Métodos formales en Gestión Global de Modelos

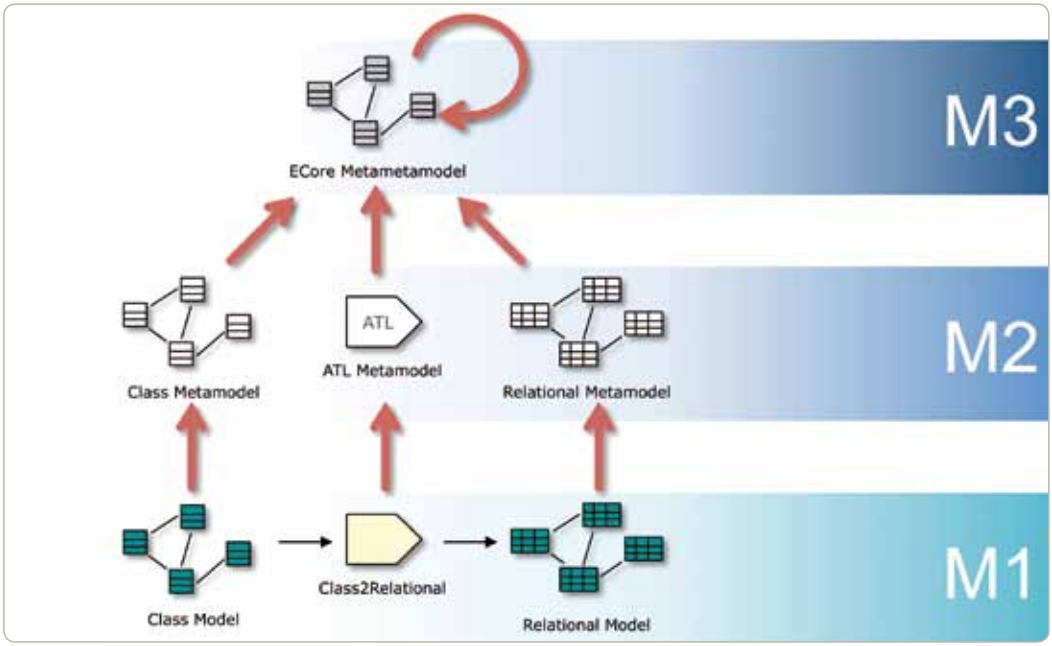
Andrés Vignaga
Gerente de Operaciones de Imagen S.A. Doctor en Ciencias mención Computación, Universidad de Chile (2011); Magíster en Informática, PEDECIBA Uruguay (2004); Ingeniero en Computación, Universidad de la República, Uruguay (2000). Áreas de interés: Ingeniería de Software, Métodos Formales.
avignaga@dcc.uchile.cl

La complejidad de los sistemas de software ha ido creciendo en las últimas décadas de manera sostenida. Ya en los noventa comenzaron a aparecer una serie de enfoques de desarrollo, con sus respectivas notaciones, orientados a manejar dicha complejidad al permitir a los desarrolladores trabajar sobre representaciones simplificadas de los sistemas a construir, las cuales se detallan sucesivamente hasta alcanzar un punto en que la brecha entre la representación y el código fuente fuese lo suficientemente pequeña. Estas representaciones recibieron el nombre de *modelos*. En la práctica, se contaba únicamente con una sintaxis gráfica informalmente definida para expresar los modelos, tipo las "cajas y flechas" que dieron origen al Unified Modeling Language (UML) [14]. Los modelos acababan siendo meros dibujos, muchas veces únicamente interpretables por sus autores, que después de servir su propósito eran descartados sin ninguna mantención. El modelamiento se percibía como una herramienta útil para pensar en un nivel de abstracción alto, pero la falta de rigor y técnicas (sí disponibles en el bajo nivel del código fuente) suponían contraproducentes los esfuerzos más allá de una diagramación rápida. El enfoque denominado Model-Driven Engineering (MDE) [15] apareció años después buscando posicionar a los modelos en el centro del desarrollo de software, precisamente atacando aquellas debilidades detectadas en la práctica. Con una definición precisa de lo que es un modelo, sería posible generar herramientas y técnicas que permitieran a los desarrolladores realmente pensar en el nivel de abstracción correcto (que por cierto está bien lejos del nivel del código fuente). Más aún, si un modelo pasa a ser una entidad precisamente definida, en lugar

