando criterios de testing para definir particiones, basándose en las propiedades del metamodelo (cardinalidades, dominios, etc) [55]. Se definen criterios de prueba, que se basan en diferentes estrategias para combinar estas particiones de propiedades. Cada criterio define un conjunto de fragmentos de modelo para un metamodelo de entrada. Estos fragmentos son transformados por Cartier a predicados acerca de las propiedades del metamodelo. Para que un conjunto de modelos casos de prueba cubra el dominio de entrada, alcanza con que al menos uno cumpla con cada uno de los fragmentos. Los fragmentos son determinados utilizando los siguiente criterios para combinar particiones:

  - Criterio de todos los rangos: cada rango en la partición de cada propiedad debe ser cubierto por al menos un modelo de testeo.
  - Criterio de todas las particiones: especifica que toda partición de cada propiedad debe ser cubierta por al menos un modelo del conjunto de casos de prueba. Para la generación de los fragmentos se utiliza la herramienta Meta-model Coverage Checker (MMCC) [45].

Finalmente, el trabajo presenta una forma de medir la efectividad de los conjuntos generados mediante análisis de mutaciones en la transformación, mostrando como resultado del análisis un puntaje para cada conjunto de modelos generado.

El enfoque de caja negra también se aplica en [12], donde se detalla un algoritmo para generar en forma automática un conjunto de casos de prueba, tomando como entradas el metamodelo origen y un conjunto de fragmentos de modelo.

Este algoritmo se enmarca en una propuesta general de tres pasos para generar casos de prueba a partir de un metamodelo.

El primer paso consiste en particionar en clases de equivalencia los dominios de los atributos de tipo simple y las cardinalidades en las asociaciones que aparecen en el metamodelo origen. Una posible estrategia de particionamiento consiste en definir las particiones a partir de la estructura o el tipo de dato (ejemplo: si se trata de un atributo string definir como particiones $\{null\}$, $\{""\}$, $\{s/|s| > 0\}$) . Otra estrategia posible es usar el conocimiento que se tiene de la transformación para seleccionar las particiones relevantes. Estas particiones pueden ser provistas por el verificador o inferirse en forma automática de la especificación de la transformación.

A continuación, el segundo paso genera fragmentos de modelo a partir de las particiones. Los fragmentos especifican las partes esenciales del metamodelo y sus valores de interés para la verificación. Pueden ser provistos por el verificador o derivarse en forma automática del metamodelo. La importancia de estos fragmentos dependerá de la estrategia utilizada para combinarlos en el siguiente paso.

Por último, se utilizan los fragmentos para generar modelos que son instancias válidas del metamodelo origen y servirán de casos de prueba. El algoritmo documentado en [12] se centra en el tercer y último paso. Este algoritmo incorpora puntos de variación de forma de permitir establecer una estrategia de verificación. Los puntos de variación son:

- *Tamaño de los modelos generados.* Los valores extremos para este punto se obtienen cuando un modelo único cubre todos los posibles fragmentos y cuando se tiene un modelo diferente para cada fragmento. En el primer extremo, se tiene un modelo muy grande y la localización de las faltas resultará difícil. El segundo extremo presentará problemas de escalabilidad, al manejar un número muy grande de modelos. Por este motivo debe balancearse este valor.

- *Validación del modelo.* Una vez generado el modelo, debe asegurarse que este cumpla con el metamodelo origen. Para esto se hace crecer el modelo hasta que cumpla con el metamodelo utilizando una estrategia "ingenua" o una estrategia de caminos. La primer estrategia no emplea heurísticas y presenta problemas para manejar ciclos originados por asociaciones bidireccionales. La segunda estrategia emplea el algoritmo de Tarjan para evitar problemas de ciclos.

- *Selección de clase de equivalencia y de valores.* Este punto de variación refiere a las diferentes estrategias que se brindan al crear un modelo para seleccionar la clase de equivalencia a cubrir y el valor dentro de la misma.

- *Selección de objetos para asociaciones.* Al manejar las restricciones de cardinalidad, deben asignarse instancias de una clase a los objetos. Este punto de variación permite seleccionar una estrategia de selección de instancias que permite o bien utilizar instancias existentes en el modelo o crear siempre una nueva instancia.

Ajustando los puntos de variación se pueden definir estrategias apropiadas para las necesidades de verificación de un escenario específico.

Una de las limitaciones de este algoritmo es que no maneja restricciones estáticas asociadas al metamodelo origen. Esto es un problema de restricciones lógicas que se suele resolver empleando herramientas tras la generación de los modelos que verifiquen el cumplimiento de estas restricciones. Como trabajo futuro, se planea utilizar algoritmos de inteligencia artificial (ej: algoritmos bacteriológicos) para considerar estas restricciones durante el proceso de generación de modelos.

Es importante señalar que este enfoque fue puesto en práctica en un prototipo denominado OMOGEN (autOmatic MOdel GENerator) en la empresa Télécom de Francia. El prototipo se utiliza para asistir la migración de un sistema de información de gran escala.

El trabajo descripto en [39] presenta un framework para verificar transformación de modelos utilizando un enfoque de verificación de caja negra. Dicho framework está integardo con un motor de transformación existente parar proveer facilidades para la construcción y ejecución de casos de prueba, comparación de las salidas con resultados esperados y visualización de resultados. Para el contexto del trabajo, se utiliza una herramienta de modelado y un motor de transformación específico, GME y C-SAW respectivamente [13]. Utilizando la herramienta de modelado planteada, se describe una técnica de comparación basada en la representación de los modelos como grafos tipados. Se presenta un algoritmo para realizar la comparación entre dos modelos.

Hay tres componentes primarios del framework de verificación planteado: el constructor de casos de prueba (Test Cases Constructor), el motor de pruebas (Testing Engine), y el analizador de la prueba (Test Analyzer), figura 2. El constructor de casos de prueba consume la especificación de la prueba para producir casos de prueba. Los casos generados son pasados al motor de pruebas el cual interactua con C-SAW y GME para ejercitar los casos de prueba especificados sobre la transformación. El motor de pruebas está compuesto por dos módulos: el módulo de ejecución y el módulo de comparación. El módulo de ejecución es el encargado de ejecutar la transformación con los juegos de casos de prueba suministrados. El módulo de comparación compara cada salida de la transformación con el resultado esperado. El analizador de la prueba visualiza los resultados proporcionados por la comparación y proporciona la capacidad de navegar entre las diferencias.

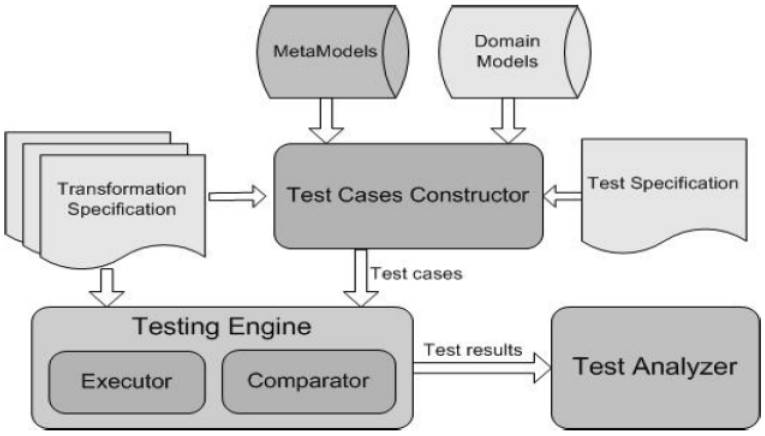

Figura 2: Framework para verificación de transformaciones [39]

2.2.3. Mutation Generation

En [19] se busca verificar implementaciones de transformaciones basadas en grafos mediante la generación de casos de prueba a partir de su especificación. Se enfoca principalmente en la fase de reconocimiento de patrones, considerada la fase más compleja en las transformaciones de grafos.

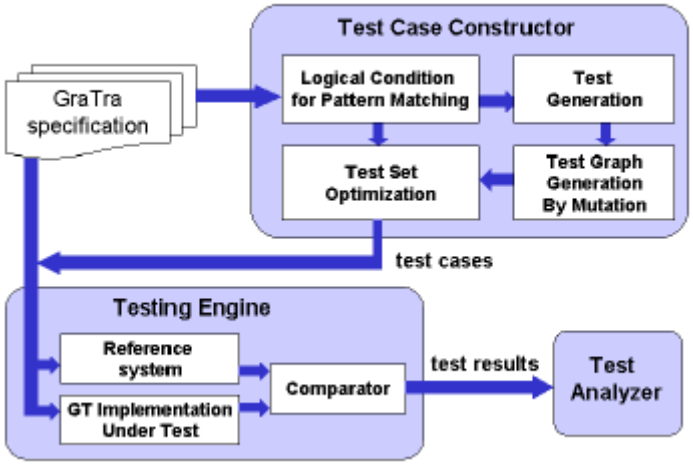

Figura 3: Generación de casos de prueba basada en circuitos combinatorios [19]

La técnica propuesta consiste en generar modelos basados en circuitos combinatorios como base para la creación de casos de prueba. Se propone utilizar modelos con defectos (fault models) para incorporar posibles defectos a la implementación y reutilizar técnicas existentes en la verificación de hardware. Un esquema del proceso de generación de casos de prueba puede verse en la Figura 3. La generación de casos de prueba *(*representada en el diagrama por *Test Case Constructor)* toma como entrada la especificación de la transformación de grafos. Para crear dichos casos de prueba utiliza una representación de circuitos combinatorios derivada de las precondiciones de las reglas de transformación de grafos. El lado izquierdo de

cada regla es representado con expresiones booleanas y a partir de él se construye el circuito combinatorio. A este circuito combinatorio se le inyectan defectos en sus entradas y se crea un conjunto de vectores de testeo binarios (*Test Graph Generation By Mutation*). A partir de estos vectores finalmente se obtiene un conjunto de casos de prueba o modelos de prueba.

