View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Centro de Servicios en Gestión de Información

Migración de software C/C++ a plataformas móviles a partir de MDD *(Model Driven Development)*

Alumnos:
Maximiliano Pablo Duthey
Carolina María Spina

Directora:
Lic. Liliana Favre

Índice general

1. Introducción **6**
1.1. Motivación . . . 7
1.2. Descripción del proceso . . . 8

2. Estado del Arte **10**
2.1. Model Driven Architecture . . . 10
2.1.1. Modelos . . . 11
2.1.2. Metamodelos . . . 12
2.1.2.1. Meta Object Facility (MOF) . . . 14
2.1.3. Transformaciones . . . 14
2.1.3.1. Query-View-Transformation (QVT) . . . 15
2.1.3.2. ATLAS Transformation Language . . . 16
2.2. Ingeniería Inversa . . . 19
2.3. Ingeniería Directa . . . 21
2.4. Architecture Driven Modernization . . . 23
2.4.1. Knowledge Discovery Metamodel (KDM) . . . 24
2.4.2. Abstract Syntax Tree Metamodel (ASTM) . . . 25

3. Metamodelo del Lenguaje HAXE **26**
3.1. Metaclases Principales . . . 27
3.1.1. Metamodelado de módulos . . . 27
3.1.2. Metamodelado de comentarios . . . 28
3.1.3. Metamodelado de paquetes . . . 28
3.2. Metaclases de Tipos . . . 28
3.2.1. Metamodelado de clases e interfaces . . . 28
3.2.2. Metamodelado de tipos de datos algebraicos . . . 29
3.2.3. Metamodelado de tipos abstractos . . . 29
3.2.4. Metamodelado de alias de tipos Typedef . . . 29
3.2.5. Metamodelado de metadatos . . . 29
3.2.6. Metamodelado de elementos parametrizados . . . 29
3.3. Metaclases de Expresiones . . . 30
3.3.1. Metamodelado de expresiones compuestas . . . 30
3.3.2. Metamodelado de constantes . . . 30
3.3.3. Metamodelado de operaciones aritméticas . . . 30

3.3.4. Metamodelado de vectores o arreglos . . . 30
3.3.5. Metamodelado de expresiones de selección . . . 30
3.3.6. Metamodelado de expresiones de selección múltiple . . . 31
3.3.7. Metamodelado de expresiones de iteración . . . 31
3.3.8. Metamodelado de expresiones de corte y continuación del flujo de ejecución . . . 31
3.4. Metamodelado de declaraciones de variables . . . 31
3.4.1. Metamodelado de atributos de instancia y de clase . . . 32
3.5. Metaclases de funciones y métodos . . . 32
3.6. Metaclases de expresiones de acceso y construcción de objetos . . . 32
3.7. Metaclases de expresiones de conversión y chequeo de tipos . . . 33
3.8. Metaclases de expresiones de manejo de excepciones . . . 33

4. Metamodelo del Lenguaje C/C++ 34
4.1. Metaclases Principales . . . 34
4.2. Metaclases de Expresiones . . . 36
4.2.1. Metaclases de Expresiones de Selección . . . 36
4.2.2. Metaclases de Expresiones de Iteración o Bucles . . . 37
4.2.3. Metaclases de Expresiones de Selección Múltiple . . . 38
4.2.4. Metaclases de Expresiones Unarias y Binarias . . . 39
4.2.5. Metaclases de Expresiones de manejo de Excepciones . . . 40
4.2.6. Metaclases de Expresiones Constantes . . . 41
4.2.7. Metaclases de Expresiones con Arreglos . . . 41
4.2.8. Metaclases de Expresiones de Salto . . . 42
4.2.9. Metaclases de Expresiones Compuestas . . . 42
4.2.10. Metaclases de Expresiones de Conversión de Tipos . . . 43
4.3. Metaclases de Variables: Declaración y Acceso . . . 43
4.4. Metaclases de Elementos Parametrizados . . . 45
4.5. Metaclases de Métodos . . . 45
4.6. Metaclases de Tipos . . . 47
4.6.1. Metaclases de Clases, Estructuras y Uniones . . . 48
4.6.2. Metaclases de Tipos Primitivos . . . 49
4.6.3. Metaclases de tipos Enum . . . 49
4.7. Metaclases de Comentarios . . . 50

5. Proceso de migración 51
5.1. Ingeniería Inversa . . . 52
5.1.1. Descripción del proceso de Ingeniería Inversa . . . 52
5.1.2. ANTLR (ANother Tool for Language Recognition) . . . 53
5.1.3. Programa Java desarrollado . . . 53
5.2. Transformación . . . 55
5.2.1. Elementos Principales . . . 56
5.2.2. Tipos Primitivos . . . 56
5.2.3. Clases y Constructores . . . 57
5.2.4. Enumeraciones . . . 57
5.2.5. Expresiones . . . 57

5.2.6. Declaración de Variables 60
5.2.7. Métodos 60
5.2.8. Variables 61
5.2.9. Comentarios 61
5.3. Ingeniería Directa 61
5.3.1. Acceleo 61

6. Conclusiones 62

Índice de figuras

1.1. Proceso de migración 9

2.1. Modelos MDA 12
2.2. Niveles de Modelado 13
2.3. Transformación MDA (en [36]) 15
2.4. Proceso MDA 15
2.5. Arquitectura de las herramientas de ingeniería inversa 20
2.6. Estrategias de generación de código 22
2.7. Estrategia PIM-to-Framework Specific Model (FSM)-to-CPC . . 23

3.1. Lenguajes de destino de HAXE 26

4.1. Metaclases relacionadas con CppModel 35
4.2. Metaclases relacionadas con CppPathReference y CppPathReferentiable 36
4.3. Metaclases correspondientes a las expresiones de selección 37
4.4. Metaclases de expresiones iterativas 38
4.5. Metaclases de expresiones de selección múltiple 39
4.6. Metaclases de expresiones unarias y binarias 39
4.7. Operadores usados en expresiones unarias y binarias 40
4.8. Metaclases de expresiones de manejo de excepciones 41
4.9. Metaclases de expresiones constantes 41
4.10. Metaclases de expresiones con arreglos 42
4.11. Metaclases de expresiones de corte y continuación 42
4.12. Metaclases de expresiones compuestas 43
4.13. Metaclases de expresiones de casting 43
4.14. Metaclases de declaración de variables 44
4.15. Metaclases de acceso a variables 45
4.16. Metaclases de métodos 46
4.17. Metaclases de invocación a métodos 47
4.18. Metaclases de tipos 48
4.19. Metaclases de clases, estructuras y uniones 49
4.20. Metaclases de tipos primitivos 49
4.21. Metaclases de tipos Enum 50

4.22. Metaclases de comentarios 50

5.1. Proceso de migración 52
5.2. Clases principales de la aplicación 54
5.3. Diagrama de relaciones dentro del paquete mmclass 54
5.4. Diagrama de paquetes del proyecto 55
5.5. Transformación entre tipos primitivos de C++ a Haxe 56

Capítulo 1

Introducción

Actualmente los dispositivos móviles acompañan a los usuarios en todo momento y lugar, y se prevé que serán el principal medio de acceso a Internet en los próximos años, sin embargo, el desarrollo de aplicaciones de software móviles no está lo suficientemente maduro.

La proliferación de diferentes plataformas móviles ha forzado a los desarrolladores a definir enfoques que permitan simplificar el desarrollo de aplicaciones . Los autores remarcan que dos de los principales desafíos de la ingeniería de software de aplicaciones móviles son por un lado, la creación de interfaces de usuario que abarquen diferentes clases de dispositivos móviles y por otro, brindar aplicaciones reutilizables en múltiples plataformas.

Desarrollar una aplicación de software para un dispositivo móvil implica adoptar y entender las características de estos dispositivos y sus restricciones. Si bien éstos cuentan con características avanzadas también se presentan importantes restricciones. Por ejemplo, incorporan interfaces de entrada más intuitivas, usualmente, pantallas táctiles, bases de datos integradas, soporte multimedia y mecanismos de comunicación y geolocalización. También se presentan importantes restricciones en cuanto al tamaño de la pantalla disponible, capacidad de procesamiento, la utilización de memoria primaria y las bibliotecas de desarrollo disponibles .

Un problema actual es la rápida proliferación de plataformas móviles. El alto costo y la complejidad técnica de desplegar una aplicación a un amplio espectro de plataformas, fuerza a desarrollar aplicaciones alineadas a cada tipo de dispositivo. Varias empresas tienen grupos de desarrollo para cada tipo de plataforma redoblando esfuerzos para aplicaciones móviles de funcionalidad similar. En aplicaciones multiplataforma, los desarrolladores prefieren implementar una aplicación por vez, y desplegarla en diversas plataformas con un mínimo esfuerzo. En esta dirección, surgió el proyecto HAXE, que es un lenguaje de programación de propósitos generales diseñado para que los desarrolladores puedan, utilizando un solo lenguaje y un conjunto de librerías, abarcar distintas plataformas de manera eficaz. Actualmente, el compilador de HAXE genera código para las siguientes tecnologías: JavaScript,

Flash, NekoVM, PHP, C++ y C#.

Otro desafío en estos desarrollos es lograr integrar variedad de información e interoperabilidad de herramientas. Model Driven Development (MDD) es considerado un enfoque promisorio para afrontarlo. MDD define un amplio rango de desarrollos basados en el uso de modelos como entidades de primera clase. Una realización específica de MDD, propuesta por OMG (Object Management Group), es la arquitectura MDA (Model Driven Architecture).

MDA propone separar la especificación de la funcionalidad del sistema de su implementación sobre una plataforma en una tecnología específica y controlar la evolución del software desde modelos abstractos a implementaciones tendiendo a aumentar el grado de automatización. Una de las características esenciales de MDA es que todos los artefactos involucrados en un proceso de desarrollo se representan a partir del lenguaje de metamodelado MOF (Meta Object Facility). Otro concepto fundamental es el de transformaciones entre modelos. En MDA, una transformación es la especificación de mecanismos para convertir elementos de un modelo en elementos de otro modelo y el estándar propuesto por OMG para especificar transformaciones es el meta modelo QVT (Query, View, Transformation).

En el contexto de MDA, OMG está involucrado en el desarrollo de estándares para la modernización de sistemas a través de la iniciativa ADM (Architecture Driven Modernization) [2] que lleva a cabo procesos de modernización considerando los principios esenciales de MDA. ADM define estándares específicos para modernización.

El proceso de modernización incluye tres etapas: ingeniería inversa, reestructuración e ingeniería forward. La ingeniería inversa, el proceso de analizar los artefactos de software existentes para extraer información proveer vistas de alto nivel del sistema, es una etapa crucial dentro de la modernización del sistema. En el contexto de ADM, la meta de la ingeniería inversa es descubrir el conocimiento del sistema existente y producir modelos en diferentes niveles de abstracción. Estos modelos serán el punto de partida para el proceso de reestructuración y la posterior generación del nuevo código.

El objetivo de esta tesis es trabajar sobre un escenario de modernización: la migración de código C/C++ a plataformas móviles a través del lenguaje HAXE.

1.1. Motivación

Es frecuente la necesidad de migrar componentes y aplicaciones de software desarrollados en C/C++ a plataformas móviles y por ende es conveniente contar con procesos genéricos reutilizables que la soporten. Esto puede lograrse a partir de procesos dirigidos por modelos integrados con el lenguaje multiplataforma HAXE.

En trabajos existentes no se ha encontrado una integración de C/C++ con HAXE a partir de MDA y consideramos de interés definir procesos de migración que los integren teniendo en cuenta la cantidad de software de calidad desarrollado en estos lenguajes que los programadores de aplicaciones móviles

deben adaptar. Los procesos deberían incluir el análisis de código existente para extraer modelos y la generación de código a partir de estos modelos para múltiples plataformas móviles.

En MDD, las transformaciones entre modelos se expresan en lenguajes específicos que permiten a los desarrolladores concentrarse en los aspectos conceptuales de las relaciones entre modelos y delegar el proceso de transformación. La generación de modelos a partir de transformaciones entre metamodelos en MDD apunta a generar modelos "correctos por construcción" con respecto a la especificación de metamodelos.

Este enfoque requiere contar con un metamodelo MOF de C/C++, produciendo además el beneficio de otorgar a los desarrolladores MDD este metamodelo y así la posibilidad de abordar otros escenarios de modernización que involucren al lenguaje C/C++.

1.2. Descripción del proceso

El proceso de migración de C/C++ a plataformas móviles a través del lenguaje HAXE puede verse gráficamente en la Figura 1.1. Los pasos a seguir del mismo son los siguientes:

1. Especificar un metamodelo del lenguaje origen (C/C++).
2. Especificar un metamodelo del lenguaje destino (HAXE).
3. Ingeniería inversa de código C/C++ para creación del modelo que conforme al metamodelo correspondiente.
4. Transformación del modelo del lenguaje origen (C/C++) a un modelo que conforme al metamodelo del lenguaje destino (HAXE).
5. Realizar ingeniería directa desde el modelo del lenguaje destino (HAXE) a texto.
6. Compilación del código fuente a plataformas móviles.

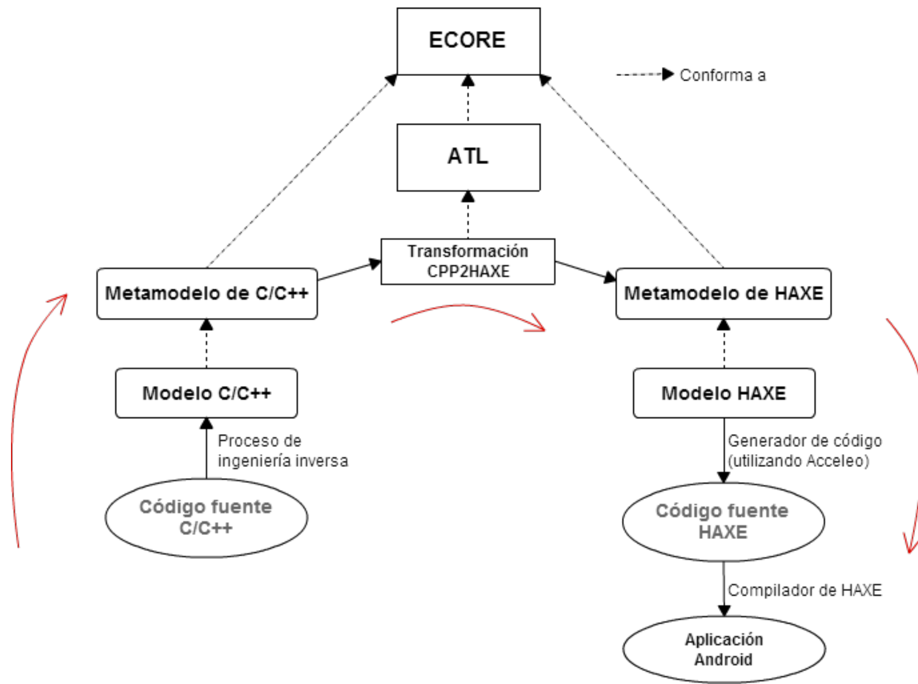

Figura 1.1: Proceso de migración

Capítulo 2

Estado del Arte

2.1. Model Driven Architecture

El Desarrollo Dirigido por Modelos (MDD) es un paradigma que utiliza modelos como artefactos primarios en el proceso de desarrollo. El Object Management Group (OMG) propone la Arquitectura Dirigida por Modelos (MDA) como la realización de esta iniciativa, donde los lenguajes de modelado y de transformaciones son estandarizados. MDA fue adoptado como estándar en 2001 y está basado en otros estándares de modelado del OMG como UML (Unified Modeling Language) , MOF (Meta Object Facility) y CWM (Common Warehouse Metamodel). Su objetivo principal es separar la especificación de la funcionalidad del sistema de su implementación sobre una plataforma y tecnología específica, y controlar la evolución del software desde modelos abstractos a implementaciones tendiendo a aumentar el grado de automatización.

La idea fundamental detrás de MDA es el uso de modelos para guiar el ciclo de vida completo de un sistema. Todos los artefactos generados durante el proceso de desarrollo, tales como la especificación de requerimientos, descripción de la arquitectura, descripciones de diseño y código, son vistos como modelos y el objetivo es tender a construir modelos capaces de ser interpretados de manera automática.

En especial, MDA pone énfasis en la creación de un modelo conceptual del sistema, independiente de la plataforma. Este modelo puede ser luego traducido automáticamente a un modelo específico para la plataforma seleccionada y posteriormente a código. La automatización de este proceso es esencial, y se logra definiendo transformaciones entre modelos que permitan a herramientas llevar a cabo toda, o al menos parte, de la conversión. Una premisa clave en MDD es que los programas se generen automáticamente a partir de sus correspondientes modelos.

Los beneficios que otorga el uso del Desarrollo Dirigido por Modelos en el

desarrollo de software son:

Portabilidad: MDA parte de un modelo de alto nivel de abstracción, independiente de cualquier plataforma, y ese modelo puede ser transformado en numerosos modelos que dependen de diferentes plataformas. Por lo tanto, cada modelo especificado independientemente de las plataformas es un modelo completamente portable al que, para cada nueva tecnología y plataforma que surja, solo será necesario definir las transformaciones correspondientes. Esto permite que se desarrollen nuevos sistemas basados en los modelos existentes de alto nivel.