de que una persona refine manualmente un modelo para obtener otro más detallado, y por ende más cercano al código fuente, sería posible también definir programas que realicen dichos refinamientos en forma (al menos idealmente) automática. Estos programas, llamados *transformaciones de modelos* [7], codifican el conocimiento experto de un desarrollador y pueden ser aplicados una y otra vez en múltiples proyectos. El ejemplo más clásico de transformación es cuando a partir de un modelo de clases (típicamente expresado mediante un diagrama de clases de UML) se genera el modelo de la base de datos relacional (típicamente expresado con un modelo entidad-relación) que almacenará los datos persistentes del sistema. Esta transformación se aplica usualmente durante el desarrollo orientado a objetos de un sistema de información. La transformación puede llevarse a cabo manualmente por un desarrollador, en un proceso que es propenso a errores, o puede implementarse en lenguajes específicos como AtlanMod Transformation Language (ATL) [11] o MOF Query/View/Transformation (QVT) [13]. Después de ser verificada, la transformación puede ser aplicada de manera sistemática por un operador que no necesita ser un especialista. De esta manera, el desarrollo de software podría realizarse mediante la construcción de una serie de modelos los cuales fluyen a través de una cadena de transformaciones que deriva en la generación del código fuente.

Este nuevo paradigma trajo soluciones, pero también nuevos problemas. Dado que un sistema usualmente se modela desde diferentes perspectivas para contar con toda la información necesaria para su construcción, en un proyecto de escala industrial la cantidad de modelos y transformaciones involucradas puede ser llamativamente grande. La gestión de tal cantidad de artefactos, que incluso podrían estar distribuidos geográficamente dadas las ubicaciones de los diferentes equipos de desarrollo, se convirtió en un problema que fue abordado por Global Model Management (GMM) [4]. Este enfoque propone almacenar todos los modelos y transformaciones intervinientes en un proyecto, en un repositorio donde dichos artefactos puedan ser organizados y gestionados apropiadamente. Este repositorio es un modelo (recordar que en MDE todo es un modelo) denominado *megamodelo*. En particular, un megamodelo no sólo permite almacenar transformaciones, sino que además provee un ambiente en el cual éstas pueden ser ejecutadas sobre otros modelos almacenados para generar nuevos modelos. Identificando transformaciones con operaciones y modelos con valores, del mismo modo en que en los lenguajes de programación se controla que una operación reciba un valor del tipo que espera para que no se produzcan errores de ejecución, el tipado de modelos [16] y transformaciones en un megamodelo se convirtió en un problema que necesitaba solución.

Figura 1



Organización en niveles de elementos básicos de MDE.

MEGAMODELOS Y TIPADO

Para que un modelo pueda ser procesado mecánicamente por una transformación sus construcciones deben estar definidas de manera precisa. Al igual que en un rincón del mapa del Metro de Santiago existe un pequeño recuadro titulado "Simbología", que explica cómo se representa una línea en operación, una línea en construcción, una estación de combinación, etc., es posible definir un modelo cuyo propósito sea expresar qué elementos podrán formar parte de nuestro modelo. A ese modelo análogo a la simbología se le denomina *metamodelo*, dado que es un modelo de un modelo. ¿Y cómo se define de manera precisa a un metamodelo? De igual forma, se puede definir un modelo que defina a un metamodelo. Este tipo de modelos es denominado *metametamodelo*. Pero esta secuencia no es infinita. Por el contrario, un metametamodelo se expresa con las construcciones que el mismo introduce, las cuales son pocas y básicas, pero que razonablemente permiten expresar un metamodelo arbitrario. Un ejemplo concreto de metametamodelo es Kernel MetaMetaModel (KM3) [9]. Estos elementos básicos de MDE se organizan en niveles [3] como se ilustra en la Figura 1. Los modelos que representan un sistema del mundo real se denominan *modelos terminales* y se ubican en un nivel M1. Los metamodelos y metametamodelos se ubican en los niveles M2 y M3 respectivamente y se denominan *modelos de referencia* ya que son utilizados para definir otros modelos. Por lo tanto, todo modelo de M1 y M2 conforma con un modelo de un nivel inmediatamente superior al suyo, mientras que un modelo de M3 conforma con un modelo de su mismo nivel, en particular, él mismo.

En MDE, dado que "todo es un modelo", las transformaciones son también consideradas modelos, más específicamente, modelos terminales. Pero existen otras clases de modelos terminales. Un megamodelo es también un modelo terminal. Adicionalmente, los *modelos de weaving* [8] se utilizan para establecer relaciones entre otros modelos. Suponiendo que se aplica la transformación antes mencionada a un modelo de clases *c* para producir un modelo relacional *r*, se podría construir un modelo de weaving *w* que representa la relación existente entre *c* y *r*. Más específicamente, los elementos contenidos en *w* serían tales que conectan a un elemento de *r* con el o los elementos de *c* que se utilizaron para producirlo.