El motor de pruebas es el responsable de ejecutar la especificación de la transformación para cada caso de prueba en el sistema de referencia (el cual actúa como oráculo) y en la implementación de la transformación. Luego, un comparador confronta los resultados del reconocimiento de patrones en ambos componentes. El analizador reúne y presenta los resultados que brinda el motor de testeo para cada caso de prueba.

2.3. Validación de casos de prueba

Existen diferentes propuestas que tienen como objetivo validar un cierto conjunto de casos de prueba. En el contexto de MDD, dado un conjunto de instancias del metamodelo origen y la transformación a verificar, se busca conocer que tan bueno es el conjunto para cubrir la mayoría de los casos de prueba.

2.3.1. Verificación de caja negra

En [23] se presenta un conjunto de reglas y un framework para evaluar la calidad de un conjunto de modelos origen utilizados como casos de prueba para verificar una determinada transformación. Se propone un asistente que brinda al usuario los elementos del metamodelo sin cubrir en el caso de prueba.

Se utilizan técnicas de caja negra para no depender del lenguaje en el cual está escrita la transformación. El dominio de entrada se define por el metamodelo origen. La idea básica es evaluar que tan adecuados son los casos de prueba con respecto al cubrimiento de los objetos del metamodelo origen. Estos casos de prueba deben instanciar al menos una vez cada clase y propiedad del metamodelo origen. Para clasificar los posibles valores de las propiedades se utiliza una adaptación de la técnica de testing conocida como partición de equivalencias. Esta técnica consiste en descomponer el modelo origen en un conjunto finito de subdominios no solapados y escoger datos para la prueba de cada uno de los subdominios. Esta forma de evaluar los casos de prueba también es utilizada en [59] y se detalla en [55].

La contribución de [23] son criterios para cubrir los metamodelos origen. Estos criterios capturan todas las nociones importantes del metamodelo y permiten evaluar los casos de prueba de las transformaciones. Dado el metamodelo origen, se pueden elegir diferentes estrategias para la selección de los casos de prueba y se utiliza un framework para verificar si los datos de prueba son adecuados. El framework automáticamente analiza un conjunto de casos de prueba y provee al usuario de información faltante en los modelos de prueba para lograr el cubrimiento de la estrategia seleccionada. Esta información puede ser utilizada para mejorar iterativamente el conjunto de casos de prueba.

La Figura 4 muestra el proceso de análisis del metamodelo origen y el proceso iterativo para la selección del conjunto de casos de prueba (modelos origen) para el testeo de la transformación.

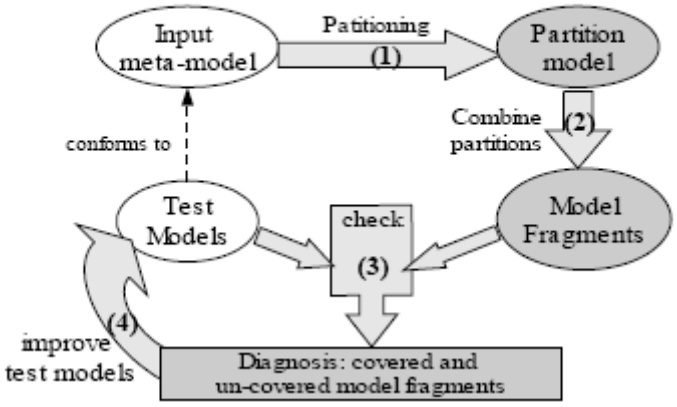

Figura 4: Proceso de selección del conjunto de casos de prueba [23]

Se comienza con el metamodelo origen y un conjunto de modelos de testeo, los pasos que propone el trabajo se enumeran a continuación:

1. Se generan las particiones para todas las clases y propiedades del metamodelo origen (partitioning).
2. Se combinan las particiones para obtener un conjunto de fragmentos del modelo indicando cual será el criterio de cubrimiento que se validará (combine partitions).
3. Se verifica que exista al menos un modelo de testeo que cubra cada fragmento del modelo (check).
4. Si algún fragmento no es cubierto por los modelos seleccionados, se deberá mejorar el conjunto de casos de prueba agregando nuevos modelos que cubran los fragmentos no alcanzados (improve).

Como caso de estudio se propone la transformación de una máquina de estados jerárquica a una máquina de estados plana. La validación consiste en ejecutar la transformación con un conjunto de máquinas de estados jerárquicas (elegido mediante las técnicas planteadas) y verificar que las máquinas de estados planas obtenidas son correctas. Como lenguaje de metamodelado en este caso se utiliza EMOF.

2.3.2. Mutation Testing

Una técnica posible para evaluar un conjunto de casos de prueba es la técnica mutation testing [46, 47]. La técnica consiste en crear sistemáticamente versiones de un programa con faltas (llamadas mutaciones) y chequear la eficiencia de los conjuntos de casos de pruebas para revelar fallas debido a las faltas introducidas. El interés principal de la técnica es proveer un estimativo de la calidad del conjunto de casos de prueba en proporción a la fallas detectadas.

Para ser efectivo, el proceso de mutación debe crear programas con faltas de forma realista. Es decir, posibles faltas que pueda insertar un programador experimentado en el lenguaje de transformación. El trabajo referenciado realiza un estudio de los problemas comunes que tienen los métodos convencionales de mutación de programas para enfoques distintos a MDD, aplicados en un enfoque MDD. La idea planteada para aplicar mutation testing es definir operadores de mutación para el enfoque MDD. Con este propósito, se identifican operaciones comunes que realizan la mayoría de los lenguajes de transformación y para éstas

se definen operadores de mutación. Con estos operadores se generan las mutaciones del programa. En el caso MDD, mutaciones de la transformación.

En [47] se presenta un ejemplo utilizando el lenguaje de propósito general Java. En [46] se presentan ejemplos de la aplicación de los operadores de mutación definidos en Kermeta [22] y Tefkat [37]. En todos los ejemplos presentados, la aplicación de los operadores de mutación no se realiza con ninguna herramienta automática.

3. Verificación utilizando Model Checking

Muchas de las técnicas planteadas en diferentes publicaciones involucran el uso herramientas de model-checking para verificar transformaciones de modelos. Como sucede con otras técnicas, esta técnica varía según la noción de verificación tomada. Podría ser una verificación sintáctica, una verificación semántica, ambas, o podría definirse una noción de verificación para un caso concreto que sea de interés.

A partir de una noción de verificación particular dentro de las mencionadas, se define un conjunto de propiedades que puedan ser comprobadas de forma automática por el model-checker. Las propiedades pueden ser restricciones sobre el modelo origen, sobre el modelo destino o sobre la propia transformación siempre y cuando esta pueda representarse con un grafo.

Las técnicas de model-checking parten del supuesto que el modelo pueda expresarse como un sistema de transiciones (grafo dirigido) que conste de un conjunto de vértices y arcos. En este modelo un conjunto de proposiciones atómicas se asocia opcionalmente a cada nodo. Los nodos representan los estados posibles de un sistema y los arcos evoluciones del mismo mediante ejecuciones permitidas (que alteran el estado). Las proposiciones representan las propiedades básicas que se satisfacen en cada punto de la ejecución.

La principal limitación de esta técnica radica en el tamaño del grafo a verificar, ya que el análisis crece exponencialmente con el número de estados del sistema. Asimismo, el lenguaje para expresar propiedades es restringido a fin de asegurar automaticidad.

3.1. Verificación basada en Redes de Petri

Existe una línea de trabajos que tienen como objetivo tomar modelos de comportamiento y transformarlos a redes de Petri, para luego analizar las propiedades de esta última. Las redes de Petri [61] constituyen un lenguaje de modelado formal y gráfico. Estas cuentan con abundantes y sólidas técnicas de análisis que permiten verificar propiedades estructurales y de comportamiento de un modelo. De esta forma, el análisis sobre una red de Petri permite estudiar propiedades sobre el modelo original que fue transformado a dicha red.

En [60] se integran un conjunto de herramientas a un framework que permite traducir modelos utilizados para el desarrollo de software a una especificación formal y posteriormente realizar análisis que permitan validar los modelos. Como caso de estudio se presenta una transformación de un workflow (BPMN [69]) a una red de Petri. El workflow es indirectamente verificado utilizando herramientas de análisis basadas en la teoría sobre redes de Petri.

En [67] se puede observar la verificación de una propiedad semántica en la transformación de máquinas de estado UML a redes de Petri. Se presenta una técnica automatizada para verificar formalmente transformaciones por medio de model-checking. La misma se basa en probar que propiedades especificadas para los modelos origen y destino se preservan al realizar la transformación. Por lo tanto, la noción de verificación utilizada consiste en definir propiedades en el modelo origen y expresar sus equivalentes en el modelo destino, para luego ser verificadas por medio de model-checking. La propiedad verificada para el caso de estudio prohíbe la activación simultánea de dos subestados diferentes desde un mismo estado OR en la máquina de estados.

La Figura 5 muestra una visión conceptual de la idea de verificación semántica para este trabajo. Las características principales se enumeran a continuación.