Productividad: el aumento en la productividad está dado por la existencia de numerosas herramientas que permiten automatizar gran parte de las transformaciones entre modelos de distintos niveles de abstracción.

Interoperabilidad: se debe establecer la relación entre los distintos modelos dependientes de plataformas, para alcanzar la interoperabilidad entre diferentes plataformas.

Mantenimiento y documentación: los modelos de alto nivel de abstracción son utilizados para generar modelos dependientes de una plataforma y estos últimos para generar código fuente. Los modelos independientes de plataforma proveen la documentación de alto nivel necesaria para cualquier sistema.

2.1.1. Modelos

Los modelos son el componente principal dentro de MDA. Se podría decir que un modelo es una descripción de un sistema, o parte de él, escrito en un lenguaje bien definido. Es posible crear muchos tipos de modelos diferentes para un sistema, desde modelos abstractos que especifican la funcionalidad en forma independiente de la plataforma, hasta modelos concretos vinculados con plataformas, tecnologías e implementaciones específicas.

En MDA, pueden distinguirse cuatro niveles de modelado en función de su grado de abstracción. Ellos son:

Computation-Independent Model (CIM): es el nivel más abstracto del modelado. Representa el contexto, los requerimientos y el propósito del sistema desde un punto de vista independiente de la computación. Es conocido como modelo de dominio y ayuda a reducir la brecha existente entre los expertos en un dominio y los ingenieros de software. Puede consistir en modelos UML y otros modelos de requerimientos.

Platform-Independent Model (PIM): este nivel describe el comportamiento y la estructura del sistema, independientemente de la plataforma. Puede verse como un modelo del sistema que será ejecutado por una máquina virtual independiente de cualquier tecnología.

Platform-Specific Model (PSM): este modelo contiene toda la información necesaria del comportamiento y la estructura del sistema para una plataforma específica. Esta vista combina la especificación del sistema capturada en el PIM con los detalles particulares de la tecnología en que se implementará.

Implementation-Specificil Model (ISM): es una descripción del sistema a nivel de código.

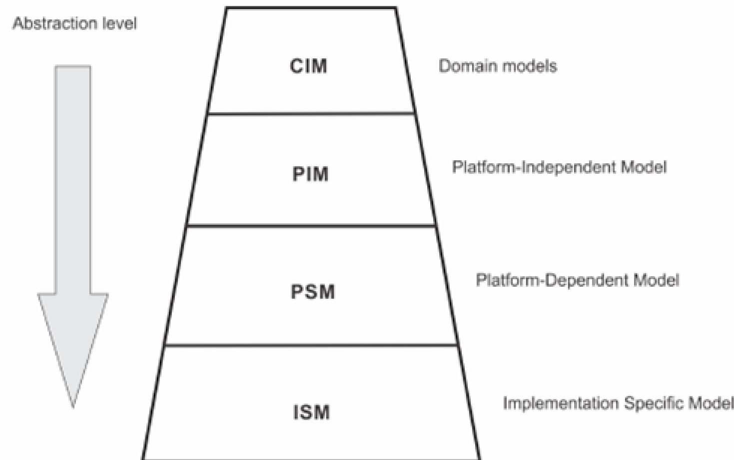

Figura 2.1: Modelos MDA

2.1.2. Metamodelos

En el contexto de MDA, es necesario que los modelos estén escritos en un lenguaje bien definido, es decir, un lenguaje formal. Una manera de definir tales lenguajes es mediante el uso de metamodelos. Un metamodelo es un modelo que define un lenguaje para expresar modelos. Es una descripción de todos los conceptos que pueden ser usados en un modelo. Cada uno de los elementos presentes en un modelo debe estar definido en el metamodelo del lenguaje utilizado.

Como un metamodelo es también un modelo, el mismo debe estar escrito en un lenguaje bien definido. Este lenguaje, a su vez, puede ser expresado mediante un modelo. Por lo que un meta-metamodelo define un lenguaje para escribir metamodelos.

El OMG propone una arquitectura de modelado de cuatro capas para sus estándares, donde cada nivel (excepto el superior) se caracteriza por conformar al nivel que está sobre él. Éstas capas se denominan M0, M1, M2 y M3:

Nivel M0 (sistema): en este se encuentran las instancias reales del sistema, es decir, los datos o información de dominio que se quiere representar.

Nivel M1 (el modelo del sistema): los conceptos que aparecen en este nivel son clasificaciones o categorizaciones de las instancias que aparecen en el nivel M0. Cada elemento en el nivel M0 será una instancia de un elemento del nivel M1.

Nivel M2 (el modelo del modelo): cada elemento del nivel M1 es una instancia de un elemento de M2, y cada elemento de M2 clasifica elementos de M1. El modelo que reside en este nivel se llama metamodelo. Cuando se construye un metamodelo se está definiendo un lenguaje con el cual escribir modelos. En este nivel se encuentran el metamodelo UML, CWM (Common Warehouse Model) y SPEM (Software Process Engineering Metamodel) entre otros estándares de OMG.

Nivel M3 (el modelo del modelo del modelo): es la capa superior de la arquitectura. Al igual que en los niveles anteriores, cada elemento del nivel M2 es una instancia de un elemento de M3, y cada elemento de M3 clasifica elementos de M2. El modelo en este nivel se llama meta-metamodelo que define el lenguaje para definir lenguajes de modelado. El lenguaje estándar de OMG del nivel M3 es MOF.

Se podría seguir agregando niveles en la jerarquía, pero como esta separación de niveles es superficial, ya que simplemente ayuda a razonar sobre modelos y clasificaciones, OMG ha declarado que todos los elementos del nivel M3 deben ser definidos como instancias de los conceptos del mismo nivel M3.

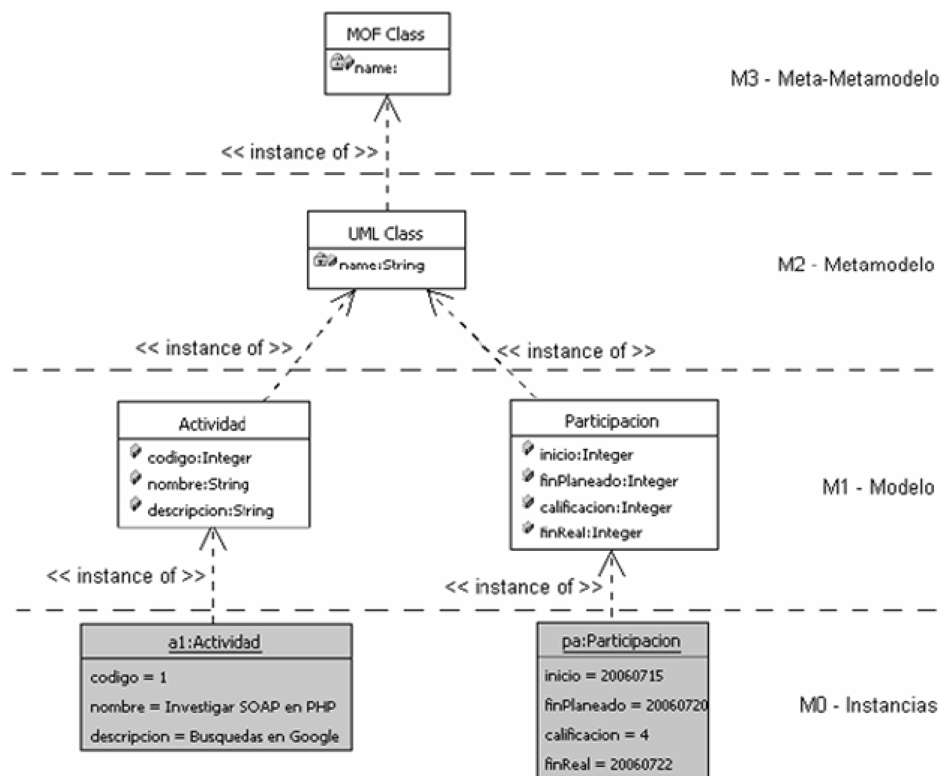

Figura 2.2: Niveles de Modelado

2.1.2.1. Meta Object Facility (MOF)

MDA se basa en el uso de un lenguaje para definir metamodelos denominado Meta Object Facility (MOF). MOF provee un lenguaje formal para describir modelos y, a su vez, proporciona un lenguaje común para manipular e intercambiar dichos modelos. Al mismo tiempo, proporciona la base para definir y ejecutar transformaciones, las cuales son un componente esencial dentro de MDA.

El requisito esencial para integrar diferentes lenguajes y herramientas es que los lenguajes involucrados se definan (modelen) en MOF.

MOF es esencialmente un conjunto mínimo de conceptos que pueden utilizarse para definir otros lenguajes de modelado. Los conceptos de MOF derivan de los conceptos definidos en UML Infrastructure en el paquete Core. En la definición de MOF 2.0 se incluyen dos meta-metamodelos: EMOF (Essential MOF) y CMOF (Complete MOF). El primero favorece la simplicidad de implementación por sobre la expresividad, mientras que el segundo es más expresivo, pero más complejo. La variante ECore que ha sido definido en el framework de modelado Eclipse es más o menos similar a EMOF.

Las construcciones de modelamiento más importantes que utiliza MOF son:
Clases: que modelan metaobjetos MOF.
Asociaciones: que modelan relaciones binarias entre metaobjetos.
Tipos de datos: que modelan tipos primitivos.
Paquetes: que modularizan los modelos.

2.1.3. Transformaciones

El Object Management Group define transformación de modelos en el contexto de MDA como "el proceso de convertir un modelo en otro modelo del mismo sistema". La transformación de modelos es un concepto central dentro del enfoque de MDD, ya que proporciona un mecanismo que permite, por un lado, conseguir una representación de todo un sistema al combinar sus diferentes modelos; por otro lado, organizar los modelos del sistema en distintos niveles de abstracción; y, por último, derivar el código fuente del sistema mediante sucesivas transformaciones entre modelos, desde la representación más abstracta, pasando por la capa intermedia de modelos. Para ello, toda transformación toma un modelo como entrada y genera otro modelo como salida, que puede estar escrito en el mismo lenguaje o en un lenguaje diferente al del modelo de entrada.

Las transformaciones son una parte fundamental dentro de MDA, dado que constituyen el mecanismo que permite convertir de un modelo a otro, por ejemplo, de un PIM a un PSM y de PSM a código. Realizar una transformación entre modelos requiere de un claro entendimiento de la sintaxis abstracta y la semántica de los modelos origen y destino.

En MDA, las transformaciones también pueden ser vistas como modelos que conforman a un metamodelo.

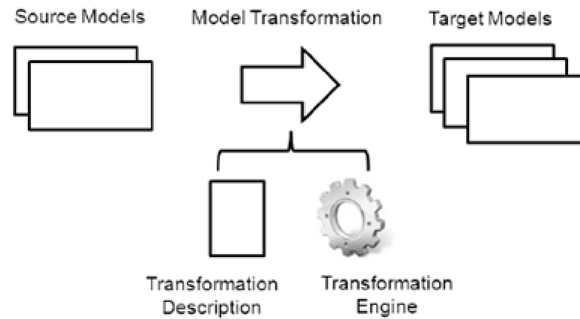

Figura 2.3: Transformación MDA (en [36])

Un desarrollo MDA que evoluciona desde modelos abstractos a implementaciones, distinge las siguientes etapas:

1. Construir un CIM
2. Construir un PIM en un alto nivel de abstracción, independiente de una tecnología específica y relacionado al CIM.
3. Transformar al PIM en uno o más modelos dependientes de la plataforma (PSM).
4. Transformar PSM a código.

Figura 2.4: Proceso MDA

Con el objetivo de reducir el esfuerzo y evitar la generación de errores, se ha buscado automatizar el proceso de transformación de distintas formas. Por ejemplo, a través lenguajes de propósito general, representaciones intermedias o lenguajes específicos de transformación. La alternativa de utilizar lenguajes específicos de transformación es la que ofrece el mayor potencial.

2.1.3.1. Query-View-Transformation (QVT)

La especificación MOF 2.0 Query, View, Transformation es el estándar propuesto por el OMG para especificar transformaciones entre modelos. El acrónimo QVT hace referencia a:

Query: es una expresión consulta para seleccionar elementos de un modelo. El resultado es una o más instancias de los tipos definidos sobre el metamodelo fuente.

View: vistas de metamodelos MOF.

Transformation: proceso que genera un modelo destino a partir de un modelo origen.

El estándar QVT está definido por un metamodelo basado en MOF que comprende tres comprende tres lenguajes para modelar transformaciones: dos de naturaleza declarativa llamados Relations y Core, y uno de naturaleza imperativa llamado Operational Mappings.

QVT define un estándar para transformar un modelo origen en un modelo destino, donde ambos deben ser instancias de metamodelos MOF. Otra característica que posee es que la transformación es considerada en sí misma un modelo que se ajusta a un metamodelo MOF. Una transformación QVT está compuesta por un conjunto de reglas y un conjunto de parámetros de los modelos asociados con la transformación.

El estándar QVT incluye sólo transformaciones modelo a modelo (M2M), en las que los modelos son instancias de metamodelos MOF 2.0. Las transformaciones de tipo modelo a texto y texto a modelo no están contempladas.

2.1.3.2. ATLAS Transformation Language

ATL (ATLAS Transformation Language) es un lenguaje de transformación de modelos y un conjunto de herramientas que dan soporte a estas transformaciones desarrollado por el grupo AtlanMod (Inria, Ecole des Mines de Nantes & LINA) como una implementación de la propuesta QVT del OMG. La especificación del lenguaje ATL está dada tanto por un metamodelo como por una sintaxis textual concreta.

ATL permite la programación declarativa e imperativa. La escritura declarativa permite simplemente expresar correspondencias entre los elementos del origen y del destino. La escritura imperativa facilita la especificación de construcciones que difícilmente se pueden expresar de manera declarativa.

El ATL Integrated Development Enviroment (IDE) se encuentra desarrollado sobre la plataforma Eclipse. Proporciona una serie de herramientas de desarrollo que tienen como objetivo facilitar el diseño de las transformaciones ATL.

Un programa de transformación ATL se compone de reglas que definen cómo se emparejan y navegan los elementos del modelo origen para crear e inicializar los elementos del modelo destino. Cada uno de estos programas constituye una transformación y se especifica mediante un módulo ATL.

Un módulo ATL está compuesto por los siguientes elementos:

Encabezado: define algunos atributos que se relacionan con el módulo de transformación.

Import: es una sección opcional que permite importar librerías ATL existentes.

Helpers: el conjunto de helpers en ATL puede verse como el equivalente de los métodos Java.

Reglas: el conjunto de reglas define la forma en que se generan los elementos del metamodelo destino a partir de los elementos del metamodelo origen.

Encabezado

La sección de encabezado define el nombre del módulo de transformación y los nombres de las variables que identifican a los modelos origen y destino. La sintaxis es la siguiente:

```
module module_name;
create out_model_name : out_metamodel_name
[from|refining] input_model_name :
input_metamodel_name;
```

El nombre del archivo ATL que contiene el código del módulo debe corresponderse con el nombre del mismo y debe tener la extensión .atl. Es posible declarar más de un modelo de entrada o de salida separándolos por coma. Dado que el nombre de un modelo constituye su identificador en el contexto de la transformación, el nombre de los modelos declarados deben ser únicos.

Import

Esta sección opcional permite declarar qué librerías ATL deben ser importadas. La declaración se realiza de la siguiente manera:

```
uses extensionless_library_file_name;
```

Helpers

El conjunto de helpers en ATL puede verse como el equivalente a los métodos Java. Permiten definir código ATL que puede ser llamado desde diferentes puntos de una transformación. Un helper consta de los siguientes elementos:

- un nombre (que se corresponde con el nombre del método);
- un tipo de contexto, que define el contexto en el que el atributo se define;
- un tipo de valor de retorno;
- una expresión ATL que representa el código del helper;
- un conjunto de parámetros opcional, en el que cada parámetro se define por un par (nombre del parámetro, tipo del parámetro).

```
helper [context metamodel!element]
def : helper_name(param_name: param_type):
return_type = OCL_expression;
```

Reglas

En ATL se pueden distinguir tres tipos de reglas que se corresponden con los dos modos de programación que provee ATL: las *matched rules* (asociadas a la programación declarativa), las *lazy rules* y las *called rules* (asociadas a la programación imperativa).

Matched rules: constituyen el núcleo de la transformación declarativa. Cada una de estas reglas se corresponde con un tipo de elemento determinado del metamodelo de entrada y genera como resultado uno o más tipos de elementos del metamodelo de salida. La regla especifica la forma en que los elementos del modelo de salida deben inicializarse a partir de los elementos seleccionados del metamodelo de entrada.

Una matched rule comienza con la palabra reservada rule y está compuesta por dos secciones obligatorias, source pattern y target pattern, y dos secciones opcionales que corresponden a las variables locales y al bloque de sentencias imperativas.

La sección de variables locales, cuando está definida, comienza con la palabra reservada using. Esta sección permite declarar e inicializar un conjunto de variables locales, que serán visibles en el ámbito de la regla actual.