MDE no significa una visión absolutista del mundo, sino que introduce el mundo mirado a través del cristal de "todo es un modelo". Esto es informalmente lo que se denomina *espacio técnico* [12]. Existen adicionalmente otros espacios técnicos donde por ejemplo todo se entiende como textos estructurados de acuerdo a gramáticas. Una misma entidad puede tener representaciones en diferentes espacios técnicos, donde una representación es una proyección de la otra y viceversa. Pensando en un programa en un cierto lenguaje de programación, podríamos contar con la representación en la forma de un código fuente y con la representación en la forma de un modelo. Cada representación presenta sus ventajas y desventajas, por lo que la idea es aprovechar las ventajas de cada una. El proceso de compilación probablemente sea mejor realizarlo sobre la representación textual, pero el proceso de factorizar propiedades comunes a varias subclases en una superclase es más fácil de realizar en forma automática sobre una representación de modelo. En MDE la acción de proyectar un modelo hacia otro espacio técnico se denomina *extracción*, mientras que la inversa se denomina *inyección*. Tanto un extractor como un inyector son operaciones binarias que involucran a un modelo como su entrada o como su salida, y son también considerados modelos terminales. Un ejemplo de lenguaje para definir proyectores entre el espacio técnico de MDE y el espacio técnico de las representaciones textuales basadas en gramáticas, es Textual Concrete Syntax (TCS) [10].

> MDE no significa una visión absolutista del mundo, sino que introduce el mundo mirado a través del cristal de "todo es un modelo". Esto es informalmente lo que se denomina *espacio técnico*.

Un megamodelo se utiliza para contener modelos de los tres niveles presentados. Por ejemplo, todos los modelos involucrados en un cierto proyecto. Pero un megamodelo es un repositorio vivo, en el sentido de que provee además la capacidad de ejecutar las transformaciones que contenga, sobre los modelos apropiados. Al igual que en los lenguajes de programación en los que se les asigna un tipo a cada valor para catalogarlos, en GMM se le asigna un tipo a cada modelo. Con las definiciones apropiadas, es posible utilizar la información de tipos para evitar que una transformación reciba como argumento a un modelo del tipo equivocado y en consecuencia produzca un error en el tiempo de ejecución. Pareciera que con una definición simple de la noción de tipo de modelos el problema quedaría resuelto. De hecho, la propuesta original de GMM fue hecha de esta forma, y al realizar la implementación de AM3 [1], herramienta de manejo de megamodelos, quedó en evidencia la necesidad de una solución más potente. La solución original consistió en que el tipo de un modelo es su metamodelo. Esto, para algunos modelos terminales, es suficiente. Por un lado, hay otros modelos que no son modelos terminales, por lo que la definición debiese corregirse indicando que el tipo de un modelo es su modelo de

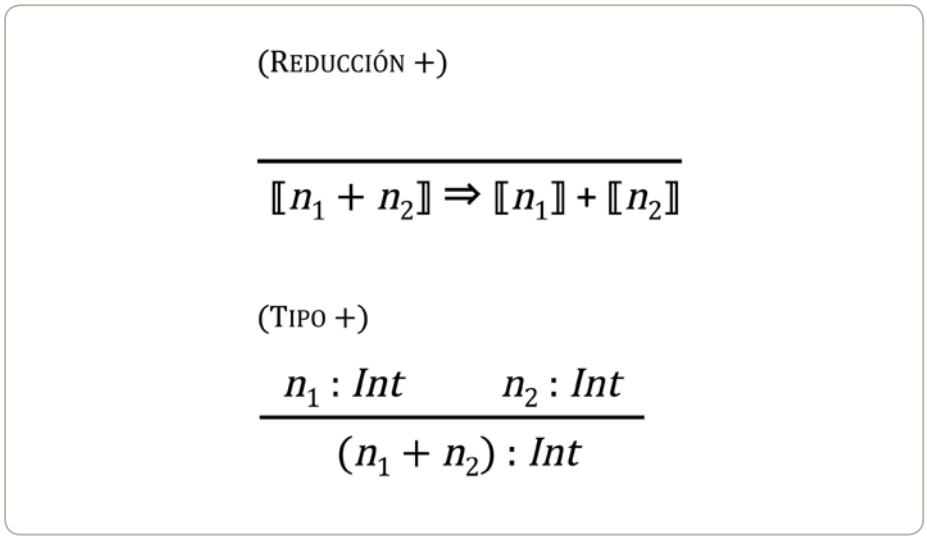
referencia. Por otro lado, algunos modelos terminales son transformaciones, y no parece apropiado tipar una operación como si fuese un valor, si es que se quiere que su tipo ayude a protegerla de entradas indeseadas. Para las transformaciones, la noción de tipo debe modificarse para corresponderse con la manejada en los lenguajes de programación. Por ejemplo, $t : A \rightarrow B$ significa que la transformación t espera como entrada un modelo de tipo A y produce como salida un modelo de tipo B. Recordar que las transformaciones son aplicables a modelos, y un metametamodelo es un modelo, e incluso una transformación es un modelo. ¿Qué sucedería si t recibiese un metamodelo y produjese una transformación de identidad para modelos tipados por dicho metamodelo? El tipo de dicha transformación sería una expresión algo más compleja: t: M:ECore $\rightarrow$ $(M \rightarrow M)$. Aquí ECore es un metametamodelo, por lo que t recibe un metamodelo M y produce una transformación de tipo $M \rightarrow M$. En este ejemplo, el tipo de t depende del valor de entrada por lo que t es de tipo dependiente, y además t produce una transformación, por lo que es una transformación de alto orden [17]. Claramente el esquema de tipado original no es suficiente para manejar casos como éste. Una posible solución al problema es la definición de un sistema de tipos que sea específico para las particularidades de GMM.