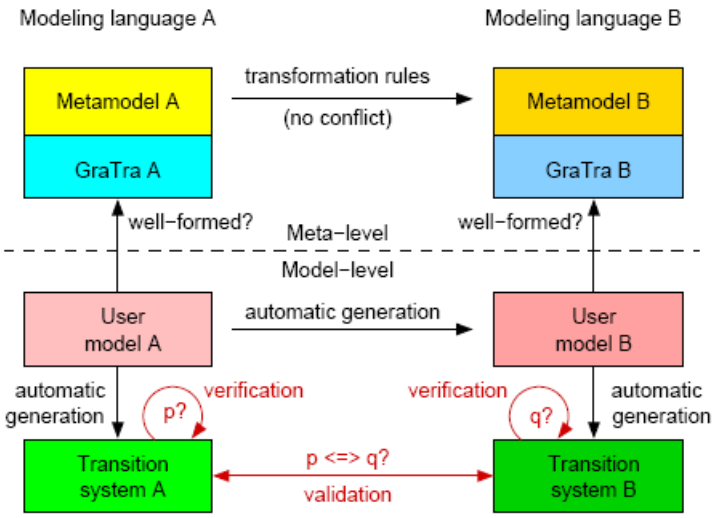

Figura 5: Modelo conceptual de la técnica [67]

1. Cada lenguaje de modelado (*A y B*) debe ser definido utilizando un metamodelo (*Metamodel A, Metamodel B*) y técnicas de transformación de grafos (*GraTra A y GraTra B*).
2. La transformación debe especificarse mediante un conjunto de reglas de transformación de grafos no conflictivas.
3. Para cada instancia bien formada del modelo origen A se deriva automáticamente la instancia correspondiente del modelo destino B (*User model A y User model B*).
4. Se genera un sistema de transiciones equivalente para los modelos origen y destino (*Transition system A, Transition system B*).
5. Se expresa una propiedad semántica p en el modelo de origen A.
6. La propiedad p es verificada automáticamente sobre el sistema de transiciones A usando un model-checker.
7. Se transforma la propiedad p en la propiedad q, expresada en el lenguaje de destino. Este paso puede ser realizado automáticamente utilizando el programa de transformación, pero un experto del dominio debe validar que la propiedad q es equivalente a la propiedad p.
8. Finalmente, la propiedad q es analizada sobre el sistema de transiciones B utilizando también un model-checker. Si la verificación es exitosa, entonces se puede concluir que la transformación es correcta respecto de las propiedades p y q. Si la verificación falla, la propiedad p no es preservada por la transformación.

Las herramientas utilizadas para el caso anteriormente mencionado son: VIATRA para generar la transformación y SAL (Symbolic Analysis Laboratory) para generar los sistemas de transiciones.

3.2. Verificación utilizando OCL

En [59] se presenta una aplicación de model-checking para realizar verificación semántica en un refinamiento de modelos basados en estados. Se define la consistencia semántica de una transformación como la preservación de la correctitud del modelo destino con respecto al modelo origen. El refinamiento es entonces correcto si el comportamiento observable de un modelo permanece incambiado.

Para esto se trabaja con una especificación abstracta y una específica. Si estas especificaciones se basan en estados, es posible definir una relación R entre ambas y verificar un conjunto de condiciones de simulación. Se utilizan reglas de simulación semántica no-bloqueante: toda operación posee precondiciones y si estas no se cumplen, el comportamiento de la operación es indefinido. Bajo estas hipótesis se definen tres condiciones de simulación:

1. *Inicialización.* Cada estado inicial concreto debe estar relacionado mediante R con un estado inicial abstracto.

2. *Aplicación.* Las precondiciones de una operación pueden debilitarse en el refinamiento: la operación concreta debe ser aplicable en todos los casos en que la abstracta lo es, pero puede definir estados adicionales.

3. *Correctitud.* Se requiere consistencia del compartamiento entre las operaciones abstractas y concretas solo en aquellos estados donde la operación abstracta es aplicable. En otras palabras, se requiere que una operación abstracta pueda ocurrir en un estado abstracto cuando en el estado concreto asociado por R puede ocurrir la operación concreta correspondiente. Además, todo estado concreto alcanzado tras una operación concreta debe estar relacionado mediante R a un estado abstracto alcanzado al realizar la operación abstracta.

Estas condiciones de simulación pueden expresarse formalmente en lenguajes formales como Z [65]. Sin embargo, estos lenguajes suelen ser complejos y requieren herramientas de análisis formal que no se utilizan frecuentemente en la práctica. En cambio, en [59] se propone utilizar UML y OCL como único formalismo para realizar una verificación dinámica tras aplicar la transformación. Estos lenguajes son lo suficientemente expresivos para visualizar la transformación, pero carecen de la semántica formal necesaria para la verificación. Para superar esta limitante se propone extraer en forma estática información de la especificación de la transformación, para luego verificar en forma dinámica tras aplicar la transformación. La verificación dinámica es imprescindible para asegurar que se obedecen las condiciones del refinamiento. Es aconsejable complementar esta verificación con verificación estática (ej: chequeo de tipos) al momento de escribir la transformación.

En la figura 6 se muestra un ejemplo de refinamiento especificado utilizando UML y OCL. Mediante OCL se expresan las condiciones iniciales, al igual que las precondiciones y postcondiciones de los métodos de las clases. A partir de esta especificación, se generan en forma automática operaciones de OCL que permiten verificar las condiciones de simulación mencionadas previamente. Los detalles respecto a la generación automática de las operaciones de verificación se encuentran en la sección 3 de [59].

Una vez generadas estas operaciones, se verifica la transformación utilizando la técnica de micromodelos de software descritas en [31]. Para ello se define una cota sobre el tamaño de los modelos y se comprueba si todos los modelos de dicho tamaño satisfacen la propiedad a demostrar. Esto permite aprovechar los mecanismos de model checking, a la vez que al trabajar sobre un conjunto de modelos se tiene cierta generalidad: si se satisface la propiedad, tenemos cierta confianza en que la propiedad se preserva para todos los modelos (aunque podría existir un modelo más grande que no la cumpla). Si en cambio no se se satisface la propiedad, tenemos certeza de que la propiedad no se cumple. Estas conclusiones parten de la hipótesis de alcance pequeño de Daniel Jackson (*small scope hypothesis* [31]), la cual establece que las respuestas negativas tienden a ocurrir en los modelos pequeños.

Como se mencionó en la sección 2.3, se debe emplear la técnica de partición de categorías para seleccionar aquellos modelos representativos de un conjunto posiblemente infinito. Si bien es posible

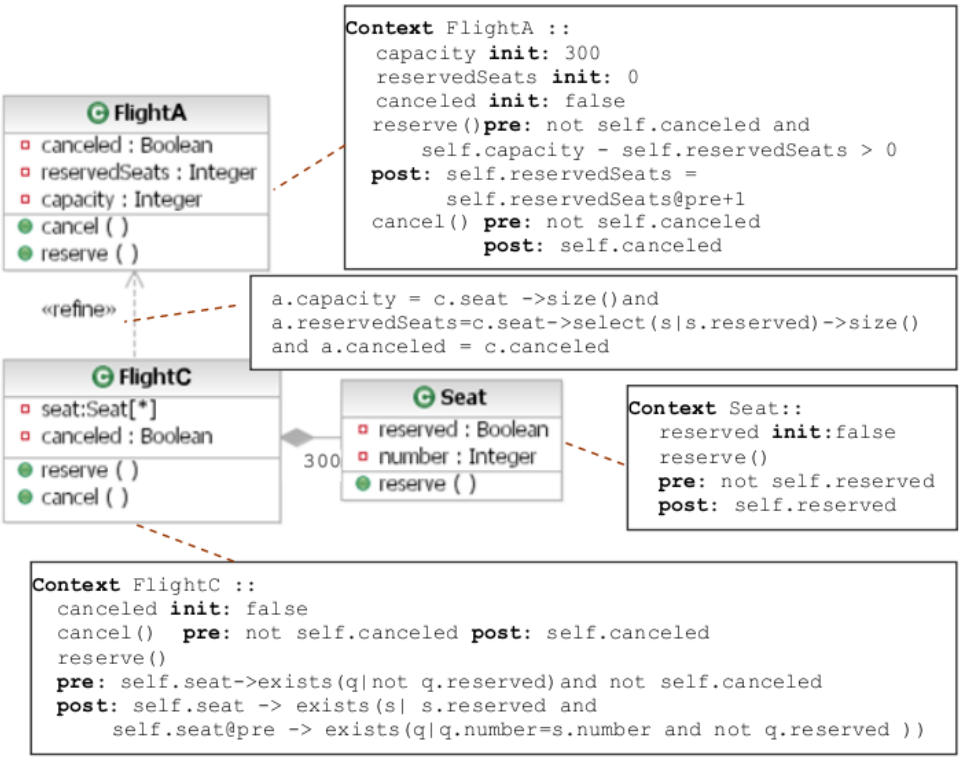

Figura 6: Especificación del refinamiento mediante UML y OCL [59]

generar en forma automática las particiones a partir del tipo de propiedad a verificar, suele ser necesaria la intervención del desarrollador para determinar los valores que posean un significado especial.

En la misma línea que [59], la técnica presentada en [48] propone identificar la correspondencia entre las estructuras de los metamodelos origen y destino definiendo los elementos críticos (denominados *pivots*). Luego se especifican las condiciones de esta correspondiencia utilizando un lenguaje como OCL y se establecen links que mapean los pivots con los elementos de los modelos origen y destino. Por último, se utiliza un model checker que mediante los links definidos verifica que se cumplan las condiciones de correspondencia.

En [16] se presenta un método y su implementación para verificar que el resultado de una transformación endógena es correcto respecto a la especificación de la transformación. El método está basado en contratos de software escritos en OCL, lo que hace al método independiente de las herramientas de modelado y transformación. Los contratos son total o parcialmente generados por una herramienta.

Un contrato de transformación es definido por tres conjuntos de restricciones:

1. Restricciones en el modelo origen: son restricciones que deben ser respetadas por el modelo origen que lo hacen adecuado para ser transformado.

2. Restricciones en el modelo destino: son restricciones generales (independientes del modelo origen) que deben ser respetadas por un modelo para ser un resultado válido de la transformación.

3. Restricciones sobre la evolución de elementos: son restricciones que deben ser respetadas en la evolución de elementos entre el modelo origen y el modelo destino, para asegurar que el modelo destino es el resultado correcto de la transformación de acuerdo al modelo origen.

El primer paso para la verificación es concatenar el modelo origen y el modelo destino en un modelo global, para luego definir un conjunto de invariantes en OCL asociadas al modelo global. Para realizar de forma automática esta tarea, se ha desarrollado una herramienta en el contexto de Eclipse/EMF que realiza la concatenación de modelos para cualquier metamodelo. La herramienta toma como entrada los metamodelos origen y destino. En primera instancia agrega una clase denominada *ModelReference* al metamodelo global que contiene un atributo que indica si el elemento pertenece al modelo origen o destino. Todas las clases del metamodelo son modificadas para extender esta nueva clase. Como segundo paso, la herramienta agrega todos los elementos de los modelos origen y destino en un modelo global que conforma al metamodelo modificado. Durante este último paso cada elemento se marca como perteneciente al origen o destino. Como resultado, la herramienta da como resultado el metamodelo modificado y el modelo global conteniendo todos los elementos de los modelos origen y destino.