El source pattern se define luego de la palabra reservada from. Permite especificar una variable para el tipo de elementos del modelo origen que serán alcanzados por la regla. Esto significa que la regla generará elementos del metamodelo destino por cada uno de los elementos del origen que coincida con el tipo del elemento de entrada. En algunos casos, puede que solo se quiera utilizar un subconjunto de elementos que conforman al tipo del elemento de entrada. Esto se consigue especificando una condición opcional para los elementos de entrada. De esta forma, la regla solo utilizará como entrada los elementos que coincidan con el tipo indicado y, que además, verifiquen la condición especificada.

El target pattern de una regla se introduce después de la palabra reservada to. Especifica los elementos que deben ser generados cada vez que un elemento en el modelo de entrada coincide con el source pattern de la regla, e indica cómo estos elementos que se generan deben inicializarse. Cada elemento declarado en el target pattern corresponde a un elemento en el metamodelo de salida con sus respectivos valores iniciales.

Finalmente, la sección imperativa opcional, que comienza con la palabra reservada do, permite especificar código imperativo que se ejecutará luego de la inicialización de los elementos de salida generados por la regla.

Lazy rules: son similares a las matched rules, pero solo se aplican cuando son llamadas por otra regla.

Called rules: estas reglas proveen algunas facilidad para la programación imperativa. Pueden verse como un tipo particular de helpers: deben ser llamadas explícitamente para ser ejecutadas y aceptan parámetros. Sin embargo, a diferencia de los helpers, éstas pueden generar elementos del modelo destino de la misma manera en que lo hacen las matched rules.

Al igual que las matched rules, comienzan con la palabra rule y pueden incluir una sección opcional de variables locales. Sin embargo, como no deben seleccionar elementos del metamodelo de entrada, no incluyen un source pattern. Por esta razón, la inicialización de los elementos que se generen debe ser una combinación de parámetros, variables locales y atributos del módulo. El target pattern, el cual es opcional, se define luego de la palabra reservada to.

Una called rule puede también tener una sección de sentencias imperativas.

2.2. Ingeniería Inversa

La Ingeniería Inversa es el proceso de analizar artefactos de software existentes con el objetivo de extraer información y proveer vistas de mayor nivel sobre el sistema subyacente. La ingeniería inversa generalmente implica extraer artefactos de diseño y construir o sintetizar abstracciones menos dependientes de la implementación. Esto se realiza con el objetivo de lograr una mejor comprensión del sistema y de su funcionamiento.

Se pueden distinguir tres procesos relacionados con la ingeniería inversa: re-documentación, recuperación del diseño y reestructuración**.**

- ***Re-documentación:*** es la creación de representaciones alternativas para artefactos de software dentro de un mismo nivel de abstracción, con el propósito de obtener documentación acerca del sistema en estudio. Las formas de representación resultantes son consideradas generalmente vistas alternativas del sistema.

- ***Recuperación del diseño:*** reconstruye abstracciones de diseño a partir de una integración entre el código, la documentación de diseño existente (si existe), la experiencia personal, y el conocimiento general sobre el dominio.

- ***Reestructuración:*** es la transformación de un artefacto de software de una forma de representación a otra, preservando el comportamiento externo del artefacto.

Se pueden distinguir algunos objetivos que motivan el desarrollo y aplicación de técnicas de ingeniería inversa sobre un sistema de software. Estos son: lidiar con la complejidad y volumen de los sistemas, generar vistas alternativas, recuperar información perdidad, detectar efectos colaterales y anomalías, sintetizar abstracciones de más alto nivel, y facilitar el reuso.

La ingeniería inversa constituye un proceso de análisis del software. Esta tarea de comprender un sistema existente es altamente costosa (en términos de tiempo y recursos), por lo que cualquier solución que ayude y permita automatizar, aunque sea parcialmente, este proceso resulta de gran utilidad.

La Ingeniería Dirigida por Modelos (MDE) ofrece una solución prometedora para este problema, dado que provee nociones y herramientas para representar sistemas complejos mediante modelos, ofreciendo así artefactos de software más maleables y transformables.

La Model Driven Reverse Engineering (MDRE) es la aplicación de los principios y técnicas de MDE para la ingeniería inversa. El objetivo es obtener vistas basadas en modelos, facilitando de esta forma el entendimiento y manipulación de dichos sistemas. Por lo que, se puede utilizar MDRE para descubrir modelos de los componentes de un sistema. y poder procesar estos modelos para generar nuevas vistas. La inciativa que propone el OMG para MDRE es Architecture Driven Modernization (ADM), que será detallada en profundidad en la siguiente sección.

Una arquitectura básica para describir las herramientas de ingeniería inversa se puede ver en la Figura , propuesta por Chikofsky y Cross.

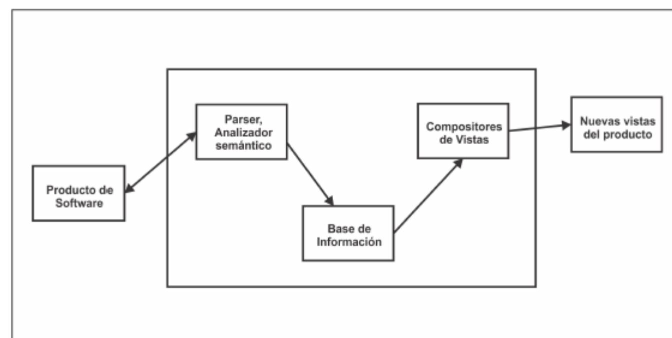

Figura 2.5: Arquitectura de las herramientas de ingeniería inversa

El producto de software que está bajo estudio es analizado y los resultados son almacenados en una base de información. Cuando se diseña una base, es conveniente hacerlo usando un esquema común compartido por diferentes herramientas. Los esfuerzos en esta dirección produjeron esquemas como el Lenguaje de Intercambio de Grafos (GXL), el Formato Estándar de Rigi (RSF), el Metamodelo FAMIX utilizado por Moose o el Metamodelo de Descubrimiento de Conocimiento (KDM), propuesto por el OMG.

Los datos de la base de información luego son utilizados por los compositores de vistas para producir vistas alternativas del producto de software. De acuerdo a las técnicas de ingeniería inversa utilizadas se pueden distinguir algunos artefactos. Si se utiliza análisis estático se pueden distinguir documentación de usuario, requerimientos, modelos de diseño y código fuente; si el análisis fuera dinámico, trazas de ejecución y registros de utilización de memoria.

El rol de la ingeniería inversa, en relación con estos artefactos, es:

- Reconstruir artefactos no disponibles o que no reflejan el sistema bajo

estudio.

- Construir otras vistas de alto nivel para proveer información necesaria para entender el sistema.
- Proveer la información necesaria para realizar tareas de cambios en el software en un escenario de reingeniería dirigida por modelos.

Muchos de estos artefactos pueden no estar disponibles y, en muchos casos, solamente se contará con el código fuente o aún peor, los binarios de un sistema como documentación del mismo. Para recuperar la información necesaria que permita construir los artefactos deseados y, por ende, entender el sistema de software bajo estudio, la ingeniería inversa ha evolucionado incorporando distintos tipos de técnicas para aplicar durante el análisis.

2.3. Ingeniería Directa

La Ingeniería Directa es el proceso de moverse desde niveles de abstracción altos y diseños lógicos, independientes de la implementación, hasta la implementación física de un sistema. La ingeniería directa sigue una secuencia de etapas que van desde el análisis de requerimientos hasta el diseño de la implementación.

La adopción del Desarrollo Dirigido por Modelos (MDD) puede simplificar drásticamente el desarrollo hacia múltiples dispositivos, reduciendo sustancialmente el costo y el tiempo. Cuando se sigue un enfoque MDD, hay varias estrategias de llegar a la generación de código dependiendo del nivel de abstracción utilizado para modelar la aplicación y del nivel de abstracción del código que se generará.

En particular, en MDA los tres niveles de abstracción son: Computation Independent Models (CIM), Platform Independent Models (PIM), y Platform Specific Models (PSM). Un conjunto de mapeos entre cada nivel y el siguiente pueden ser definidos a través de transformaciones entre modelos. Específicamente, cada CIM puede ser transformado en diferentes PIMs, los cuales pueden ser transformados a PSMs. Sin embargo, se pueden dar algunas otras combinaciones, por ejemplo, omitiendo algún nivel.

A continuación se describen en detalle los diferentes enfoques posibles para la generación de código para aplicaciones móviles representados en la Figura 2.6 extraída de [35].

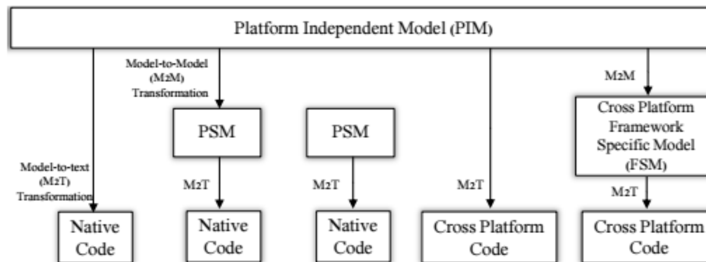

Figura 2.6: Estrategias de generación de código

- ***PIM-to-Native Code (NC):*** el enfoque PIM-a-NC establece que el código de aplicación se genera a partir del PIM. Para llegar a múltiples plataformas, se deberá proporcionar una nueva transformación para el código nativo de cada una de ellas.

- ***PIM-to-PSM-to-NC:*** este enfoque consiste, en primer lugar, en la especificación de la aplicación de manera independiente de la plataforma que será utilizada para implementar el código ejecutable a través de un PIM. El segundo paso, es la creación de diferentes PSMs, uno para cada plataforma específica, a través de transformaciones modelo-a-modelo. Estos modelos PSMs serán la entrada del generador de código para las plataformas correspondientes, a través de transformaciones modelo-a-texto.

- ***PSM-to-NC:*** consiste en definir directamente los PSMs, uno por cada plataforma de desarrollo y luego generar el código desde estos PSMs, omitiendo el nivel de abstracción de PIMs.

- ***PIM-to-Cross Platform Code (CPC):*** proporcionar una implementación nativa no siempre es conveniente, debido a los costos en el desarrollo que la misma conlleva (recursos y tiempo asociado con la actividad para cada plataforma). La utilización de un código destino que permita utilizarse en múltiples plataformas simplifica la tarea y disminuye los costos, a costa de perder el control de la parte nativa de las aplicaciones.

- ***PIM-to-Framework Specific Model (FSM)-to-CPC:*** con respeto a la opción PIM-a-CPC, este enfoque introduce el Modelo Específico del Framework. Este modelo contiene la información que el framework multiplataforma utiliza para generar las aplicaciones. Específicamente, un FSM es un PSM en el que la plataforma es un framework multiplataforma para el desarrollo de aplicaciones móviles. En la Figura 2.7 extraída de [35] puede verse gráficamente este proceso.

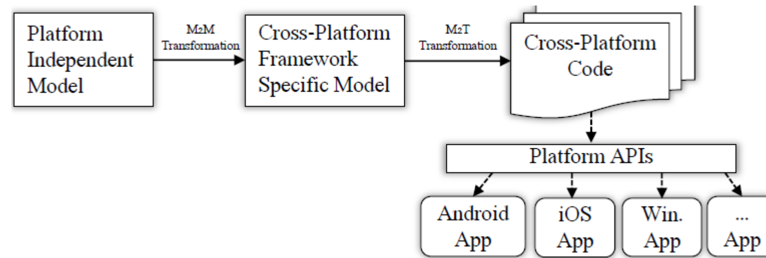

Figura 2.7: Estrategia PIM-to-Framework Specific Model (FSM)-to-CPC

2.4. Architecture Driven Modernization

La modernización de sistemas legados es una nueva área de investigación en la industria del software que está destinada a proporcionar apoyo para la transformación de un sistema de software existente a uno nuevo que satisfaga las nuevas demandas. Esta tarea requiere marcos técnicos para la integración de la información y una herramienta de interoperabilidad que permita la gestión de las nuevas plataformas, técnicas de diseño y procesos. La iniciativa del OMG (Object Management Group) para la modernización es la Architecture Driven Modernization (ADM). En 2003, el OMG conformó el Architecture-Driven Modernization Task Force (ADMTF) con el objetivo de crear especificaciones y generar consenso dentro de la industria del software en relación a la modernización de sistemas legados. Este grupo elaboró y publicó en una hoja de ruta que dejó sentado un plan para la definición de una serie de estándares de modernización.

Entre los estándares definidos por el ADMTF se encuentran Knowledge Discovery Metamodel (KDM) y Abstract Syntax Tree Metamodel (ASTM). KDM define una especificación que permite representar información semántica de un sistema, mientras que ASTM establece una especificación para representar la sintaxis del código fuente por medio de árboles de sintaxis abstracta. ASTM actúa como la base para modelar software dentro del ecosistema de estándares del OMG, mientras que KDM sirve como puerta de entrada a los modelos del OMG de más alto nivel. A su vez, ASTM complementa KDM proveyendo un framework para efectuar mapeos entre modelos de software de bajo nivel representados en ASTM, y vistas conceptuales de alto nivel representadas en KDM y otros estándares de modelado del OMG, como UML.

Hasta la fecha, ADMTF ha publicado KDM (Knowledge Discovery Metamodel), ASTM (Abstract Syntax Tree Metamodel) y SMM (Software Metrics Metamodel).

El proceso de modernización incluye tres etapas: la ingeniería inversa, la reestructuración y la ingeniería directa. La ingeniería inversa, es el proceso de análisis de los artefactos de software disponibles con el fin de extraer información y proporcionar puntos de vista de alto nivel del sistema. En el contexto de ADM, el objetivo de la ingeniería inversa es descubrir el conocimiento de la

solución existente y producir modelos con diferentes niveles de abstracción. Estos modelos son el punto de partida para el proceso de reestructuración y generación del nuevo sistema (ingeniería directa). El éxito del enfoque ADM depende de la existencia de herramientas CASE que ayuden en la automatización de cada etapa del proceso de modernización.

2.4.1. Knowledge Discovery Metamodel (KDM)

El estándar KDM define un metamodelo para el descubrimiento de conocimiento en software independiente de cualquier lenguaje de implementación y plataforma. Su objetivo es proporcionar un formato común para el intercambio de metadatos de aplicaciones facilitando la interoperabilidad entre herramientas.

El metamodelo representa los activos de software en varios niveles de abstracción como entidades, relaciones y atributos. Está especificado en MOF y utiliza como lenguaje de intercambio XMI.

La especificación está organizada en las siguientes 4 capas y, a su vez, cada capa se organiza en paquetes que definen un conjunto de elementos del metamodelo cuya finalidad es representar un cierto aspecto independiente del conocimiento en relación con los sistemas de software existentes. Las capas son:

- Capa de Infraestructura: esta capa está conformada a suvez por tres paquetes: el paquete Core que provee las construcciones básicas para crear y describir clases del metamodelo, el paquete Kdm que describe diferentes elementos de infraestructura que se encuentran presentes en cada una de las instancias KDM y, por último, el paquete Source que representa los artefactos físicos de un sistema existente como entidades KDM y especifica el mecanismo de trazabilidad que permite asociar elementos KDM con su representación "original" en el código fuente del sistema de software representado. El paquete kdm junto con los elementos definidos en el paquete Core constituyen el KDM Framework.

- Capa de Elementos del Programa: esta capa provee una representación intermedia para construcciones comunes a los lenguajes de programación. Define un único modelo KDM (CodeModel) y está compuesta por los paquetes Code y Action. El paquete Code representa elementos de código determinados por los lenguajes de programación (tipos de datos, clases, métodos, variables) y las relaciones estructurales existentes entre ellos. El paquete Action captura los elementos de bajo nivel que determinan el flujo de control y flujo de datos entre componentes del código.

- Capa de Recursos de Ejecución: representa el entorno operacional del sistema existente y está formada por cuatro paquetes: Platform (representa los recursos de la plataforma de ejecución tales como comunicación entre procesos, el uso de registros y manejo de datos), UI (representa el conocimiento relacionado con las interfaces de usuario), Event (contiene el conocimiento asociado a eventos y transiciones de

estado) y Data (representa el conocimiento relacionado con la persistencia de datos).

- Capa de Abstracciones: esta capa representa abstracciones del dominio y de la aplicación así como artefactos relacionados con el proceso de compilación del sistema de software existente. Está compuesta por tres paquetes: *Conceptual* (provee construcciones para crear un modelo conceptual durante la fase de análisis al realizar el descubrimiento de conocimiento de código existente), *Structure* (representa la organización lógica del sistema de software en subsistemas, capas y componentes) y *Build* (representa la vista de ingeniería del sistema de software, esto es, la información relacionada con su proceso de compilación)

2.4.2. Abstract Syntax Tree Metamodel (ASTM)

En el contexto de MDD, los Árboles de Sintaxis Abstracta (AST) son utilizados como un modelo del código fuente. En este sentido, ASTM establece una especificación para representar árboles de sintaxis abstracta, es decir, provee una definición de los elementos utilizados para componer modelos AST. Un modelo AST es un modelo que describe como se estructuran las sentencias de un programa de acuerdo a la gramática del lenguaje de programación particular en que el programa se encuentra implementado. Para proporcionar uniformidad y al mismo tiempo un framework universal capaz de ser extendido, la especificación ASTM se compone de:

- El Metamodelo de Árbol de Sintaxis Abstracta Genérico (GASTM): representa un conjunto genérico de elementos para modelar lenguajes común a numerosos lenguajes de programación, constituyendo, de esta forma, un core común para el modelado de lenguajes.