SISTEMA DE TIPOS PARA GMM

Un sistema de tipos [5] es un caso particular de sistema formal. A su vez, un sistema formal es un sistema simbólico no interpretado que incluye reglas y axiomas mediante los cuales se pueden realizar deducciones. Esto significa que dada una expresión del sistema no es necesario comprender qué significa, para determinar mecánicamente si una deducción hecha a partir de éste es válida o no. Un sistema de tipos es entonces un sistema formal que clasifica expresiones en tipos. El propósito de un buen sistema de tipos es garantizar el buen comportamiento de un lenguaje, evitando que ocurran errores de tipo, que en este contexto son el uso de una operación sobre los argumentos equivocados o la aplicación de algo que no es una operación. La ejecución de una expresión bien tipada no debiese causar errores, mientras que la de una mal tipada probablemente sí los cause. Por ejemplo 1+2 es una expresión intuitivamente bien tipada, mientras que 1+'a' no lo es. La primera expresión produce 3 como resultado, mientras que la segunda nunca producirá resultado alguno, sino un error. Anteriormente se habló de un "buen" sistema de tipos. Un sistema de tipos es bueno si al menos satisface dos propiedades fundamentales: consistencia (el buen tipado implica buen comportamiento) y decidibilidad (la capacidad de decidir si una expresión está bien tipada o no).

Para poder mostrar que un sistema de tipos satisface las dos propiedades mencionadas es necesario incorporarle un ingrediente adicional: una semántica, la que le otorga un significado a las expresiones. Este agregado extiende al sistema formal convirtiéndolo de un sistema de tipos a un lenguaje formal. En el caso de GMM, el lenguaje sería "el lenguaje de ejecución de transformaciones en un megamodelo". Como fue indicado antes, las deducciones que se realizan en un sistema de tipos son específicamente la asignación de un tipo a una expresión. Las deducciones que se realizan en el sistema con el agregado referido son la asignación de un valor a una expresión. Es decir, las reglas del sistema de tipos (reglas de tipado) permiten deducir tipos, y las reglas semánticas (reglas de reducción) permiten computar valores. En la Figura 2 se puede apreciar un ejemplo simplificado de cada tipo de regla. La regla de tipado indica que si dos números son de tipo entero, entonces su suma es de tipo entero. La regla de reducción indica que el significado de la suma de dos elementos es la suma de los significados de los elementos (cualquiera de los dos elementos puede requerir ser reducido con la misma u otra regla del sistema). Lo importante es que con una regla podemos deducir el resultado de una suma, y con la otra el tipo de dicho resultado.

Figura 2

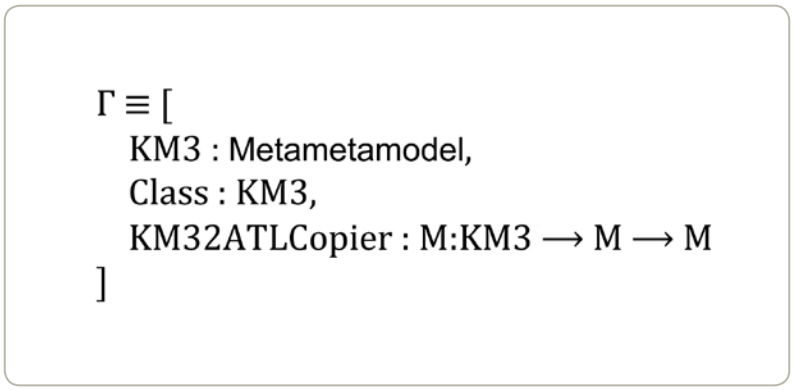


Regla de reducción y regla de tipado.

El sistema cGMM

El sistema de tipos para GMM desarrollado se denominó cGMM, y conjuga algunas ideas tomadas del Cálculo Predicativo de Construcciones Inductivas (pCIC), el cálculo utilizado por el asistente de pruebas Coq [6], con definiciones específicamente diseñadas para la realidad de GMM. Recordar que el sistema de tipos debe ser capaz de determinar qué expresión (i.e., qué aplicación de transformaciones) producirá un error de ejecución, por lo cual debe ser rechazada, y qué expresión es segura para su ejecución. Los conceptos involucrados en el sistema de tipos cGMM son los siguientes:

Figura 3



Ambiente representando a un megamodelo con tres artefactos.