El segundo paso de la verificación es expresar los contratos mediante restricciones OCL. Para los modelos origen y destino las restricciones se expresan de la forma habitual, a partir de su especifiación. Para expresar las restricciones sobre la evolución de elementos, se debe trabajar sobre el modelo global indicando que un elemento en un modelo se mapea con un elemento en el otro. Para ello es necesario determinar cuáles son los elementos del modelo origen que tienen su equivalente en el modelo destino, y definir los mapeos correspondientes. Para ayudar al usuario a realizar dicha tarea, existe una herramienta que analiza cualquier metamodelo y brinda al diseñador la posibilidad de elegir para cada elemento del metamodelo el tipo del mapeo entre elementos. Luego la herramienta genera todas las funciones OCL de los mapeos especificados. El diseñador podrá completarlas o modificarlas para definir partes específicas.

Todas las funciones e invariantes que forman cada contrato son escritas en OCL estándar, lo que las hace verificables por cualquier evaluador OCL que implemente el estándar y sea capaz de leer modelos Ecore. De esta forma se podría utilizar ATL para realizar las validaciones. Este método no aplica a transformaciones exógenas.

Un tercer trabajo basado en la utilización de invariantes expresadas en OCL es el presentado en [14]. En este trabajo se parte de una especificación declarativa de la transformación de modelos para obtener en forma automática restricciones expresadas en OCL. Estas restricciones establecen las condiciones que deben preservarse entre el modelo origen y destino para satisfacer la definición de la transformación. Se define al *modelo de transformación* como el conjunto de metamodelos origen y destino, y las restricciones OCL previamente mencionadas. Esta conceptualización permite utilizar las herramientas existentes para verificación y validación de UML/OCL: solucionadores SAT, solucionadores de restricciones o asistentes de pruebas, como los presentados en la sección 4 de este documento.

El trabajo referido presenta como caso de estudio una transformación de un modelo de clases a un modelo relacional y ejemplifica el enfoque utilizando los lenguajes QVT relations y Triple Graph Grammars (TGG) [62]. En las secciones 3 y 5 de [14] se detalla como lograr la extracción de las restricciones a partir de una especificación en cada uno de estos lenguajes. Esta extracción es una transformación en sí misma de un modelo TGG (o QVT) a un modelo UML/OCL.

Utilizando la herramienta UMLtoCSP [66] se verifica que estas restricciones se cumplan en el modelo de transformación. De esta forma se logran verificar propiedades de correctitud como la aplicabilidad de una regla o de la transformación completa (esto se demuestra encontrando un modelo origen en que sea aplicable). Es posible además utilizar la herramienta para realizar validaciones, ya sea generando pares de modelos origen y destino, o generando un modelo destino a partir de una especificación parcial del modelo origen. El procedimiento consiste en utilizar UMLtoCSP para traducir el modelo de transformación en un problema de satisfacción de restricciones, y tras resolverlo encontrar los modelos deseados.

Este trabajo aporta criterios para la selección de herramientas de verificación de UML/OCL y un análisis general de como cada categoría de herramientas contribuye a la verificación y validación de una transformación. Además, presenta una formalización de propiedades de verificación a nivel de regla y a nivel de toda la transformación.

3.3. Verificación en compiladores de modelos

Dado el crecimiento del desarrollo guiado por modelos, este ha provocado que los compiladores de modelos asuman el rol de los compiladores de los lenguajes de propósito general. Así la fiabilidad en ellos se ha vuelto de suma importancia.

En [25] se propone una manera de verificar transformaciones para eliminar la brecha entre el lenguaje en el que es especificada la transformación y los procedimientos necesarios para verificar propiedades esperadas en la transformación. El método involucra la traducción automática de los metamodelos de entrada y salida en un formalismo en el cual, procedimientos de decisión son definidos para responder si una transformación satisface las propiedades de interés. Por ejemplo, propiedades básicas deseables a verificar en una transformación podrían contemplar que todas las sentencias de salida pertenezcan al lenguaje objetivo y que la función de transformación cubra todo el lenguaje de entrada para el cual fue diseñada.

El trabajo tiene como objetivo eliminar la ardua tarea de preparar una especificación que sea posible de verificar mediante model-checking para transformaciones no triviales reutilizando datos de la semántica estática (EMOF + reglas OCL) contenida en los metamodelos de los lenguajes participantes en la transformación. Además, muestra cómo desarrollar algoritmos robustos de transformación empleando un motor para model-checking. En particular se investiga el lenguaje +CAL (lenguaje de especificación diseñado para remplazar pseudo-código para escribir algoritmos de alto orden de descripción) y el model-checker TLC.

El método de verificación se rige por los siguientes pasos:

1. Automáticamente se traducen las definiciones contenidas en los metamodelos EMOF + OCL en el lenguaje +CAL. Aquí se obtienen los árboles de sintaxis abstracta (ASTs en inglés) correspondientes.

2. Se expresa la transformación como un algoritmo +CAL que opera sobre los ASTs.

3. Si es apropiado, se anota la transformación con supuestos sobre la entrada que vayan mas allá de las restricciones expresadas en los metamodelos. El que invoca la transformación es responsable de satisfacer estas suposiciones (similar a las pre-condiciones en el diseño por contratos).

4. Se anota la transformación con características indispensables sobre el estado del sistema luego de cada ejecución exitosa de la transformación. El algoritmo es responsable de satisfacer estas características (similar a las post-condiciones en el diseño por contratos).

Una vez que el algoritmo es especificado, en tiempo de diseño el model-checker puede detectar situaciones donde la transformación no termina o termina sin cumplir las propiedades de interés. Esta técnica exige que los metamodelos sean expresados en EMOF + OCL. Como se mencionó, se utiliza el model-checker TLC y el lenguaje +CAL para especificar formalmente los metamodelos y las propiedades.

3.4. Verificación utilizando sistemas de transformaciones de grafos tipados

En [27] se presenta una técnica aplicada a los refinamientos de modelos de comportamiento. En este trabajo se representan los modelos utilizando metamodelos dinámicos, formalizados como un sistema de transformaciones de grafos tipados. Informalmente, un sistema de transformaciones de grafos tipados consiste en:

1. un grafo tipado que define el vocabulario de los elementos del modelo permitidos y sus relaciones,

2. un conjunto de restricciones, y

3. un conjunto de reglas de transformación de grafos.

Los grafos tipados y las restricciones pueden ser vistos como el análogo a un metamodelo estático. En el caso de sistemas dinámicos, una instancia de un grafo modela el estado del sistema en un punto específico del tiempo. Para modelar las evoluciones de un sistema, el metamodelo dinámico provee reglas de transformación de grafos. Las mismas son especificaciones ejecutables que pueden ser usadas para definir reglas de transformación que especifiquen posibles computaciones, comunicaciones u operaciones que puedan ser aplicadas a estados e impliquen transiciones a nuevos estados.

Para la verificación del modelo de comportamiento, se deben tener en cuenta los sistemas de transición que se pueden generar dentro de la dinámica subyacente del metamodelo. Formalmente, se considera un grafo G instancia del sistema abstracto y una instancia G' del sistema concreto, donde G' es un refinamiento de G. Luego, todo camino $G \rightarrow G_1 \rightarrow ... \rightarrow G_n$ en el sistema de transiciones abstracto tiene su correspondiente camino $G' \stackrel{*}{\rightarrow} G'_1 \stackrel{*}{\rightarrow} ... \stackrel{*}{\rightarrow} G'_n$ en el sistema de transiciones concreto, siendo G'_i el refinamiento de G_i, para todo $i = 1...n$. Cada paso en el sistema abstracto se corresponde con una secuencia de pasos en el sistema concreto. Un paso simple $G_i \rightarrow G_{i+1}$ en el sistema de transiciones abstracto es "refinado" por una secuencia de transiciones $G'_i \stackrel{*}{\rightarrow} G'_{i+1}$ en el nivel concreto, dado que pueden ser necesarios varios pasos. La idea planteada es formular los caminos que definen los sistemas de transiciones del refinamiento, como un problema de "alcanzabilidad" en el sistema de transiciones concreto. Es decir, verificar que para cada paso de la transformación $G \rightarrow H$ en el sistema abstracto, exista una secuencia de $G' \stackrel{*}{\rightarrow} H'$ en el sistema concreto que refine H.

En [11] se presenta un enfoque para formalizar y verificar transformaciones del estilo QVT que reutilizan el concepto principal de los sistemas de transformaciones de grafos. Debido a la naturaleza de los modelos en la teoría de grafos, los sistemas de transformaciones de grafos y su soporte tecnológico proveen un ambiente adecuado para formalizar y verificar trasformaciones de modelos. Específicamente, la técnica consiste en formalizar la transformación de modelos en una lógica de reescritura, para así poder verificar la transformación. El trabajo presenta una formalización ejecutable para transformaciones del estilo QVT en lógica de reescritura, donde la transformación puede ser ejecutada y analizada mediante model-checking de invariantes y por propiedades lógicas temporales [3]. El trabajo se enfoca en transformaciones endógenas, pero las ideas pueden extenderse fácilmente.

A nivel semántico, se presenta cómo la semántica algebraica para metamodelos de tipo MOF puede extenderse en MOMENT2[4] [10] a una semántica en lógica de reescritura. Esta semántica en lógica de reescritura es el homólogo algebraico de un sistema de transformación de grafos, en el sentido que:

- los modelos vistos como grafos se corresponden al mismo modelo visto en términos de tipos de datos algebraicos para un metamodelo dado, y
- las reglas de transformación de grafos se corresponden a reglas en lógica de reescritura sobre sus representaciones algebraicas.