- Los Metamodelos de Árbol de Sintaxis Abstracta Especializado (SASTMs): representan un conjunto de especificaciones complementarias que extienden el core genérico para lenguajes tales como Ada, C, Fortran, Java, etc. Estas especificaciones se encuentran modeladas en MOF o formas compatibles con MOF y expresadas en términos del metamodelo genérico junto con extensiones de modelado que permiten capturar las particularidades del lenguaje específico.

- Los Metamodelos de Árbol de Sintaxis Abstracta Propietario (PASTMs): expresan árboles de sintaxis abstracta para lenguajes como Ada, C, COBOL, etc. modelados en formatos que no son compatibles con MOF, GASTM o SASTM.

Capítulo 3

Metamodelo del Lenguaje HAXE

El proyecto Haxe fue iniciado en el año 2005 por el desarrollador francés Nicolas Cannase y consta de un lenguaje de programación open source de alto nivel y un compilador. El lenguaje es orientado a objetos, aunque con características funcionales y fuertemente tipado, con una sintaxis similar a ECMAScript. Empleando las abstracciones correctas, es posible mantener un único código base para múltiples plataformas.

Actualmente, hay nueve lenguajes de destino compatibles que permiten diferentes casos de uso: Javascript, Neko, PHP, Python, C++, Actionscript3, Flash, Java y C#.

Name	Output type	Main usages
Javascript	Sourcecode	Browser, Desktop, Mobile, Server
Neko	Bytecode	Desktop, Server
PHP	Sourcecode	Server
Python	Sourcecode	Desktop, Server
C++	Sourcecode	Desktop, Mobile, Server
Actionscript 3	Sourcecode	Browser, Desktop, Mobile
Flash	Bytecode	Browser, Desktop, Mobile
Java	Sourcecode	Desktop, Server
C#	Sourcecode	Desktop, Mobile, Server

Figura 3.1: Lenguajes de destino de HAXE

Dado que el lenguaje Haxe puede ser compilado para diferentes plataformas,

es muy útil en una amplia variedad de dominios como lo son juegos, aplicaciones web, aplicación móviles, entre otros.

La plataforma Haxe se encuentra integrada con MDA, ya que se encuentra definido un metamodelo Ecore de la misma y un generador de código (en [12]). La integración de esta plataforma permite la definición de procesos tales como los de ingeniería inversa e ingeniería directa en forma rigurosa y sistematizada que involucra a Haxe como plataforma de desarrollo.

El metamodelo utilizado se encuentra expresado en Ecore para el lenguaje Haxe en su versión 3.1.3. Desde la óptica de MDA, las instancias del metamodelo corresponden a un modelo específico de la plataforma, o PSM.

A continuacion se explicarán brevemente las metaclases que componen el metamodelo.

3.1. Metaclases Principales

Las metaclases principales del metamodelo son aquellas que permiten la especicación de una aplicación utilizando a Haxe como lenguaje. Una de las más importantes del metamodelo es HaxeModel, la cual sirve de contenedora de los elementos utilizados para describir una aplicación y almacenar información adicional sobre la misma.

Las metaclases utilizadas directamente por HaxeModel son HaxeModule (para accedre a los distintos módulos de Haxe usados en el proyecto) y HaxePathReferentiable (para acceder a los distintos elementos del árbol de paquetes que son declarados en el proyecto y para acceder a los elementos a los que se hace referencia en el proyecto, pero que no han sido denidos de manera completa para el mismo). La metaclase HaxePathReferentiable (padre de las metaclases HaxeType y HaxePackage) es utilizada para poder aceptar el almacenamiento de tipos que estén contenidos en la raíz del árbol de paquetes.

Adicionalmente, por medio de la relación *mainClass*, la metaclase HaxeModel puede referenciar a una clase capaz de ser usada como punto de entrada al momento de la ejecución del proyecto cuando ya esté compilado.

Para modelar las características más comunes del metamodelo, se encuentran las metaclases abstractas HaxeModelElement, HaxeNamedElement y HaxeASTNode.

3.1.1. Metamodelado de módulos

La metaclase HaxeModule permite modelar el concepto de módulo o unidad de compilación. Además, la relación de composición con la metaclase HaxeField permite expresar las relaciones de uso estáticas, las cuales permiten usar métodos estáticos de clases sin referenciar directamente a la clase a la que pertenecen en el ámbito de la unidad de compilación.

3.1.2. Metamodelado de comentarios

Los distintos tipos de comentarios están modelados mediante metainstancias de la metaclase HaxeComment, y, según el tipo de comentario, asignando el valor correspondiente al atributo *lineComment* para comentarios de una o múltiples líneas, o utilizando la metaclase HaxeHaxedocComment en el caso de comentarios que correspondan a la documentación.

3.1.3. Metamodelado de paquetes

Los paquetes de Haxe permiten agrupar lógicamente distintas declaraciones de tipos y asociarlas a un determinado nombre. Los paquetes pueden ser agrupados de manera jerárquica, incluso con otros tipos. Esto está modelado mediante la metaclase HaxePackage. Mediante las relaciones *childrenPackages* y *parentReference*, es posible acceder a los paquetes hijos y padre respectivamente, y por medio de la relación *containedTypes* se contienen a los tipos declarados para ese paquete. Mediante el par de metaclases HaxePathReferentiable y HaxePathReference, se encuentran modelados, de manera abstracta, aquellos tipos que pueden ser referenciados, y las referencias en sí que pueden ocurrir en distintos ámbitos de un programa Haxe.

3.2. Metaclases de Tipos

Las metaclases representativas de los distintos tipos de Haxe, permiten expresar los diferentes tipos que pueden ser declarados o utilizados en un proyecto Haxe. En el metamodelo existen metaclases para cubrir cinco de las siete clases de tipos disponibles. En particular, para los tipos Clase (los tipos clase e interfaz), Enumeración, Función, Tipo Abstract y Estructura Anónima.

Además, se encuentran clases como HaxeTypeAccess, la cual permite especificar referencias a tipos de manera abstracta.

3.2.1. Metamodelado de clases e interfaces

A través de la metaclase HaxeClass se modelan las distintas clases e interfaces que aparecen en un proyecto.

HaxeClass tiene una relación de composición con la metaclase HaxeField, que contiene a todos los elementos que definen la estructura de la clase, como son los métodos, constructores, atributos y propiedades. De HaxeFields se derivan las relaciones HaxeConstructors, HaxeOperations y HaxeAttribute, obteniendo de ellas todos los constructores, operaciones y atributos de una determinada clase o interfaz. Las relaciones de herencia entre clases e interfaces Haxe, la generalización y la implementación, se encuentran modeladas mediante las relaciones *implementation* y *generalization*.

3.2.2. Metamodelado de tipos de datos algebraicos

El lenguaje Haxe incluye un mecanismo para declarar tipos algebraicos de datos generalizados, los cuales son declarados utilizando la palabra reservada *enum*. En el metamodelo se encuentra implementada esta característica a través de las metaclases HaxeEnum y HaxeEnumConstructor.

3.2.3. Metamodelado de tipos abstractos

Los tipos *abstract* permiten agregar funcionalidad a otro tipo Haxe que haya sido declarado anteriormente. Estos tipos se encuentran modelados mediante la clase HaxeAbstract. La misma tiene tres relaciones que son *underlyingType* (que hace referencia al tipo subyacente del tipo abstract) y las referencias *directCastingFromType* y *directCastingToType* (que modelan las referencias a los tipos usados para realizar conversiones implícitas desde o hacia el tipo abstract declarado).

3.2.4. Metamodelado de alias de tipos Typedef

Los tipos *typedef* de Haxe permiten asignar un nombre de tipo a una ocurrencia de algún otro tipo, de manera análoga a un alias. Los mismos se encuentran modelados a través de una única relación de composición, llamada *refType*, la cual hace referencia a una instancia de HaxeTypeAccess.

3.2.5. Metamodelado de metadatos

Los metadatos de Haxe, permiten agregar información a distintos elementos del lenguaje. Éstos, en términos sintácticos, consisten de un nombre, y un conjunto opcional de valores. Para ello, se encuentra la metaclase HaxeMetadata la cual tiene una relación de composición con la metaclase HaxeMetadataContainer. Esta metaclase es heredada por los elementos que pueden ser anotados con metadatos, como son las metaclases abstractas HaxeClassifier y HaxeField, dado que son los elementos que, por denición del lenguaje, pueden ser anotables.

3.2.6. Metamodelado de elementos parametrizados

Las características que permiten la parametrización de distintos elementos de Haxe se encuentran implementadas a través de la metaclase HaxeTypeParameter. Las distintas instancias de esta metaclase pueden agregarse mediante composición a las relaciones *typeParameter* declaradas en las metaclases HaxeType y HaxeAbstractFunction, que son los elementos que pueden alojar estas declaraciones.

3.3. Metaclases de Expresiones

3.3.1. Metamodelado de expresiones compuestas

Dentro de las metaclases de expresiones, existen metaclases que tienen relaciones de composición con la metaclase HaxeExpression. Por ejemplo, HaxeBlock, la cual por medio de la relación de composición *statements* se compone de un conjunto, posiblemente vacío, de expresiones Haxe; o la metaclase HaxeExpressionStatement, la cual sirve para separar distintas expresiones en forma de secuencia.

3.3.2. Metamodelado de constantes

Para las constantes de Haxe, se encuentran definidas las metaclases correspondientes a los seis tipos de constantes aceptadas por Haxe: HaxeStringLiteral, HaxeNullLiteral, HaxeBooleanLiteral, HaxeNumberLiteral, HaxeIdentifierLiteral y HaxeRegexLiteral.

3.3.3. Metamodelado de operaciones aritméticas

Para las operaciones aritméticas que pueden realizarse en Haxe se encuentran, en principio, dos metaclases abstractas: HaxeUnaryExpression y HaxeBinaryExpression con las cuales se expresan operaciones con uno y dos operandos, respectivamente. Dependiendo del orden entre operador y operando y, para las operaciones binarias hay dos metaclases: HaxeInxExpression, la cual permite expresar operaciones infijas de dos o más términos y HaxeAssignment, la cual solamente requiere de dos operandos y expresa las distintas asignaciones posibles de realizar.

3.3.4. Metamodelado de vectores o arreglos

Aunque el lenguaje Haxe no provee tipos vector o arreglo como construcciones sintácticas del lenguaje, éste sí es provisto mediante un tipo, llamado Array<T>, que aparece declarado en las librerías provistas por la plataforma. Para esto, en el metamodelo se encuentran dos metaclases: HaxeArrayInitializer, que corresponde con un inicializador de arreglos, y HaxeArrayAccess, que modela las expresiones que acceden como arreglo, mediante una expresión índice.

3.3.5. Metamodelado de expresiones de selección

Las expresiones condicionales de Haxe permiten, dada una expresión booleana, elegir entre dos flujos de programa distintos o dos expresiones distintas. Para ello, existe una metaclase, HaxeConditionalExpression, con las relaciones comunes para las expresiones de selección, realizadas con las metaclases concretas HaxeIfStatement y HaxeTernaryExpression .

3.3.6. Metamodelado de expresiones de selección múltiple

Las expresiones de selección múltiple (*switch*) permiten, a partir de una expresión, elegir entre varios caminos de ejecución según el valor obtenido de evaluar esta última. Las metaclases HaxeCase y HaxeSwitch permiten escribir este tipo de expresiones de selección. La metaclase HaxeSwitch permite crear sentencias *switch* a través relaciones de composición con las metaclases HaxeCase y HaxeExpression.

3.3.7. Metamodelado de expresiones de iteración

Para las estructuras de control, se encuentra una metaclase abstracta, llamada HaxeLoopStatement, que contiene los elementos comunes a las expresiones de control. Las metaclases hijas, en particular HaxeWhileStatement y HaxeDoWhileStatement, si bien no cuentan con más atributos y relaciones que su clase padre, permiten expresar las diferencias de orden entre el cuerpo y la condición de corte del bucle.

Para la expresión for de Haxe, que comparte una sintaxis similar con la sentencia *Enhanced For* de Java, existe la metaclase HaxeForStatement, la cual está relacionada con la metaclase HaxeSingleVariableDeclaration. Esta relación con esta metaclase modela la ocurrencia de la variable de iteración del bucle, en la cual se almacenan los distintos valores que se obtienen durante la iteración.

3.3.8. Metamodelado de expresiones de corte y continuación del flujo de ejecución

Las expresiones de corte y continuación, conocidas como *break* y *continue*, permiten escapar de un bucle de expresiones o seguir su iteración sin ejecutar las instrucciones posteriores. Esto se encuentra modelado a través de las metaclases HaxeBreak y HaxeContinue.

3.4. Metamodelado de declaraciones de variables

La metaclase que permite expresar las distintas combinaciones posibles de declaraciones de variables es HaxeVariableDeclaration. Esta metaclase es la base para la declaración de variables, y luego se encuentran dos metaclases hijas, HaxeVariableDeclarationFragment y HaxeSingleVariableDeclaration. Una instancia de HaxeSingleVariableDeclaration representa la declaración de un solo identificador con, opcionalmente, algún tipo asignado o algún valor por defecto. Para el modelado de las declaraciones múltiples de variables, hay una metaclase adicional, llamada HaxeVariableDeclarationGroup, que representa un conjunto de variables asociadas a un mismo tipo.

3.4.1. Metamodelado de atributos de instancia y de clase

Para modelar los atributos de instancia y de clase de Haxe existe la clase HaxeAttribute. Esta metaclase puede expresar simultaneamente una propiedad o una variable de clase. Las propiedades extienden el concepto de las variables de clase y brindan características que permiten un control de acceso granularmente más fino a los valores de los atributos.

3.5. Metaclases de funciones y métodos

En el metamodelado de funciones y operaciones, existe una metaclase base para modelar todos los tipos de funciones que pueden ser declaradas, que es HaxeAbstractFuction, metaclase que contiene dos relaciones de composición básicas para las funciones, que son la relación *theBody* (que modela el cuerpo de la función) y la relación *formalParameters* (que modela la lista de parámetros formales). Esta metaclase base es luego especializada por HaxeFunctionExpression, la cual realiza las funciones locales, y HaxeAbstractOperation, que es la metaclase base que modela los distintos métodos y constructores; y luego, las metaclases HaxeOperation y HaxeConstructor que corresponden a las distintas funciones y constructores en particular.

El modelado de las distintas invocaciones a métodos y funciones locales está dado por las metaclases HaxeCallExpression y HaxeAbstractMethodInvocation. Esta última metaclase es luego especializada por las metaclases HaxeMethodInvocation, HaxeSuperMethodInvocation y HaxeSuperConstructorInvocation, que son utlizadas para realizar las llamadas a métodos, llamadas a constructores de la clase padre y las invocaciones a métodos sobrecargados.

3.6. Metaclases de expresiones de acceso y construcción de objetos

La metaclase HaxeThisExpression modela las expresiones que, en el contexto de un método, acceden al objeto actual que recibe una llamada a método, siendo éste expresado por la palabra reservada *this*.

En la metaclase HaxeSuperConstructorInvocation, se modelan las llamadas a constructores pertenecientes a la clase padre, los cuales son usados en el ámbito de los constructores de clase y con la metalcase HaxeSuperMethodInvocation, se modelan las llamadas a los métodos pertenecientes a las clases padre, los cuales no podrían ser accedidos si estos fueran sobrecargados por clases más especializadas.

3.7. Metaclases de expresiones de conversión y chequeo de tipos

Las expresiones de conversión de tipos (*casting*) permiten cambiar la interpretación de un valor de una expresión a la de un tipo distinto al esperado. En el lenguaje Haxe, fueron definidas dos tipos de conversiones, una que retorna el valor de una expresión convertida convenientemente a un tipo predeterminado (*casting* seguro) y, la otra, envuelve el valor en una instancia de un tipo monomorph, el cual luego el compilador determina mediante unificación, si esta nueva expresión puede ser usada en expresiones subsiguientes (*casting* inseguro). El metamodelo implementa dos metaclases para las conversiones de tipos, una denominada HaxeUnsafeCastExpression, la cual modela las expresiones de conversión inseguras y la metaclase HaxeCastingExpression, con la cual se puede modelar una expresión de conversión segura.

Para las expresiones de chequeo de tipos existentes en Haxe, el metamodelo incluye la metaclase HaxeTypeCheckExpression.

3.8. Metaclases de expresiones de manejo de excepciones

El lenguaje Haxe provee soporte para el manejo de excepciones, utilizando expresiones *throw* y bloques *try/catch*. Las metaclases que permiten modelar esto son: HaxeThrowExpression, HaxeTryExpression y HaxeCatchClause.