Universos. En el sistema de tipos toda expresión debe tener necesariamente un tipo. Un tipo, es un caso particular de expresión, por lo que tiene que a su vez tener un tipo. Un universo es una constante que es el tipo de un tipo. En cGMM existe el universo Type que es el tipo de todos los tipos. Pero también existen otros universos que son el tipo de algunos tipos. Entonces todo tipo es de tipo Type y de alguno de estos (sub)universos. Algo similar ocurre con el número 5, que es un entero y en particular un natural.

Términos. Determinan la sintaxis de las expresiones válidas y se utilizan para representar los elementos contenidos en un megamodelo y las combinaciones que se hacen con ellos. Los términos se definen inductivamente. Los nombres de los universos, las variables y las constantes son los términos básicos. Además, si X e Y son dos términos válidos, entonces por ejemplo (X Y) es un término que representa la aplicación de X sobre el argumento Y, o (X,Y) es un término que representa un par. Notar que los términos se ocupan únicamente de la sintaxis, por lo que en el caso de la aplicación, es tarea de alguna regla determinar si X es o no una transformación y por ende si la aplicación es una expresión errónea.

Ambiente. Es un registro de las variables libres que ocurren en los términos. En otras palabras, si en un término se hace referencia a "algo que ya fue definido antes" entonces ese "algo" está almacenado en el ambiente. Algo similar ocurre en los lenguajes de programación, donde se declara una variable y se le asigna un valor. La declaración ocurre cronológicamente antes de la asignación, por lo que la expresión que le asigna el valor necesita que en el "entorno" exista la definición de la variable.

Juicios. Son afirmaciones, que en el caso de un sistema de tipos refieren al tipado de términos. Los juicios se realizan en el contexto de un cierto ambiente, y se puede demostrar que es válido o no. La regla de tipado mostrada en la Figura 2 contiene tres juicios: uno para el tipado de n_1, otro para el tipado de n_2 y otro para el tipado de $n_1 + n_2$ (la regla fue simplificada para no incluir ambientes en los juicios). En este caso, por tratarse de una regla, si los dos primeros resultan ser ciertos, entonces el tercero es necesariamente cierto.

Reglas de tipado. Determinan implicaciones entre juicios. Tal como se mencionara en el punto anterior, una regla indica que si ciertos juicios específicos son válidos, eso es suficiente para demostrar la validez de un juicio puntual. Un caso particular de regla es aquella que no necesita premisas (supuestos) para validar un juicio (conclusión). Este tipo de regla se denomina axioma.

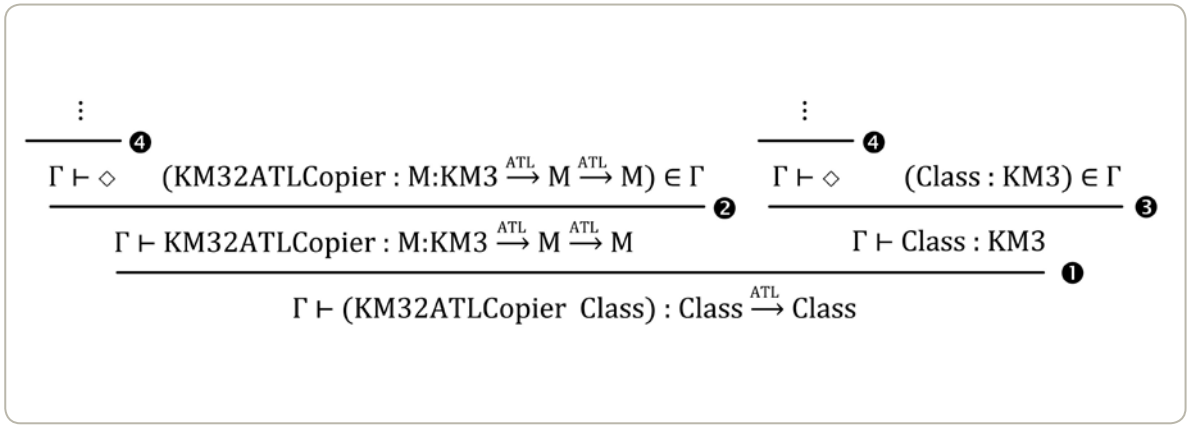
Derivaciones. Son deducciones basadas en reglas de tipado. A partir de un cierto juicio que se quiere demostrar como válido, el mecanismo consiste en encontrar una regla que tenga como conclusión una estructura igual a la del juicio (la idea es que siempre exista una única regla para esto). Sobre la conclusión se colocan las premisas de la regla, realizando las sustituciones de nombres que correspondan, y éstas se convierten en nuevos juicios que recursivamente deben demostrarse como válidos. Esta estructura toma usualmente la forma de un árbol y cada rama termina necesariamente con un axioma, para el cual no es necesario demostrar ninguna premisa. El árbol referido se conoce como árbol de derivación y sirve de demostración para el juicio que se ubica en su raíz.