El trabajo referenciado presenta además algunos detalles interesantes en la relación entre las gramáticas de grafos y la lógica de reescritura. Uno de ellos es el soporte consistente de tipos algebraicos complejos en la lógica de reescritura, ya que frecuentemente los atributos de los nodos en los grafos son de tipos de datos complejos.

3.5. Verificación semántica dependiente e independiente de la plataforma

Existen una variedad de publicaciones en las cuales se utiliza model-checking junto con otras técnicas para verificar transformación de modelos. En esta sección se presentan trabajos en los cuales se utiliza un enfoque *híbrido*, o sea técnicas que no sólo involucran model-checking.

[3] El artículo además propone Maude [17] como lenguaje de programación basado en lógica de reescritura, para realizar análisis de alcanzabilidad.

[4] MOMENT2 es una herramienta que soporta transformaciones del estilo QVT y su análisis via model-checking.

En [32] se presenta una técnica que utiliza model-checking y verificación basada en casos de prueba. El trabajo se basa en la diferenciación de características independientes de la plataforma (PI - Platform Independent) y específicas de la plataforma (PS - Platform Specific) del sistema bajo prueba.

Se definen dos términos para luego describir la técnica.

- La propiedad PIS (Platform Independent Semantics) de un lenguaje de programación cubre los aspectos del comportamiento del programa que no dependen de la plataforma en la que se ejecuta el programa
- La propiedad PSS (Platform Specific Semantics) de un lenguaje de programación cubre los aspectos del comportamiento del programa que son específicos de la plataforma en la que se ejecuta el programa.

En este trabajo se presenta un marco unificado que muestra la correctitud de un sistema embebido utilizando dos pasos. En primer lugar se verifica formalmente la semántica independiente de la plataforma (PIS) mostrando que el sistema cumple con la especificación. Luego, se verifica mediante casos de prueba, si la semántica específica de la plataforma (PSS) del sistema cumple los requerimientos definidos en la especificación (en particular, se utiliza la técnica de mutation testing).

Se plantea como caso de estudio un ejemplo sobre el dominio automotriz, donde el sistema es modelado con Matlab Simulink y el código (en lenguaje C) es generado automáticamente con TargetLink. Este código define la PIS del sistema, que es luego compilado a código objeto, para ser ejecutado en una plataforma específica.

La aplicación de la técnica se realiza en dos pasos.

1. Verificación formal: La PIS es verificada por medio de model checking utilizando NuSMV [1]. Desde el código fuente del sistema a ser verificado se genera automáticamente un modelo que representa la semántica del programa. Sobre este modelo se verifican formalmente las propiedades de la especificación. De este modo se prueba si el comportamiento del programa cumple con los requerimientos de la especificación.

2. Testing: Para probar la PSS del sistema se ejecutan casos de prueba en la plataforma objetivo. Para esto se generan casos de prueba desde el modelo autómata del sistema. Los casos de prueba ejecutados en la plataforma objetivo verifican si la PSS del sistema cumple con el comportamiento del modelo. El objetivo es verificar que el comportamiento de la PSS se ajusta a los requerimientos de la especificación. Los casos de prueba son generados añadiendo pequeños cambios sintácticos a la especificación del modelo (mutation testing).

3.6. Verificación utilizando lenguajes formales específicos

En [7] se presenta un método sistemático para representar transformaciones declarativas en el lenguaje Alloy. Se muestra como el analizador de Alloy puede ser usado para conducir un análisis totalmente automatizado de la especificación de una transformación de modelos representada con Alloy.

La verificación se realiza en dos etapas. La primera etapa consiste en traducir la especificación de la transformación al lenguaje Alloy. El primer paso en la traducción requiere que los metamodelos de los lenguajes de origen y destino y las restricciones del lenguaje de origen sean traducidos a Alloy. Este proceso puede ser automatizado mediante metodologías que traducen metamodelos MOF enriquecidos con restricciones OCL. Esta traducción fue implementada en la herramienta UML2Alloy [6]. El segundo paso es traducir las reglas de transformación hacia el lenguaje Alloy, en donde son representadas mediante lógica de primer orden. Se introducen relaciones de mapeo en Alloy para especificar la relación entre los elementos del metamodelo origen y el metamodelo destino. Esta relación es similar a la noción de trazas definida en la especificación de QVT.

La segunda etapa involucra el análisis utilizando el analizador de Alloy. El procedimiento definido en la etapa anterior produce un modelo Alloy de la transformación. El analizador de Alloy puede ser utilizado para analizar este modelo y así encontrar defectos en la especificación de la transformación. El analizador puede simular la transformación, lo que brinda la producción de una instancia aleatoria del metamodelo origen que se ajusta a las restricciones establecidas, una instancia del mapeo que transforma elementos del modelo origen y el modelo destino generado por la transformación. Si el analizador no puede producir una instancia de la transformación, entonces existe una inconsistencia en la definición de las reglas de transformación. El analizador además brinda facilidades para encontrar errores, como resaltado de inconsistencias y enumeración de instancias que se ajustan a la especificación de la transformación. Puede ser utilizado también para verificar afirmaciones. Las afirmaciones pueden ser usadas para verificar si el modelo destino cumple con las restricciones del lenguaje destino o también para verificar cuando un modelo generado por las reglas de transformación satisface determinadas propiedades. Si una propiedad no se satisface, el analizador presenta un contraejemplo instancia del modelo destino que viola la propiedad.

No obstante, existen algunas limitaciones para esta técnica. Una de ellas es que no son soportadas definiciones de transformaciones expresadas en forma imperativa o híbrida, sólo las puramente declarativas son contempladas. El tipado de Alloy solo soporta el tipo de datos Integer, así que no es posible usar Alloy para analizar propiedades que involucren otros tipos de datos. Dado que los modelos en Alloy son estáticos, sólo se puede utilizar esta aproximación para razonar sobre propiedades estáticas de la transformación. Sin embargo, sí se pueden modelar sistemas dinámicos en Alloy. Es de esperar que esta aproximación no escale bien cuando grandes metamodelos y reglas de transformación complejas sean utilizados.

Como caso de estudio el trabajo presenta una transformación que trata con un DSL (*Domain Specific Language* - Lenguaje Específico del Dominio) utilizado para modelar procesos de negocio, lenguaje similar a los diagramas de actividad de UML. Se utiliza el analizador de Alloy para verificar si la transformación produce modelos destino sintácticamente correctos.

En [54] se presenta un método específico basado en el uso de patrones para especificar las propiedades que caracterizan la correctitud de una transformación dada. Se proponen además las bases para un método que verifique la correctitud de una transformación. El trabajo introduce la noción de *restricción algebraica* como la traducción de la noción de restricción sobre grafos. Se presenta el concepto de *álgebras triples*, siguiendo la idea de los enfoques *triple graph*, y se muestra como pueden ser usados los patrones (patrones triples) para describir las propiedades de una transformación de modelos. El concepto de triple graph es definido en [62] para describir transformaciones de modelos cuando las modelos y metamodelos son representados como grafos. En lugar de considerar que la transformación está solamente caracterizada por la especificación de modelos origen y destino, los autores consideran de gran importancia establecer una conexión entre las correspondencias entre elementos del modelo origen con los elementos del modelo destino. Para ello se utiliza un grafo intermedio (grafo de conexiones) y se introduce un mapeo entre este grafo y los grafos origen y destino. En [54], se utiliza la misma idea pero utilizando álgebras en lugar de grafos.

El aporte principal de los autores en [54] es la definición de un método para verificar transformaciones mediante álgebras triples y patrones triples. Básicamente, como se comentó, un álgebra triple es un conjunto de tres álgebras: un álgebra de origen, una de destino y una de conexión, que expresa las relaciones del álgebra origen con el álgebra destino. Esta última es de gran utilidad para realizar verificaciones. Los patrones triples describen propiedades que deben ser satisfechas por un álgebra triple. Utilizando estos patrones se puede definir la transformación. En la figura 7 se puede observar un ejemplo de transformación utilizando patrones triples.

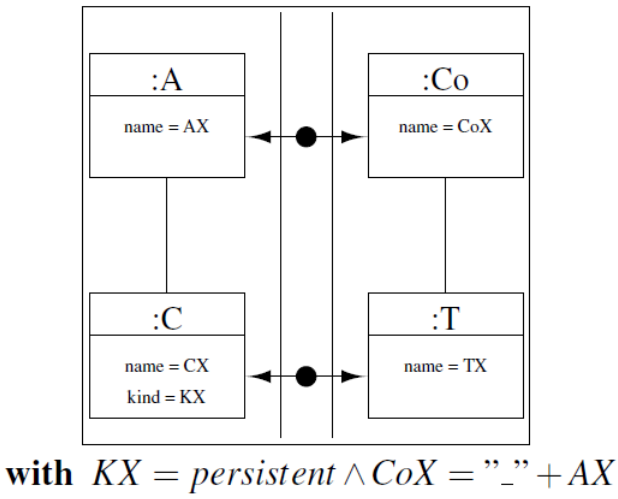

Figura 7: Transformación utilizando patrones triples [54]

De acuerdo con la transformación, las clases persistentes serán transformadas en tablas y sus atributos en columnas de dichas tablas. En este caso, la fórmula en el patrón expresa que el tipo de la clase debe ser persistente y que el nombre de la columna es obtenido agregando el caracter "_" delante del nombre del atributo. Teniendo ambos modelos y la transformación expresados en este formalismo, es sencillo expresar propiedades que se deben cumplir tanto en el modelo origen como en el modelo destino luego de la transformación, ya que se necesita un único formalismo para expresarlas.

En el trabajo se toman en cuenta solo descripciones declarativas de transformaciones de modelos para modelos estructurales (esto excluye, por ejemplo, modelos de comportamiento). Además, es importante notar que no existe una implementación del método propuesto en el artículo. Como trabajo futuro se comenta que el método podría ser implementado utilizando Maude (que permite trabajar directamente sobre álgebras) o se podría especializar esta aproximación para el caso de grafos y así utilizar alguna herramienta de transformación de grafos.