Capítulo 4

Metamodelo del Lenguaje C/C++

Se presenta en esta sección un metamodelo desarrollado en Ecore para el lenguaje C/C++. Desde el punto de vista de MDA, las instancias del metamodelo corresponden a un modelo específico de la plataforma (PSM).

Para el desarrollo se utilizaron las herramientas de modelado del proyecto Eclipse y, en particular, el lenguaje de modelado Ecore. Además, a la hora del diseño e implementación se utilizó una representación textual llamada OclInEcore. Esta representación permite una mayor legibilidad, modificabilidad e integración con herramientas de versionado. Otra ventaja que presenta es que permite especificar expresiones en el lenguaje OCL en el mismo archivo en el que se escribe el metamodelo.

Para la presentación del metamodelo desarrollado, se explicarán a continuación las metaclases principales y sus relaciones y, finalmente, se mostrarán las metaclases más específicas del lenguaje.

4.1. Metaclases Principales

Las metaclases principales del metamodelo son aquellas que permiten la especificación de una aplicación utilizando a C++ como lenguaje.

Una de las metaclases más importantes es CppModel. La misma sirve de contenedora de todos los elementos utilizados para describir una aplicación en el lenguaje C++.

Las metaclases utilizadas directamente por CppModel, como se muestra en la Figura 4.1, son:

- CppClassFile: por medio de la relación *modules* se accede a los diferentes archivos de la aplicación.
- CppPathReferentiable: por medio de la relación *elements* se accede a los distintos elementos del árbol de paquetes que son declarados en el proyecto.

- CppClass: a través de la relación *mainClass* se obtiene la clase principal de la aplicación.
- CppType: por medio de la relación *orphanTypes* se accede a los tipos primitivos.

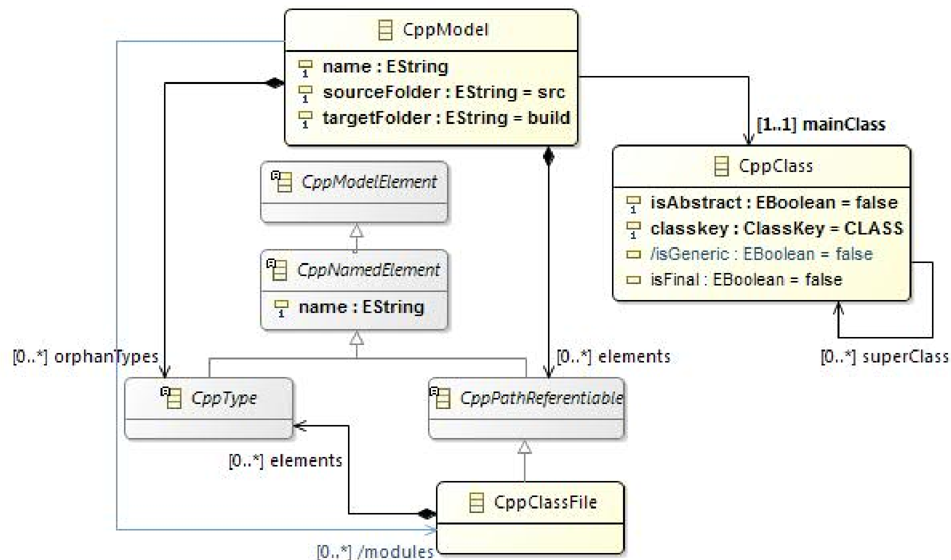

Figura 4.1: Metaclases relacionadas con CppModel

Se creó la metaclase CppModelElement como metaclase padre de todos los elementos del modelo y la metaclase CppNamedElement como padre para todos los elementos con nombre.

Mediante el par de metaclases CppPathReferentiable y CppPathReference, se modelaron de manera abstracta aquellos tipos que pueden ser referenciados, y las referencias que pueden ocurrir en distintos ámbitos de un programa (Figura 4.2). De la metaclase CppPathReferentiable heredan las siguientes metaclases:

- CppClassFile: permite modelar los diferentes archivos de clases (.cpp o .c) que componen un proyecto. Esta metaclase, a través de la relación *elements*, sirve de contenedora de los elementos que componen un archivo y, a través de la relación *imports*, contiene todos los archivos que importa.
- CppPackage: los paquetes en C++ no son como los paquetes en otros lenguajes tipo Java, o incluso Haxe. En este caso, se considera como paquete una carpeta que agrupa diferentes archivos del programa, y que puede seguir una estructura de árbol. Esta metaclase contiene tres realaciones:

- *childrenPackages* para acceder a los paquetes hijos.
- *parentReference* para acceder al paquete padre.
- *containedTypes* contiene a los tipos declarados para ese paquete.

Además, se creó la metaclase CppImportDeclaration para modelar los imports que puede tener cada CppClassFile.

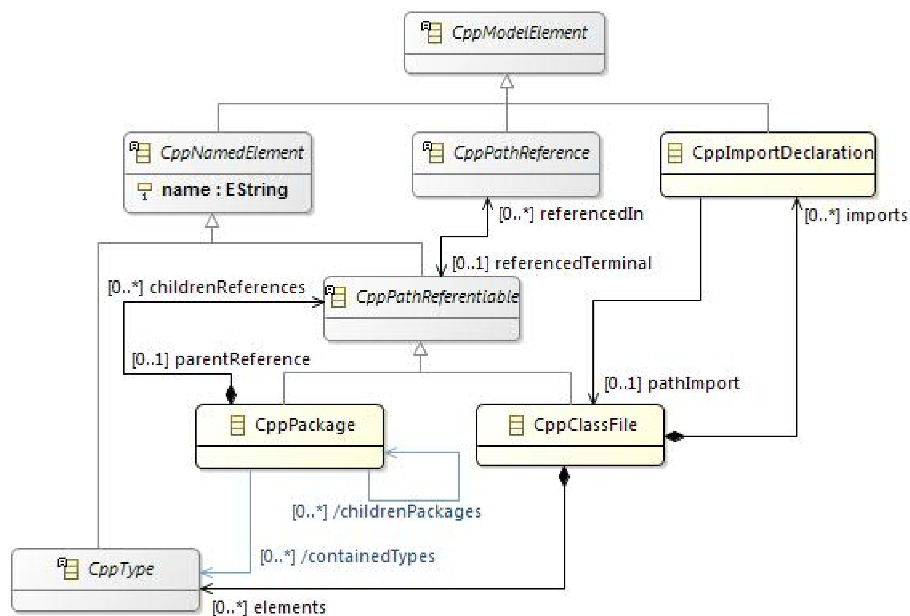

Figura 4.2: Metaclases relacionadas con CppPathReference y CppPathReferentiable

4.2. Metaclases de Expresiones

Las metaclases de expresiones, permiten construir las distintas expresiones que permite el lenguaje C++. Las mismas heredan de la metaclase abstracta «CppExpression». A continuación se mostrarán como se modelaron los diferentes tipos de expresiones.

4.2.1. Metaclases de Expresiones de Selección

Las expresiones de selección de C/C++, permiten dada una condición expresada como una expresión booleana, elegir entre flujos de ejecución distintos de acuerdo al resulado de evaluar dicha condición.

En el metamodelo, se realizaron las metaclases y relaciones mostradas en la Figura 4.3 para expresar este tipo de construcción, donde:

- La metaclase abstracta «SelectionStatement» hereda de CppExpression contiene las relaciones comunes para las expresiones de selección: la relación *condition* (que referencia a la condición que se evaluará) y *statement* (que referencia al cuerpo de ejecución).

- Las metaclases concretas IfStatement e IfElseStatement heredan de SelectionStatement. La diferencia entre las mismas es que la primera no acepta una rama *else* y la segunda si. Además, la metaclase IfElseStatement tiene un atributo booleano que indica si es de una única línea o no, esto indica si el *if* en ese caso está escrito como una expresión ternaria.

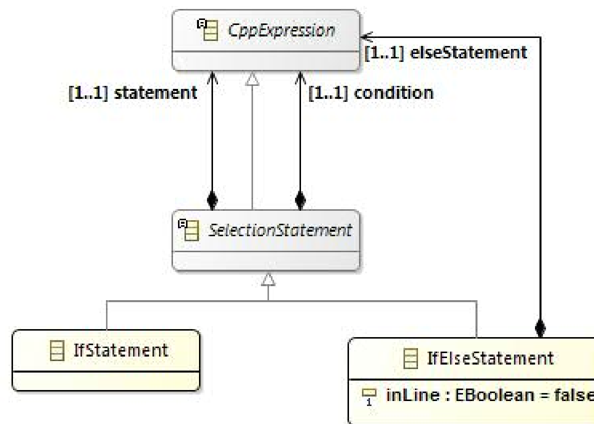

Figura 4.3: Metaclases correspondientes a las expresiones de selección

4.2.2. Metaclases de Expresiones de Iteración o Bucles

Las sentencias de iteración permiten repetir un conjunto de sentencias ejecutando un bucle. En C/C++ existen tres formas de iteraciones: los bucles *While, Do While* y *For*. Estas expresiones se modelaron a partir de una metaclase llamada IterationStatement que hereda de CppExpression, como se muestra en la Figura 4.4. Esta metaclase contiene los elementos comunes a las expresiones *While*, *Do While* y *For*, como lo son la condición (a través de la relación *condition*) y el cuerpo de ejecución (a través de la relación *theBody*). De IterationStatement heredan las metaclases: DoWhileStatement, WhileStatement y ForStatement. Las primeras dos no agregan nada con respecto a su clase padre. En cambio, ForStatement agrega dos relaciones nuevas: una expresión que inicializa la variable con la que se itera (a través de la relación

initializer) y una expresión que actualiza el valor del iterador (a través de la relación *updater*).

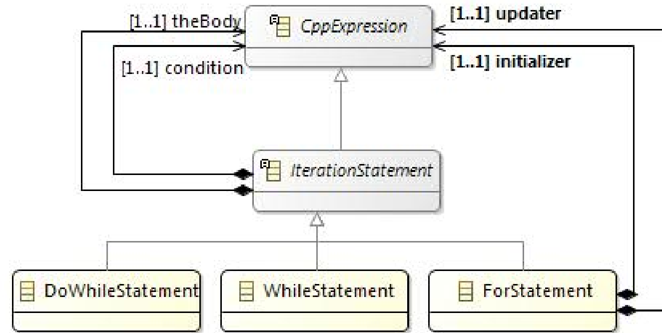

Figura 4.4: Metaclases de expresiones iterativas

4.2.3. Metaclases de Expresiones de Selección Múltiple

Las expresiones de selección múltiple (expresiones *switch*) permiten a partir de una expresión, elegir entre varios caminos de ejecución según el valor obtenido de evaluar esta última. Como se muestra en la figura 4.5, las metaclases SwitchExpression y CppCase, que heredan de la metaclase HaxeExpression, permiten escribir este tipo de expresiones de selección. La metaclase SwitchExpression permite crear sentencias *switch* y tiene tres relaciones de composición con las metaclases HaxeCase y HaxeExpression:

- A través de la relación *expression* se modela la expresión que se evalúa para elegir la rama que se ejecutará.
- A través de la relación *default* se permite agregar opcionalmente una rama de ejecución por defecto, que se ejecutará si el valor de la expresión no coincide con ninguno de los valores considerados en cada rama.
- A través de la relación *cases* con la metaclase CppCase, contiene las diferentes ramas de ejecución posibles. En la metaclase CppCase, por medio de la relación *values*, se configura la lista de valores que puede tomar la expresión de la sentencia *switch* que la contiene y, por medio de la relación *expression*, se realiza la composición de las expresiones que se ejecutarán en esa rama.

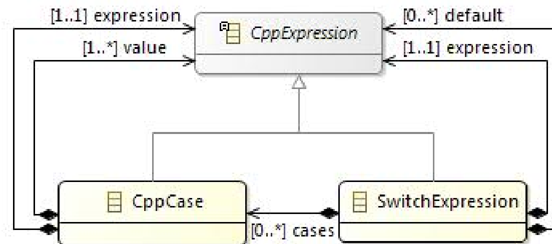

Figura 4.5: Metaclases de expresiones de selección múltiple

4.2.4. Metaclases de Expresiones Unarias y Binarias

Para el metamodelado de las operaciones se optó por realizar dos metaclases abstractas, UnaryExpression y BinaryExpression, las cuales heredan de CppExpression (como se muestra en la Figura 4.6). Con las mismas se permite expresar operaciones con uno y dos operandos, respectivamente. Los operandos se obtienen a través de las relaciones *rightOperand* y *leftOperand* en las binarias, y por la relación *expression* en las unarias.

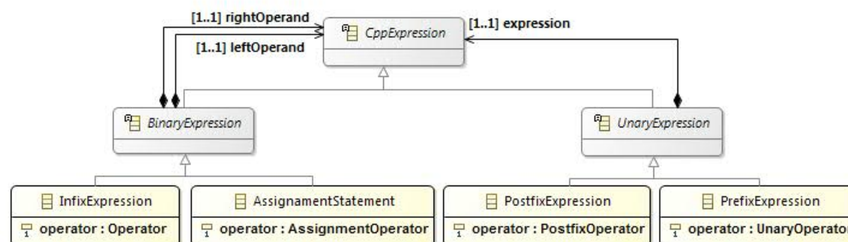

Figura 4.6: Metaclases de expresiones unarias y binarias

- Operaciones Unarias: de la metaclase UnaryExpression heredan las metaclases PostfixExpression y PrefixExpression, de acuerdo al orden en el que se encuentre el operador y el operando. En las expresiones postfijas (PostfixExpression) se encuentra primero el operando y luego el operador, y viceversa para las expresiones prefijas. Cada metaclase contiene una relación con una enumeración que describe los posibles operadores que se pueden utilizar en cada operación, como se muestra en la figura 4.7.
- Operaciones Binarias: de la metaclase BinaryExpression heredan las metaclases InfixExpression y AssignamentStatement, para diferenciar si se trata de una expresión infija o de una asignación. Cada metaclase contiene una relación con una enumeración que describe los posibles operadores que se pueden utilizar en cada operación, como se muestra en la Figura 4.7.

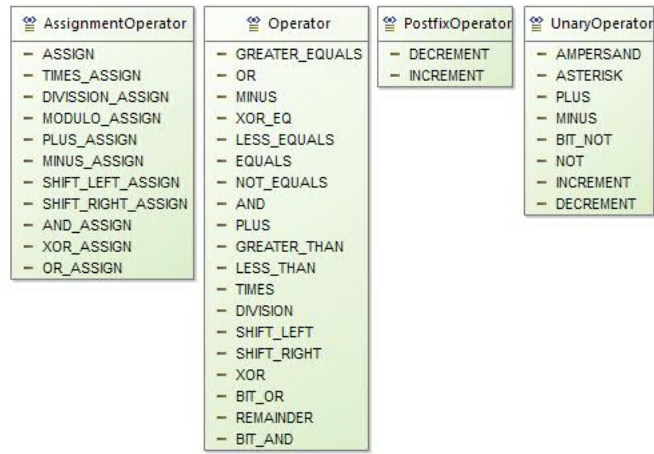

Figura 4.7: Operadores usados en expresiones unarias y binarias

4.2.5. Metaclases de Expresiones de manejo de Excepciones

El manejo de excepciones en C++ se realiza a través de expresiones *throw* y bloques *try*/*catch* (Figura 4.8). Para ello, se crearon las siguientes metaclases que heredan de CppExpression:

- ThrowExpression: esta metaclase contiene una relación *expression* en la que contiene la expresión que utiliza como argumento.
- TryExpression: esta metaclase contiene una relación *theBody* que contiene la expresión que se ejecutará dentro del *try* y una relación *catchClause* en la que contiene las posibles expresiones de *catch*.
- CatchClause: esta metaclase, al igual que TryExpression, contiene una relación *theBody* con la expresión que se ejecutará dentro del *catch*.

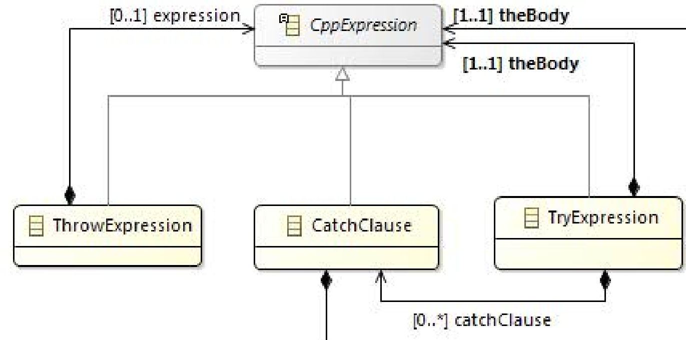

Figura 4.8: Metaclases de expresiones de manejo de excepciones

4.2.6. Metaclases de Expresiones Constantes

Para las expresiones constantes, se crearon las siguientes metaclases que heredan de CppExpression: BooleanLiteral (para valores booleanos), CharacterLiteral (para caracteres), NumberLiteral (para valores numéricos), NullLiteral (para valores nulos), StringLiteral (para cadenas de caracteres) y RegexLiteral (para expresiones regulares). Cada instancia de estas metaclases almacenará el valor del literal, según corresponda en cada caso.