Ejemplo

A continuación se ilustra la operación del sistema de tipos al realizar (parcialmente) la deducción del tipado de una expresión. Para ello se utiliza como base la transformación KM32ATLCopier [2], cuyo tipo fue utilizado como ejemplo en la sección anterior. El ambiente Γ se encuentra definido en la Figura 3 y representa a un megamodelo. En ese megamodelo existe un metametamodelo llamado KM3 (Metametamodel es un universo), existe Class que por ser tipado por un metametamodelo necesariamente es un metamodelo, y existe la transformación KM32ATLCopier ya comentada. Notar que si KM3 no hubiese estado declarado en Γ antes, entonces Class no podría haber sido declarado. Esto es análogo a lo que hubiese ocurrido en el megamodelo; primero debe existir el metametamodelo para poder definir un metamodelo a partir de él. Suponiendo que se aplica la transformación al metamodelo Class, queremos demostrar que el resultado de dicha aplicación es de tipo Class → Class (el índice no es importante en este momento). Este juicio, junto con su árbol de derivación, se muestra en la Figura 4.

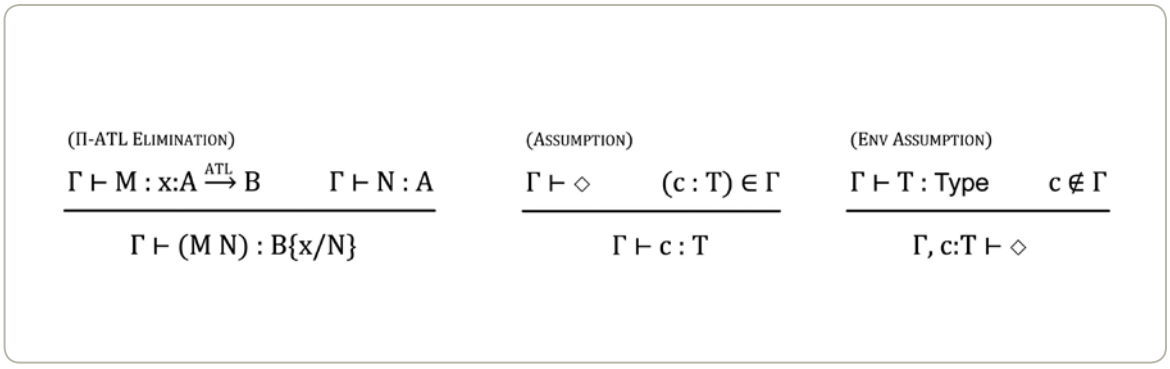
Los pasos en la deducción están numerados y pudieron realizarse gracias a la aplicación de las reglas de tipado de la Figura 5, que son un pequeño subconjunto de las

Figura 4



Árbol de derivación parcial de un juicio.

Figura 5



Reglas de tipado aplicadas al ejemplo.

definidas para cGMM. La regla (∏-ATL Elimination) indica cómo deducir el tipo de la aplicación de la transformación M al argumento N. Es el tipo de co-dominio de M donde cualquier ocurrencia libre de x es sustituida por N. Naturalmente, el tipo de N tiene que ser igual al dominio de M. Realizando la misma sustitución mediante la cual la conclusión de esta regla se convierte en la conclusión de (1) sobre las premisas de la regla, se obtienen las premisas de (1). Tal como se indicó antes, para que la conclusión valga es necesario demostrar las premisas. Para demostrar la premisa izquierda se toma el paso (2) aplicándose la regla (Assumption). Esta regla indica que si un ambiente está bien formado y que en ese ambiente la constante c tiene tipo T entonces se puede derivar que c tiene tipo T. La misma regla aplica para el paso (3). En ese caso, c se corresponde con Class y T con KM3. Tanto para (2) como para (3) la prueba de que la constante tipada pertenece al ambiente es trivial. Finalmente, con el paso (4) resta probar que el ambiente Γ está bien formado. Para ello se utiliza la regla (Env Assumption), la cual indica que si una constante c no está ya incluida en un ambiente Γ, y que en ese ambiente T es un tipo (con esto implícitamente Γ se considera bien formado), entonces se puede incluir c:T al ambiente y el resultado estará bien formado. Claramente la aplicación de esta regla no basta para finalizar la deducción completa, por lo que habrá que aplicar más reglas a las premisas de (Env Assumption), pero esta deducción parcial sirve para ilustrar cómo se construyen las deducciones.