En [35] se presenta un método para verificar modelos de comportamiento para un contexto específico. En particular, la verificación se realiza sobre modelos estilo workflows, que modelan sistemas multiagente utilizando la herramienta agentTool [4]. La técnica plantea transformar los modelos expresados en la herramienta a un formalismo que pueda ser comprobado por un model-checker. En particular, se utiliza Promela [29] como lenguaje formal y Spin [28] como model-checker. La noción de verificación en este caso es comprobar que el modelo no presente deadlocks. En caso que un deadlock sea detectado, el analizador escribe un archivo de trazas que puede ser utilizado posteriormente para reparar el error en el modelo de comportamiento.

4. Verificación utilizando Métodos Deductivos

La verificación utilizando casos de prueba es de gran ayuda para asegurar la calidad de la transformación, explotando la especificación de la transformación y de los modelos para derivar casos de prueba. Sin embargo, solo es posible probar un número finito de casos, lo cual resulta insuficiente en aplicaciones críticas para asegurar la equivalencia semántica entre el modelo origen y el modelo destino, por ejemplo.

La verificación mediante métodos deductivos se basa en la especificación formal de la transformación y de los metamodelos origen y destino. No se emplean instancias de los metamodelos como casos de prueba,

por lo que se trata de un análisis *offline* (no se realiza mediante la ejecución) que produce resultados formales para todas las posibles instancias válidas. 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 [8].

Si bien la verificación mediante métodos deductivos cuenta con madurez en el campo de investigación de la ingeniería de software, su aplicación práctica se limita a sistemas embebidos y a aplicaciones donde la confiabilidad es crítica. Entre los motivos se encuentran la complejidad de las técnicas de especificación formal y la falta de entrenamiento de los desarrolladores de software para la aplicación de dichas técnicas [21].

En esta sección se describen distintos problemas y técnicas de verificación donde se utilizan métodos deductivos que permiten asegurar con certeza absoluta propiedades de una transformación. Las técnicas trabajan con especificaciones formales de los metamodelos origen y destino, y/o de la transformación. Dependiendo del contexto y de la verificación realizada, las técnicas hacen uso de asistentes de prueba o de herramientas similares donde se brinda ayuda para especificar y/o construir la prueba.

4.1. Descripción formal de transformaciones

En [8] se busca desarrollar un sistema de razonamiento semi-automático sobre los metamodelos que permita probar propiedades de una transformación basada en grafos en forma offline. Para esto se propone una descripción formal de la transformación y de las propiedades, con el fin de permitir la verificación. Este trabajo no se centra en la verificación, sino en la descripción formal y en el sistema de razonamiento que asiste al usuario para las pruebas.

Se introduce el concepto de *assertion*: una expresión formal que describe características de la transformación o del modelo. Se desea que el sistema pruebe si las assertions se cumplen o no. Para esto se expresan en un lenguaje formal las assertion iniciales y las reglas de deducción, permitiendo que el sistema de razonamiento pueda derivar assertions adicionales.

Este lenguaje formal debe permitir la extensión de los tipos de assertion y de reglas de deducción. Se considera apropiado para este objetivo la utilización de lógica de primer orden extendida con manejo de atributos y consultas topológicas. Además es deseable que el sistema sea modular (diferentes partes de la transformación deben ser descritas en forma separada) y derive assertions iniciales en forma automática a partir de la definición de la transformación. Finalmente, debe permitir agregar assertions en forma manual.

Este trabajo presenta estructuras matemáticas para describir el flujo de control de la transformación y el lenguaje ADL (Assertion Description Language) para describir assertions. ADL presenta la extensibilidad deseada que se mencionó en el párrafo anterior. Además, permite expresar patrones y fórmulas. Los patrones establecen restricciones sobre los elementos del modelo (ej: elementos de tipo T). Estos patrones son a su vez utilizados para construir fórmulas (ej: no existen elementos de tipo T). Los componentes del grafo que describe el flujo de control, así como las sentencias de ADL que expresan assertions, los patrones y las fórmulas se encuentran detallados en [8].

Los conceptos expuestos son ejemplificados mediante la implementación de un caso de estudio en Visual Modeling and Transformation System (VMTS) [38]. VMTS fue utilizado para la generación automática de una descripción formal muy básica y limitada a partir de la definición de la transformación. El sistema de razonamiento fue implementado usando SWI-Prolog [70].

Existen aún limitaciones en el alcance de las estructuras propuestas. Aún debe estudiarse cómo propagar las fórmulas en la estructura de control de flujo y cómo manejar iteraciones. Otro objetivo pendiente es la generación de una descripcion formal más completa a partir de una implementación de la transformación en herramientas específicas para MDD. Finalmente, restan preguntas abiertas sobre cómo analizar la consistencia de la descripción formal dadas assertions iniciales y assertions derivadas.

4.2. Verificación mediante Triple Graph Grammars

En [26] se describe la técnica de verificación mediante un caso de estudio que plantea la transformación de un autómata a código PLC (Programmable Logic Controllers). En dicho trabajo se verifica formalmente que la transformación asegure equivalencia semántica entre cualquier modelo y el código fuente generado por la transformación. Este tipo de pruebas resulta de utilidad si el código generado automáticamente es correcto sintácticamente, lo cual debe verificarse en forma previa.

Se prueba que la relación de equivalencia semántica es una congruencia con respecto a una representación adecuada de las reglas de transformación. En este contexto, la noción de equivalencia semántica significa que las propiedades de interés verificadas en el modelo origen se mantienen en el código generado.

En la Figura 8 se puede observar un esquema del proceso de verificación de una trasformación utilizando un probador de teoremas.

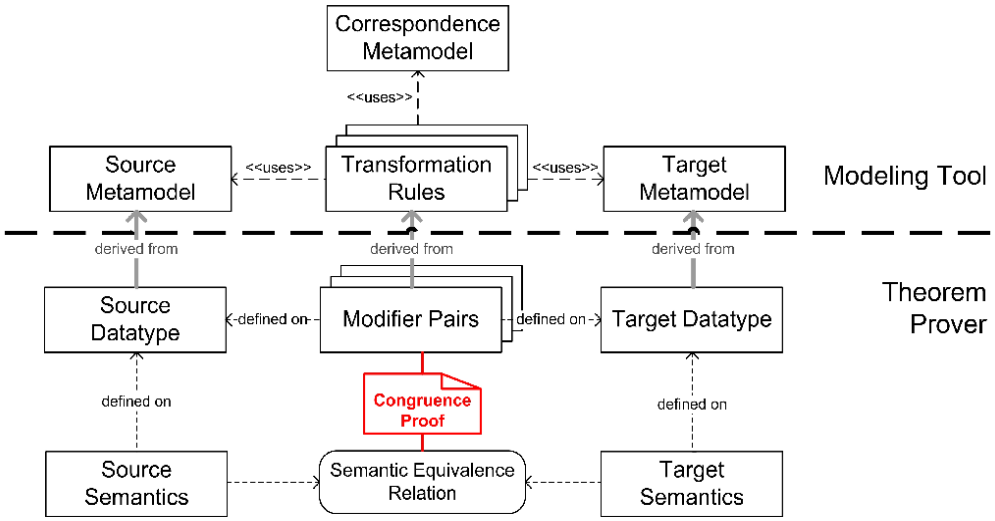

Figura 8: Esquema de verificación formal [26]

Por medio del probador se verifica la correctitud de las reglas de transformación que son aplicadas a cualquier modelo a transformar. Se expresan los metamodelos en un lenguaje formal que el probador entiende (en la figura 8, *Source Datatype* y *Target Datatype*), para luego utilizarlos en la definición de las propiedades semánticas que se quieren comprobar *(Source Semantics* y *Target* Semantics).

La verificación es realizada utilizando Fujaba Tool Suite4 [49] y se basa en una técnica de especificación formal llamada Triple Graph Grammars (TGG) [62] para especificar una transformación de modelo a código. La correctitud de la especificación de la transformación y del generador de código es probada utilizando Isabelle/HOL [50]. El código PLC generado automáticamente es definido mediante el lenguaje ST (Structured Text), el cual emplea una notación similar a Pascal.

4.3. Verificación usando el método formal B

Una técnica similar a la anterior, es la presentada en [36]. El artículo describe el uso del método formal B para verificar propiedades semánticas de modelos gráficos UML y la correctitud de transformaciones definidas sobre estos modelos.

Se define un subconjunto de UML llamado UML-RSDS (*Reactive System Development Support*) el cual posee semántica precisa basada en la teoría de conjuntos ZF y el cálculo de predicados. La especificación

de un modelo UML-RSDS se compone básicamente de:

- Un diagrama de clases UML que incluye restricciones adjuntas a las operaciones, clases y asociaciones.
- Un diagrama de casos de uso.
- Modelos de máquinas de estado adjuntos a las clases, a las operaciones en el diagrama de clases o a los casos de uso.

El método formal B provee una notación en la cual la semántica de los modelos puede ser expresada y utilizada para la verificación de los modelos. En la Figura 9 se puede ver un esquema del proceso de verificación.