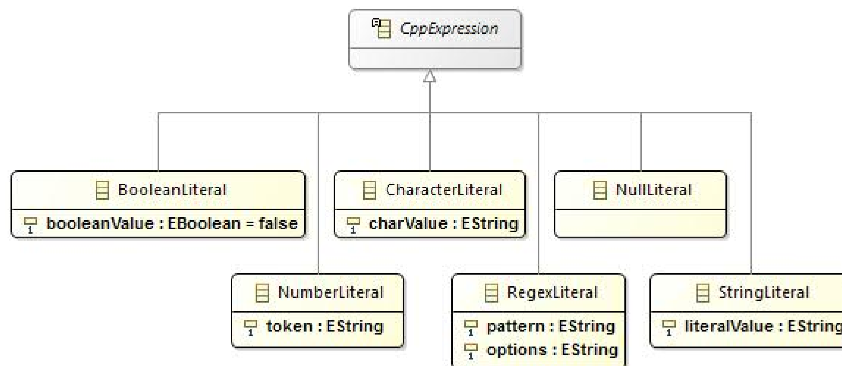

Figura 4.9: Metaclases de expresiones constantes

4.2.7. Metaclases de Expresiones con Arreglos

En cuanto a las expresiones que sirven para la manipulación de arreglos, se crearon dos metaclases que heredan de CppExpression:

- ArrayInitializer: que corresponde con un inicializador de arreglos. Esta metaclase, por medio de la relación *expressions*, contiene a las distintas expresiones que permiten la inicialización.
- ArrayAccess: permite modelar las expresiones que acceden a un arreglo. Las relaciones *index* y *array* contienen el índice y el arreglo al que se accede, respectivamente.

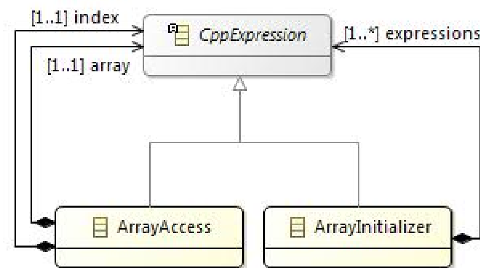

Figura 4.10: Metaclases de expresiones con arreglos

4.2.8. Metaclases de Expresiones de Salto

Para modelar las expresiones de salto *GoTo*, *Break*, *Continue* y *Return* se creó una metaclase abstracta JumpStatement de la cual heredan las metaclases: BreakStatement, ReturnStatement (la cual tiene una relación *returnExpression* con la expresión de retorno), GotoStatement (la cual tiene una relación *label* con la expresión que tiene la etiqueta a la que se salta) y ContinueStatement.

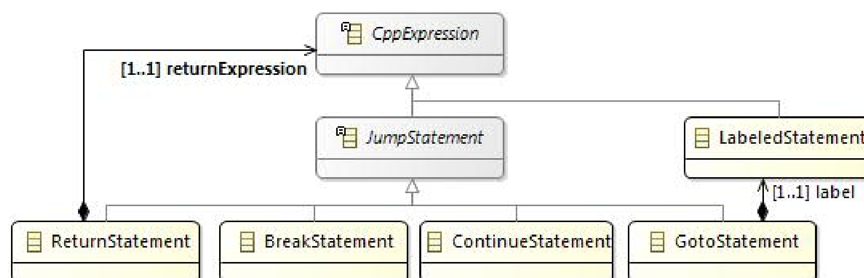

Figura 4.11: Metaclases de expresiones de corte y continuación

4.2.9. Metaclases de Expresiones Compuestas

Para las expresiones compuestas se creó la metaclase CppBlock, como muestra la Figura 4.12. En esta metaclase, a través de la relación de composición

satements, sirve de contenedora de cero o más expresiones.

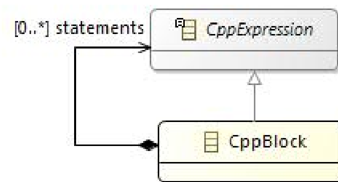

Figura 4.12: Metaclases de expresiones compuestas

4.2.10. Metaclases de Expresiones de Conversión de Tipos

Las expresiones de conversión explícita de tipos (o expresiones de *casting*) permiten cambiar la interpretación de un valor de una expresión a la de un tipo distinto al esperado. La sintaxis puede ser: (<nombre de tipo>)<expresión> o <nombre de tipo>(<expresión>). Para ello, como se ve en la figura 4.13, se creó la metaclase CastExpression, que hereda de CppExpression. Esta metaclase contiene una referencia a una expresión (que es la que se convertirá) y una referencia al tipo al que se quiere convertir.

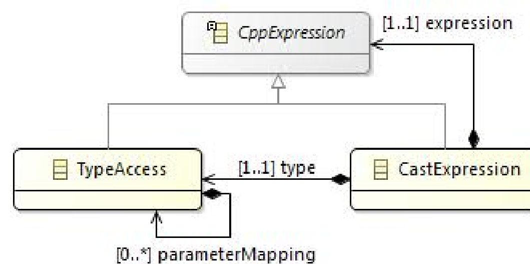

Figura 4.13: Metaclases de expresiones de casting

4.3. Metaclases de Variables: Declaración y Acceso

Para poder declarar variables y atributos en el contexto del metamodelo, se crearon las metaclases mostradas en la Figura 4.14:

- DeclarationExpression: esta metaclase que hereda de CppExpression permite modelar una expresión de declaración de variables.

- VariableDeclaration: esta metaclase abstracta se creó con el objetivo de expresar las diferentes combinaciones posibles de declaraciones de variable. Contiene como atributos el tipo de variable (del tipo enum VarType) y un booleano que indica si se trata de un arreglo o no. Además, cuenta con dos referencias a CppExpression: una para modelar si la variable declarada tiene un inicializador y otra para expresar las dimensiones en caso de tratarse de un arreglo. De esta metaclase heredan VariableDeclarationFragment y SingleVariableDeclaration.

- VariableDeclarationGroup: esta metaclase representa la declaración de múltiples variables asociadas a un mismo tipo. Por lo tanto, se creó una relación de composición con VariableDeclarationFragment, a través de la cual se modelan los diferentes fragmentos que componen el grupo de variables.

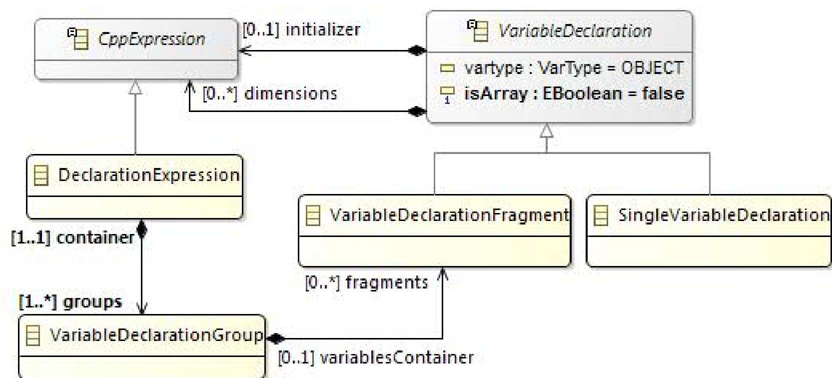

Figura 4.14: Metaclases de declaración de variables

La metaclase que representa cada variable es CppVariable, la cual hereda de CppField, VariableDeclaration y CppTypedElement. La misma agrega dos atributos: el tipo de almacenamiento (*StorageType*) y un booleano que indica si se trata de una constante o no. Para modelar el acceso a las variables se creó la metaclase VariableAccess.

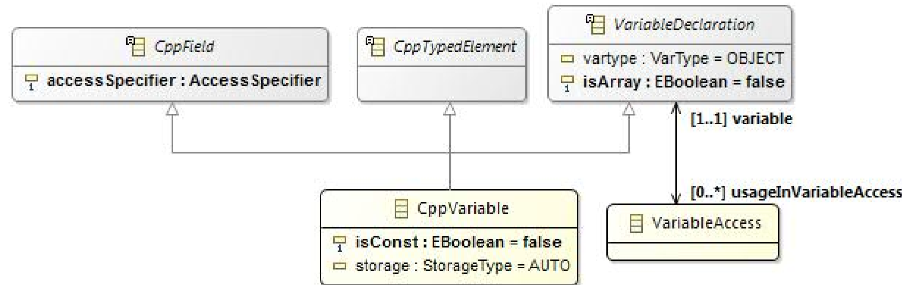

Figura 4.15: Metaclases de acceso a variables

4.4. Metaclases de Elementos Parametrizados

Las plantillas (*Templates*), también denominadas tipos parametrizados, son un mecanismo de C++ que permite que un tipo pueda ser utilizado como parámetro en la definición de una clase o una función.Ya se trate de clases o funciones, la posibilidad de utilizar un tipo como parámetro en la definición, posibilita la existencia de entes de nivel de abstracción superior al de la función o clase concreta.

template <T> void fun(T& ref); // declaración de función genérica

template <T> class C {/*...*/}; // declaración de clase genérica

Para modelar esto, se creó la metaclase CppTypeParameter, la cual está vinculada con relaciones de composición con las metaclases CppType y CppFunction.

4.5. Metaclases de Métodos

Para modelar los métodos se crearon las metaclases que se muestran en la Figura 4.16. La metaclase principal es CppFunction. La misma contiene una relación *ownedParameters* con SingleVariableDeclaration para modelar los parámetros que puede recibir una función. De ella y de CppField hereda la metaclase CppMemberFunction, la cual es la clase padre de los métodos, los destructores y los constructores.

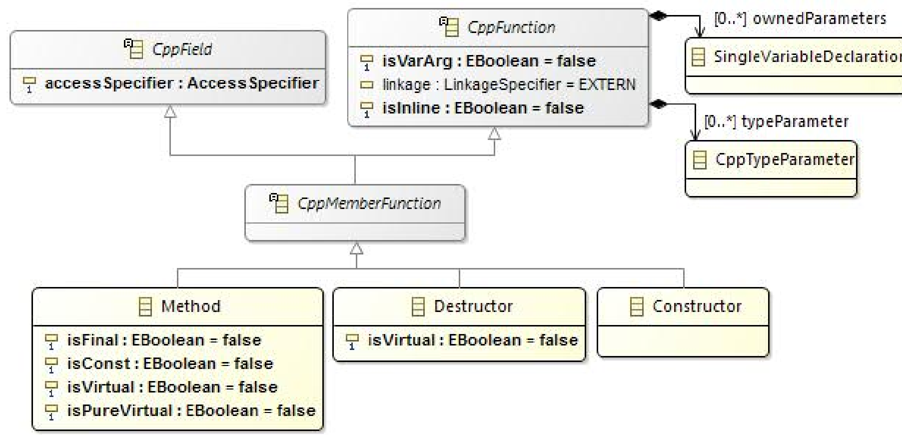

Figura 4.16: Metaclases de métodos

Para modelar la invocación a los métodos se crearon las metaclases que se muestran en la Figura 4.17. La metaclases abstracta AbstractMethodInvocation contiene, mediante la relación de composición *arguments*, los argumentos con los que se invoca al método (que está dado por la relación *method*). Las metaclases concretas para el modelado de estas invocaciones son:

- MethodInvocation: para modelar el acceso a cualquier tipo de método.
- SuperMethodInvocation: para modelar el acceso a un método de una clase padre.
- SuperConstructorInvocation: para modelar el acceso al constructor de la clase padre.

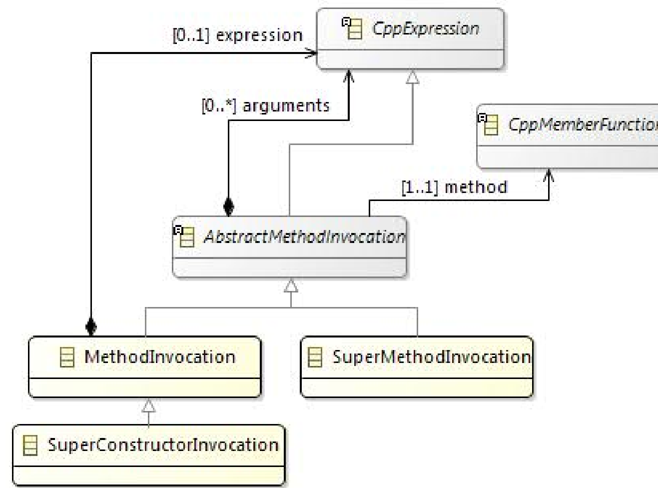

Figura 4.17: Metaclases de invocación a métodos

4.6. Metaclases de Tipos

Para modelar los diferentes tipos de datos de C/C++ se creó la metaclase abstracta CppType como padre de todos ellos. De la misma heredan PrimitiveType (para modelar tipos de datos primitivos), CppClassifier (para modelar clases, estructuras y uniones) y CppEnum (para modelar enumeraciones). Para modelar el acceso a todos los tipos se creó la metaclase TypeAccess.

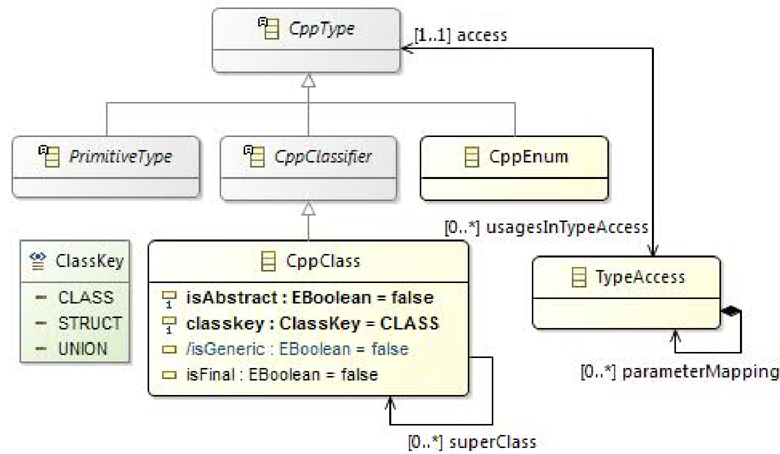

Figura 4.18: Metaclases de tipos

4.6.1. Metaclases de Clases, Estructuras y Uniones

Para el modelado de Clases, Estructuras y Uniones se creó la metaclase CppClass, que hereda de CppClassifier (como muestra la Figura 4.19). El atributo que define de cuál de estos tres tipos (*class, struct* o *unión*) se trata es *classkey*. En caso de tratarse de clases (solo C++), cada CppClassifier tiene relaciones de composición con CppMemberFunction para definir las operaciones (a través de la relación *cppOperations*), el constructor (a través de la relación *cppConstructor*) y el destructor (a través de la relación *cppDestructor*). Además, tiene una relación de composición (*cppAttributes*) con CppVariable para definir los atributos que serán las variables de clase o componentes de la estructura o unión.

La metaclase CppFieldContainer contiene a través de la relación *cppFields* todos los elementos que defienen la estructura de una clase: las operaciones (métodos, constructor y destructor) y los atributos. Además, cada CppClass contiene la relación *superClass* para modelar la herencia múltiple del lenguaje C++.

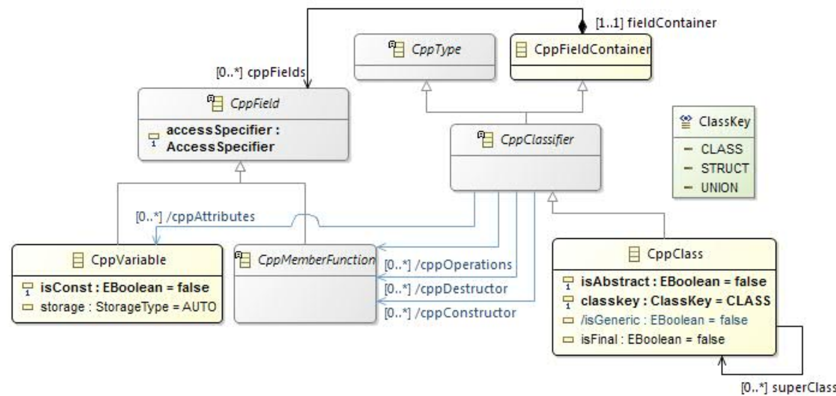

Figura 4.19: Metaclases de clases, estructuras y uniones

4.6.2. Metaclases de Tipos Primitivos

Para modelar los tipos primitivos se creó la metaclase abstracta «PrimitiveType», como muestra la Figura 4.20. De la misma heredan las clases concretas: BooleanType, CharType, DoubleType, FloatType, UnsignedType, SignedType, IntType, LongType, ShortType y VoidType.