Es importante recalcar que cualquier interpretación que se haya hecho aquí tanto de términos como de reglas persigue únicamente un fin ilustrativo. De hecho, el proceso anterior se lleva a cabo observando la forma de los términos y correspondiéndolos con las reglas (de ahí el nombre "formal") y sin necesitar conocimiento adicional de qué representa cada término. Esto hace que este proceso sea tanto confiable como mecanizable. Finalmente, respecto al árbol de derivación presentado, si éste se construye desde la raíz hacia las hojas el proceso se denomina chequeo de tipos (se está chequeando que el término sea de un tipo dado). Por el contrario, si el árbol se construye desde las hojas hacia la raíz, el proceso se denomina inferencia de tipos (se está encontrando un tipo para el término). En

cGMM siempre es posible inferir tipos para los términos (para los cuales exista un tipo, por supuesto).

Consistencia y decidibilidad

Para analizar la consistencia de un sistema de tipos es necesario considerar la semántica de los términos involucrados. Para cGMM se definió una semántica operacional estructural que describe pasos de computación individuales. Esto se expresa mediante un conjunto de reglas que permiten reducir términos mediante la relación $\Rightarrow$. A partir de esta relación se define su clausura transitiva $\Rightarrow^*$ que representa una cantidad arbitraria pero finita de pasos de reducción. Una función semántica entrega el valor al que reduce un término. En cGMM, para un término M y un ambiente Γ, la función semántica es parcial y se define como $[\![M]\!]_\Gamma \stackrel{\text{def}}{=} V$ si se cumple que $\Gamma \vdash M \Rightarrow^* V$, es decir, si el término M reduce en una cantidad finita de pasos a V. En cambio, la función semántica devuelve $\perp$ (el valor nulo) si la reducción de M en algún momento no puede avanzar más sin haber alcanzado un valor, o devuelve ∞ si la reducción de M nunca termina. En base a todas estas definiciones, es posible enunciar y demostrar un teorema de consistencia para el sistema de tipos. El mismo dice que un término arbitrario que esté bien tipado (es decir, que pase el chequeador de tipos) siempre reduce a un valor del mismo tipo que el término original. En símbolos, el enunciado es el siguiente: si $\Gamma \vdash M : A$ entonces $\Gamma \vdash [\![M]\!]_\Gamma : A$. Para demostrar este teorema es necesario contar con la demostración de varias propiedades básicas:

Propiedad 1: El valor $[\![M]\!]_\Gamma$, si existe (es decir, es diferente a $\perp$ o ∞), es único. Esta propiedad se conoce con el nombre de Church-Rosser.

Propiedad 2: Los términos bien tipados nunca divergen. Esta propiedad es conocida como *normalización fuerte* y significa que para términos bien tipados la función semántica nunca devuelve ∞.

Propiedad 3: Los términos que no terminan de reducir no son tipables. Esto significa que para términos bien tipados la función semántica nunca devuelve $\perp$.

Propiedad 4: Los tipos se preservan en cualquier paso de reducción. Esta propiedad es conocida como *subject reduction* y significa que si un término tiene un cierto tipo, después de un paso de reducción, el término resultante tiene el mismo tipo.

La demostración del teorema de consistencia se basa en casos aplicando las propiedades anteriores. Las propiedades 2 y 3 indican que para términos bien tipados, la función semántica no puede devolver ni $\perp$ ni ∞. Por lo tanto, para cualquier término bien tipado la función semántica debe devolver un valor, y en virtud de la propiedad 1 este valor tiene que ser único. Finalmente, gracias a la propiedad 4, el tipo de dicho valor es el mismo que el del término original.

La decidibilidad del sistema de tipos viene dada por la definición de un algoritmo de inferencia de tipos. Para un cierto término M considerado en un ambiente Γ, el algoritmo Type(M, Γ) devuelve el tipo de M si éste existe, o devuelve $\perp$ si no existe un tipo posible para M. El algoritmo de inferencia de tipos se define por inducción en la estructura de M, por lo que para cada tipo de término se define cómo se infiere su tipo. Para un término arbitrario, se aplica una única de las cláusulas, y recursivamente se aplican las cláusulas correspondientes a sus subtérminos. En una implementación, el funcionamiento del algoritmo consiste en construir desde las hojas hasta la raíz el árbol de derivación mediante el que se deduce que $\Gamma \vdash M : A$, donde A es la respuesta del algoritmo. Se debe tener en cuenta que todos los términos en Γ se encuentran ya tipados y que solamente una cláusula inductiva aplica a M. Esta información es suficiente para construir el tipo A. El detalle del algoritmo de inferencia de tipos puede encontrarse en [18].

> Un sistema de tipos es bueno si al menos satisface dos propiedades fundamentales: consistencia (el buen tipado implica buen comportamiento) y decidibilidad (la capacidad de decidir si una expresión está bien tipada o no).

CONCLUSIÓN

MDE promueve centrar el esfuerzo del desarrollo de software en artefactos de mayor nivel de abstracción que el código fuente, de forma de controlar la complejidad de los sistemas computacionales actuales. Dichos artefactos, al ser definidos en forma precisa, pueden ser procesados mecánicamente mediante transformaciones de modelos, las cuales automatizan el conocimiento experto y cuya aplicación manual es propensa a errores. En este contexto (partes de) un proceso de desarrollo de software podría realizarse por medio de un conjunto de transformaciones. Sin embargo la cantidad y complejidad de artefactos involucrados en un proyecto de escala industrial introduce la necesidad de contar con mecanismos de gestión para estos. GMM introduce la noción

> MDE promueve centrar el esfuerzo del desarrollo de software en artefactos de mayor nivel de abstracción que el código fuente, de forma de controlar la complejidad de los sistemas computacionales actuales.

de megamodelo como un repositorio de artefactos de MDE donde, en particular, es posible ejecutar transformaciones. AM3 es la herramienta que implementa la noción de megamodelo. Tanto GMM como AM3 no contaron originalmente con una definición apropiada de tipos, lo cual es un reflejo de la inmadurez de este aspecto dentro de MDE. En AM3 se permiten ciertas ejecuciones de transformaciones que conducen a errores. Para solucionar este problema se definió un sistema de tipos consistente y decidible específicamente diseñado para GMM, y se generó una implementación en la forma de plug-in que será integrado con AM3. Ese trabajo es la única aplicación de métodos formales a GMM hasta el momento, y una de las pocas con impacto tanto teórico como práctico al área de MDE en general. Adicionalmente al sistema de tipos, la definición de una semántica utilizada para la prueba de consistencia derivó en la formalización de un lenguaje de programación básico sobre megamodelos. Actualmente las "sentencias" del lenguaje se ejecutan individualmente como producto de la interacción del usuario con AM3, mediante su interfaz gráfica, pero una versión textual permitiría la definición de programas que manipulan megamodelos, pudiendo evolucionar incluso con construcciones más elaboradas en lo que a composición de transformaciones se refiere. Formalizar conceptos de la Ingeniería de Software no es una tarea simple, más si se busca un impacto en la práctica profesional. Por esta razón este tipo de esfuerzos debe enfocarse en aquellas situaciones que lo ameriten, como por ejemplo cuando una demostración es requerida. Desde notaciones hasta lenguajes de transformaciones de modelos, existe una gran cantidad de escenarios en MDE que pueden beneficiarse de una aplicación de métodos formales. BITS

REFERENCIAS

[1] AM3 Project. http://www.eclipse.org/gmt/am3/, 2009.

[2] ATL Transformations Zoo. http://www.eclipse.org/m2m/atl/atlTransformations/, 2009.

[3] J. Bézivin. On the Unification Power of Models. Software and System Modeling. 4(2):171-188, 2005.

[4] J. Bézivin, F. Jouault, P. Rosenthal, P. Valduriez. Modeling in the Large and Modeling in the Small, MDAFA 2004. LNCS 3599, pp. 33-46, 2004.

[5] L. Cardelli. Type Systems. The Computer Science and Engineering Handbook, pp. 2208-2236. CRC Press, 1997.

[6] The Coq Project Team. The Coq Proof Assistant Reference Manual. Version 8.2. http://coq.inria.fr/doc-eng.html, 2009.

[7] K. Czarnecki, S. Helsen. Feature-based Survey of Model Transformation Approaches. IBM Systems Journal, 45(3):621-646, 2006.

[8] M. Didonet Del Fabro, J. Bézivin, F. Jouault, E. Breton, G. Gueltas. AMW: A Generic Model Weaver. 1ères Journées sur l'Ingénierie Dirigée par les Modèles, pp. 105-114, 2005.

[9] F. Jouault, J. Bézivin. KM3: A DSL for Metamodel Specification. FMOODS 2006, LNCS 4037, pp. 171-185, 2006.

[10] F. Jouault, J. Bézivin, I. Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. GPCE 2006, pp. 249-254, 2006.

[11] F. Jouault, I. Kurtev. Transforming Models with ATL. MoDELS 2005, LNCS 3844, pp. 128-138, 2005.

[12] I. Kurtev, J. Bézivin, M. Aksit. Technological Spaces: An Initial Appraisal. DOA'2002 Federated Conferences, pp. 1-6, 2002.

[13] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.1. http://www.omg.org/spec/QVT/1.1, January 2011.

[14] OMG. OMG Unified Modeling Language (OMG UML) Infrastructure Specification. Version 2.3. http://www.omg.org/spec/UML/2.3/Infrastructure/PDF, 2010.

[15] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering.IEEE Computer, 39(2):25-31, 2006.

[16] J. Steel, J-M. Jézéquel. On Model Typing. Software and System Modeling, 6(4):401-413, 2007.

[17] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, J. Bézivin. On the Use of Higher-Order Model Transformations. ECMDA-FA 2009. LNCS 5562, pp. 18-33, 2009.

[18] A. Vignaga. A Type System for Global Model Management. PhD Thesis, Universidad de Chile, 2011.