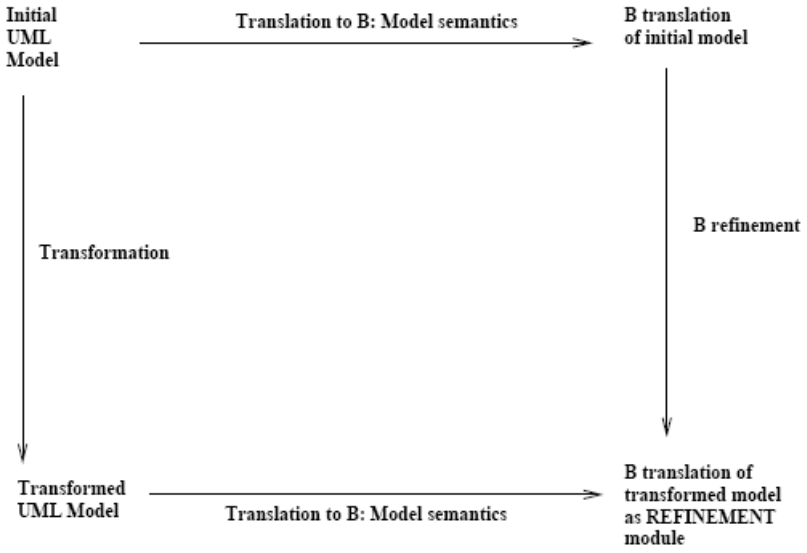

Figura 9: Verificación utilizando B [36]

Inicialmente, se tiene un modelo inicial expresado en UML con su correspondiente transformación. Luego el modelo origen y el modelo destino son traducidos automáticamente (mediante una herramienta UML-RSDS) a B, el cual expresa la semántica de los modelos. El modelo destino es definido en un módulo B, el cual es declarado como un refinamiento del módulo B inicial que expresa la semántica del modelo original. A partir de aquí, las pruebas pueden ser generadas utilizando una herramienta para el lenguaje B.

La correctitud de una transformación se verifica probando que la semántica del modelo refinado preserva la semántica del modelo original. La prueba de refinamiento realizada en B establece que todas las pre y post condiciones de las operaciones y que todas las propiedades invariantes del modelo UML original son válidas en el modelo destino.

4.4. Verificación basada en teoría de tipos

Otro ejemplo de aplicación de métodos deductivos es el presentado por [57], donde se muestra cómo construir una transformación correcta a partir de una especificación funcional de precondiciones y

postcondiciones. Es importante señalar que este enfoque no resulta de utilidad para verificar una transformación existente, sino que se enfoca en el proceso de construcción de una transformación correcta. El enfoque propuesto se basa en un formalismo matemático denominado *Constructive Type Theory* (CTC) y en el método formal de síntesis *Proof-as-Programs* [18, 58].

CTC es un formalismo de alto orden que permite expresar programas, tipos de programas y tipos de tipos. Según el autor, es precisamente este alto orden lo que lo vuelve adecuado para expresar naturalmente las relaciones de alto orden entre modelos y metamodelos. De esta forma, la especificación funcional de un programa se representa como un tipo especial. Sin embargo, este trabajo se restringe a un subconjunto de los modelos representables por MOF. En particular, el trabajo define un metamodelo como un conjunto de objetos de tipo *Classifier*, *Class*, *Datatype*, *Attribute*, *Association*, *AssociationEnd* y *Constraint*. Estos objetos pueden referirse únicamente a otros objetos de este conjunto. Esta limitación impide la utilización en la implementación de la transformación de características presentes en MOF como reflection.

El método de proof-as-programs ha sido empleado con fines similares por otros enfoques. En este enfoque, se emplea el isomorfismo de Curry-Howard para mostrar la relación entre la lógica constructiva y CTC, donde las fórmulas lógicas se corresponden con tipos y las pruebas con términos. Se prueba entonces que a partir de la prueba de una fórmula basada en tipos, es posible extraer un programa que cumpla con las restricciones de la transformación de modelos. En otras palabras, se obtiene la implementación de la transformación a partir de una prueba formal de su especificación.

Este enfoque presenta dos desafíos técnicos:

- La extracción del programa a partir de la especificación es una extensión no trivial de la extracción realizada en otros enfoques proofs-as-program. Esta extensión requiere consideraciones especiales para la teoría de tipos empleada en este trabajo. Los detalles de esta extensión son desarrollados en [58].

- La formalización mediante tipos de metamodelos basados en MOF la cual es descrita en líneas generales en [57]. Además, este trabajo presenta un simple ejemplo en el que se formaliza la especificación de una transformación de un modelo de clases UML a un modelo relacional. No se detalla cómo se trata la herencia en el metamodelado usando la teoría de tipos. Se sugiere que puede introducirse el subtipado como mecanismo para trabajar con herencia basándose en [56].

Finalmente, se debe señalar que la prueba de las fórmulas que son luego sintetizadas en transformaciones es una tarea laboriosa para el desarrollador. Por este motivo se sugiere utilizar probadores de teoremas basados en tácticas como Nuprl[2], Coq[5] y PVS[3].

En [15] se utiliza CTC para representar los metamodelos y sus respectivos modelos, y se usa dicho formalismo para verificar semiformalmente la transformación. Con este fin se propone un framework basado en el *Cálculo de Construcciones Inductivas* (CIC). Este trabajo parte de las mismas premisas de [57], pero hace especial énfasis en demostrar la factibilidad de dicho enfoque. Para lograrlo se estudia como traducir transformaciones especificadas en un lenguaje de transformación de modelos al formalismo CIC empleado. Para mostrar las ideas de este trabajo se implementaron casos de estudio de transformaciones expresadas en ATL sobre metamodelos definidos en Kernel MetaMetaModel (KM3), y se utilizó el asistente de pruebas Coq.

CIC es una versión de la teoría de tipos que fue elegida por tratarse de una teoría consolidada que cuenta con varias herramientas para trabajar sobre la misma. El trabajo propone un framework basado en CIC donde los metamodelos son representados como tipos y la transformación como fórmulas lógicas que deben cumplirse (assertions). Dado que las pruebas de dichas fórmulas en CIC son constructivas, es posible derivar a partir de ellas programas sin fallas que construyan el modelo destino a partir del modelo origen. El trabajo deja de lado estas consideraciones para enfocarse en la representación formal de transformaciones pre-existentes y en el desarrollo de un método semi-formal para su verificación.

De esta forma se muestra como expresar en CIC un subconjunto de los elementos constructores KM3 que definen un metamodelo. En [15] se presentan ejemplos para un caso de estudio. Los modelos

que conforman un metamodelo se representan como un registro (record) que contiene una lista de las instancias pertenecientes a cada clase del metamodelo.

De forma similar, se muestra como traducir la especificación de una transformación en ATL (sin perder generalidad para otras herramientas) a una representación Coq. Se considera para esto un subconjunto de los constructores de ATL que excluye a los módulos y los modos de ejecución.

Una vez obtenida esta traducción, es posible verificar propiedades de la especificación original mediante la construcción de una prueba sobre su representación en Coq. Esto se realiza expresando especificaciones que debe cumplir la transformación en forma de formulas cuantificadas con $\forall$ (para todo) y $\exists$ (existe). Estas pruebas interactivas, se realizan mediante el asistente de pruebas Coq utilizando tácticas.

5. Conclusiones

En este documento se presentaron una gran variedad de enfoques y técnicas para la verificación de transformaciones de modelos. Algunas técnicas toman la experiencia obtenida en la verificación dentro de otros paradigmas de desarrollo de software y la adaptan a las características de MDD. Esto muestra que los diferentes paradigmas comparten muchos conceptos de verificación como el análisis de caja blanca y caja negra, las mutaciones y las técnicas de cubrimiento. En cambio, otros métodos toman en consideración que toda transformación tiene como entrada y salida uno o más modelos que conforman un metamodelo. Esta característica es la que explotan los enfoques basados en model-checkers, donde se busca comprobar en forma automática, bajo ciertas restricciones, características de los modelos involucrados.

Una de las principales diferencias entre los distintos métodos documentados radica en la certeza que éstos brindan. Los métodos de verificación basados en el uso de casos de prueba sólo brindan certeza sobre los casos de prueba verificados por lo que mediante técnicas de partición y cubrimiento se puede afirmar, sólo con cierto grado de confiabilidad, que las propiedades se preservarán para modelos no verificados. Por otro lado, las técnicas basadas en model-checking y en métodos deductivos brindan certeza absoluta sobre las propiedades que se analizan. Bajo métodos deductivos, muchas veces se realizan demostraciones constructivas que permiten generar transformaciones certificadas como correctas (ej: usando teoría de tipos).

Surge entonces la interrogante sobre por qué no se utilizan siempre métodos deductivos. El primer motivo es que los métodos deductivos requieren la utilización de lenguajes formales y herramientas matemáticas con los que los desarrolladores no están habituados a trabajar. Es un requisito indispensable el obtener (en el mejor de los casos de forma semiautomática) una especificación formal de la transformación, de los metamodelos y los modelos involucrados para poder luego utilizar asistentes de pruebas. Para superar esta dificultad existen algunas propuestas donde se intenta reducir la cantidad de formalismos mediante el uso de especificaciones semiformales. Sin embargo, estas técnicas que emplean semiformalismos suelen perder alcance y generalidad en los resultados de la verificación, o asumen la existencia de una transformación intermedia que preserva propiedades semánticas. En nuestro criterio, esa suposición sobre la transformación intermedia correcta nos devuelve al principio sobre el problema de verificación.

El segundo motivo que parece aún más difícil de superar radica en el tipo de propiedades que se puede probar usando un enfoque u otro. Los métodos deductivos utilizan un mayor nivel de abstracción para expresar propiedades, al basarse en la especificación de metamodelos. Los métodos basados en el uso de casos de prueba en cambio permiten probar propiedades más específicas sobre la transformación.

Estas consideraciones generales sobre la verificación de transformaciones son motivo de la existencia de una gran variedad de enfoques. Ninguno de estos enfoques es capaz de contemplar por sí mismo todas las necesidades de verificación, por lo que deben emplearse criterios para determinar cuál es la técnica más apropiada para el problema concreto teniendo en cuenta la criticidad de la aplicación. Debido a esta gran variedad, las herramientas de verificación no se encuentran aún integradas a las herramientas de desarrollo de MDD. La tendencia en herramientas de desarrollo MDD es permitir la especificación más general

posible de transformaciones. Dada esta diversidad de transformaciones que se pueden implementar, es difícil pensar en herramientas específicas de verificación para integrar a los entornos de desarrollo.

6. Tabla de referencias

Tema	**Referencias**
Casos de Prueba	
Dificultades generales	[9, 68]
Caja blanca	[34]
Caja negra	[23, 59, 64, 12, 39]
Mutation testing	[19, 32, 46]
Model-checking	
Basado en Redes de Petri	[60, 67]
Basado en especificación UML/OCL	[14, 16, 21, 48, 59]
+CAL	[25]
Basado en sistemas de transformaciones de grafos	[27, 11]
NuSMV	[32]
Model-checking con lenguajes formales	[7, 54, 35]
Métodos deductivos	
Descripciones formales	[8]
Triple Graph Grammars	[26]
Método formal B	[36]
Basados en teoría de tipos	[15, 57]

Cuadro 1: Referencia de trabajos agrupados por tema

Referencias

[1] NuSMV: a new symbolic model checker. http://nusmv.irst.itc.it/. Último acceso, Abril 2010.

[2] PRL Automated Reasoning Project at Cornell. http://nuprl.org/. Último acceso, Abril 2010.

[3] PVS Specification and Verification System. http://pvs.csl.sri.com/. Último acceso, Abril 2010.

[4] The agentTool Project. http://macr.cis.ksu.edu/projects/agentTool/agentool.htm. Último acceso, Abril 2010.

[5] The Coq Proof Assistant. http://coq.inria.fr/. Último acceso, Abril 2010.

[6] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. 2007.

[7] K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of Model Transformations via Alloy. In Benoit Baudry, Alain Faivre, Sudipto Ghosh, and Alexander Pretschner, editors, *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.

[8] M. Asztalos, L. Lengyel, and T. Levendovszky. A formalism for describing modeling transformations for verification. In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–10. ACM, 2009.

[9] B. Baudry, S. Ghosh, F. Fleurey, R. France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 2009.

[10] A. Boronat. *MOMENT: A Formal Framework for MOdel manageMENT*. PhD thesis, Universidad Politécnica de Valencia, 2007.

[11] A. Boronat, R. Heckel, and J. Meseguer. Rewriting logic semantics and verification of model transformations. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 18–33. Springer-Verlag, 2009.

[12] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. pages 85–94, 2006.

[13] C-SAW. Constraint-Specification Aspect Weaver. http://www.gray-area.org/Research/C-SAW/.

[14] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, August 2009.

[15] D. Calegari, C. Luna, N. Szasz, and A. Tasistro. Experiment with a Type-Theoretic Approach to the Verification of Model Transformations.

[16] E. Cariou, N. Belloir, F. Barbier, and N. Djemam. OCL contracts for the verification of model transformations.

[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, and N. Martí-Oliet. *All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Springer-Verlag New York Inc, 2007.

[18] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, et al. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1986. Último acceso, Abril 2010.

[19] A. Darabos, A. Pataricza, and D. Varró. Towards testing the implementation of graph transformations. *Electronic Notes in Theoretical Computer Science*, 211:75–85, 2008.

[20] EMFCompare. EMF Compare. http://wiki.eclipse.org/index.php/EMF_Compare/. Último acceso, Abril 2010.

[21] G. Engels, J. Küster, R. Heckel, and M. Lohmann. Model-based verification and validation of properties. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[22] F. Fleurey, Z. Drey, D. Vojtisek, C. Faucher, and V. Mahé. Kermeta Language, Reference Manual. *Internet: http://www. kermeta. org/docs/KerMeta-Manual. pdf. IRISA*, 2006.

[23] F. Fleuri, B. Baudry, P. Muller, and Y. Le Traon. Towards dependable model transformations: Qualifying input test data. *Journal of Software and Systems Modeling (SoSyM)*, 2007.

[24] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. 2007.

[25] M. Garcia and R. Möller. Certification of Transformations Algorithms in Model-driven Software Development. 105(ISBN 978-3-88579-199-7):107–118, 2007.

[26] H. Giese, S. Glesner, J. Leitner, W. Schafer, and R. Wagner. Towards verified model transformations. *MoDeV^{2}a: Model Development, Validation and Verification.*, page 78.

[27] R. Heckel and S. Thöne. Behavioral refinement of graph transformation-based models. *Electr. Notes Theor. Comput. Sci.*, 127(3):101–111, 2005.

[28] G.J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley Publishing Company, 2004.

[29] R. Iosif. The PROMELA Language. http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html. Último acceso, Abril 2010.

[30] D. Jackson. Alloy Community. http://alloy.mit.edu. Último acceso, Abril 2010.

[31] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. *ACM SIGSOFT Software Engineering Notes*, 26(5):62–73, 2001.

[32] S. Kandl, R. Kirner, and G. Fraser. Verification of platform-independent and platform-specific semantics of dependable embedded systems. Oct. 2006.

[33] R.E. Korf. Space-efficient search algorithms. *ACM Computing Surveys (CSUR)*, 27(3):337–339, 1995.

[34] J.M. Kuster and M. Abd-El-Razik. Validation of model transformations-first experiences using a white box approach. *Lecture Notes in Computer Science*, 4364:193, 2007.

[35] T. Lacey and S. DeLoach. Verification of agent behavioral models, 2000.

[36] K. Lano. Using b to verify uml transformations. *MODEVA workshop at MoDELS 2006*, 2006.

[37] M. Lawley and J. Steel. Practical declarative model transformation with tefkat. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2005.

[38] L. Lengyel, T. Levendovszky, and G. Mezei. Visual Modeling and Transformation System - VMTS. http://avalon.aut.bme.hu/ tihamer/research/vmts/. Último acceso, Abril 2010.

[39] Yuehua Lin, Jing Zhang, and Jeff Gray. A testing framework for model transformations. pages 219–236, 2005.

[40] H. López, F. Varesi, M. Viñolo, D. Calegari, and C. Luna. Estado del Arte de Lenguajes y Herramientas de Transformación de Modelos. Technical Report RT09-19, InCo-PEDECIBA, Uruguay, 2009. Último acceso, Abril 2010.

[41] Y. Mahajan. Boolean Satisfiability Research Group at Princeton. http://www.princeton.edu/ chaff/zchaff.html. Último acceso, Abril 2010.

[42] ATLAS Group. *KM3: Kernel MetaMetaModel.* LINA & INRIA, manual v0.3 edition, 2005.

[43] ATLAS Group. *ATL: Atlas Transformation Language.* LINA & INRIA, User Manual v0.7 edition, 2006.

[44] S. Mellor, A. Clark, and T. Futagami. Model-Driven Development. In *IEEE Software*, volume 20, pages 14–18, 2003.

[45] MMCC. Metamodel Coverage Checker – Projet Triskell. http://www.irisa.fr/triskell/Softwares/protos/MMCC/. Último acceso, Abril 2010.

[46] J. Mottu, B. Baudry, and Y. Le Traon. Mutation analysis testing for model transformations. *Lecture Notes in Computer Science*, 4066:376, 2006.

[47] J.M. Mottu, B. Baudry, and Y. Le Traon. Reusable MDA Components: A Testing-for-Trust Approach. *Lecture Notes in Computer Science*, 4199:589, 2006.

[48] A. Narayanan and G. Karsai. Specifying the correctness properties of model transformations. In *GRaMoT '08: Proceedings of the third international workshop on Graph and model transformations*, pages 45–52, New York, NY, USA, 2008. ACM.

[49] U. Nickel, J. Niere, and A. Zundorf. The FUJABA environment. In *International Conference on Software Engineering*, volume 22, pages 742–745, 2000.

[50] T. Nipkow, L.C. Paulson, and M. Wenzel. Isabelle/HOL - A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS, 2002.

[51] OMG. Meta Object Facility (MOF) 2.0 Core Specification. Specification Version 2.0, Object Management Group, 2003.

[52] OMG. UML 2.0 Object Constraint Language. Formal Specification formal/06-05-01, Object Management Group, 2006.

[53] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation. Specification Version 1.0, Object Management Group, 2008.

[54] F. Orejas and M. Wirsing. On the specification and verification of model transformations. pages 140–161. 2009.

[55] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, 1988.

[56] I. Poernomo. A type theoretic framework for formal metamodelling. In *Architecting Systems with Trustworthy Components*, pages 262–298, 2004.

[57] I. Poernomo. Proofs-as-Model Transformations. In *ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, pages 214–228, Berlin, Heidelberg, 2008. Springer-Verlag.

[58] I.H. Poernomo, J.N. Crossley, and M. Wirsing. *Adapting proofs-as-programs: the Curry-Howard protocol*. Springer-Verlag New York Inc, 2005.

[59] C. Pons and D. Garcia. Proving the Correctness of Refinement in Model Driven Software Engineerings. *Electronic Notes in Theoretical Computer Science*, 2008.

[60] I. Raedts, M. Petković, A. Serebrenik, J.M van der Werf, L. Somers, and M. Boote. A software framework for automated verification. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1031–1032, New York, NY, USA, 2007. ACM.

[61] W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[62] A. Schiirri. Specification of graph translators with triple graph grammars. In *Graph-theoretic concepts in computer science: 20th international workshop, WG'94, Herrsching, Germany, June 16-18, 1994: proceedings*, page 151. Springer Verlag, 1995.

[63] S Sen, B. Baudry, and J.M. Mottu. On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. In *Proceedings of ICST'08 (International Conference on Software Testing Verification and Validation)*, 2008.

[64] S. Sen, B. Baudry, and J.M. Mottu. Automatic model generation strategies for model transformation testing. *International Conference on Model Transformation, ICMT'09*, 2009.

[65] J.M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[66] UMLtoCSP. UMLtoCSP: Tool for the Verification of UML/OCL models. http://gres.uoc.edu/UMLtoCSP/. Último acceso, Abril 2010.

[67] D. Varró and A. Pataricza. Automated formal verification of model transformations. (TUM-I0323):63–78, September 2003.

[68] J. Wang, S.K. Kim, and D. Carrington. Verifying Metamodel Coverage of Model Transformations. In *Proceedings of the Australian Software Engineering Conference*, page 282. IEEE Computer Society, 2006.

[69] S.A White et al. Business Process Modeling Notation (BPMN) Version 1.0. *Business Process Management Initiative, BPMI. org*, 2004.

[70] J. Wielemaker. SWI-Prolog reference manual. *University of Amsterdam*, 1996.