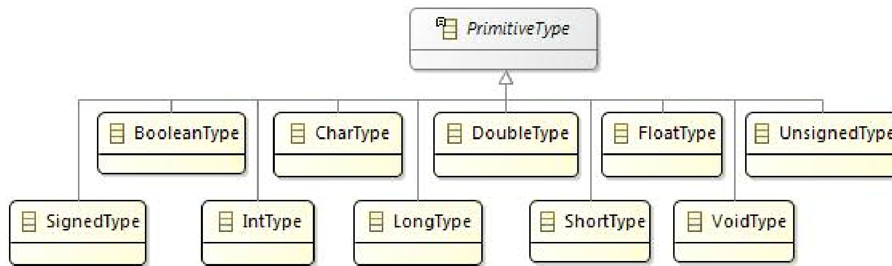

Figura 4.20: Metaclases de tipos primitivos

4.6.3. Metaclases de tipos Enum

Una enumeración es un conjunto de constantes enteras. A la enumeración se le puede asignar un nombre, que se comportará como un nuevo tipo de dato que solo podrá contener los valores especificados en la enumeración. Como muestra la figura 4.21, para modelar este tipo de datos se crearon las metaclase CppEnum (que hereda de CppType) y CppEnumConstructor (que hereda de CppNamedElement). Ambas metaclases se encuentran vinculadas por una relación de composición, ya que CppEnum contiene, mediante la relación *literals*, CppEnumConstructors.

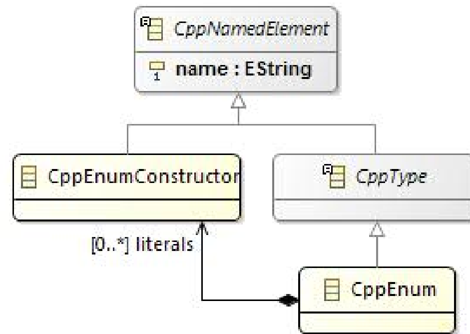

Figura 4.21: Metaclases de tipos Enum

4.7. Metaclases de Comentarios

Para el metamodelado de los comentarios, se creó la metaclase CppComment (como muestra la Figura 4.22). Los comentarios pueden abarcar una línea o varias, debiendo éstos ser delimitados de manera apropiada y no anidados. Por lo tanto, la metaclase contiene como atributos dos valores booleanos que indican si el comentario es de una línea o multilínea. Además, contiene el contenido del comentario.

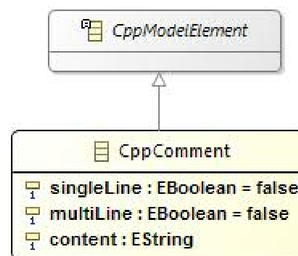

Figura 4.22: Metaclases de comentarios

Capítulo 5

Proceso de migración

El proceso de migración de C/C++ a plataformas móviles a través del lenguaje HAXE puede verse gráficamente en la Figura 5.1.

El primer paso de la migración consiste en una transformación código-a-modelo, en la que el código fuente del programa escrito en C/C++ es traducido a un modelo que conforme al metamodelo definido para tal lenguaje. Para llevar a cabo este proceso en primer lugar se debe diseñar un metamodelo que describa los elementos del lenguaje de origen, por lo que fue desarrollado un metamodelo que describa al lenguaje C/C++ (Capítulo 4). Este metamodelo es el principal factor que asegura la calidad y la completitud de la fase de descubrimiento encargada de generar instancias del metamodelo a partir del sistema. En el presente trabajo, se utilizó ANTLR y se desarrolló un programa Java para llevar a cabo la tarea de descubrimiento del modelo a partir del programa. .

El siguiente paso consiste en transformar el modelo que conforma al lenguaje de origen, a un modelo que conforme el metamodelo del lenguaje destino. Esta etapa se llevó a cabo mediante una transformación a nivel de metamodelos, implementada en ATL, que toma como entrada el modelo C/C++ para el programa y genera como salida el correspondiente modelo HAXE. Dicha transformación se especifica mediante un conjunto de reglas de transformación que definen mapeos semánticos entre los elementos del metamodelo de entrada (metamodelo C/C++) y los elementos del metamodelo de salida (metamodelo HAXE).

Finalmente, el último paso es la ingeniería directa, una transformación modelo-a-texto desde el modelo que conforma al metamodelo HAXE hacia código fuente. Para ello, se utilizó el generador de código Acceleo.

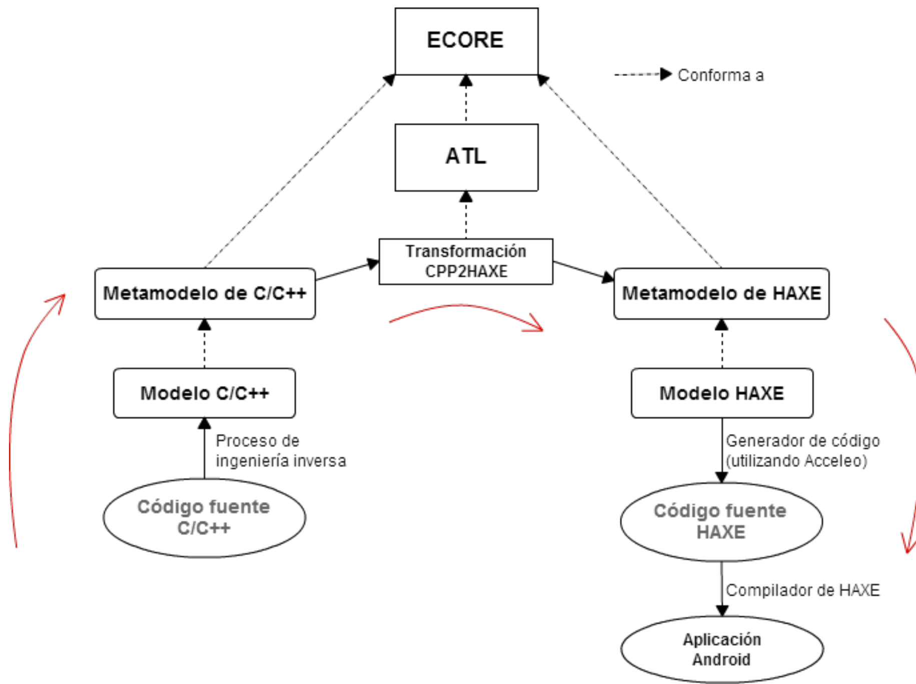

Figura 5.1: Proceso de migración

5.1. Ingeniería Inversa

5.1.1. Descripción del proceso de Ingeniería Inversa

El primer paso de la migración consiste en una transformación código-a-modelo, en la que el código fuente del programa escrito en C/C++ es traducido a un modelo que conforme al metamodelo definido para tal lenguaje. Esta representación es un paso muy importante, dado que permite obtener la implementación del programa en términos de un modelo compatible con MOF.

Hoy en día, la herramienta más completa alineada con ADM es MoDisco, un framework open source para MDRE (Model Driven Reverse Engineering) que forma parte de la plataforma Eclipse. MoDisco provee un framework genérico y extensible cuyo objetivo es facilitar el desarrollo de herramientas para extraer modelos a partir de sistemas legados. Como un componente Eclipse, MoDisco puede integrarse con otros plugins y tecnologías disponibles en el entorno Eclipse tales como el Eclipse Modeling Framework (EMF), Graphical Modeling Framework (GMF) y transformaciones modelo-a-modelo.

Las limitaciones que tiene MoDisco es que solo provee herramientas para el descubrimiento del modelo de proyectos escritos en el lenguaje Java. Por lo tanto, fue necesario desarrollar una herramienta propia para realizar esta tarea.

5.1.2. ANTLR (ANother Tool for Language Recognition)

ANTLR es una poderosa herramienta de generación de analizadores léxicos y sintácticos para lectura, procesamiento, ejecución y/o traducción de texto estructurado e incluso archivos binarios.

Esta herramienta es ampliamente utilizada para el desarrollo de lenguajes, frameworks e incluso de otras herramientas.

Partiendo de gramáticas de distintos tipos, ANTLR genera lexers y parsers capaces de construir y recorrer árboles de parsing. Permite además insertar acciones semánticas en cada punto de las reglas gramaticales, e incluso posee un sistema de predicados, de gran utilidad para desambiguar casos complejos, o bien simplificar considerablemente gramáticas complicadas con el agregado de simples condiciones.

En anteriores versiones de ANTLR, sólo se contemplaba la generación de analizadores léxicos y sintácticos. En las versiones actuales se puede, además, escribir gramáticas especialmente diseñadas para recorrer y procesar árboles sintácticos *on the fly*, pudiendo así separar el parsing, del recorrido y proceso de las estructuras obtenidas. También provee en las versiones actuales un template engine para la traducción entre lenguajes, es decir, para la generación de textos estructurados escritos en un segundo lenguaje, tomando como entrada el análisis sintáctico resultante del texto escrito en el lenguaje original modelado por la gramática.

5.1.3. Programa Java desarrollado

Una vez generado el árbol de parsing con la herramienta ANTLR, se desarrolló un programa escrito en el lenguaje Java para el recorrido del mismo y creación del modelo correspondiente a cada programa C/C++ de entrada. La salida del programa es un archivo XML que representa el modelo. El tipo de modelo obtenido luego de esta conversión se denomina modelo inicial dado que posee una correspondencia directa con los elementos del sistema original con la menor pérdida de información.

En el diagrama que se encuentra en la Figura 5.2 se pueden observar las relaciones existentes en las clases principales del proyecto. Desde la aplicación principal, al seleccionar un proyecto escrito en C++, se ejecuta el "Compiler". Esta clase es la encargada de centralizar y llamar a los métodos necesarios para realizar la lectura de código y transformarlo en modelos. Se ayuda de la clase "ExtractListenerCPP" y "TypesController". La primera, es la encargada de recorrer el árbol de sintaxis abstracta generado por ANTLR al pasarle un archivo escrito en C/C++. Este extractor, utiliza la clase "Expressions" que se encarga principalmente de reconocer las diferentes expresiones del lenguaje. La clase "TypesController" permite llevar el conocimiento de los elementos principales encontrados en el código (clases, métodos y variables). De esta manera, es posible recuperar una instancia de cualquier elemento si es necesario (por ejemplo, llamado a una función y crear la relación). También,

se pueden encontrar la clase "XmiReference", que es la implementación de la interfaz "XmiReferenceInterface". Esta implementación permite tener herencia "múltiple" dentro del lenguaje Java. Es utilizada para remarcar que la clase puede ser referenciada por otra y ofrece los métodos necesarios para realizar esta tarea. La interfaz "Class2XMI" provee un método que debe ser implementado. Este método permite generar el código XML para su utilización/visualización. Es implementado por las clases principales dentro del paquete mmclass.

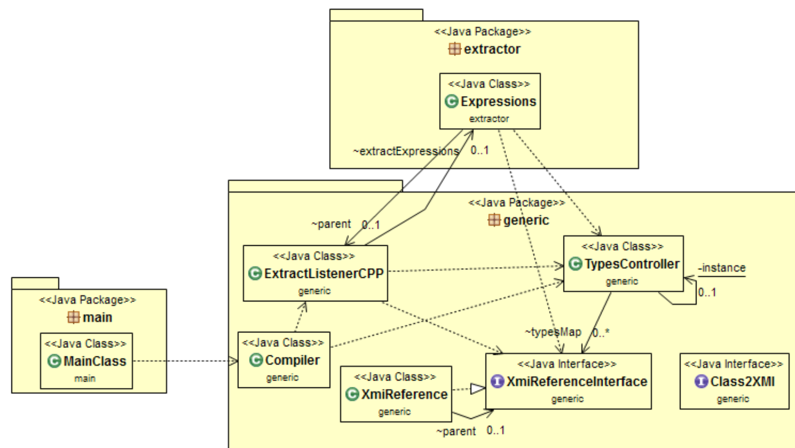

Figura 5.2: Clases principales de la aplicación

En el diagrama que se encuentra en la Figura 5.3 se pueden observar las relaciones entre los subpaquetes del paquete mmclass. Este paquete contiene las clases .java generadas a partir del Metamodelo. De esta manera, cada clase se puede implementar siendo fiel al metamodelo y generar de manera sencilla el XML resultante ya que cada una sabe como representarse.

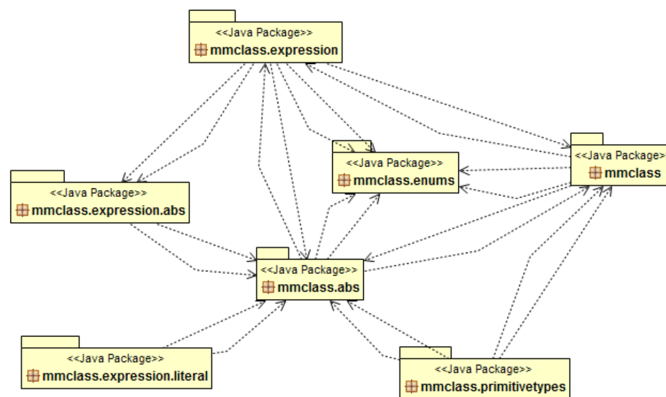

Figura 5.3: Diagrama de relaciones dentro del paquete mmclass

En el diagrama que se encuentra en la Figura 5.4 se muestran las relaciones del paquete mmclass con el resto de los paquetes del proyecto. El paquete "antlr4.ccparser" es generado a partir de la gramática de C++ con la herramienta ANTLR 4.5. Estas clases son utilizadas para generar el árbol de sintaxis abstracta de un archivo escrito en C++ y ofrece una clase base para recorrerlo. A partir de esta clase se generan los extractores. En el paquete "main" se encuentra el programa principal. Dicho programa permite seleccionar el proyecto que se desea procesar. El paquete "generic" contiene las clases principales utilizadas para el procesamiento, las principales son "Compiler" y "ExtractListenerCPP". El paquete extractor posee la clase "Expressions" que se encarga del procesamiento de expresiones del lenguaje.

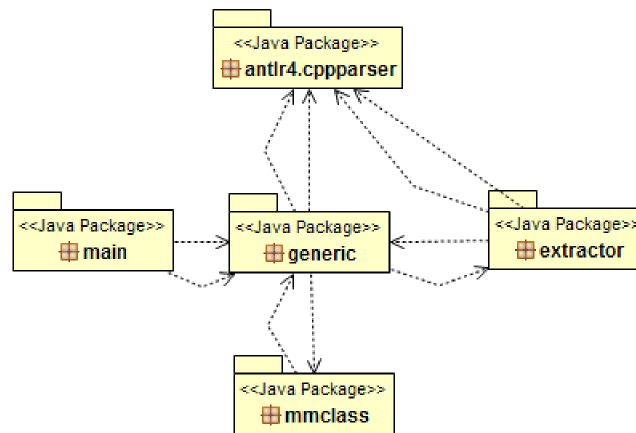

Figura 5.4: Diagrama de paquetes del proyecto

5.2. Transformación

En el nivel de metamodelos las transformaciones están dadas por mapeos entre metamodelos. Para cada transformación se deben definir los metamodelos origen y destino. El metamodelo origen describe la familia de modelos para el cual es posible aplicar la transformación. El metamodelo destino indica las caracterísiticas de los modelos generados.

La transformación cpp2haxe, de C++ a Haxe, permite la conversión desde modelos que conformen con el metamodelo de C/C++ a su modelo equivalente que conforme al metamodelo de Haxe. El objetivo es la traducción de los distintos elementos sintácticos de un programa codificado en el lenguaje C/C++ para poder ser expresados como construcciones válidas en el lenguaje Haxe. La implementación de las transformaciones fue utilizando el lenguaje ATL. Se utilizó el lenguaje ATL debido a su madurez, integrabilidad y estabilidad.

A continuación, se explican las características de las transformaciones para las diferentes construcciones.

5.2.1. Elementos Principales

La metaclase principal del metamodelo de C++ es «CppModel» y, la del metamodelo de Haxe es «HaxeModel». Por lo tanto, la transformación de una instancia de CppModel a una de HaxeModel se realiza mediante la regla «model», en la que:

- La referencia «elements» se resuelve utilizando un helper llamado «get_allpackages» que obtiene todos los CppPackages que contiene un CppModel.
- La referencia «referenced» se resuelve filtrando los elementos de la relación «orphanTypes», pero eliminando los elementos sin ocurrencias en el modelo, y quedándose únicamente con los tipos primitivos.
- La referencia «haxeModules» se resuelve utilizando un helper llamado «get_classfiles» que obtiene todos los CppClassFiles de todos los paquetes.
- La referencia «mainClass» se resuelve de manera directa con la mainClass del CppModel.

Transformaciones entre metaclases simples se realizaron de manera directa, tales como:

- CppNamedElement a HaxeNamedElement, mediante la regla «namedElement».
- CppPackage a HaxePackage, mediante la regla «packages».
- CppClassFile a HaxeModule, mediante la regla «modules». En la misma se utilizó, para resolver la referencia «dependencies» un helper llamado «get_imports».

5.2.2. Tipos Primitivos

La conversión entre tipos primitivos, se muestra en la Figura 5.5.

Tipo Primitivo de C++	Tipo Primitivo de Haxe
int	Int
boolean	Bool
void	Void
char	String
short	Int
long	Int
float	Float
double	Float
signed	Int
unsigned	Uint

Figura 5.5: Transformación entre tipos primitivos de C++ a Haxe

5.2.3. Clases y Constructores

La transformación de las clases públicas se hizo de manera directa a través de la regla «classes», y la transformación de los constructores de clase se realizaron por la regla «constructors». Sin embargo, se tuvieron que tener en cuenta los siguientes puntos:

- El lenguaje Haxe no admite que las clases sean abstractas. En cambio, se puede simular esta situación convirtiendo al constructor de la clase en un método abstracto. Por lo tanto, cada clase que en C++ fuera declarada como abstracta, fue transformada como una clase en Haxe cuyo constructor es abstracto.
- A diferencia del lenguaje C++, Haxe no permite la declaración de los destructores de las clases. Por lo tanto, éstos fueron descartados al momento de la transformación.
- El lenguaje C++ admite herencia mútliple de clases y Haxe solo permite herencia simple. En la transformación se optó por generar solo la herencia del primer elemento (a través de la lazy rule «get_generalization»), descartando las siguientes. Esto claramente puede tener inconvenientes al momento de generar el código que deberán ser solucionados manualmente.

5.2.4. Enumeraciones

Para la transformación de las enumeraciones, se utilizaron las siguientes reglas:

- «enumTransformation» que se encarga de transformar un CppEnum en un HaxeEnum.
- «enumConstructor» que se encarga de transformar un CppEnumConstructor en un HaxeEnumConstructor.

5.2.5. Expresiones

- **Expresiones binarias:** BinaryExpression a HaxeBinaryExpression se transformó mediante la regla abstracta «binaryExpression», de la cual heredan las reglas:
  - «infixExpression», que transforma una InfixExpression a una HaxeInfixExpression
  - «assignmentExpression», que transforma un AssignamentStatement a un HaxeAssignment.
- **Expresiones unarias:** UnaryExpression a HaxeUnaryExpression se transformó a través de la regla abstracta «unaryExpression», de la cual heredan las reglas:

- «prefixExpression», que transforma una PrefixExpression a una HaxePrefixExpression.
- «postfixExpression», que transforma una PostfixExpression a una HaxePostfixExpression.

- **Expresiones condicionales:** SelectionStatement a HaxeConditionalExpression, se transformó mediante la regla abstracta «conditionalExpression», de la cual heredan las reglas:
    - «ifStatement», que transforma un IfStatement a un HaxeIfStatement.
    - «ifElseStatement», que transforma un IfElseStatement a un HaxeIfStatement (agregándole la sección «else»).
    - «ifElseStatementInline», que transforma un IfElseStatement como operación ternaria a una HaxeTernaryExpression.
- **Expresiones iterativas:** IterationStatement a HaxeLoopStatement, se transformó mediante la regla abstracta «loopStatement», de la cual heredan las reglas:
    - «doWhileStatement», que transforma un DoWhileStatement a un HaxeDoWhileStatement.
    - «whileStatement», que transforma un WhileStatement a un HaxeWhileStatement.
- **Expresión For:** Haxe no soporta las sentencias «for» tradicionales, sino que solo se puede aplicar sobre tipos que pueden ser iterados o sobre intervalos. Por ejemplo: for (v in e1) e2 o for (i in 0...10) trace(i). Por lo tanto, para la transformación de un for tradicional como es el caso de los que tiene el lenguaje C++, se recurrió a convertirlo en una sentencia while a través de la regla «forStatement».
- **Expresión Switch:** SwitchExpression a HaxeSwitch, se transformó mediante la regla «switch». Además, la estructura de los posibles casos dentro de un switch se transformó mediante la regla «case», la cual transforma un CppCase a un HaxeCase.
- **Bloques de sentencias:** la transformación de un CppBlock a un HaxeBlock se realiza a través de la lazy rule «generateBlock». Se recurrió a utilizar una regla lazy para evitar la transformación innecesaria de bloques vacíos.
- **Expresiones de excepciones:** para modelar las excepciones se utilizaron tres expresiones: TryExpression, CatchClause y ThrowExpression, las cuales fueron transformadas a sus equivalentes en el lenguaje Haxe a través de las siguientes reglas:

- La regla «try» transforma una TryExpression en una HaxeTryExpression.
- La regla «catch» transforma una CatchClause en una HaxeCatchClause.
- La regla «throw» transforma una ThrowExpression en una HaxeThrowExpression.

- **Expresiones para el manejo de arreglos:** las expresiones que participan en el manejo de arreglos son ArrayAccess y ArrayInitializer, las cuales fueron transformadas a HaxeArrayAccess (a través de la regla «arrayAccess») y a HaxeArrayInitializer (a través de la regla «arrayInitializer»), respectivamente.

- **Expresiones constantes:** las expresiones constantes se transformaron de manera directa a través de las siguientes reglas:

  - La regla «stringLiteral» transforma un StringLiteral a un HaxeStringLiteral.
  - La regla «nullLiteral» transforma un NullLiteral a un HaxeNullLiteral.
  - La regla «booleanLiteral» transforma un BooleanLiteral a un HaxeBooleanLiteral.
  - La regla «numberLiteral» transforma un NumberLiteral a un HaxeNumberLiteral.
  - La regla «charLiteral» transforma un CharacterLiteral a un HaxeStringLiteral.
  - La regla «regexLiteral» transforma un RegexLiteral a un HaxeRegexLiteral.

- **Expresiones de casting de variables:** la regla «castRule» se encarga de transformar una CastExpression en una HaxeCastingExpression.

- **Expresiones break, continue, return y this:** estas expresiones se transformaron de manera directa a través de las siguientes reglas:

  - La regla «breakStatement» transforma un BreakStatement a un HaxeBreak.
  - La regla «continueStatement» transforma un ContinueStatement a un HaxeContinue.
  - La regla «return» transforma un ReturnStatement» a un HaxeReturn,
  - La regla «thisExpression» transforma una ThisExpression a una HaxeThisExpression.

- **Expresiones de acceso a elementos:** las expresiones de acceso a diferentes elementos fueron transformadas de la siguiente manera:
  - La regla «fieldAccess» transforma un FieldAccess en un HaxeFieldAccess.
  - La regla «variableAccess» transforma un VariableAccess en un HaxeSingleVariableAccess.
  - La regla «typeAccessClass» transfroma un TypeAccess en un HaxeClassifierAccess, en el caso de que el tipo sea CppClassifier.
  - La regla «typeAccessPrimitive» transfroma un TypeAccess en un HaxeClassifierAccess, en el caso de que el tipo sea PrimitiveType.

5.2.6. Declaración de Variables

Las expresiones que permiten la declaración simple de variables y las declaraciones de grupos de variables, se transformaron a través de las siguientes reglas:

- La regla «variableDeclarationGroup» se encarga de la transformación de un VariableDeclarationGroup a un HaxeVariableDeclarationGroup.
- La regla abstracta «variableDeclaration» se encarga de la transformación de una VariableDeclaration a una HaxeVariableDeclaration. De esta regla heredan las siguientes:
  - «variableDeclarationFragment» que se encarga de la transformación de un VariableDeclarationFragment en un HaxeVariableDeclarationFragment. Este caso se aplica cuando la variable que se está declarando no es de tipo arreglo.
  - «variableDeclarationFragmentArray» que se encarga de la transformación de un VariableDeclarationFragment en un HaxeVariableDeclarationFragment. Este caso se aplica cuando la variable que se está declarando es de tipo arreglo, ya que es necesaria la creación de una instancia de la metaclase «ArrayCreation».
  - «singleVariableDeclaration» que se encarga de transformar una SingleVariableDeclaration en una HaxeSingleVariableDeclaration.
- La regla «declarationExpression» se encarga de la transformación de una DeclarationExpression a una HaxeDeclarationExpression.

5.2.7. Métodos

- La regla abstracta «abstractFunction» se encarga de la transformación de una CppFunction a una HaxeAbstractFunction. De la misma hereda la regla «abstractOperation» que se encarga de transformar una CppFunction a una HaxeAbstractOperation. De la regla

«abstractOperation» hereda la regla «methods», cuyo objetivo es transformar un Method en una HaxeOperation.

- Para la transformación de las metaclases que modelan el acceso a los métodos, se partió de la regla abstracta «abstractMethodInvocation» que transforma una AbstractMethodInvocation en una HaxeAbstractMethodInvocation. De dicha regla heredan las siguientes reglas:
  - «methodInvocation» que se encarga de transformar un MethodInvocation en un HaxeMethodInvocation.
  - «superMethodInvocation» que se encarga de transformar un SuperMethodInvocation en un HaxeSuperMethodInvocation.
  - «superConstructorInvocation» que se encarga de transformar un SuperConstructorInvocation en un HaxeSuperConstructorInvocation.

5.2.8. Variables

La regla que se encarga de transformar las variables es la llamada «classVariables», la cual parte de una CppVariable y la transforma en un HaxeAttribute. En el caso de que se trate de una variable de tipo arreglo, la regla que se ejecuta es diferente y es la llamada «classVariablesArray», ya que también debe crear una instancia de «HaxeArrayCreation».

5.2.9. Comentarios

La transformación de los comentarios se realizó de manera directa a través de la regla «comments».

5.3. Ingeniería Directa

5.3.1. Acceleo

Para el proceso de Ingeniería Directa para la transformación del modelo del lenguaje Haxe en código, se utilizó Acceleo. Acceleo es un sistema de generación de código basado en el estándar MOFM2T (Modelo a Texto) del OMG. Fue desarrollado por la empresa francesa Obeo en el año 2005.

Acceleo contiene todas las herramientas que se esperan de un IDE de generación de código de calidad: sintaxis simple, generación de código eficiente, herramientas avanzadas, entre otras. Su enfoque basado en prototipos y modelos facilita la creación de generadores de texto basados en el código fuente de los prototipos existentes. Es compatible con muchos de los modelos que actualmente se pueden utilizar para representar realidades, entre los que se encuentran modelos UML, UML2, EMF, ECORE, XML, XMI, modelos generados por MoDisco (JAVAXMI).

Capítulo 6

Conclusiones

A lo largo del presente trabajo se describió un proceso general para realizar una migración desde el lenguje C/C++ hacia plataformas móviles a través del lenguaje HAXE.

Se pueden mencionar como principales contribuciones de este trabajo:

- La especificación de un metamodelo Ecore para el lenguaje C/C++, que permite su integración en MDA.
- La definición de un proceso para ingeniería inversa de código fuente C/C++ a un Modelo Específico de la Plataforma.
- La transformación de código existente en C/C++ de manera semiautomática hacia mútilples plataformas utilizando el lenguaje Haxe y el enfoque dirigido por modelos para llevarla a cabo.

Para finalizarlo, se continuará refinando el metamodelo de C/C++ desarrollado y se plantearán diferentes casos de estudio. De esta forma, se podrán evaluar las limitaciones, corregir errores que puedan surgir y validar la migración completa. Se planea también trabajar en la integración del metamodelo y del proceso de ingeniería inversa con las herramientas provistas por Eclipse, ya que solo se encuentran disponibles para el lenguaje Java.

Este trabajo será finalizado y presentado como tesis de grado de la carrera Ingeniería de Sistemas (UNICEN).

Bibliografía

[1] Acceleo (2015). Obeo. Acceleo Generator. http://www.eclipse.org/acceleo/

[2] ADM (2015). Architecture-driven modernization task force. http://www.adm.org

[3] ASTM (2011). OMG Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM), versión 1.0, http://www.omg.org/spec/ASTM.

[4] Bowen, J., Hinze, A. (2011). Supporting mobile application development with modeldriven emulation. Journal of the ECEASST, Volumen 45 (pp. 1-5).

[5] Brambilla, M. Cabot, J., Wimmer, M (2012). Model-Driven Software Enginneering in Practice, Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers.

[6] Braun, P., Eckhaus, R. (2008). Experiences on model-driven software development for mobile applications. In Proceedings of Engineering of Computer-Based Systems, IEEE International Conference and Workshop on the Engineering of computer Base Systems (pp. 490-493), Los Alamitos: IEEE Computer Society.

[7] Bruneliere, H. Cabot, J. Dupé, G., Madiot, F. (2014). MoDisco: a Model Driven Reverse Engineering Framework. Information and Software Technology 56 (pp. 1012- 1032).

[8] Cannasse, N. (2014). HaXe. Too Good to be True? GameDuell Tech Talk. http://www.techtalk-berlin.de/news/read/nicolas-cannasse-introducing-haxe/

[9] CASE MDA. (2015). Committed companies and their products. www.omg.org/mda/committed-products.htm

[10] Dasnois, B. (2011). HaXe 2 Beginner's Guide. Packt Publishing.

[11] Dehlinger, J., Dixon, J. (2011). Mobile application software engineering: Challenges and research directions. Proceedings of the Workshop on Mobile Software Engineering (pp. 29-32), Berlin Heidelberg: Springer-Verlag.

[12] Díaz Bilotto, Pablo; Favre, Liliana (en prensa) Migrating JAVA to Mobile Platforms through HAXE: An MDD Approach," Aceptado como capítulo del libro "Modern Software Engineering Methodologies for Mobile and Cloud Environments". Editores: Antonio Miguel Cruz , Sara Paiva. IGI GLOBAL, USA. 2016

[13] Dunkel, J., Bruns, R. (2007). Model-driven architecture for mobile

applications. In Business Information Systems, W. Abramowicz, Ed., Lecture Notes in Computer Science, Volumen 4439 (pp. 464-477), Berlin Heidelberg: Springer-Verlag.

[14] EMF (2015) Eclipse Modeling Framework (EMF) Retrieved April 15, 2015 from http://www.eclipse.org/modeling/emf/

[15] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I (2008). ATL: A model transformation tool. Science of Computer Programming 72, 1 (pp. 31-39).

[16] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P: (2006) ATL: a QVT-like transformation language. Companion to the 21st ACM SIGPLAN Symposium on Objectoriented programming systems, languages, and applications (New York, NY, USA, 2006), OOPSLA '06 (pp. 719-720), New York: ACM Press.

[17] Jouault, F., Kurtev, I. (2005). Transforming models with ATL. In Satellite Events at the MoDELS 2005 Conference (Berlin, 2006), Lecture Notes in Computer Science, vol. 3844, Berlin Heidelberg: Springer Verlag (pp. 128-138).

[18]KDM. OMG (2011). Architecture-Driven Modernization: Knowledge Discovery MetaModel (KDM), versión 1.3, http://www.omg.org/

[19] Kim, H. K. (2008). Frameworks of process improvement for mobile applications. Engineering Letters 16, 4 (pp. 550-555).

[20] Kramer, D., Clark, T., Oussena, S. (2010). MobDSL: A domain specific language for multiple mobile platform deployment. Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference (pp. 1-7), Los Alamitos: IEEE Press.

[21] Lettner, M., Tschernuth, M., Mayrhofer, R. (2012). Mobile platform architecture review: Android, Iphone, Qt. In Computer Aided Systems Theory EUROCAST 2011, R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, Lecture Notes in Computer Science, Volumen 6928 (pp. 544-551). Berlin Heidelberg: Springer-Verlag.

[22] MMT (2015). Model-to-Model Transformation. Eclipse Modeling Framework. Retrieved April 15, 2015, https://www.eclipse.org/mmt/

[23] MoDisco. (2012). Model discovery. Retrieved April 15, 2015 http://www.eclipse.org/MoDisco

[24] MOF (2011). OMG Meta Object Facility (MOF) core specification version 2.4.1. OMG Document Number:formal/2011-08-07. http://www.omg.org/spec/MOF/2.4.1

[25] MOFM2T (2008). MOF Model to Text Transformation Language, Version 1.0. http://www.omg.org/spec/MOFM2T/1.0/

[26] OCL (2014). OMG Object constraint language (OCL), version 2.4. http://www.omg.org/spec/OCL/2.4

[27] OCLinEcore (2015) OCLinEcore Editor. http://wiki.eclipse.org/MDT/OCLinEcore.

[28] OMG MDA (2014). MDA guide version rev. 2.0 OMG Document ormsc/2014-06-01. http://www.omg.org/cgi-bin/doc?ormsc/14-06-01

[29] Parr, T. (2013) The Definitive ANTLR 4 Reference. Pragmatic Bokkshelf, Second Edition.

[30] Pérez Castillo, R., García Rodriguez, I., Gómez Cornejo, R., Fernández Ropero, M., Piattini, M. (2013). ANDRIU. A Technique for Migrating Graphical User Interfaces to Android. Proceedings of The 25th International Conference on Software Engineering and Knowledge Engineering (SEKE 2013) (pp. 516-519) Boston: Knowledge Systems Institute.

[31] QVT (2012). QVT: MOF 2.0 query, view, transformation:Version 1.1. OMG Document Number: formal/2011-01-01. http://www.omg.org/spec/QVT/1.1/SMM

[32] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E. (2009). EMF: Eclipse Modeling Framework, 2 ed. Addison-Wesley, Boston, MA.

[33] Thompson, C., Schmidt, D., Turner, H., White, J. (2011). Analyzing Mobile Application Software Power Consumption via Model-Driven Engineering, Proceedings of PECCS 2011 (pp. 101-113).

[34] Wasserman, A. I. (2010). Software engineering issues for mobile application development. Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10), (pp. 397-400) New York, NY: ACM.

[35] Umuhoza, E., Ed-douibi, H., Brambilla, M., Cabot, J., Bongio, A. (2015). Automatic Code Generation for Cross-platform, Multi-Device Mobile Apps: Some Reflections from an Industrial Experience. In 3rd International Workshop on Mobile Development Lifecycle, MobileDeli 2015.

[36] Biehl, Matthias (2010). Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology.