UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Lenguajes formales y derivación automática de código de pruebas a partir de modelos de software con restricciones OCL
Autores: Ilan Rosenfeld
Director: Claudia Pons
Codirector: -
Asesor profesional: Gabriela Alejandra Perez
Carrera: Licenciatura en Informática

Resumen

Qué testear es un tema siempre vigente. Con tal propósito, y en caso de querer derivar casos de prueba automáticamente desde un modelo de software, no contamos con la precisión necesaria en los mismos para generar los tests acordes a nuestros intereses. Lenguajes formales como OCL permiten enriquecer cualquier modelo mediante información adicional o restricciones sobre sus elementos; entonces, la derivación de código y casos de prueba a partir de un modelo enriquecido con restricciones y especificaciones en este lenguaje permitirá contar con un soporte mucho más robusto de nuestro sistema.
Tras el análisis de varias herramientas de generación automática de código a partir de modelos de software, se llegó a la conclusión de que casi ninguna de ellas incluye la traducción de restricciones en el modelo escritas en un lenguaje formal. Es por ello que la tesina consistió en un análisis de varios lenguajes formales de especificación (o modelado), especialmente de los lenguajes UML/OCL y Alloy. A partir de ello se desarrolló una herramienta para Eclipse, que permite, a partir de una especificación de un modelo UML poseyendo restricciones OCL, la generación automática de código Java, incluyendo las clases del modelo junto con sus respectivos Casos de Prueba, regulados por OCL. Paralelamente se genera de forma automática una especificación Alloy que permite el análisis formal estático del modelo.

Palabras Claves

Acceleo, Alloy, AlloyMDA, Desarrollo de Software dirigido por Modelos (MDD), EasyMock, Eclipse, Java, JUnit, Mockito, OCL, Papyrus, Pruebas de Software Dirigidas por Modelos (MDT), PSM, Traducción de código, U2TP, UML, Verificación formal.

Trabajos Realizados

Investigación teórica del MDD, del MDT y de lenguajes de especificación formal, específicamente OCL y Alloy. Desarrollo de código de traducción automática Acceleo para generar casos de prueba Java a partir de un modelo de datos con restricciones OCL. Uso y aprovechamiento de la herramienta AlloyMDA para verificación formal del código resultante.

Conclusiones

Se creó una herramienta que permite traducir un modelo de datos con restricciones formales a su correspondiente código Java, automatizando la generación de casos de prueba robustos y especificados no sólo en este lenguaje sino también en dos lenguajes formales (OCL y Alloy), lo que le da un soporte confiable y verificable con diversas técnicas. Se lograron integrar distintos lenguajes, cada uno con su sintaxis propia, para concluir en un resultado que es más que utilizable por el usuario que así lo desee y que le permite detectar inconsistencias en su modelo.

Trabajos Futuros

Permitir especificar las restricciones en el modelo fuente directamente en el lenguaje Alloy; lograr que Acceleo regenere el código sin alterar las actualizaciones realizadas o aquel texto que estuviera delimitado por separadores especiales; contar con tests menos abstractos y desprenderse de a poco del uso de mocks, para generar tests más específicos; generar contraejemplos en lenguaje natural/Java y soluciones posibles a las inconsistencias; realizar el mismo proceso permitiendo la selección de diversos lenguajes de programación destino.

Fecha de la presentación: Noviembre, 2015

Lenguajes formales y derivación automática de código de pruebas a partir de modelos de software con restricciones OCL

Tesina de Licenciatura

Alumnos:
Ilan Rosenfeld

Director:
Dra. Claudia Pons

Asesor:
Gabriela Alejandra Perez

2015

Facultad de Informática
Universidad Nacional de La Plata

A mi familia, por su apoyo incondicional.
A mis amigos, especialmente aquellos que me acompañaron a lo largo de esta hermosa carrera.
A Claudia, por su ayuda y cariño.

Índice General

Índice de Figuras 7

Índice de Tablas 9

1 Introducción 11

2 Desarrollo de Software Dirigido por Modelos 13
- 2.1 Introducción 13
- 2.2 El origen del MDD 13
- 2.3 Modelos 15
- 2.4 Metamodelos 16
- 2.5 La Arquitectura Dirigida por Modelos (MDA 16
- 2.6 MOF 17
- 2.7 Resumen 17

3 Pruebas de Software Dirigidas por Modelos 19
- 3.1 De qué hablamos al hablar de Testing 19
- 3.2 Testing Dirigido por Modelos 21
- 3.3 Perfiles de Testing Aplicados a Modelos de Software 24
- 3.4 JUnit 26
- 3.5 EasyMock 28
  - 3.5.2 Mockito 29
- 3.6 Resumen 29

4 Lenguajes Formales 31
- 4.1 Introducción 31
- 4.2 Z 32
- 4.3 UML/OCL 33
- 4.4 Alloy 37
- 4.5 Comparando Lenguajes Formales 40
  - 4.5.1 Alloy – Z 40
  - 4.5.2 Alloy –UML/OCL 40
    - 4.5.2.1 Diferencias de Complejidad 41
    - 4.5.2.2 Diferencias de Precisión 42
    - 4.5.2.3 Diferencias de Expresividad 42
    - 4.5.2.4 Diferencias Textuales 44
    - 4.5.2.5 Conclusiones 45
- 4.6 Traducción entre Alloy y UML/OCL 45
  - 4.6.1 UML/OCL a Alloy 45
    - 4.6.1.1 Metamodelo Alloy 47
    - 4.6.1.2 Mapeando diagramas de clases a Alloy 48
    - 4.6.1.3 Diferencias entre UML y Alloy que influyen en la transformación 49
    - 4.6.1.4 Herramientas de Traducción 52
    - 4.6.1.5 Conclusión 53
  - 4.6.2 Alloy a UML/OCL 54
- 4.7 Resumen 56

5 Transformaciones de Modelos 57
5.1 Transformaciones 57
5.2 Tipos y Elección de Transformaciones de Modelos 58
5.3 Testing de Transformaciones de Modelos 60

6 Herramientas para el modelado, traducción y verificación en Eclipse 63
6.1 Eclipse Modeling Framework (EMF) 63
6.2 Herramientas para el Desarrollo de Modelos MDT 63
6.3 Acceleo 65

7 Implementación de la Solución 69
7.1 Creando el diagrama de clases con *Papyrus* 69
7.2 Código de Traducción Acceleo 72
7.2.1 Introducción 72
7.2.2 Código Acceleo Final 74
7.3 Análisis de los resultados obtenidos 79
7.4 Traduciendo de OCL a Alloy – Verificación formal 83
7.5 Resumen 93

8 Trabajos Relacionados 95
8.1 TestEra 95
8.2 Modelo de base para UML y verificación OCL 96
8.3 Generación de Tests basado en modelos para aplicaciones Web 97
8.4 Fokus!MBT – Un entorno de modelado de Tests multiparadigma 98
8.5 AlloyMDA 98
8.6 Casos de Prueba del Sistema Generados en el Contexto MDD/MDT 99
8.7 Perfiles de testing aplicados a modelos de software 99

9 Conclusiones y Trabajos Futuros 101

10 Referencias 103

Índice de Figuras

Figura 2.1 Arquitectura de 4 capas de modelado del OMG 17
Figura 3.1 Representación 3D de los tipos de Testing 21
Figura 3.2 El proceso de MDT 23
Figura 3.3 Metamodelo de UTP 25
Figura 3.4 Diagrama de clases de JUnit 27
Figura 4.1 Schema de Z 33
Figura 4.2 Diagrama de Clases ejemplo 1 34
Figura 4.3 Diagrama de Clases ejemplo 2 35
Figura 4.4 Modelo de Objetos de Alloy para un árbol familiar 38
Figura 4.5 Modelo Alloy de los estudiantes 43
Figura 4.6 Modelo UML de los estudiantes 43
Figura 4.7 Adelanto del método de transformación 46
Figura 4.8 Metamodelo UML simplificado 46
Figura 4.9 Parte del metamodelo Alloy correspondiente a las declaraciones de signatures 47
Figura 4.10 Modelo del protocolo SSL utilizado en el servicio de login 48
Figura 4.11 Múltiples capas en UML2Alloy 53
Figura 4.12 Pasos para convertir el modelo Alloy en uno UML 55
Figura 4.13 Transformación de modelos multi-nivel, con ejemplos 56
Figura 5.1 Definiciones de transformaciones dentro de las herramientas de Transformación 57
Figura 6.1 Template por defecto (generate.mtl) en el lenguaje de templates Acceleo 65
Figura 6.2 Código por defecto del archivo generate.mtl 66
Figura 7.1 Creando un modelo Papyrus 69
Figura 7.2 Proyecto Papyrus 70
Figura 7.3 Paleta Papyrus 70
Figura 7.4 Diagrama de clases del caso de estudio 71
Figura 7.5 Creando un módulo Acceleo 73
Figura 7.6 Estructura de directorios de un proyecto Acceleo 73
Figura 7.7 Código por defecto del archivo generate.mtl 73
Figura 7.8 Inicio del código Acceleo 74
Figura 7.9 Generación del Chequeador 75
Figura 7.10 Generando los métodos de las clases 76
Figura 7.11 Generando los casos de prueba 77
Figura 7.12 Archivos generados tras ejecutar el código Acceleo 79
Figura 7.13 Código del Test de Integración 79
Figura 7.14 Éxito en la ejecución del Test de Integración 79
Figura 7.15 Clase Student generada 81
Figura 7.16 Clase de prueba TestStudent generada 83
Figura 7.17 Código OCL centralizado generado 84
Figura 7.18 Validando el modelo a nivel OCL 84
Figura 7.19 Éxito en la validación 85
Figura 7.20 Modelo OCL simplificado a utilizar con AlloyMDA 85
Figura 7.21 Porción del código del Modelo UML a utilizar con AlloyMDA 86
Figura 7.22 Corriendo AlloyMDA sobre nuestros archivos 87
Figura 7.23 Modelo Alloy pasado en limpio 88
Figura 7.24 Pantalla principal del Alloy Analyzer 89

Figura 7.25 Mensaje de la consola del Alloy Analyzer tras correr el programa 90
Figura 7.26 Modelo generado por el analizador 90
Figura 7.27 Sin instancias del modelo a mostrar 92
Figura 7.28 Se logró detectar un contraejemplo 92
Figura 7.29 Contraejemplo encontrado 93
Figura 8.1 Programa Java de LinkedList 95
Figura 8.2 Ejemplo de Test de JUnit 96
Figura 8.3 Flujo de proceso empleando el modelo base 97
Figura 8.4 Transformación de una agregación en una asociación con una restricción 97

Índice de Tablas

Tabla 4.1 Operación allParents() en OCL y Alloy 41
Tabla 4.2 Operación allParents() sin circularidad en OCL y Alloy 41
Tabla 4.3 Correspondencia entre los metamodelos UML y Alloy 49
Tabla 5.1 Transformaciónes de modelos: Endógena vs. Exógena, Horizontal vs Vertical 59
Tabla 8.1 Comparando ambos proyectos 99

1. *Introducción*

En los últimos años, el Desarrollo de Software Dirigido por Modelos (*denominado MDD por su acrónimo en inglés, model-driven development*) ha ido ganando territorio en el ámbito informático, con el desarrollo de tecnologías e innovaciones cuyo objetivo es atribuir a los modelos el papel principal y activo en el proceso de desarrollo de software, frente a las propuestas tradicionales, logrando la independencia del software y la portabilidad de los sistemas, y separando el diseño de la arquitectura. A través de una serie de transformaciones, tal modelo puede ser traducido a código fuente, dependiente de una plataforma específica.
Como consecuencia, se mejora la productividad del sistema, se aumenta su calidad, y se facilita su comprensión, evolución, mantenimiento y reutilización/reimplementación en otras tecnologías.

Una de las ramas del MDD es el Model-Driven Testing o MDT (Pruebas de Software Dirigidas por Modelos), un enfoque nuevo y prometedor para la automatización del testeo del software, que puede reducir significativamente los esfuerzos en el tedioso ciclo de testing de todo desarrollo de software. Consiste en una forma de prueba de caja negra que utiliza modelos estructurales y de comportamiento para automatizar el proceso de generación de casos de prueba. Cabe destacar que el modelo a traducir debe ser conciso, preciso y completo: conciso de forma de que no lleve mucho tiempo escribirse y que sea fácil de validar con respecto a sus requerimientos, y lo suficientemente preciso y completo como para describir el comportamiento que debe ser testeado.
Los tests producidos desde el modelo serán abstractos, debiendo transformarse en ejecutables; esto requiere una participación del ingeniero de tests, aunque la mayoría de las herramientas de testeo basadas en modelos proveen cierta asistencia durante este proceso.

Al pensar en representación de modelos, suele pensarse en notaciones gráficas como UML. Observando que los modelos deben ser precisos y completos, debemos eliminar toda ambigüedad de los mismos. Tradicionalmente, en el caso de UML, lo que suele hacerse es completar los modelos con descripciones en lenguaje natural y cualquier otra información adicional sobre el modelo que sea relevante para aportar a esa causa. El problema de las descripciones en lenguaje natural es que, si bien son fáciles de escribir y leer por cualquier persona, suelen ser ambiguas, y desde luego, no son manipulables por otros programas.
En este contexto, surge OCL (Object Constraint Language), un lenguaje textual con base formal, basado en teoría de conjuntos y lógica de primer orden, y con mecanismos y conceptos muy cercanos a los de UML por su naturaleza orientada a objetos, lo que facilita su uso por parte de los modeladores. UML ve en OCL el candidato idóneo para la expresión de restricciones de integridad sobre sus modelos.

Tras el análisis de varias herramientas de generación automática de código a partir de modelos de software, se llegó a la conclusión de que ninguna de ellas incluye la traducción de restricciones en el modelo escritas en un lenguaje formal. Es por ello que el trabajo consistirá en un análisis de los lenguajes formales, particularmente de los lenguajes OCL y Alloy, quedándonos con el primero para el desarrollo de una herramienta para Eclipse, que permita, a partir de una especificación de un modelo poseyendo restricciones OCL, la generación automática de código Java, incluyendo las clases del

modelo junto con sus respectivos Casos de Prueba, regulados por OCL; además, el código OCL se traducirá luego a su correspondiente código Alloy para su verificación formal.

La tesina se organiza de la siguiente manera:

- En el capítulo 2, se realiza una introducción al mundo del Desarrollo de Software Dirigido por Modelos.
- En el capítulo 3, se definen varios conceptos relacionados con el Testing de sistemas de información y su aplicación en el perfil de testing de UML, además de la especificación de las herramientas de testing más importantes con las que Eclipse cuenta.
- En el capítulo 4, se introducen los lenguajes formales, específicamente Alloy y OCL, detallando una comparación entre los mismos, ventajas y desventajas de cada uno, herramientas disponibles para traducción entre estos lenguajes.
- En el capítulo 5, se trata sobre las transformaciones de modelos, definiendo previamente el concepto de transformación, para luego tipificar, indicar qué tipo de transformación es más conveniente en cada caso particular, y observar cómo se testean dichos procesos.
- En el capítulo 6, se mencionan las herramientas con las que Eclipse cuenta para llevar a cabo el modelado, la verificación y la traducción de código, especificando aquellas que serán utilizadas para el objetivo final.
- En el capítulo 7, se procede a implementar la solución, con un caso de estudio y su especificación en Eclipse mediante la herramienta *Papyrus*, su traducción mediante *Acceleo*, el análisis del modelo resultante, y el uso de la herramienta *AlloyMDA* para traducir este último a su respectivo código Alloy y así verificarlo formalmente.
- En el capítulo 8, se pueden observar trabajos relacionados que sirvieron tanto para motivar la realización del proyecto como para mejorarlo.
- En el capítulo 9, se finaliza con las conclusiones teóricas y prácticas de la investigación desarrollada, mencionando una serie de trabajos futuros que pueden llevarse a cabo a partir de ella.
- En el capítulo 10, se citan las referencias.

2. Desarrollo de Software Dirigido por Modelos

El desarrollo de software dirigido por modelos [3] es una propuesta para el desarrollo de software en la que se atribuye a los modelos el papel principal, frente a las propuestas tradicionales basadas en lenguajes de programación y plataformas de objetos y componentes software.
En este capítulo, definiremos sus componentes principales, sus ramas más importantes y su utilización y aprovechamiento dentro de la industria, tanto en la actualidad como a futuro.

2.1 . Introducción

El desarrollo de software dirigido por modelos [2] (desde aquí, abreviado MDD por su acrónimo en inglés, *model-driven development)* se basa en la noción de que los sistemas de software deben ser desarrollados a través de la utilización de modelos.
Los procesos de MDD suelen comenzar con el objetivo de desarrollar un sistema de software con una fase de requerimientos, en la cual se define un modelo de requerimientos para describir las necesidades del usuario de una manera independiente a la computación. Luego, este modelo es refinado en uno o más modelos conceptuales, que describen el sistema sin considerar aspectos tecnológicos; estos son utilizados principalmente en las fases de análisis.
Finalmente, los modelos mencionados son refinados en modelos de diseño que describen el sistema utilizando conceptos de una tecnología específica y son traducidos a código, o bien son directamente derivados a código en el caso de contener suficiente información para implementar un producto de software de una manera precisa y completa.

2.2 . El Origen del MDD

El proceso de desarrollo de software ha resultado históricamente caro, riesgoso, incierto y demasiado lento para las condiciones de negocio modernas [1], donde una demanda costosa y creciente atormenta a aquellos sistemas de software cuyas bases no están definidas en base a buenas prácticas. Estos inconvenientes dieron lugar en su momento a la conocida “crisis del software”, que surgió prácticamente de la mano con la creación del software mismo.
El término “crisis del software” fue acuñado en 1968, en la primera conferencia organizada por la OTAN sobre el desarrollo de software, etiquetando junto a él a aquellos problemas que surgían ocasionalmente en los sistemas de software. En esta misma conferencia se utilizó por primera vez el término “ingeniería de software”, para describir los conocimientos que existían en aquel entonces sobre el software.
La crisis se origina por varias razones no mutuamente excluyentes, sino que la verdadera razón puede darse como una mezcla de todas ellas. Todas estas razones pueden vincularse a la complejidad en general del proceso de software y a la relativa inmadurez de la ingeniería de software como una profesión.
Uno de los principales problemas recae en que la mayoría de los proyectos dan comienzo a la programación tan pronto son definidos, concentrando la mayor parte de su esfuerzo en la escritura del código. Los principales “síntomas” que encontramos en este proceso de desarrollo cada vez más lento son, entre otros:

- Baja calidad del software

- Tiempo y Presupuesto excedido
- Confiabilidad Cuestionable
- Altos requerimientos de personal para desarrollo y mantenimiento

Sobre esta crisis escribía Edsger Dijkstra en su paper titulado “The Humble Programmer” (*El programador humilde*) [4]. El mismo detalla una discusión que el autor disputó junto a su jefe, A. van Wijngaarden, en 1955, año en que Dijkstra era un programador al mismo tiempo que estudiaba física teórica. Preguntas filosóficas recorrían la cabeza del autor.

> *“Debía decidirme entre dejar de programar y convertirme en un físico teórico real y respetable, o usar mis estudios de la física tan sólo para una terminación formal y convertirme en un… ¿programador? Pero, ¿era esa una profesión respetable? Después de todo, ¿qué era programar?*

Tras oírlo respetuosamente, su jefe Wijngaarden coincidió con el preocupado personaje en que hasta ese momento no existía una disciplina de programación (tanto así que al casarse Dijkstra y deber especificar por obligación su profesión en el acta de casamiento, debió especificar, para su enojo, “físico teórico”), pero continuó explicándole que las computadoras automáticas estaban allí para quedarse.
Y así fue. Pero, en vez de aportar esto a un crecimiento de calidad del software y a resolver todos los problemas de programación, los programadores se encontraron en la denominada “crisis del software”. Mientras no había habido máquinas, la programación no había sido un problema en absoluto; al desarrollarse las primeras, la programación se volvió un leve problema; y tras el desarrollo masivo de ellas, un problema gigante.
A medida que el poder disponible en las computadoras crecía, la ambición de la sociedad para aprovechar las mismas crecía en proporción, encontrándose el pobre programador en las vísperas de abandonar su profesión. Y esta crisis no era una sorpresa, sino que había sido predicha con antelación.

Actualmente, la magnitud de este problema continúa creciendo. Las organizaciones operan en entornos hipercompetitivos con el objetivo de satisfacer al cliente con servicios de calidad cada vez mayor. Esto supone un desafío de complejidad y la ocurrencia de cambios incrementales, lo que requiere evolución, agilidad y flexibilidad de la organización y de sus sistemas de información. Como consecuencia, la estabilidad se torna una propiedad fundamental de un sistema, que debe responder a los cambios constantes a los que tiene que someterse el software para adaptarse a las necesidades cambiantes de los usuarios y a las innovaciones tecnológicas
Si asumimos la evolución ilimitada de los sistemas en una cantidad de tiempo infinita, el número total de requerimientos y sus dependencias se tornará intratable [5], a no ser que tratemos con principios y teoremas que aporten a un desarrollo de software limpio, claro, mantenible y escalable desde sus orígenes.
El costo de mantenimiento de un sistema es muy elevado, dividiéndose este en:

- Mantenimiento perfectivo: 65% [6]
- Proveer mejoras (adaptativo y perfectivo): 75% [7]
- Entender código existente: 50% del tiempo de mantenimiento

La incapacidad de cambiar el software de manera rápida y segura implica que se pierden oportunidades de negocio.
En este contexto, surge el MDD.

2.3 Modelos

Los modelos son tan antiguos como las ingenierías. Los ingenieros tradicionales siempre construyen modelos antes de construir sus obras y artefactos. El modelo del sistema es una conceptualización del dominio del problema y de su solución [1], que se focaliza sobre el mundo real: identificando, clasificando y abstrayendo los elementos que constituyen el problema y organizándolos en una estructura formal. Es una representación conceptual o física a escala de un proceso o sistema, con el fin de analizar su naturaleza, desarrollar o comprobar hipótesis o supuestos y permitir una mejor comprensión del problema.

Los modelos sirven para:

- Especificar el sistema: su estructura, comportamiento, comunicación con los distintos *stakeholders*
- Comprender el sistema
- Razonar y validar el sistema: detectar errores y omisiones en el diseño, prototipado, inferir y demostrar propiedades
- Guiar la implementación
- Documentación del sistema

Claro está que el acrónimo MDD enfatiza el hecho de que los modelos son el foco central de MDD. Entre sus características principales, tenemos que un modelo es **abstracto**. En palabras de Edsger W. Dijkstra:

> *"ser abstracto no significa en absoluto ser impreciso [...] El principal propósito de la abstracción es definir un nivel semántico en el que poder ser totalmente preciso".*

Es por ello por lo que es vital determinar el punto de vista desde donde plantear el problema, que identifica de forma precisa tanto los conceptos que se deben tener en cuenta a la hora de plantear el mismo y su solución, como aquellos aspectos que podemos ignorar por no ser relevantes desde ese punto de vista.

Estas definiciones abarcan muchos tipos distintos de modelos interdependientes a diferentes niveles de abstracción (análisis, diseño, implementación), cada uno representando un aspecto del sistema, que puede ser especificado a un nivel más alto de abstracción y de forma independiente de la tecnología utilizada.

A grandes rasgos, podemos clasificar a los modelos como:

- **Modelo independiente de la computación (CIM):** representa una vista del sistema desde un punto de vista independiente a la computación. Se enfoca en el sistema y en su entorno; los detalles de la estructura del sistema no son tenidos en cuenta o no están aún especificados. Suele ser llamado el *modelo de dominio* o el *modelo de negocios*, y es especificado utilizando un vocabulario familiar a los profesionales del dominio en cuestión. Se asume que está destinado a usuarios sin conocimientos técnicos acerca de los artefactos a utilizar para implementar el sistema.
- **Modelo Independiente de la Plataforma (PIM):** es una vista del sistema desde un punto de vista que exhibe independiente de la plataforma. Adecuado para usar junto a un número de plataformas distinto de tipo similar. Se tiene en cuenta el cómo soportar mejor al negocio más que cómo va a ser generado, ignorando sistemas operativos, lenguajes de programación, hardware, etc.
- **Modelo específico de plataforma (PSM):** es una vista del sistema desde el punto de vista específico de la plataforma. Combina las especificaciones en el PIM con los detalles que especifican cómo usa el sistema una plataforma de tipo particular. El PIM puede igualmente transformarse en uno o más PSM, cada uno para una

tecnología en particular (por ejemplo, uno para Java contiene términos como clase, interfaz, etc)

- **Modelo de la implementación (Código):** paso final en el desarrollo, transformando cada PSM a código fuente, en un proceso bastante directo ya que el PSM está orientado al dominio tecnológico específico.

2.4 Metamodelos

Tras definir los modelos, es importante señalar el hecho de que el lenguaje utilizado para describir a los mismos debe estar bien definido y ofrecer un nivel de abstracción adecuado para expresarlo y razonar sobre él. Si bien se contaba con mecanismos útiles y precisos para definir la sintaxis de los lenguajes, estos estaban más bien pensados para lenguajes textuales, y no tanto así para los gráficos (como lo son los lenguajes de modelado), surgiendo la necesidad de recurrir a un mecanismo distinto para definirlos. La idea compartida por todos los paradigmas encerrados dentro del MDD es la conveniencia de utilizar para el modelado lenguajes de mayor nivel de abstracción que los lenguajes de programación, que manejen conceptos más cercanos al dominio del problema. Estos se llaman Lenguajes específicos de dominio (o DSL, *domain-specific language*). A su vez, estos lenguajes requieren una descripción precisa, siendo lo más apropiado definir a la misma también como un modelo.

Surge entonces una técnica específica para facilitar la definición de los lenguajes gráficos, llamada “metamodelado”. Un **metamodelo** es un modelo que especifica los conceptos de un lenguaje, las relaciones entre ellos y las reglas estructurales que restringen los posibles elementos de los modelos válidos, así como aquellas combinaciones entre elementos que respetan las reglas semánticas del dominio [3]. Un metamodelo es también un modelo, por lo que debe estar escrito en un lenguaje bien definido: este lenguaje se denomina **metalenguaje** (por ejemplo, *BNF* es un metalenguaje).

Por ejemplo, el metamodelo de UML es un modelo que posee los elementos para describir modelos UML, como *Package*, *Classifier*, *Class, Operation*, *Association*, etc. El mismo también define las relaciones entre estos conceptos, y las restricciones de integridad de los modelos UML (una de ellas, la que obliga a que las asociaciones sólo puedan conectar *classifiers*, y no paquetes u operaciones).

2.5 La Arquitectura Dirigida por Modelos (MDA)

El metamodelado entonces es un mecanismo que permite definir formalmente lenguajes de modelado, como por ejemplo UML. OMG (*Object Management Group)* propuso en el 2000 una arquitectura de cuatro capas de modelado, orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos.

Los 4 niveles definidos en esta arquitectura se denominan M3, M2, M1 y M0 (del más abstracto al más concreto), como se observa en la figura 2.1.

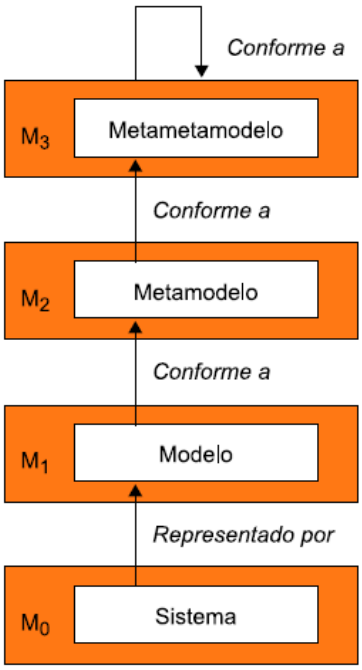
MDA concibe la construcción de modelos de software a distintos niveles de abstracción, como artefactos principales en el desarrollo de software. Cumple a su vez un rol importante en la transformación de modelos para automatizar la derivación de un modelo a otro.

Los principales objetivos de MDA son:

• Interoperabilidad (independencia de los fabricantes a través de estandarizaciones).

• Portabilidad (independencia de la plataforma) de los sistemas de software.

• Separación del diseño del sistema tanto de la arquitectura como de las tecnologías de construcción, facilitando así que el diseño y la arquitectura puedan ser alterados independientemente.



- En el modelo M0 se encuentran las instancias "reales" del sistema, los objetos de la aplicación.
- Por encima de la capa M0 se sitúa la capa M1, que representa el modelo de un sistema de software. Sus conceptos representan categorías de las instancias de M0: cada elemento de M0 es una instancia de un elemento de M1.
- Análogamente a como ocurre entre M0 y M1, los elementos de M1 son a su vez instancias del nivel M2, recibiendo la capa el nombre de Metamodelo.
- Por último, podemos ver los elementos de M2 como instancias de la capa M3 o capa de meta-metamodelo, un modelo que define el lenguaje para representar un metamodelo. Este es el nivel más abstracto, que permite definir metamodelos concretos. Dentro del OMG, MOF es su lenguaje estándar, siendo entonces que todos los metamodelos de M2 son instancias de MOF.

Figura 2.1 Arquitectura de 4 capas de modelado del OMG

En síntesis, los beneficios de utilizar MDA son:

- Contar con un soporte completo para el ciclo de vida de la aplicación
- Reducción de costos de principio a fin
- Reutilización de código, aplicaciones, experiencia
- Representación independiente de la tecnología
- Escalabilidad y robustez
- Interoperabilidad entre distintas tecnologías

2.6 MOF

El lenguaje MOF (*meta-object facility*) es el lenguaje de la OMG para describir metamodelos, siendo UML un lenguaje de modelado conforme a él. Es un estándar para la ingeniería conducida por modelos, ubicada como vimos recientemente en la capa superior de la arquitectura de 4 capas.
MOF provee un meta-metalenguaje que permite definir metamodelos en la capa M2. El ejemplo más conocido de un elemento en la capa M2 es el metamodelo UML, que describe al lenguaje UML. Comprende una arquitectura de metamodelado cerrada y estricta: cerrada, porque el metamodelo de MOF se define en términos de sí mismo; y estricta, pues cada elemento de un modelo en cualquiera de las capas posee una correspondencia estricta con un elemento del modelo de la capa superior.

2.7 Resumen

Actualmente, diferentes factores nos permiten considerar MDD como una realidad [2]. Por un lado, en los últimos años han surgido un conjunto de métodos que proponen modelos basados en el lenguaje de modelado unificado (UML) o en los lenguajes de modelado de dominio específico (DSL) para construir software a partir de modelos. Además, el surgimiento de las metodologías MDD es alentado por la arquitectura dirigida por modelos MDA, creada por el grupo OMG. Se están desarrollando nuevas tecnologías que aportan a la causa de MDD, tales como Ecore (editor de diagramas de Eclipse), el Framework de

Modelado de Eclipse (EMF) y el Framework de Modelado Gráfico (GMF) (ambos asociados al proyecto de modelado de Eclipse), entre otros.
En síntesis, las razones para utilizar MDD son:

- Proveer a nuestros expertos del dominio una forma de especificar formalmente su conocimiento, y luego los expertos en tecnología definirán cómo esto se implementa (utilizando transformaciones de modelos)
- Proveer distintas implementaciones del mismo modelo (por ejemplo, con modelos más concretos), con el objetivo de correrlo en diferentes plataformas (.NET, Java, CORBA).
- Capturar conocimiento sobre el dominio y la tecnología libres de los detalles de otras áreas, no relevantes.
- En general, uno no desea contaminar la especificación de la funcionalidad con detalles de implementación. A través de MDD, esto puede lograrse.

Para finalizar el capítulo, citamos dos frases del célebre Daniel Jackson:

"The core of software development is the design of abstractions."
(El desarrollo de software se centra en el diseño de abstracciones)

"An abstraction is not a module, or an interface, class, or method; it is a structure, pure and simple - an idea reduced to its essential form."
(Una abstracción no es un módulo, o una interface, o un método; es una estructura, pura y simple – una idea reducida a su forma esencial)

3. *Pruebas de Software Dirigidas por Modelos*

El testeo de software es vital en el proceso de desarrollo del mismo, reflejado esto en un mercado cada vez más demandante de herramientas de testing automáticas. Específicamente, el testeo de software consume generalmente entre el 30 y el 60 por ciento del esfuerzo de desarrollo total [8].
Muchas compañías ya están utilizando herramientas de ejecución de tests automáticas.
El testing basado en modelos (MDT desde ahora, por su acrónimo en inglés *Model-Driven Testing*) incrementa aún más el nivel de automatización, automatizando no sólo la ejecución de los casos de prueba, sino también su diseño; los casos de prueba son generados automáticamente desde un modelo del producto de software. Esto deriva en una base repetible y racional para testear el producto, asegurando el cubrimiento de todos los comportamientos del producto, y permitiendo que las pruebas sean ligadas directamente a los requerimientos.
Entre los últimos cinco y diez años, se han llevado a cabo investigaciones intensivas sobre el MDT, llegando a demostrarse la factibilidad de este enfoque y su efectividad en relación al costo; por su parte, se han desarrollado un conjunto de estrategias de generación de tests y criterios de cobertura de modelos. Gracias a esto, hoy en día contamos con un rango de herramientas comerciales disponibles de MDT.

Tras una rápida descripción del aporte y utilidad del Desarrollo de Software Dirigido por Modelos, enfocaremos nuestra atención ahora en el aprovechamiento que puede hacerse de esta nueva técnica para aplicarla en el testeo del software. Previo a ello, realizaremos una breve introducción para definir claramente de qué hablamos al hablar del testeo de software.

3.1 De qué hablamos al hablar de Testing

La mayoría de los programadores posee una noción de qué es un Test. Gran parte de ellos desarrolla día a día casos de prueba para sus sistemas, e incluso, varios lo hacen sin notarlo. El test es una parte natural e imprescindible de nuestros sistemas. Pero, ¿qué queremos decir al hablar de Testing?
Existen varias definiciones sobre la materia; he aquí algunas de ellas:

"Las pruebas de software son las investigaciones empíricas y técnicas cuyo objetivo es proporcionar información objetiva e independiente sobre la calidad del producto a la parte interesada o stakeholder. Es una actividad más en el proceso de control de calidad". Wikipedia

"El testeo de software es el proceso de analizar un ítem de software para detectar las diferencias entre las condiciones existentes y requeridas (esto es, bugs) y evaluar las características del ítem mencionado". Realsearch group.

"El testing es una actividad realizada con el fin de evaluar la calidad del producto y mejorarlo, a través de la identificación de sus defectos y problemas. Consiste de una verificación dinámica del comportamiento de un programa en un set finito de casos de

prueba, seleccionados de manera adecuada desde el dominio de ejecución y contra el comportamiento esperado". IEEE Software Engineering Body of Knowledge, 2004.

Seguramente, usted como lector tendrá otra definición del término o una combinación de las tres mencionadas. Esta elección es libre, pero es importante desarrollar tests que verifiquen lo que se quiere verificar, y siempre respetando buenas prácticas.
Otra definición que define de forma general y precisa el testing es la que enunció nuestro ya mencionado Edsger W. Dijkstra, haciéndose públicamente conocida en [9]:

"El testing revela la presencia de errores, pero no su ausencia"

Tenemos entonces que el testing debe tener lugar en cada etapa del proceso de software, con el objetivo de encontrar defectos, y siendo un elemento de un tema más amplio que se conoce como Verificación y Validación (V&V). Este proceso de V&V comienza con revisiones de los requerimientos y continúa con revisiones del diseño e inspecciones de código hasta la prueba del producto. Si bien verificación y validación suelen confundirse, no son lo mismo. La validación responde a la pregunta: *¿Estamos construyendo el producto correcto?*; por su parte, la verificación responde a: *¿Estamos construyendo el producto correctamente?*. Esto nos dice que la verificación implica comprobar que el software está de acuerdo con su especificación, comprobando que satisface tanto los requerimientos funcionales como los no funcionales. La validación, sin embargo, es un proceso más general, cuyo objetivo es asegurar que el software satisface las expectativas del cliente.
Antes de continuar, definiremos algunos conceptos más relacionados con las pruebas de software:

- Un *verdict* (veredicto) es el que determina si una prueba "pasó" o no, o, en otras palabras, si tuvo éxito o no.
- Un *failure* (fracaso) es un comportamiento no deseado. Generalmente, se observan durante la ejecución de sistema a testear.
- Un *fault* (falla) es la causa del *failure*. Es un error en el software, generalmente causado por un error humano en la especificación, diseño, o código. La ejecución de los *faults* en el software produce los *failures*. Una vez que observamos un *failure*, podemos investigar para encontrar el *fault* que lo produjo y corregirlo
- *Testing* es entonces la actividad de ejecutar un sistema para detectar *failures*.

Que un software ha fallado significa que no hace lo que especifican los requerimientos, y esto puede deberse a:

- Una especificación errónea
- Requerimientos imposibles con las estructuras previas
- Defectos en el diseño del programa
- Defectos en el diseño del sistema
- Defectos en el código.

Hay distintos tipos de testing. Algunos de ellos se reflejan en la figura 3.1.
Las siglas SUT describen el sistema bajo prueba (*System Under Test*), y su eje va desde unidades pequeñas hacia el sistema completo, incluyendo: el Test de Unidad (*Unit Testing*), que requiere testear una única unidad a la vez; el Test de Composición (*Component Testing*), que testea cada componente/subsistema de forma separada; y el test de integridad (*Integration Testing*) busca asegurar que los distintos componentes funcionan correctamente juntos. El test del sistema (*System Testing*) se basa en testear el sistema como un todo. El MDT puede aplicarse en cualquiera de estos niveles.

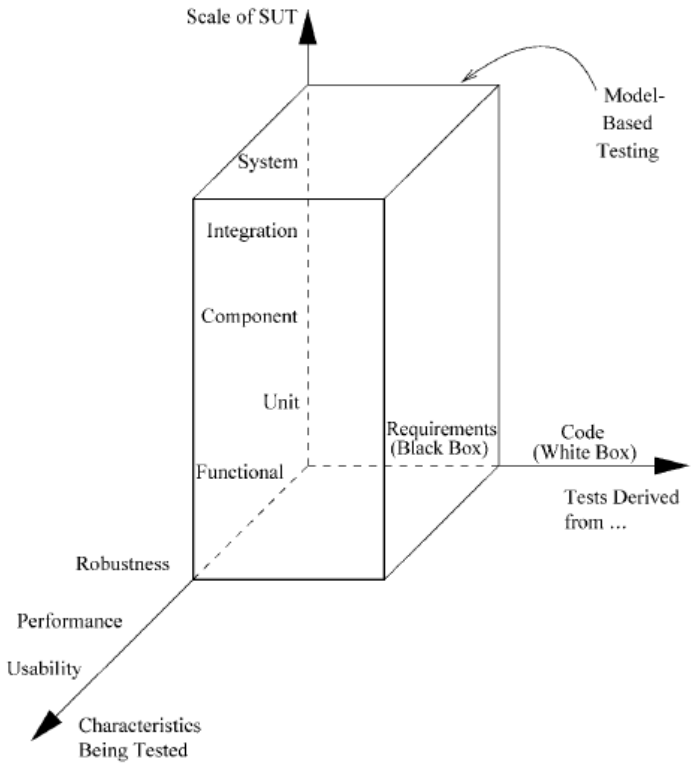


Figura 3.1 Representación 3D de los tipos de Testing

Otro eje nos muestra las diferentes *características* que podemos querer testear. El tipo más común de pruebas es el *Testing Funcional* (también conocido como *Testing de Comportamiento*), donde buscamos encontrar errores en la funcionalidad del sistema (por ejemplo, testear que para determinada entrada se produce una salida correcta). Por su parte, el *Testing de Robustez* busca encontrar errores en el sistema bajo determinadas condiciones, como entradas inesperadas, aplicaciones dependientes o no disponibles, y errores de hardware o de la red. Por último, el *Test de Usabilidad* se enfoca en encontrar problemas de interfaz de usuario, que pueden hacer dificultar el uso del software o causar la malinterpretación de la salida.
El tercer eje muestra el tipo de información que utilizamos para diseñar los tests. El *Test de Caja Negra* requiere tratar al SUT como una caja negra, sin usar información sobre su estructura interna; en cambio, se diseñan los tests a partir de los requerimientos del sistema, que describen el comportamiento esperado de esa caja negra. Por otro lado, el *Test de Caja Blanca* se basa en el minucioso examen de los detalles procedimentales; se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles.

3.2 Testing Dirigido por Modelos

Las técnicas tradicionales de testing pueden no siempre ser las indicadas para adecuar exhaustivamente un testeo extensible de software crítico y complejo en un ambiente de desarrollo de software limitado en recursos y tiempo. El testing dirigido por modelos (MDT por su acrónimo en inglés *Model Driven Testing*), también conocido como Testing *basado* en modelos (MBT o *Model Based Testing*) es una técnica evolutiva para generar casos de prueba automáticamente a partir de un modelo de comportamiento de un sistema bajo prueba.

Definido por Wikipedia como el *"test de software, donde se derivan casos de prueba completos o parciales desde un modelo que describe algunos (si no todos) los aspectos del sistema bajo prueba SUT"*, el SUT puede ser algo tan simpe como un método o una clase, o tan complejo como un sistema completo o una solución que consiste en múltiples sistemas. Para testear, un modelo provee una descripción del comportamiento del SUT, que puede ser procesada para generar un conjunto de casos de prueba útiles para determinar si el SUT está acorde a sus requerimientos, representados estos en el modelo [12].
En los últimos años, MDT se ha convertido en una palabra de moda, siendo utilizado por el ambiente informático para describir una gran variedad de técnicas de generación de tests.
Según IBM [10], MDT es un enfoque nuevo y prometedor para la automatización del testeo del software. Este enfoque puede reducir significativamente el doloroso esfuerzo invertido en el ciclo de generación de pruebas. Se trata de una nueva tecnología y un conjunto de herramientas que permitirán a los desarrolladores de software y a los testers alcanzar un nivel de productividad altamente superior, manteniendo los estándares de calidad de software. Sobre este ciclo tedioso que se busca reducir, Brian Kernighan comenta:

"Debuggear es al menos dos veces más difícil que escribir un programa en primer lugar. Por lo que si tu código es tan inteligente como potencialmente puedes hacerlo, entonces por definición no eres lo suficientemente inteligente como para debuggearlo".

En base a la clasificación de los tests que desarrollamos en 3.1, el propósito principal del MDT es el de generar tests funcionales, aunque también puede ser usado para algunos tests de robustez, como es el testeo del sistema bajo entradas inválidas.
Precisamente, el objetivo de MDT es permitir que, en vez de realizar la tediosa tarea de escribir manualmente cientos de casos de prueba, el diseñador de tests debe tan sólo escribir un modelo abstracto del SUT, para que luego la herramienta de MDT genere un set de casos de prueba a partir del mismo. Además de reducir el tiempo, existe la ventaja de que uno puede generar una variedad de casos de prueba a partir del mismo modelo simplemente usando distintos criterios de selección de ellos.
El proceso de MDT puede ser dividido en cinco pasos generales, como se muestra en la figura 3.2:

1) **Modelar el SUT y/o su entorno:** debemos escribir un modelo abstracto (más pequeño y simple que el mismo SUT) del sistema que queremos testear, enfocado sólo en los aspectos clave que queremos probar y omitiendo muchos de los detalles del SUT. Para ello se cuenta con notaciones de modelado que nos ayudan a escribirlo, y con herramientas que permiten comprobar si el modelo generado es consistente y posee el comportamiento deseado (la mayoría de ellas automáticas, con chequeos de tipos y análisis estático).
2) **Generar casos de prueba abstractos a partir del modelo:** debemos seleccionar un criterio de selección de tests, para filtrar los que queramos generar desde el modelo, pues estos suelen ser infinitos. Al surgir del modelo, que utiliza una vista simplificada del SUT, estos tests carecen de los detalles necesarios por el SUT y no son directamente ejecutables.
3) **Concretizar los casos de prueba abstractos para hacerlos ejecutables:** puede ser realizado por una herramienta de transformación, que utiliza varias plantillas y mapeos para trasladar cada caso de prueba abstracto a un script ejecutable; o escribiendo un código adaptador que envuelva el SUT e implemente cada operación abstracta en términos de las facilidades de más bajo nivel del SUT. De

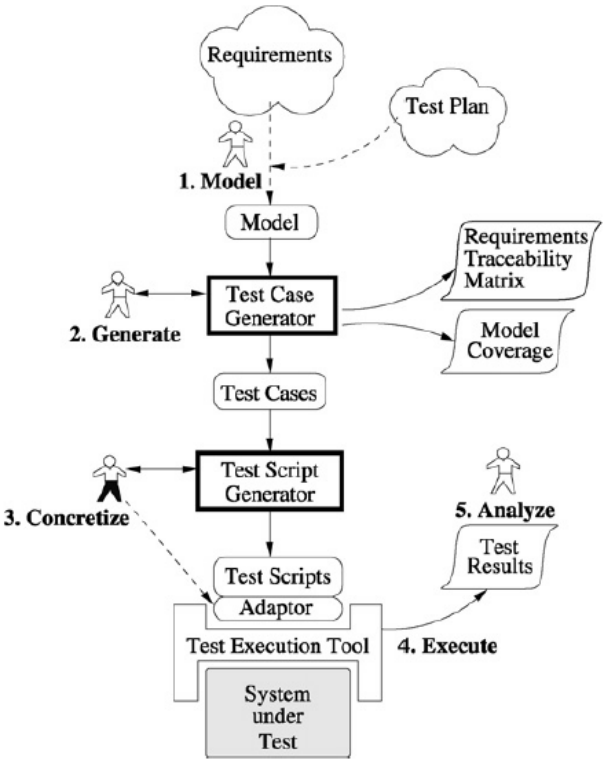


Figura 3.2 El proceso de MDT (Las herramientas de testing se encuentran en las cajas con borde marcado)

cualquiera de las dos maneras, el objetivo de este paso es eliminar la brecha entre los tests abstractos y el SUT concreto, adicionando detalles de bajo nivel del SUT que no hayan sido mencionados en el modelo abstracto.

4) **Ejecutar los casos de prueba en el SUT y asignar veredictos:** estos pueden ser ejecutados mientras van siendo producidos, de manera que la herramienta MDT maneje el proceso de ejecución y grabe los resultados; o podemos ejecutarlos con nuestra herramienta existente de ejecución de tests.
5) **Analizar los resultados de los casos de prueba y realizar cambios adecuados:** por cada test que reporta fracaso, debemos determinar el error que causó ese fracaso (similar al proceso original de pruebas). Usualmente, al fallar un test, se debe a una falla del SUT o una falla del caso de prueba mismo. Dado que utilizamos en este caso MDT, una falla en el caso de prueba debe deberse a una falla en el código adaptador o en el modelo (e incluso quizás en el documento de requerimientos); por lo que aquí obtenemos nuevamente un feedback sobre la correctitud del modelo.

Claramente, los pasos 4 y 5 son parte de cualquier proceso de testing, incluso el manual. El paso 3 es similar a la fase adaptativa del testeo basado en palabras clave, donde el significado de cada palabra clave es definido. Los primeros dos pasos distinguen al MDT de cualquier otro tipo de testing.

Son claros entonces los beneficios del MDT. Sus herramientas reducen el tiempo de testeo total significativamente, soportando la reutilización de muchas funciones de testeo comunes. Además, mejoran la calidad de las pruebas y de la complejidad, ofreciendo un enfoque sistemático para la generación de entornos de prueba. La cobertura es altamente

funcional, exponiendo un mayor número de defectos de manera temprana en el ciclo de desarrollo del software. Utilizando estas herramientas, el testing se vuelve menos monótono, pues las mismas automatizan las tareas repetitivas y no creativas asociadas con el proceso. Por otro lado, se reducen drásticamente los costos de mantenimiento del testing, pues los cambios de implementación son capturados en el modelo; los desarrolladores sólo deben regenerar el caso de prueba para afectar a todos los tests. Se logra además una buena comunicación del equipo, poseyendo una vista unificada, clara y no ambigua del sistema bajo prueba.

3.3 Perfiles de Testing Aplicados a Modelos de Software

A medida que el software se vuelve más grande y complejo, la necesidad de la calidad y confianza en un sistema de software crece con él. Los métodos basados en modelos han sido un desarrollo importante que ayudó a las organizaciones a crear software con alta calidad. En ese contexto y aportando a la causa, surge el Perfil de Pruebas de UML (desde ahora, ***UTP*** o *UML Testing Profile*), basado en el lenguaje de modelado más popular, el cual, antes de su aparición, no poseía soporte suficiente para las actividades relacionadas con el testing. Fue desarrollado por un consorcio de testers, vendedores de UML y usuarios dedicados a convertir a UML en algo aplicable para el testeo de software, junto con empresas inscriptas (entre ellas, IBM, Motorola, Ericsson).

Al estar definido sobre UML 2.0, permite que la generación y definición de tests esté basada en aspectos estructurales (estáticos) y de comportamiento (dinámicos) de los modelos UML, y la buena relación con otras tecnologías existentes de testing de caja negra [14], heredando además la característica de la división de capas, con una arquitectura de metamodelo de 4 niveles.

Provee extensiones de UML para soportar el diseño, la visualización, la especificación, en análisis, la construcción y la documentación de los artefactos involucrados en el testing [13]. Es independiente de la implementación de cualquier lenguaje o tecnología, y puede ser aplicado en una variedad de dominios de desarrollo. Las características principales de UTP incluyen:

- **Minimalidad:** siempre que sea posible, utilizar construcciones proveídas directamente por UML, para reducir el overhead necesario para aprender y usar el perfil.
- **Claridad:** proveer una clara separación de los conceptos de testing, ayudando a los ingenieros de calidad a desarrollar su trabajo y referirse a conceptos complejos de la forma más fácil posible.
- **Extensibilidad:** crear un perfil genérico que pueda ser extendido y aplicado a una variedad de dominios y tecnologías.

Al utilizar MOF, permite crear meta-metamodelos que se pueden transformar a texto. Además, define el lenguaje para modelar pruebas a través de:

- **Arquitectura**: SUT, Test Component, Test Context, Test Configuration, Test Control, Arbiter, Scheduler.
- **Comportamiento**: Test Objective, Test case, Defaults, Validation action, Verdict.
- **Datos**: Wildcards, Data pool, Data Partition, Data selector, Coding rules.
- **Tiempo**: Timer, Time zone.

En la figura 3.3, podemos observar el metamodelo de la arquitectura de pruebas de UTP. La misma se compone por una serie de conceptos que especifican los aspectos estructurales de una situación de pruebas [15]:

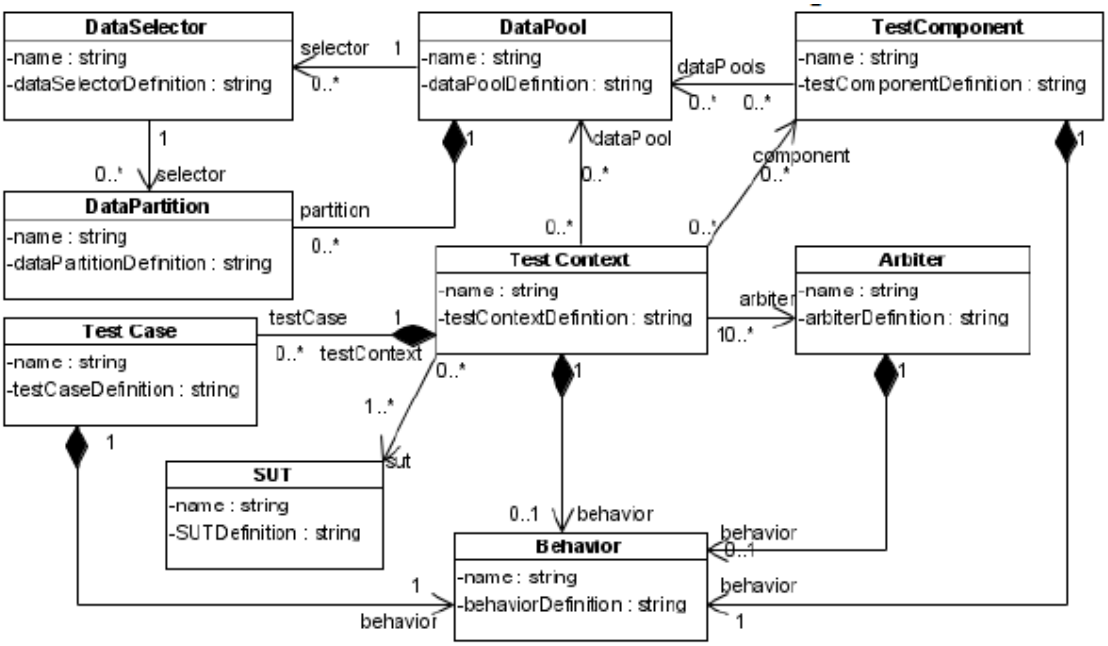


Figura 3.3 Metamodelo de UTP

- ***TestContext*** (Contexto de pruebas): clase estereotipada que contiene los casos de prueba (como operaciones), y cuya estructura compuesta define la configuración de las pruebas.
- ***SUT:*** sistema bajo pruebas.
- ***TestComponents*** (Componentes de pruebas): objetos dentro del sistema que pueden comunicarse con el SUT u otros componentes para realizar comportamientos de prueba
- ***Arbiter*** (Árbitro): provee una manera de evaluar los resultados de tests derivados por diferentes objetos en el sistema de pruebas, con el objetivo de enunciar un veredicto total para un caso o un contexto de pruebas.

El ***Test case behaviour*** (comportamiento del caso de prueba): especifica las acciones y operaciones requeridas para evaluar el ***Test Objective*** (objetivo del test), que describe qué debe testearse. El caso de prueba es el concepto más importante en un modelo de pruebas. En UTP, se describe el *test case behaviour* utilizando el concepto de *Behaviour*, y puede mostrarse utilizando diagramas de interacción UML o máquinas de estados. Los conceptos de UTP utilizados para describir el *test case behaviour* pueden resumirse en los siguientes:

- ***Test Case*** (Caso de prueba): especificación técnica completa de cómo un set de componentes de prueba interactúan con un SUT para realizar un *TestObjective* y devolver un veredicto. La implementación de los *TestCases* está especificada por un *testBehaviour*, que también determina su semántica
- ***Verdict*** (Veredicto): enumeración predefinida que especifica posibles resultados de tests, como *pass* (pasó, éxito), *inconclusive* (resultado incierto), *fail* (falló), y *error*.
- ***Validation action*** (Acción de Validación): realizada por un *TestComponent* para indicar que el árbitro es informado del resultado del test del componente de pruebas.

- ***DataPool*** (Pool de datos): contiene un set de valores que pueden asociarse a un contexto de pruebas particular y sus casos de prueba.

En síntesis, ***UTP*** es una especificación formal para la definición de un modelo de pruebas, que:

- Permite modelar dominios de pruebas estructurales y de comportamiento.
- Utiliza el concepto de prueba de caja negra.
- Incluye las reglas para transformar los modelos a código ejecutable.

3.4 JUnit

En los últimos años, se han desarrollado un conjunto de frameworks denominados XUnit que facilitan la elaboración de pruebas unitarias en distintos lenguajes. Esta familia se compone, entre otros, de las siguientes herramientas [1]:

- SUnit, para Smalltalk.
- JUnit, JTest, JExample, para Java.
- CPPUnit, para C++.
- NUnit, para .NET y Mono.
- DUnit, para Borland Delphi.
- PHPUnit, para aplicaciones realizadas en PHP.

Estos frameworks son muy útiles para controlar las pruebas de regresión, validando que el nuevo código cumple con los requisitos anteriores, sin alterar su funcionalidad tras la nueva modificación.

JUnit fue creado por Erich Gamma y Kent Beck a mediados de los años ’90, con el objetivo de proveer un framework fácil de utilizar que alentara el testing de unidad, y basado en SUnit. Desde allí que se ha vuelto muy popular, pudiendo hoy en día considerarse la cabeza de la familia de XUnit mencionada [16]. Actualmente se encuentra en su versión 4.12. En la imagen 3.4 podemos observar el diagrama de clases del framework JUnit, con el paquete de Java junit.framework.

En pocas palabras, JUnit trabaja de la siguiente manera [17]:

La ejecución completa de un test posee dos fases: la primera es la configuración, y la segunda es la ejecución del caso de prueba. Esta última puede ser reforzada con un análisis y una presentación visual de los resultados a un usuario.

Durante la configuración, se construye una jerarquía de casos de prueba, en donde un objeto representa un caso de prueba. Cualquier objeto es una instancia de una subclase de *TestCase.* Los programadores implementan tales subclases, y todo método en ellas que empiece con la palabra clave *“test”* referencia un objeto de la jerarquía. Estos métodos *“test”* contienen el código de prueba real, y las clases que los contienen son siempre objetos hoja de la jerarquía; los nodos intermedios y el nodo raíz son instancias de *TestSuite*, quien provee la funcionalidad de agrupamiento necesaria para la estructura de árbol. Tanto *TestCase* como *TestSuite* implementan la interface *Test*, para permitir un tratamiento homogéneo de cada nodo del árbol.

La verdadera ejecución del test comienza al invocar el *“run”* en el nodo raíz de la jerarquía de pruebas. Entonces, el árbol se recorre en profundidad primeramente, determinando el orden de los nodos en base al orden en que fueron agregados al árbol. En todo nodo hoja (es decir, en todo objeto *TestCase*), el *“run”* será llamado eventualmente; en otras palabras, se correrán tantos casos de prueba como subclases de la clase *TestCase* haya. El resultado de cada *TestCase* se graba en un objeto recolector, que es pasado de prueba en prueba, mientras ocurre la ejecución, a lo largo del árbol. Este objeto se denomina *TestResult.*

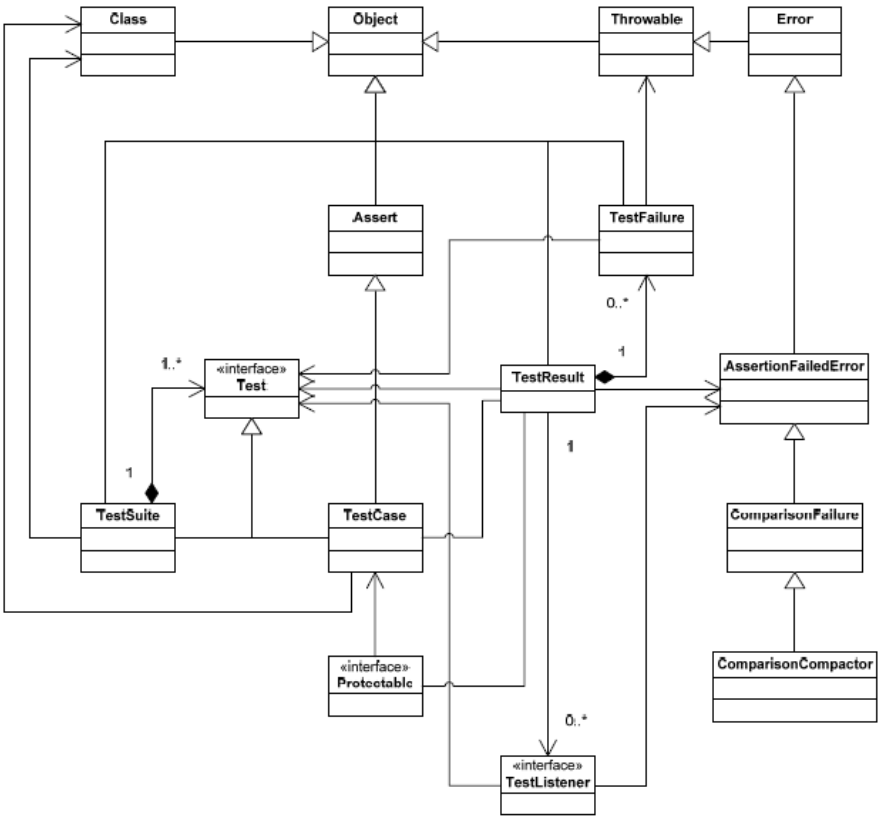


Figura 3.4 Diagrama de clases de JUnit

JUnit distingue entre *failures* y *errors*, términos ya nombrados previamente. Los *Failures* son instancias de la *AssertionFailureError*, creadas por el código de test al fallar una prueba. Tanto un error como un *failure* ocurren cuando algo anduvo mal, manejados con excepciones en Java, y recolectados por el *TestResult*, que al almacenarlos, los envuelve en un objeto *TestFailure*.

Al invocar la ejecución de un caso de prueba por primera vez, este no lo ejecuta directamente, sino que delega ese trabajo al objeto *TestResult*, que lo recibe como un argumento y, tras preparar algunas cosas (como notificar a los *TestListeners* sobre el comienzo/fin de una prueba, utilizado esto por ejemplo por una UI para mostrarle el progreso al usuario), llama nuevamente al caso de prueba para la ejecución real del código de prueba.

En un caso de prueba, el código de test real se envuelve en llamadas a los llamados *setUp* y *tearDown*, que inicializan el objeto de prueba respectivamente. El framework encuentra luego el método de test a ejecutar por matcheo de nombre, utilizando reflexión, y luego lo ejecuta. Si el test tiene éxito, la ejecución correrá correctamente a través del código y volverá de la llamada al método sin problemas. Los programadores implementan códigos de prueba generando aserciones sobre el estado del programa bajo test, utilizando llamadas a métodos *"assert"* estáticos heredados de la clase *Assert.* Si una aserción falla, *TestResult* almacena la excepción lanzada.

Finalmente, y tras completar la ejecución de la prueba (o interrumpirse la misma), el cliente que inició la prueba puede inspeccionar los resultados recolectados por el objeto *TestResult,* y mostrárselos a un usuario o utilizarlos programáticamente de otra manera.

¿Por qué trabajar con JUnit, entonces? [1]

- Incrementa la calidad y estabilidad del software, permitiendo refactorizar el código de manera más continua y libre
- Es simple, incentivando el uso de pruebas
- Sus tests chequean sus propios resultados, con feedback inmediato
- Es de código abierto
- Sus tests pueden componerse como un árbol de conjuntos de tests, como mencionamos.

3.5 EasyMock

Un programa orientado a objetos comprende una web de objetos en colaboración que se envían mensajes entre ellos para lograr una tarea específica. En el testeo de unidades, el objetivo es testear una clase de forma aislada. Un objeto ***mock*** de un objeto real es un colaborador del objeto bajo prueba [18]. Los objetos *mock* son reemplazos livianos y controlables de los objetos reales; su utilización permite testear un objeto aisladamente. Algunos los definen como objetos que están *"pre-programados con expectativas que forman una especificación de las llamadas que se supone que recibirán".* Pueden lanzar una excepción si reciben una llamada que no esperaban, y se chequean durante la verificación para asegurar que poseen todas las llamadas que estaban esperando.
Astels [19] lista diversos usos que se le pueden asignar a un objeto mock. Algunos de los más importantes son:

- Utilizar mocks para clases no existentes, de forma de postergar la implementación
- Promover un diseño basado en la interfaz
- Refinar interfaces
- Alentar la composición sobre la herencia
- **Pruebas de unidad y verdadero aislamiento**

Se cuenta con varios frameworks de objetos mock de código abierto [20], pero aquí nos enfocaremos en el utilizado para Java, llamado ***EasyMock***. *EasyMock* [21], en su versión actual 3.3.1, es una librería que provee una API fácil de utilizar para generar objetos mock para interfaces determinadas. Utilizándolo, uno puede generar objetos mock de un objeto bajo prueba, y asociarlos con el código de dominio. Puede utilizarse junto a JUnit para generar casos de prueba más robustos, completos y confiables.
En *EasyMock*, los objetos mock se generan dinámicamente, en tiempo de ejecución, con un estilo de uso de *"record and replay"*, lo que involucra grabar (el objeto mock no se comporta como tal, sino que simplemente almacena llamadas a métodos esperadas y valores de retorno), y el modo de volver a ejecutar (donde se comporta como un objeto mock, chequeando si las llamadas a los métodos esperadas son realmente hechas). Comúnmente, el patrón de uso de estos objetos es:

1) Crear un objeto mock para la interfaz deseada.
2) Asociar el objeto mock con el objeto del dominio como un parámetro pasado al constructor o a un método de seteo relevante.
3) Especificar las llamadas esperadas y sus valores de retorno correspondientes (fase *record*).
4) Activar el modo *replay* del objeto mock.
5) Ejercitar la funcionalidad del código de dominio (la unidad bajo test).
6) Verificar el resultado esperado del código de dominio afirmando alguna postcondición.
7) Finalmente, verificar que el objeto mock fue utilizado correctamente.

Es bueno entender la importancia de los objetos mock en el testing, teniendo tres ventajas principales:

1) Al testear unidades de forma aislada, suele ocurrir que tal unidad debe interactuar con uno o más colaboradores, debiendo reemplazar estos con los apropiados objetos mock.
2) Antes de pasar a la implementación de una unidad, debemos preparar la especificación de las interfaces para ella. Para simular el comportamiento en esta etapa, contamos nuevamente con los objetos mock.
3) El diseñador de cada unidad (clase/interfaz) debe esforzarse por usar interfaces en vez de clases concretas para los tipos de parámetros y valores de retorno de los métodos de la unidad; las variables miembro definidas en la unidad también deben especificar interfaces. Con el uso de objetos mock reemplazando a los objetos reales, las unidades pueden ser compiladas con éxito sin las verdaderas clases, además de ser ejecutadas y testeadas sin tener que esperar por los objetos colaboradores reales.

3.5.2 Mockito

El framework EasyMock ha sufrido diversas modificaciones a lo largo de los años. Uno de los patrones más comunes en este tipo de frameworks con objetos *mock* es el famoso *expect-run-verify*, el cual se vuelve molesto con el tiempo, pues:

- Ensucia el código, estando obligado el tester a programar los expect no por necesitar testearlos, sino porque en caso contrario el *mock* se quejará.
- No es natural en el ciclo de testeo: las llamadas esperadas a un mock son en si aserciones, debiendo comprobarse tras la ejecución; hacerlo al revés no resulta intuitivo.
- Los tests quedan frágiles: los cambios en el código que requieran nueva interacción entre objetos suelen romper los tests, que no cuentan con los expect necesarios
- Es difícil identificar errores: estos suelen ocurrir dentro del código de los expect, no permitiendo descifrar cuál fue el lugar específico del error.

En este contexto surge ***Mockito*** [53], una librería Java para la creación de objetos *mock* utilizados en el testeo unitario MDD, basado en *EasyMock*. Fue creado con el objetivo de simplificar y solucionar algunos de los temas antes mencionados. *Mockito* y *EaskyMock* pueden realizar exactamente lo mismo, pero *Mockito* posee una API más natural y práctica. Sus características principales son:

- Se pueden crear mocks de interfaces y clases concretas
- Verificación de invocaciones
- Limpieza del stack trace, con errores ocurridos en los asserts que se hagan y no dentro del método bajo prueba, como en su predecesor.
- Claridad
- Existe un solo tipo de mock, a diferencia de los diversos tipos de EasyMock, entre ellos "*nice*", "*default*", "*strict*". Aquí todos los mocks son *nice*, es decir, devuelven valores al invocarse (con las colecciones vacías y no nulas)
- Verificación explícita

3.6 Resumen

Como se vio en el capítulo, se cuenta con varias tipos de técnicas de V&V, tales como la verificación formal de programas y modelos, chequeo de modelos, pruebas de caja blanca y caja negra, pruebas funcionales, testing. Dentro de estas, se introdujo la nueva técnica de prueba de caja negra MDT.

El MDT se basa en generaciones previas de herramientas de testeo, tales como las mencionadas de Java, *JUnit* y *EasyMock,* que permiten refinar las pruebas. MDT se basa

en la abstracción, y al igual que el enfoque de scripts de prueba automáticos, produce scripts de pruebas ejecutables.
No es recomendable ni práctico reutilizar nuestros modelos de desarrollo para MDT. Usualmente, es necesaria una personalización o construcción de modelos de prueba específicos. Pero aun incluyendo el trabajo de desarrollar el modelo, MDT es efectivo en cuanto al costo, pues automatiza la generación de scripts de prueba, y facilita el manejo de la evolución de los requerimientos, modificando el modelo y regenerando los tests, en vez de manteniendo el *test suite* mismo. Esto puede reducir notablemente el costo de mantenimiento de las pruebas.
Antes de adoptar MDT, es deseable poseer un proceso de testing razonablemente maduro, y las investigaciones han probado que es una manera efectiva de detectar errores en los SUTs y una técnica efectiva en cuanto al costo.
Una limitación del MDT es que requiere diferentes habilidades de los testers: habilidades de modelado para los diseñadores de pruebas, y habilidades de programación para los implementadores del adaptador de pruebas. Esto podría implicar costos de entrenamiento y una curva inicial de aprendizaje.

4. *Lenguajes Formales*

En los capítulos previos, detallamos la importancia del uso de modelos para la comprensión del problema de manera no ambigua, incompleta o contradictoria, especificando un enunciado preciso, abstrayendo el problema (siendo este independiente de la perspectiva del observador), facilitando la comunicación con otros desarrolladores de software, y analizando las posibilidades de reutilización. En este capítulo, introduciremos los lenguajes de modelado formales utilizados para construir estos modelos; específicamente, a los lenguajes formales OCL y Alloy.

4.1 Introducción

Como Daniel Jackson menciona en [23], todos queremos un buen nivel de abstracción: los usuarios principiantes desean programas cuyas abstracciones sean simples y fáciles de comprender; los expertos, por otro lado, desean abstracciones robustas y lo suficientemente generales como para ser combinadas en nuevas maneras. De ahí que el objetivo del desarrollo de software es el ***diseño de abstracciones***, definiendo a una abstracción como una estructura pura y simple – una idea reducida a su forma esencial.
El proceso de desarrollo de software debería ser simple y secuencial. Primero, uno diseña las abstracciones; luego, realiza la transición al código (interfaces y módulos, algoritmos). Desafortunadamente, este enfoque raramente funciona. El problema, como alguna vez lo llamó Bertrand Meyer, es el "pensamiento deseoso" (*wishful thinking*). Un conjunto de abstracciones que parecen ser simples y robustas, al ser implementadas, resultan ser incoherentes e incluso inconsistentes.
Una primera solución para el problema del *wishful thinking* surgió con la programación extrema, eliminando al diseño como una fase separada en conjunto, y en donde el diseño del software evoluciona junto con el código. Pero el código es tan solo un medio pobre para explorar las abstracciones, expresándolas de manera torpe, y provocando que un simple cambio global genere ediciones extensivas a lo largo de diversos archivos.
Otra alternativa es atacar el diseño de abstracciones de frente, con una notación elegida para facilitar la expresión y exploración; haciendo la notación precisa y no ambigua, el riesgo del *wishful thinking* se reduce. Este enfoque, conocido como ***especificación formal***, ha tenido un gran éxito. ¿Por qué entonces no es usada ampliamente? Según varios autores, hay dos obstáculos principales que limitan su atracción:

- Sus notaciones poseen una sintaxis matemática que intimida a los diseñadores de software (siendo en realidad más simples que la mayoría de los lenguajes de programación)
- La falta de soporte de herramientas más allá del chequeo de tipos: si bien los probadores de teoremas han avanzado en los últimos 20 años, todavía demandan más investigación y esfuerzo.

Para construir entonces abstracciones consistentes y robustas, expresadas en modelos, requerimos un lenguaje de modelado, el cual puede ser:

- Informal
- Semi-formal (UML): semántica de modelos/diagramas generalmente imprecisa. Suele ser sólo sintaxis.
- Formal (OCL, Z, Alloy): un lenguaje es formal si su sintaxis y su semántica están definidos formalmente (matemáticamente) [25].
    - Remueven las ambigüedades e introducen precisión

- Información estructurada en un nivel de abstracción apropiado
- Permiten la verificación de las propiedades de diseño
- Soportados por herramientas y sistemas
- Utilizar matemáticas y métodos formales puede parecer caro en el corto plazo, pero en el largo, se justifica.

Un lenguaje de modelado (también llamado de especificación), es un lenguaje cuya función es construir modelos de los sistemas que se desea elaborar. A diferencia de los lenguajes de programación, que son lenguajes interpretables o traducibles por una computadora hacia una representación ejecutable, los lenguajes de especificación no son utilizados para implementar el sistema, sino para especificarlo, conceptualizarlo o incluso validarlo, aunque suelen ser legibles para un programa de computadora, que puede asistir en el proceso de validación. Existen una gran variedad de lenguajes de especificación, que difieren en la vista del sistema (estática, dinámica, funcional, orientada a objetos), en el grado de abstracción, en la formalidad, entre otras; algunos de los más representativos son:

- **OCL**, el cual es un lenguaje para la descripción formal de expresiones en los modelos UML.
- **Alloy**, lenguaje de especificaciones que utiliza la lógica de primer orden y se basa en el uso de relaciones.
- **Z,** lenguaje formal expresivo basado en lógica de primer orden.
- **Autómatas** es un formalismo utilizado para modelar sistemas discretos en general.
- **B**, lenguaje de descripción formal basado en la lógica de predicados.
- **Cálculo Pi**, lenguaje de especificación para sistemas distribuidos y paralelos.
- **CSP**, lenguaje formal basado en el álgebra de procesos.
- **Estelle**, lenguaje formal basado en autómatas de estado finito para la especificación de sistemas distribuidos.

En el modelado orientado a objetos, y de aquí surge la motivación principal de esta tesina, un modelo gráfico como el de clases no posee la suficiente información como para lograr una especificación precisa y no ambigua. Existen ciertas características adicionales que necesitan ser descriptas sobre los objetos del modelo. Muchas veces esto se hace, pero en un lenguaje natural, con la desventaja de no ser procesable por una computadora, y llevando a menudo a ambigüedades. De aquí que surge este tipo de lenguajes, todos ellos aplicables a problemas específicos dentro de distintas ramas de la informática, ingeniería eléctrica y ramas afines. A continuación, nos enfocaremos en los lenguajes formales, específicamente en algunos de ellos.

4.2 Z

Z (*"zet", aunque pronunciado "Zed"*) fue desarrollado en la Universidad de Oxford, en los años '80. Ha sido muy influyente en la educación y la investigación, aplicado exitosamente en varios proyectos largos, notablemente por la Universidad de Oxford e IBM en el sistema CICS [24], en una serie de proyectos de sistemas críticos, y para la verificación de seguridad del banco *NatWest*. Su fundación, en base a una semántica simple y limpia, ha sido una inspiración para el diseño de Alloy, del que hablaremos luego.
El lenguaje se basa en una lógica de predicados de primer orden, teoría axiomática y lambda-cálculo. Todas sus expresiones son tipadas. Posee un kit de herramientas matemáticas, que representa un catálogo estandarizado, con una librería de definiciones y tipos de datos abstractos (sets, listas, bags). Se basa en PL/1 [26], al igual que otros

lenguajes, como VDM y B. Permite el modelado estructurado de un sistema, tanto estático como dinámico:

- Modelado/especificación de los datos de un sistema
- Descripción funcional de un sistema (transiciones de estados)

Z es básicamente una lógica, aumentada con algunas construcciones sintácticas que facilitan la descripción de abstracciones de software. En Alloy, estas construcciones son la firma (*signature*), para declaración de paquetes, y hechos/predicados/funciones (*facts, predicates, functions*) para las restricciones. En Z, la misma construcción – el *schema*, es usada tanto para las declaraciones de paquetes como para sus restricciones. El lenguaje de schemas, llamado *schema calculus*, es lo suficientemente rico para soportar una gran variedad de idiomas. Una especificación en Z es creada como una serie de declaraciones de schemas, que son bloques de construcción básicos que permiten modularidad.
En la siguiente figura, puede observarse un *schema* tipo:

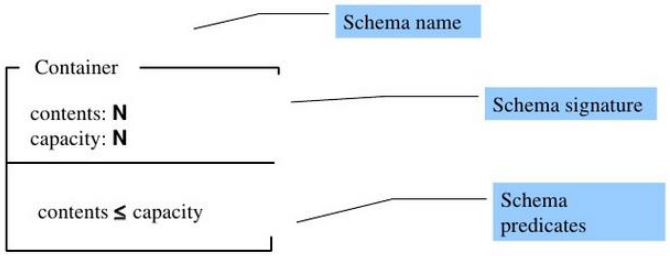


Figura 4.1 Schema de Z

Los predicados de un *schema* son siempre verdaderos, y pueden referirse sólo a elementos en la *signature*.

4.3 UML/OCL

OCL (*Object Constraint Language, o lenguaje de restricciones de objetos),* es un lenguaje formal notacional, subconjunto del UML 2.0 estándar (adoptado en 2003 por el grupo OMG como parte de tal), que permite a los desarrolladores de software escribir restricciones sobre modelos de objetos [22] (pre y pos condiciones, invariantes, reglas de derivación, guardias y restricciones sobre operaciones). Estas restricciones son particularmente útiles, permitiendo a los desarrolladores crear un amplio conjunto de reglas que rigen el aspecto de un objeto individual; suelen especificar condiciones invariantes que deben cumplirse tanto en el sistema modelado como en las queries sobre los objetos descriptos en el mismo.
Inicialmente desarrollado por IBM, es un lenguaje formal para expresar restricciones libres de efectos colaterales. Cualquier usuario de UML (que hoy en día, es la mayoría) puede usarlo para especificar restricciones y otras expresiones incluidas en su modelo. Los lenguajes formales tradicionales se usaban por personas con conocimientos matemáticos, dificultando su uso para muchas empresas y modeladores de sistemas. OCL ha sido desarrollado para ampliar el uso de este tipo de lenguajes a toda la comunidad.
Algunas de las características principales de OCL son:

- Es un lenguaje de expresiones puro: un estado del sistema nunca cambiará por una expresión OCL; incluso, una expresión OCL podría usarse para describir tal cambio de estado (como por ejemplo, una postcondición). Al evaluar una expresión OCL, esta simplemente retorna un valor

- OCL es un lenguaje de modelos y no de programación: no puedo escribir un programa lógico o un flujo de control en él; todos los aspectos de implementación están fuera de él.
- OCL es un lenguaje formal donde todos los constructores poseen un significado formal definido: su especificación es parte de la especificación de UML.
- OCL es un lenguaje tipado, por lo que cada una de sus expresiones posee un tipo. Para estar bien formada, una expresión OCL debe cumplir las reglas de conformación del lenguaje, por ejemplo, no está permitido comparar un entero con un String.

¿Por qué utilizar OCL? Pues, como se especifica en el objetivo de esta tesina, UML no es suficiente! Un diagrama UML, tal como lo es un diagrama de clases, no está lo suficientemente refinado como para proveer todos los aspectos relevantes de una especificación.
¿Dónde usar OCL? OCL puede ser usado para un número distinto de propósitos:

- Como un lenguaje de queries
- Para especificar invariantes en las clases y tipos del modelo de clases
- Para especificar invariantes en estereotipos
- Para describir guardas
- Para especificar restricciones sobre las operaciones
- Para especificar reglas de derivación para los atributos de cualquier expresión sobre un modelo UML

¿Cuáles son los requerimientos de OCL? Según el manual del lenguaje, estos son:

1) OCL debe ser capaz de expresar información extra (necesaria) en los modelos y otros artefactos utilizados en el desarrollo orientado a objetos
2) OCL debe ser un lenguaje preciso y no ambiguo que pueda ser leído y escrito fácilmente por todos los practicantes de la tecnología de objetos y sus clientes. Esto significa que el lenguaje debe ser entendible por gente que no sean matemáticos o científicos de la computación
3) OCL debe ser un lenguaje declarativo. Sus expresiones no pueden tener efectos laterales, siendo que el estado del sistema no debe cambiar en respuesta a una expresión OCL
4) OCL debe ser un lenguaje tipado

Veamos algunos ejemplos simples que nos ayuden a entender el lenguaje, definiendo restricciones en OCL. Para esto, nos basaremos primero en la figura 4.2, que posee un simple diagrama de clases UML, en dónde se especifica la relación padre-madre-hijo.

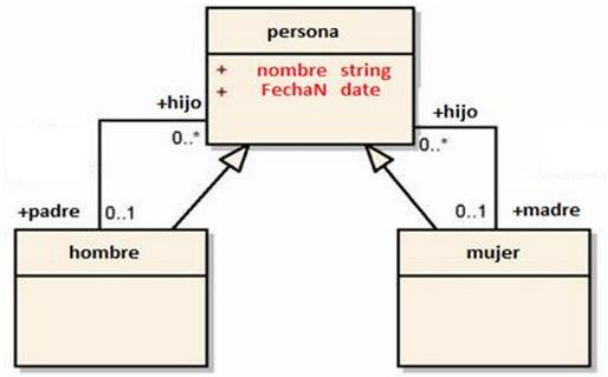


Figura 4.2 Diagrama de Clases 1

La restricción deseada es especificar el hecho de que los padres nacen antes que sus hijos. En OCL, tal restricción queda de la siguiente manera:

```
context Persona
inv: padre.FechaN < self.FechaN
    and madre.FechaN < self.FechaN
```

En la figura 4.3 contamos con otro diagrama de clases, correspondiente a una gestión de una flota de vehículos.

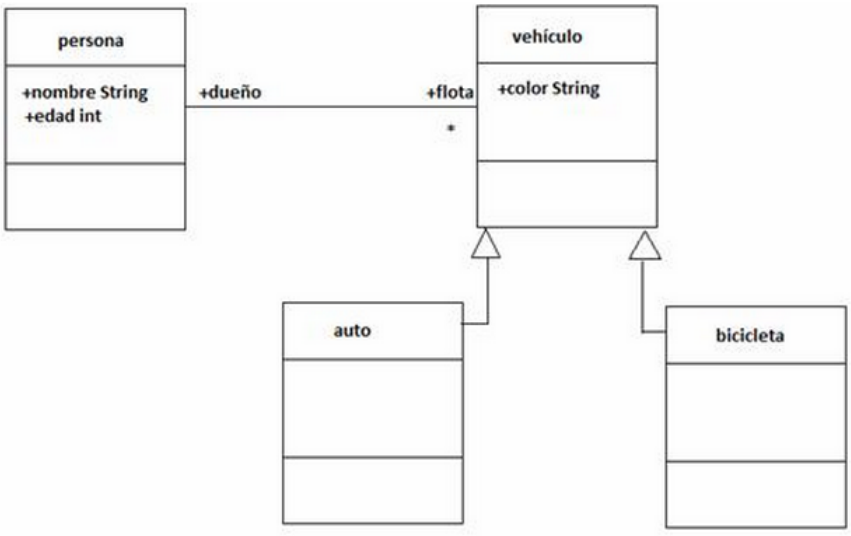


Figura 4.3 Diagrama de Clases 2

En este caso, se quiere restringir la cantidad de vehículos por dueño, la edad requerida para ser dueño de un auto, obligar a que todos los autos con dueño sean negros, contar la cantidad de vehículos negros de una persona, e impedir a un menor poseer un auto. En OCL, esto se especifica de la siguiente manera:

```
context Auto
    Inv: self.dueño.edad >= 18 (el dueño de un auto debe ser mayor de 18)
context Persona
      Inv: self.flota->size() <= 3 (nadie tiene más de tres vehículos)
context Persona
      Inv: self.flota->forall(v| v.color = 'negro' ) (todos los vehículos de una persona son negros)
context Persona                                    (la cantidad de vehículos negros de una
    inv: self.flota>select(v| v.color='negro')->size()<=3          persona no supera los 3)
context Person                                   (un menor no posee ningún auto)
    inv: edad<18 implies self.flota -> forAll(v | not v.oclIsKindOf(Auto))
```

Hasta aquí, consideramos sólo invariantes de clase, pero OCL también puede especificar operaciones, por ejemplo:

```
context Persona::setEdad(nuevaEdad:int)
pre:    nuevaEdad >=0
post:   self.edad=nuevaEdad
```

```
context Person::birthday()
post: self.age = self.age@pre + 1
```

Al llamarse *setEdad(valor)* con un valor no negativo de argumento, el mismo se transforma en el nuevo valor del atributo de edad. Llamar a *birthday()* incrementa la edad de una persona.

Vemos entonces algunas características más de OCL:

- Es utilizado para especificar invariantes de los objetos, y pre y post condiciones de las operaciones, haciendo a los diagramas de clase de UML más precisos.
- Las expresiones OCL utilizan un vocabulario UML
- OCL permite navegar dentro de un diagrama de clases UML
- El "contexto" especifica de qué elementos estamos hablando; "self" indica el objeto actual, y "result" el valor a retornar.
- OCL puede hablar sobre colecciones y realizar operaciones sobre ellos (select, forAll, iterate)

A pesar de las ventajas que hemos nombrado de OCL, el mismo posee los siguientes defectos con respecto a sus requerimientos [31]:

- OCL parece estar más cerca de ser un lenguaje de implementación que de ser un lenguaje conceptual, pues utiliza operaciones en restricciones. Esto es problemático con respecto a dos cuestiones:
    - Primero, suma al sabor operacional de OCL. Una operación puede caer en un loop infinito o ser indefinida. Esto agrega complejidad innecesaria al lenguaje y genera cuestionamientos con respecto a su precisión, tales como: ¿Qué significa para un modelo satisfacer una restricción indefinida? ¿Cómo sabemos que una restricción es indefinida?
    - Segundo, consideremos una operación aplicada a una colección de objetos que son instancias de una clase determinada. Un subset de estos objetos pueden ser también instancias de una subclase de tal clase que redefina la operación. No está claro qué operación se aplica a cada objeto en la colección en tal caso. Incluso, en caso de considerar aplicar la operación redefinida en la subclase, esto implica que el significado de una restricción puede cambiar mientras el modelo evoluciona y se agregan subclases. Esto es algo indeseado, dado que la expresión de las restricciones de un sistema puede tener que ser revisada mientras el modelo evoluciona.
- El sistema de tipos de OCL es innecesariamente complicado. Conceptualmente, una clase es un set de objetos, y una subclase un subset de esos objetos, por lo que si una clase hereda de dos clases, entonces estas deben ser sets no-disjuntos. Sin embargo, en OCL es posible tener una clase que herede de dos clases aparentemente disjuntas.
- OCL utiliza dos símbolos distintos para navegar a través de los sets y los escalares (-> y ., respectivamente). Esta falta de uniformidad agrega complejidad innecesaria y hace menos sucintas a sus expresiones
- Las clases no se tratan simplemente como colecciones de objetos, lo que impide que usemos operadores de sets para manipularlas directamente, lo que incrementa el uso de cuantificadores. Esto genera la necesidad de frecuentes coerciones (*oclIsKindOf*).
- OCL no es un lenguaje *stand-alone*: un modelo siempre necesita un diagrama de clases UML que lo acompañe, cuando existen muchas ventajas de tener un lenguaje de restricciones *stand-alone*, entre ellas:

- La chance de elegir de manera más flexible entre expresar modelos gráficamente o textualmente
- Mejor integración entre el lenguaje de modelado y el lenguaje de restricciones
- La semántica del lenguaje de restricciones es más fácil de definir
- El lenguaje de restricciones es más ameno para el análisis automático.

A pesar de ello, y de que, como explicaremos a continuación, Alloy surge para suplir tales deficiencias de OCL y dispuesto a desplazarlo en el futuro, dada la disponibilidad actual de herramientas con las que se cuenta de un lenguaje y del otro, utilizaremos OCL para definir restricciones en nuestro modelo de prueba a traducir.

4.4 Alloy

Alloy es un lenguaje de modelado formal, con sintaxis y semántica formal, basado en lógica relacional de primer orden. Sus especificaciones están escritas en ASCII. Posee una representación visual (similar a la de los diagramas de clase de UML y a los diagramas ER), pero esta no posee la expresividad del lenguaje completo [27]. Si bien puede ser utilizado para modelar datos en general (siendo muy útil para especificar objetos de clases, asociaciones entre ellos, y restricciones en estas últimas), su principal objetivo es la especificación formal de los modelos de datos orientados a objetos. Básicamente, Alloy es una combinación de los diagramas de clase UML combinados con OCL, aunque con una semántica más simple y limpia que UML/OCL, y soportado además por una herramienta de verificación. Esta herramienta se denomina *Alloy Analyzer* (Analizador de Alloy), que puede ser utilizada para analizar automáticamente las propiedades de sus modelos.
Este analizador utiliza una verificación delimitada, limitando el número de objetos en cada clase a un número fijo, y chequeando aserciones sobre la especificación dentro de ese límite. Utiliza un *SAT-solver* para responder las queries de verificación, convirtiendo a las mismas en satisfacibilidad de fórmulas booleanas lógicas, y llamando luego al solver para responderlas. Tanto el lenguaje como su analizador fueron desarrollados por el grupo de Daniel Jackson en MIT. Información sobre la actual versión 4.2 puede encontrarse en [28].

Utilizando Alloy, al soportar éste abstracciones de software, podemos comenzar desde un modelo simple y pequeño del dominio y extenderlo gradualmente. En cada paso, el *Alloy Analyzer* automáticamente nos dará un feedback inmediato. Las instancias de nuestro modelo pueden ser útiles para detectar defectos del mismo, y desde allí alcanzar un mejor entendimiento de los requerimientos del usuario [29].
Alloy no posee una sintaxis matemática compleja, y los modelos que provee son fáciles de entender, proveyendo además visualizaciones su analizador, lo que la hacen una muy buena chance para el modelado de dominios.

A modo de definir vagamente la sintaxis de un modelo Alloy, contamos con un pequeño ejemplo en la figura 4.4, donde se ilustra un árbol familiar. Cada caja de la figura 4.4 denota un set de objetos (átomos), correspondiente a una clase en UML/OCL, siendo llamada *signature* en Alloy. Un objeto es una entidad abstracta, atómica y no cambiante. El estado del modelo está determinado por:

- Las relaciones entre los objetos
- El agrupamiento de los objetos en sets
- Estos pueden cambiar en el tiempo

Como mencionamos, Alloy es sólo un lenguaje textual, siendo la notación gráfica sólo una manera útil de visualizar las especificaciones, pero no la forma de escribir un modelo Alloy.

Su representación textual representa el modelo completamente, siendo para la figura 4.4 la siguiente:

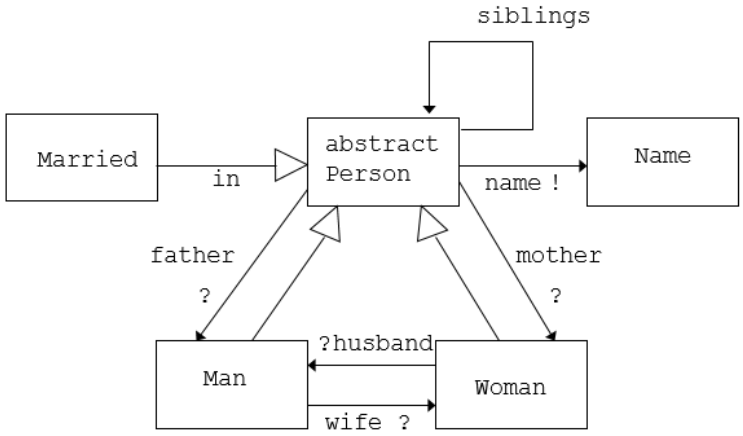


Figura 4.4 Modelo de Objetos de Alloy para un árbol familiar

```
module lenguaje/Family
sig Name { }
abstract sig Person {
        name: one Name,
        siblings: Person,
        father: lone Man,
        mother: lone Woman
        }
sig Man extends Person {
        wife: lone Woman
        }
sig Woman extends Person {
        husband: lone Man
        }
sig Married in Person {
        }
```

La representación textual comienza con declaraciones **sig**, definiendo las *signatures*. En un módulo Alloy, hay un número de ellas, definiendo sets de átomos. La definición de una *signature* puede contener un número de campos que definen relaciones entre sus átomos. Las *signatures* sirven también como tipos, y la subclasificación se hace posible mediante extensiones.

Por supuesto, contamos con formas de definir restricciones en el modelo, utilizando párrafos de restricciones. Hay cuatro tipos de ellos:

- ***Fact:*** una restricción que siempre persiste
- ***Predicate:*** fórmulas nombradas y parametrizadas que pueden ser utilizadas en cualquier otro lugar
- ***Function:*** expresiones nombradas y parametrizadas que pueden ser utilizadas en cualquier otro lugar
- ***Assertion:*** una restricción que debe respetarse a partir de los hechos (*facts*) de un modelo

Los *facts* pueden definirse de dos formas: siguiendo la declaración de una *signature*, o en otro lugar del modelo.

En base al módulo previo, podríamos definir los siguientes invariantes:

```
inv Basics{
        all p | some p.wife <-> p in Man & Married
        //Una persona posee una esposa si solo si esa persona es un hombre y está casado.
        no p | p.wife in p.siblings
        //Ninguna esposa de una persona es a su vez su hermana.
        all p | (sole p.parents & Man) && (sole p.parents & Woman)
        //Una persona posee a lo sumo un padre y a lo sumo una madre.
        no p | p in p.+parents
        //Ninguna persona es ancestra de sí misma.
}
```

Un modelo en Alloy significa una colección de instancias, siendo estas uniones de valores a variables. El *Alloy Analyzer* encuentra instancias de un modelo automáticamente asignando valores a variables, siempre satisfaciendo las restricciones definidas. El análisis involucra resolver restricciones, utilizando el analizador el *SAT solver*, y proveyendo visualización para mostrar el sentido de las soluciones y contraejemplos que encuentra.
Las instrucciones enviadas al analizador para ejecutar su análisis se denominan *comandos*. Un comando en ejecución provoca que el analizador busque una instancia que testifique la consistencia de una función o un predicado; un comando de chequeo provoca que busque un contraejemplo para mostrar que una aserción no persiste. La búsqueda de instancias se realiza dentro de límites finitos, especificados por el usuario con el alcance (*scope*) deseado. Es por ello que al fallar la búsqueda, no significa que no existe instancia alguna que satisfaga el modelo (por ejemplo, el modelo es inconsistente).

Son muchas las ventajas que presenta Alloy frente a otros lenguajes formales, entre otras:

- Utilizándolo junto con su poderosa y eficiente herramienta, el *Alloy Analyzer*, podemos completar un modelo de dominio gradualmente, y en cada paso podemos consultar las instancias visualizadas de nuestro modelo con los expertos del dominio, corrigiendo nuestro modelo utilizando el feedback que obtenemos de ellos.
- No requiere preparar un modelo inicial elaborado conteniendo todos los detalles, siendo este uno de los mayores beneficios de Alloy, permitiendo a su analizador ejecutar su análisis en cualquier etapa del proceso de desarrollo Además, para testear un modelo no hay necesidad de implementarlo.
- Al poseer una sintaxis simple, comparándola con la de otros lenguajes formales, sus modelos son fácilmente entendibles por los expertos del dominio, involucrándonos efectivamente en la ingeniería de requerimientos.
- El aporte de los contraejemplos es fundamental para descubrir los errores del modelo.

Por otro lado, presenta algunas desventajas:

- Al estar basado en lógica relacional, los conceptos de la orientación a objetos puros, como la herencia, no están incluidos directamente, por lo que para utilizarlos en los modelos Alloy debemos descubrir una forma de simularlos en el lenguaje. Por ejemplo, modelar la herencia con *subsignatures*, pero estas caen en un subset de sus superclases, no pudiendo sobreescribir los campos de ellas, causando ciertas limitaciones en algunos casos.

- No provee un fuerte set de tipos primitivos: no existen los strings de caracteres o los puntos flotantes en este lenguaje, debiendo definir nuevas *signatures* para ellos, lo que incrementa el tamaño y la complejidad de nuestros modelos.

4.5 Comparando Lenguajes Formales

4.5.1 Alloy-Z

Dado que Alloy fue altamente influenciado por Z, es sólo natural comparar a ambos. Estando ambos basados en lógica y teoría de sets, Alloy se asemeja altamente a Z; de hecho, Alloy puede verse como un subset de Z [30].
Una de las ventajas de Z es que posee una notación matemática rica que lo hace más expresivo que Alloy. Sin embargo, Alloy es automáticamente analizable, siendo los verificadores de teoremas de Z más limitados (automatizados hasta un punto; las pruebas complejas requieren guía de un usuario con experiencia).
Alloy posee una notación ASCII pura y no requiere herramientas de seteo de tipos especiales, a diferencia de Z que posee la notación y el estilo del cálculo de schemas, teniendo la habilidad de soportar muchos idiomas distintos. Escribir los schemas puede resultar tedioso comparado con escribir una *signature* en Alloy.
En síntesis, Z posee un uso mucho más extendido en la educación y en la investigación, ya habiendo sido aplicado en varios proyectos, incluyendo el sistema CICS de IBM y la Universidad de Oxford. Con el desarrollo continuo de investigaciones, quizás Alloy pueda algún día emular e incluso superar Z.

4.5.2 Alloy-UML/OCL

Alloy comprende una notación de modelado orientada objetos que surgió en los últimos años, más concisa y precisa que OCL, contando con un analizador para validar los modelos expresados en él. Tanto uno como el otro pueden ser usados para especificar los requerimientos de un sistema de software complejo, describiendo sus estados y las transiciones entre ellos.
La sintaxis de Alloy es altamente compatible con la de OCL, y ambos poseen una sintaxis y semántica formal. OCL es utilizado para especificar restricciones en un modelo UML, pudiendo especificar invariantes, precondiciones, postcondiciones y transiciones entre estados. Alloy es similar, pero con una sintaxis y semántica más simple, siendo totalmente declarativo, mientras que OCL es tanto declarativo como operacional [32].
OCL está basado en una lógica de predicados de primer orden, pero utiliza una sintaxis similar a los lenguajes de programación y muy cercana a la de UML. Alloy es similar a OCL, pero sus creadores ostentan que Alloy posee una sintaxis mucho más convencional y una semántica más simple. Alloy es completamente declarativo, mientras que OCL permite mezclar los elementos declarativos y operacionales. Los creadores de Alloy reclaman que OCL es demasiado orientado a la implementación y por lo tanto no lo suficientemente apto para el modelado conceptual; otros creen que la experiencia puede ayudar a mantener el proceso de especificación en un nivel abstracto con OCL.
Hay muchas herramientas disponibles que soportan OCL, como *Octopus* y las herramientas de desarrollo de modelos de *Eclipse*. Las características típicas incluyen la interpretación de restricciones OCL sobre los casos de prueba y generación de código. Algunas soportan análisis del diseño del tiempo, y búsqueda exhaustiva sobre un espacio finito de casos similares a Alloy.

Las restricciones OCL y las operaciones son locales a una clase particular. Esto no es así en Alloy, donde los invariantes son globales. Veamos en un ejemplo los problemas de

poseer operaciones en restricciones (desventaja de OCL nombrada previamente), considerando la definición de la operación *allParents* (todos los padres) en OCL:

OCL	GeneralizableElement allParents: Set (GeneralizableElement); allParents = self.parent → union (self.parent.allparents)
Alloy	all e: GeneralizableElement | e.allParents = e.+parent
Español	La operación *allParents* devuelve un set conteniendo todos los GeneralizableElement heredados por este GeneralizableElement (clausura transitiva), excluyendo a sí mismo.

Tabla 4.1 Operación allParents() en OCL y Alloy

La operación puede entrar en loop infinito si hay una circularidad en la jerarquía de padres, siendo en tal caso indefinida. En Alloy, las restricciones son fórmulas lógicas verdaderas o falsas, sin estados indefinidos. *allParents* está definida como una relación utilizando el operador de clausura transitiva. La operación *allParents* es utilizada en la siguiente restricción OCL:

OCL	GeneralizableElement not self.allParents → includes (self)
Alloy	all e: GeneralizableElement | e !in e.allParents
Español	La herencia circular no está permitida

Tabla 4.2 Operación allParents() sin circularidad en OCL y Alloy

La restricción OCL intenta prohibir la circularidad mencionada. Sin embargo, si la misma existiera, entonces *allParents* entra en un loop infinito y es indefinida, causando que la restricción también lo sea, no siendo falsa como se intenta. Por otro lado, la restricción Alloy está bien definida y prohíbe tal circularidad en la jerarquía.
Veamos los cuatro tipos de diferencias principales que existen entre estos dos lenguajes:

4.5.2.1 Diferencias de complejidad: La descripción en OCL es más orientada a objetos, con un sistema de tipos cercano al de un lenguaje como tal. El estilo de las expresiones de Alloy es más declarativo, de manera de poder especificar la computación en términos de los datos de entrada sin una secuencia de comandos paso-a-paso.
Si bien las notaciones de OCL son similares a las de Alloy, es más complicado utilizar al primero por ser aplicado en el contexto que incluye subclases, polimorfismo paramétrico, herencia múltiple, etc. OCL se implementa orientado, mientras que Alloy está conceptualmente orientado.
La estructura de los modelos Alloy será construida sólo a partir de átomos y relaciones. Las relaciones son utilizadas para relacionar a los átomos, siendo de primer orden y tipadas. Una relación puede ser vacía, unaria, binaria, ternaria, etc. Alloy sólo considera relaciones finitas, sin haber sets o escalares (los sets están representados con relaciones unarias, y los escalares con relaciones unarias de singleton). Toda expresión denota una relación, permitiendo al modelo Alloy ser más sucinto.

Por su parte, las expresiones de OCL a veces son largas y difíciles de leer. La estructura de las expresiones con operadores lógicos a veces son difíciles de descifrar. A continuación, tenemos algunas diferencias sintácticas y semánticas entre OCL y Alloy:

1) En OCL, los operadores de sets no pueden aplicarse a los objetos de la misma clase; en este caso, se deben usar cuantificadores. El sistema de tipos de OCL requiere tipos compatibles, obtenido utilizando coerción (*oclIsKindOf*). En Alloy, los tipos son implícitos y son asociados con los dominios en el modelo.
2) Alloy no posee las nociones de campo, método o aritmética de enteros. UML incluye más nociones y tipos que Alloy, soportando OCL una variedad de tipos que incluye los básicos (Integer, Boolean, String, Real), tipos de modelo definidos por el usuario (clases, interfaces), y tipos de colección de objetos (sets, secuencias, bags). Alloy se maneja con las nociones de tuplas, sets, etc, sin una noción de composición incorporada en absoluto.
3) OCL distingue la navegación de sets y escalares. Para los sets, navega utilizando *'→'*, mientras que para los escalares utiliza el símbolo *'.'*. Esto agrega complejidad para explicar y chequear sus semánticas. Alloy trata los sets y escalares uniformemente, dado que los escalares son considerados sets singletons. El operador clave de Alloy *"punto navegacional"* puede usarse en más de una forma uniforme y flexible que en UML, con una sintaxis uniforme que puede utilizarse tanto para sets como escalares, navegando ambos con el símbolo *'.'*. El símbolo *'→'* se utiliza para representar producto o unión en Alloy
4) Alloy considera a los atributos y a las relaciones como lo mismo, mientras que UML utiliza diferentes sintaxis y semánticas para ambos. Los operadores en Alloy pueden aplicarse a los sets y a las relaciones; sus modelos pueden manejar relaciones con aridad arbitraria, permitiendo reutilización de fragmentos del modelo con un mecanismo de estructuración.
5) La clasificación en UML es estática por defecto. Será necesaria una anotación textual para permitir a una clasificación ser dinámica. Para expresar disjunción y exhaustividad en Alloy, no se requieren anotaciones textuales. El set de Alloy tiene a lo sumo un superset, mientras que UML puede tener múltiples herencias. En UML, la exhaustividad se representa como una propiedad del superset.

4.5.2.2 Diferencias de precisión: para describir las restricciones, OCL utiliza operaciones, quienes generan algunos problemas en OCL, entrando por ejemplo en un loop infinito o siendo indefinidas, haciendo a OCL menos preciso. Otro problema es que las operaciones pueden ser aplicadas a muchas clases que poseen relación de herencia, y luego las mismas pueden ser refinadas por los objetos de esas clases, haciendo ambiguo el significado de la expresión conteniendo las operaciones. La mayoría de los problemas pueden ser descriptos utilizando sólo un pequeño porcentaje de los diagramas UML, y UML no es preciso, siendo difícil crear herramientas para validar los sistemas descriptos en él. En cambio, Alloy posee una fundación semántica más rigorosa.
Alloy es un lenguaje de modelado simple comparado con OCL, siendo más preciso y tratable, permitiendo un análisis automático (diseñado para ello). Puede describir un sistema de forma más precisa que UML, siendo soportado y analizado por una herramienta asociada. El analizador Alloy puede chequear la consistencia del modelo y generar capturas del mismo. Alloy soporta modelado abstracto de alto nivel, pero su objetivo principal no es modelar arquitectura de código como el diagrama de clases de UML.

4.5.2.3 Diferencias de Expresividad: generalmente, UML es más expresivo que Alloy, poseyendo más tipos de datos, y teniendo UML muchas más maneras de describir una

arquitectura de sistema, modelar problemas de dominio, capturar comportamientos, etc. Sin embargo, las expresiones para especificar relaciones son bastante poderosas en Alloy. Por ejemplo, Alloy posee un operador de clausura transitiva, sin haber uno en UML.
Veamos un ejemplo de un modelo *Student* para ilustrar estas diferencias, tomando a Alloy y a UML de forma separada. En el modelo, tenemos estudiantes no graduados y estudiantes graduados, sin ningún estudiante siendo ambos; un estudiante debe registrarse, y sólo así es legal; todo estudiante posee un ID único, y él o ella posee un *major* y sólo puede tener

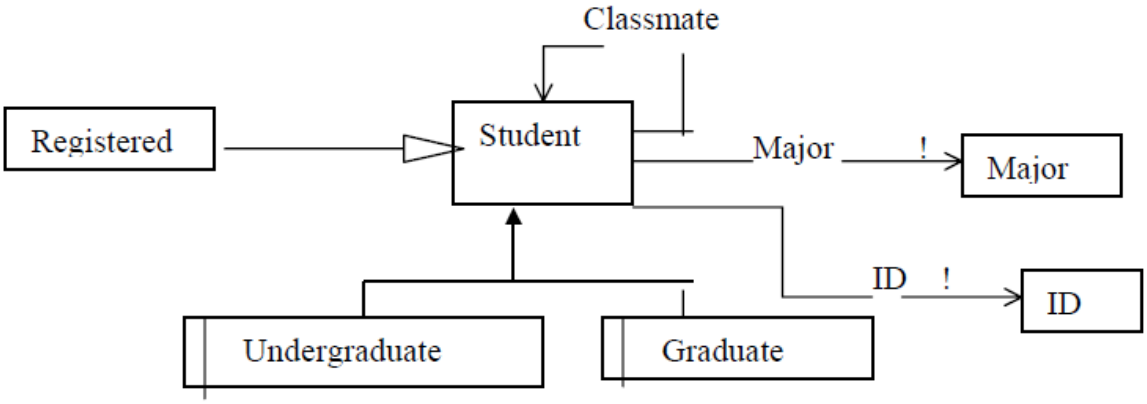


Figura 4.5 Modelo Alloy de los estudiantes

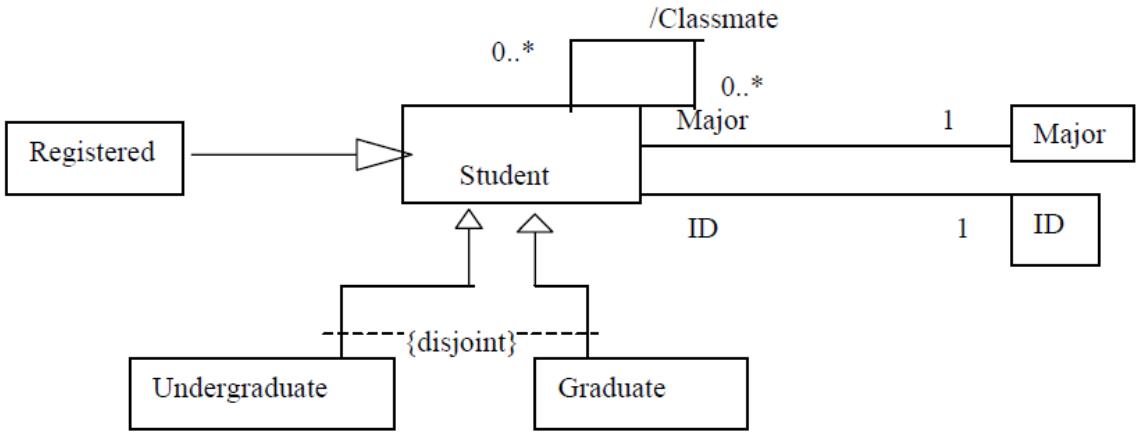


Figura 4.6 Modelo UML de los estudiantes

uno para estudiar; los estudiantes con el mismo *major* se denominan compañeros de clase (*classmates*), pudiendo tener un estudiante múltiples compañeros.
A partir de las figuras 4.5 y 4.6, podemos observar que el grafo Alloy es más simple y abstracto, mientras que el UML es más detallado aunque simple de entender. Cuando un modelo se vuelve más complejo, la simplicidad del grafo Alloy se vuelve más evidente. Algunas diferencias entre las representaciones gráficas de modelos Alloy y UML se listan a continuación:

(1) Alloy utiliza cajas para denotar sets de objetos, así también UML. Pero Alloy diferencia entre tipos de cajas, como sets estáticos (con una línea vertical en la parte izquierda de la caja) y sets fijos (con una línea vertical en la parte derecha de la caja).

(2) Para conectar entidades y sus atributos en Alloy, se utilizan flechas, mientras que en Alloy se utilizan líneas. En Alloy, una flecha con una cabeza cerrada denota subset, al igual que en UML. Pero en Alloy, los subsets que comparten la misma flecha son disjuntos; si los subsets son exclusivos, entonces la cabeza de la flecha debería pintarse. En UML, los subsets son disjuntos, pudiendo utilizar el símbolo *"{disjoint}"* para indicarlo. En Alloy, la marca tras la cabeza de flecha indica que la relación es estática, mientras que en UML, se utiliza un símbolo *"{frozen}".*

(3) Alloy utiliza operadores de expresiones regulares para representar multiplicidad en vez de utilizar rangos enteros como en UML. En Alloy, las marcas al final de las flechas indican restricciones de multiplicidad. El símbolo "!" significa exactamente uno; "?" significa cero o uno; "*" significa cero o más; y "+" significa uno o más. La omisión de las marcas toma por defecto a "*". En UML, la multiplicidad se indica utilizando enteros, utilizando "1" para exactamente uno, "0..1" para cero o uno, "0..*" para cero o más, y "1..*" para uno o más.

4.5.2.4 Diferencias Textuales: el grafo de Alloy refleja las partes del dominio y de estado del modelo. Sólo el texto del modelo Alloy puede representar el modelo equivalente, incluyendo la descripción del grafo y de las restricciones. La parte textual del modelo Alloy de la figura 4.3 es la siguiente:

```
model Students{
    domain { Student, Major, ID}
    state{
        partition Undergraduate, Graduate: static Student
        Registered: Student
        Classmate: Student -> static Student
        Major: Student -> Major !
        ID: Student -> ID !
    }
    def Classmate{
            all a, b | a in b.Classmate <-> (a.major = b.major)
    }
    inv Basics{
            all s | s in Registered
            no s | s in Undergraduate && s in Graduate
    }
    op Register(s: Student, m: Major, d: ID){
             s not in Registered
             all s| s.Major' = s.Major
             all s| s.ID' = s.ID
             all s| s.Classmate' = s. Classmate
             Student' = Student
    }
    assert LegalStudent{
            all s: Registered
    }
}
```

Con la descripción del modelo especificada antes y el grafo de la figura 4.3, el significado correspondiente de las expresiones en este modelo Alloy es muy fácil de entender, dado que las nociones textuales mapean al grafo y la descripción del modelo sin rodeos.
Por su parte, los diagramas UML y OCL son combinados para representar modelos. A pesar de que el grafo UML no posee una contraparte textual equivalente, se utiliza OCL para describir las restricciones. El OCL para el grafo de la figura 4.4 es el siguiente:

```
Student
    self.Classmate = Student.allInstances->select(Major=self.Major)
    self.oclIsKindOf(Registered) implies self.ID. notEmpty
    self.oclIsKindOf(Registered) implies self.Major.notEmpty
    self.oclIsKindOf(Undergraduate) implies not self.oclIsKindOf(Graduate)
    self.oclIsKindOf(Graduate) implies not self.oclIsKindOf(Undergraduate)
Student::Registered (m: Major)
    pre: not self.oclIsKindOf(Registered)
    post: self.Major=m
        Student.allInstances -> forall (s | s!=self implies s.Major@pre = s.Major )
        Student.allInstances -> forall (s | s!=self implies s.ID@pre = s.ID )
        Student allInstances -> forall (s | s.Classmate@pre = s.Classmate )
```

El mapeo del OCL textual al diagrama no es directo, y las expresiones son más largas y difíciles de entender que aquellas en Alloy.

4.5.2.5 Conclusiones: tras comparar las características y las diferencias entre UML y el nuevo lenguaje de modelado Alloy, podemos ver que algunas de las ventajas de Alloy pueden ser referenciadas como una mejora del altamente usado UML. UML es más complicado, mientras que Alloy es más conciso. UML es más ambiguo, mientras que Alloy es más exacto. UML es más expresivo, mientras que Alloy es más abstracto. Como resultado, Alloy provee una manera de validar la etapa de diseño utilizando herramientas de soporte. A futuro, se promete formalizar y transmitir parte de UML a Alloy para permitir un análisis de validación de modelos automático, para reducir los errores en las etapas de requerimientos y de diseño.

4.6 Traducción entre Alloy y UML/OCL

4.6.1 UML/OCL a Alloy

El lenguaje de modelado unificado UML es el lenguaje de facto utilizado en la industria para las especificaciones de software. Es importante contar con una herramienta que permita transformar automáticamente un modelo UML y OCL en un modelo Alloy, quien puede ser luego analizado por el *Alloy Analyzer.*
Un enfoque de transformación se presenta en [33], en donde se ha desarrollado un metamodelo MOF utilizando la representación EBNF de la gramática de Alloy. Para conducir la transformación de modelos desde UML a Alloy, se definieron un conjunto de reglas de transformación, que mapean elementos de un grupo de metamodelos de diagramas de clases y OCL a elementos del metamodelo de Alloy.
Un adelanto del método de transformación se puede observar en la figura 4.7.
Grupo de diagramas de clase UML a utilizar
Este grupo es lo suficientemente expresivo como para representar los conceptos de diagramas de clases básicos, tales como clases, atributos, asociaciones y restricciones OCL. En este caso, se excluyen a las características UML menos populares, cuya

semántica no puede expresarse utilizando clases, asociaciones y OCL, siendo estas las interfaces, dependencias y señales (*signals*).

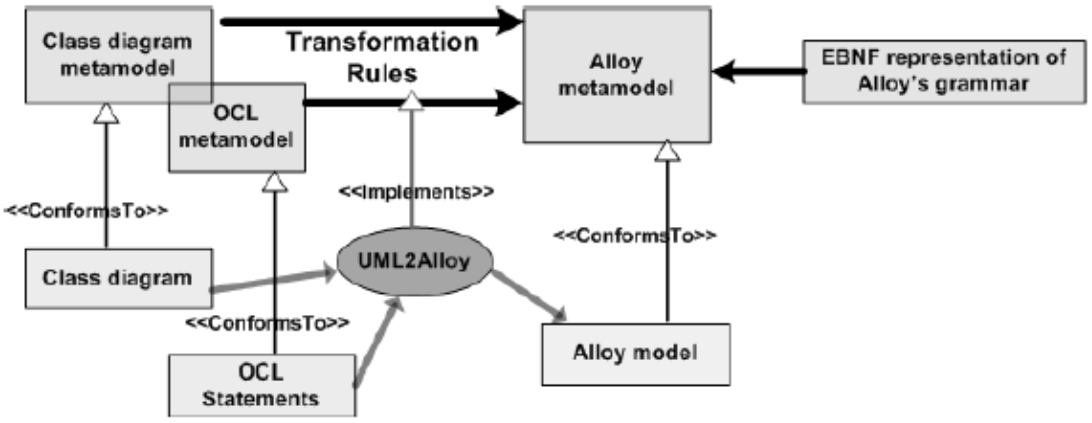


Figura 4.7 Adelanto del método de transformación

El subset de UML utilizado consiste en el paquete del Kernel del metamodelo UML. En la figura 4.8 podemos observar una versión simplificada de este, conteniendo sólo las metaclases concretas (por ejemplo, sin mostrar la metaclase común *Element*).

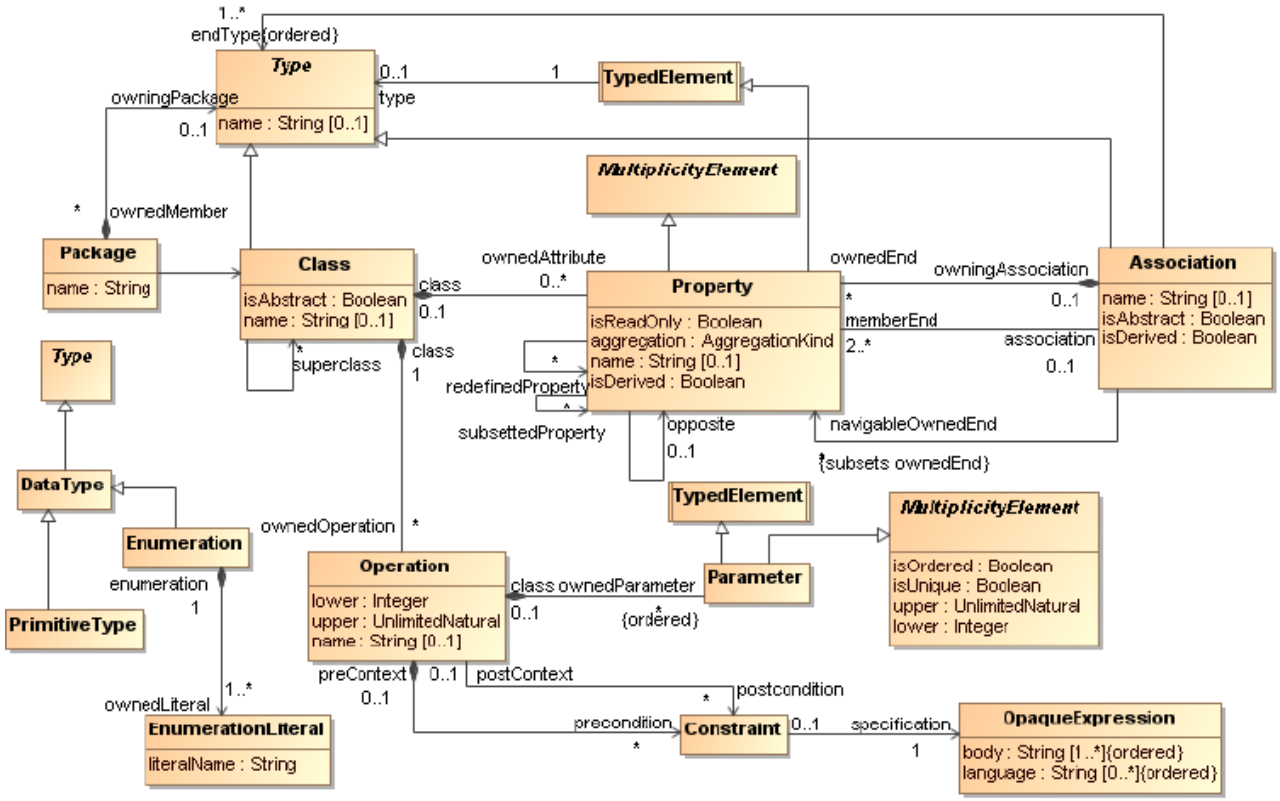


Figura 4.8 Metamodelo UML simplificado

4.6.1.1 Metamodelo Alloy

Alloy es un lenguaje textual, definido en términos de su gramática EBNF, que representa la sintaxis concreta del lenguaje. Para usar MDA, debemos convertir la sintaxis concreta del lenguaje en una representación de sintaxis abstracta acorde a MOF. Este metamodelo Alloy se construye en base al método de *Wimmer y Kramler* para generar metamodelos a partir de representaciones EBNF [34]. En la figura 4.9 vemos una porción del metamodelo Alloy construido para las declaraciones de *signatures*. Una declaración de *signature* (*SigDecl*) es una metaclase abstracta, que peude ser tanto *ExtendSigDecl* como *InSigDecl*, usados para subtipear y subclasificar las *signatures*, respectivamente. Una *SigDecl* posee un cuerpo (*SigBody*), que puede contener una secuencia de restricciones (*ConstraintSequence*), y especifica cero o más declaraciones (*Decl*), utilizadas para definir *fields*. Además, puede declarar una o más variables (*VarId*) y se relacionan a una expresión de declaración (*DeclExp*). Una expresión de declaración puede declarar una relación binaria entre *signatures* (*DeclSetExp*), o una relación que asocie más de dos *signatures* (*DeclRelExp*). Similarmente, se definieron las partes del metamodelo Alloy que representan expresiones, restricciones y operaciones.

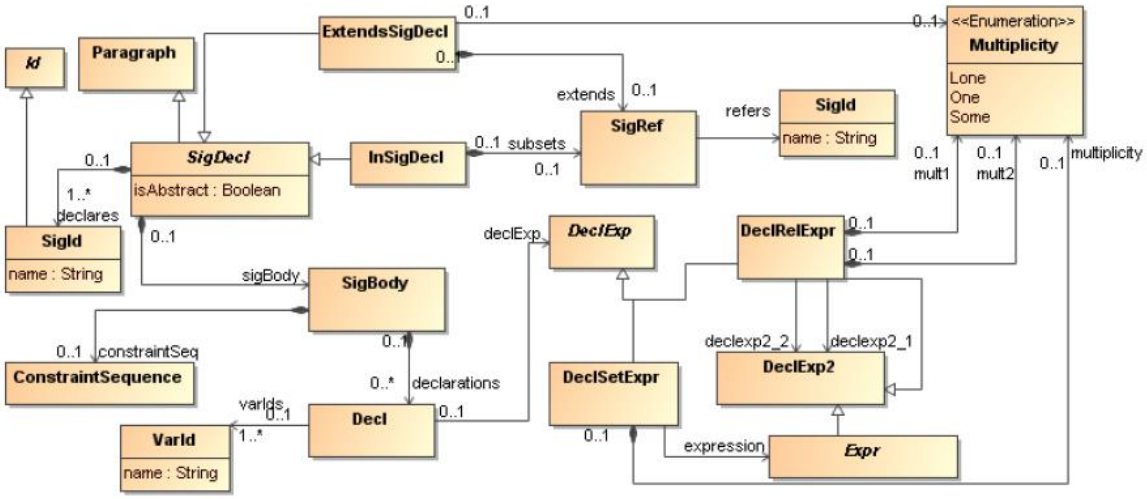


Figura 4.9 Parte del metamodelo Alloy correspondiente a las declaraciones de signatures

En base al manual de referencia de Alloy, podemos consultar algunas reglas de buena formación. Por ejemplo, una *signature* no puede extenderse a sí misma, expresando esto formalmente en OCL:

```
context ExtendsSigDecl
inv: (self.extends -> size() > 0) implies
        (self.declares.name <> self.extends.refers.name)
```

Esta regla no permite que una *signature* se extienda a si misma directamente, pero permite que lo haga indirectamente (por ejemplo, a través de otra *signature*); impedir ello requiere un operador de clausura transitivo, del cual OCL carece.
En la figura 4.10 se presenta un diagrama de clases UML ejemplo, que será utilizado para ilustrar las reglas de transformación de UML y OCL a Alloy, así como las partes desafiantes de la transformación. Este representa el servicio de login de una aplicación de comercio electrónico, que permite a los clientes (como *Client*) adquirir bienes a través de

la red. De allí que es susceptible a varios ataques, como la interceptación de información confidencial (conocido como *man-in-the-middle attack*). El servicio de login debe entonces ser protegido con el protocolo de autenticación y confidencialidad SSL (Capa de Sockets de Seguridad o *Secure Sockets Layer*). El *man-in-the-middle attack* es modelado en el ejemplo agregando una clase *Attacker* que intercepta todas las comunicaciones entre el cliente y el servidor de comercio electrónico; el atacante podría cambiar el contenido de los mensajes intercambiados. El protocolo SSL funciona de la misma manera si el cliente es un SoftwareClient o es un *WebClient,* pero el primero provee alguna funcionalidad extra al usuario. Si el saludo SSL se completa con éxito, una clave de sesión secreta, que puede utilizarse para cifrar y descifrar los mensajes, habrá sido intercambiada entre el cliente y el servidor; todas las comunicaciones posteriores estarán encriptadas, siendo confidenciales. Si el saludo falla, todas las comunicaciones entre cliente y servidor son abortadas.
La figura 4.10 muestra una representación de alto nivel del sistema, donde los atributos de las clases contienen los valores de los mensajes intercambiados entre las entidades participantes en las interacciones. Las clases *SoftwareClient* y *WebClient* fueron agregadas para ilustrar puntos clave en la transformación de UML a Alloy, demostrando diferentes propiedades de herencia que se vuelven importantes.

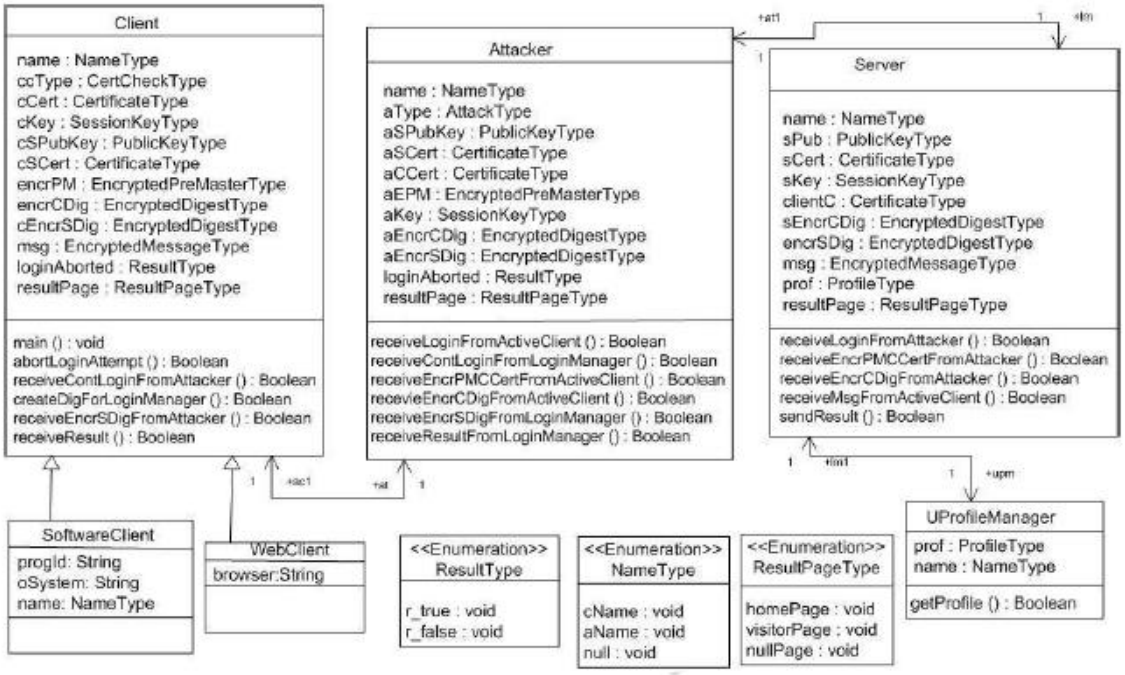


Figura 4.10 Modelo del protocolo SSL utilizado en el servicio de login

4.6.1.2 Mapeando diagramas de clases a Alloy

En la tabla 4.3, se presentan las correspondencias entre los elementos principales de los metamodelos UML y OCL, y Alloy.
Las clases UML de mayor nivel (por ejemplo *Classes*, que no son subclases de ninguna otra clase), se mapean como *signatures* de mayor nivel a Alloy. El metaelemento UML *Class* se mapea al metaelemento de Alloy *ExtendsSigDecl*, quien define un *SigId* con el mismo nombre que la clase. El metaatributo *isAbstract* de la clase se mapea al metaatributo *isAbstract* de *ExtendsSigDecl*. Las *signatures* de mayor nivel, es decir,

Elementos del metamodelo UML+OCL	Elementos del metamodelo Alloy
Package	ModuleHeader
Class	ExtendsSigDecl
Property	Decl
Multiplicity	Expr
Operation	Predicate
Parameter	Decl
Enumeration	ExtendsSigDecl
EnumerationLiteral	ExtendsSigDecl
Constraint	Expression
DataType	ExtendsSigDecl

Tabla 4.3 Correspondencia entre los metamodelos UML y Alloy

clases que no son especializaciones, no se relacionan a ningún *SigRef*, que referencia la *signature* que podrían extender.
Las subclases (clases que extienden de otras) se transforman en las *subsignatures* de Alloy, que son instancias de *ExtendsSigDecl* al igual que las *signatures* de mayor nivel, pero relacionada en este caso a *SigRef*, que referencia la *signature* que extiende. Por ejemplo, la clase *Client* del modelo UML de la figura 4.10 se transforma en un *ExtendsSigDecl* que declara un *SigId* cuyo *name* es *Client*; al no ser *Client* una subclase, no se relaciona a ningún *SigRef*. De manera similar, *SoftwareClient* y *WebClient* se transforman en *ExtendsSigDecl*, pero a diferencia de *Client*, se relacionan a un *SigRef*, que refiere al *SigId* generado para representar la clase *Client*.

4.6.1.3 Diferencias entre UML y Alloy que influyen en la transformación
Si bien ambos lenguajes están diseñados para ser utilizados en el paradigma de orientación a objetos (OO), UML y Alloy poseen distintos enfoques en cuanto a los mismos aspectos fundamentales de OO, tales como herencia, sobrecarga y tipos predefinidos. Ya hemos mencionado previamente grandes diferencias entre los lenguajes, pero ahora nos enfocaremos en algunas de ellas, que influyen directamente en el proceso de transformación de modelos:
A) Herencia: tanto UML como Alloy soportan la herencia. En UML, una clase hija hereda y puede especializar las propiedades de una o más clases. El estándar UML usa el término "*redefine"* para denotar sobreescritura de operaciones o atributos.
En Alloy, una *signature* puede extender a otra, y los elementos de la hija son un subset de los elementos del padre. Sin embargo, la hija no puede declarar un campo cuyo nombre es igual al nombre de un campo de su padre, sin poder sobreescribir los campos de tal. Este problema se resuelve en la transformación, renombrando todos los campos Alloy que poseen conflictos de nombres. Esto se explica mediante un ejemplo:
Consideremos la clase *SoftwareClient* de nuestro diagrama, quien posee el atributo *name*, que sobrecarga el *name* de la clase *Client*. Para transformar este modelo, debemos crear un campo con un nombre único en Alloy y cambiar todas las referencias al atributo *name* de la clase *SoftwareClient*, para referenciar al campo creado. Sin embargo, esto trae nuevas dificultades. De acuerdo con la especificación UML, las restricciones de una superclase se propagan a las subclases. Supongamos que existe la restricción: *self.name <> NameType::null* en *Client* y: *self.name <> NameType::aName* en *SoftwareClient*. Durante la transformación, el atributo *name* de esa clase se renombra a *name1*. La restricción original debe entonces traducirse a *name1!=aName* en Alloy; pero esto permite a *name1* poseer un valor nulo, lo que no es aceptable en el modelo UML original.

Entonces, necesitamos incluir una nueva restricción (por ejemplo, *name1!=null*) en el modelo Alloy resultante, para reflejar las restricciones aplicadas a *name* en *Client.*

B) Espacio de nombres (*namespace*): todos los elementos de los modelos UML se definen en un *namespace*, por ejemplo: generalmente, las clases en un diagrama de clases están definidas en el *namespace* del paquete, mientras que los atributos están definidos en el *namespace* de la clase a la que pertenecen.

Los elementos de los modelos de Alloy también pertenecen a un *namespace*; sin embargo, la noción de tal es levemente distinta en ambos lenguajes. Por ejemplo, la especificación UML define que *"El conjunto de los nombres de atributos y nombres de clases no necesitan ser disjuntos"*. En Alloy, por otro lado, los nombres de los *signatures* deben ser distintos a sus nombres de campos.

Por lo tanto, debemos asegurarnos que, durante la transformación, se cree un nombre único para los elementos Alloy que pertenecen al mismo *namespace*. Primero, identificaremos los elementos del metamodelo UML que pertenecen a distintos *namespaces* UML, pero son traducidos al mismo *namespace* de Alloy (por ejemplo, operaciones de clase). Si esos elementos no poseen un nombre único, les asignaremos uno durante la transformación; todas las referencias a ellos en el modelo UML original también se actualizan en el proceso.

Otro inconveniente es que en OCL la instancia de la clase en la que la operación se aplica puede ser accedida utilizando la palabra clave *self*. En Alloy no existe tal concepto para los predicados, dificultando la referencia a la instancia de la *signature* en la que el predicado Alloy se aplica. Como solución, se pasa la instancia como un parámetro al predicado. Como ejemplo, tomemos la especificación OCL para la operación *abortLoginAttempt* de la clase *Client:*

```
context Client::abortLoginAttempt (): Boolean
    post abortLoginAttempt:
            self.loginAborted = ResultType::r_true and
            self.resultPage = ResultPageType::nullPage
```

Como vemos, se hace uso de *self*. Siguiendo nuestras reglas de transformación, traduciremos tal operación a Alloy como el siguiente predicado, donde se pasa como parámetro al predicado a *act*, instancia de la *signature Client*:

```
pred abortLoginAttempt(act:Client){
act.loginAborted = r_True && act.resultPage = nullPage }
```

C) Sets, Escalares, Relaciones e Indefinición: Alloy trata a los sets y a los escalares como relaciones; una relación Alloy denota un set de tuplas, dependiendo el número de elementos de cada tupla de la aridad de la relación. Por ejemplo, una relación binaria es representada por una 2-tupla, un set como una relación unaria y un escalar como una relación unaria singleton.

Por otro lado, en UML los sets y los escalares comparten el significado estándar de la teoría de sets. El equivalente de las relaciones en UML es una asociación entre las clases, representado como un set de tuplas.

Estas diferencias entre los dos lenguajes parten del hecho de que UML y Alloy poseen distintas filosofías de diseño. Más específicamente, uno de los propósitos de UML es representar los conceptos de la programación OO, con una clara distinción entre escalares y sets. Por otro lado, Alloy fue diseñado para analizar especificaciones abstractas, y la manera uniforme en que trata sets, escalares y relaciones contribuye a su sintaxis sucinta.

Como ejemplo, consideramos el punto de navegación (.). En Alloy, se lo trata como un join relacional. Navegar una relación vacía con este lleva a un set vacío, de ahí que Alloy no requiere afrontar el problema de funciones parciales introduciendo un valor indefinido, como en UML. Más específicamente, asumamos que en el diagrama de clases poseemos la siguiente sentencia OCL:

```
context Client inv: self.at.name = self.at.lm.name
```

En UML, si la instancia de Cliente en que el invariante OCL se evalúa está relacionado a un no atacante, la parte de la sentencia *self.at.name* denotará un valor indefinido, siendo el resultado del invariante *undefined*. Sin embargo, en un modelo Alloy equivalente, si el cliente estaba relacionado a un no atacante, ¡tal restricción siempre denotara verdadero! Esto se da ya que el lado izquierdo de una expresión denotará una relación de sets vacía. Similarmente, el lado derecho, esto es *self.at.lm.name*, evaluará un set vacío, haciendo al invariante evaluado siempre verdadero.
Esto trae serias implicaciones, pues la declaración producirá un resultado distinto en ambos lenguajes. Para solucionar este problema, chequeamos si una sentencia OCL puede evaluar un valor indefinido, utilizando la operación OCL *oclIsUndefined()*:

```
context Client
        inv: if not self.at.name.oclIsUndefined() and
                not self.at.lm.name.oclIsUndefined() then
                        self.at.name = self.at.lm.name
                else false endif
```

En este caso, se especifica que si alguna parte de la expresión no está definida, el invarante debe evaluarse a falso, asegurando que la sentencia OCL evaluará verdadero o falso, pero no *undefined*. Esta expresión se transforma a Alloy utilizando las reglas de transformación estándar vistas.

D) Tipos predefinidos: la especificación UML define un número de tipos primitivos (String, Real, etc.). Estos tipos pueden utilizarse al desarrollar modelos UML; por ejemplo, el atributo *browser* de *WebClient* es de tipo String. Por otro lado, Alloy posee un sistema de tipos simple, y el único tipo predefinido es Integer. Sin embargo, algunos tipos del resto de los tipos predefinidos de UML pueden modelarse en Alloy, tal es el caso de String, que puede modelarse como una secuencia de caracteres, y cada carácter pueden representarse por un átomo.
Una consecuencia de este enfoque es que, mientras que los tipos primitivos de UML y sus operaciones son parte del metamodelo, en Alloy los mismos requieren ser definidos en el nivel de modelo (por ejemplo, String debe declararse como una *signature* de Alloy). Las reglas de transformación definidas realizan esto automáticamente para algunos tipos de atributos, como String.

E) Mecanismo de extensión de UML: UML provee dos mecanismos de extensión. Uno de ellos es crear un perfil, y el otro es extender el metamodelo UML, caso en el que necesitaríamos incorporar las reglas que involucran los nuevos elementos en la transformación.
La transformación actual trata con un subconjunto de los metamodelos estándar UML y OCL. Si los mismos fueran extendidos, deberíamos incorporar la nueva semántica en la transformación. Por ejemplo, si asumimos que UML se extendió con la habilidad de definir un estereotipo *Singleton*; al usar el mismo en una clase, restringe a la misma a poseer sólo una instancia, expresado con el invariante OCL: self.allInstances() → size () = 1. En tal

caso, las reglas de transformación necesitan ser ajustadas acordemente. Particularmente, al encontrar en una clase un estereotipo *Singleton*, debemos incluir una restricción en el modelo Alloy resultante, que imponga que la *signature* resultante sólo tendrá una instancia.

F) Agregación y Composición: UML trata a la agregación y a la composición como tipos especiales de asociaciones. Alloy no soporta directamente las nociones de estos conceptos. Afortunadamente, existe una manera metódica de refactorizarlas como asociaciones con restricciones OCL adicionales, que representan las semánticas de estos conceptos, utilizando esto para las reglas de transformación.

G) Modelos Estáticos vs. Dinámicos: los modelos en Alloy son estáticos, capturando por ejemplo entidades de un sistema, sus relaciones y restricciones acerca del mismo. Un modelo Alloy define una instancia de un sistema, donde las restricciones se satisfacen. Sin embargo, estos modelos no poseen una noción inherente de estados: Alloy no incluye ninguna noción de máquinas de estado.
En UML, el término estático se utiliza para describir una vista del sistema, que representa las relaciones estructurales entre los elementos, así como las restricciones y la especificación de operaciones, con la ayuda de pre y post condiciones. En UML, a diferencia de Alloy, los modelos estáticos poseen una noción inherente de estados. Un *estado de sistema* se construye a partir de los valores de los objetos, links y atributos en un punto particular del tiempo.
Tenemos entonces que UML posee una noción implícita de estados, mientras que Alloy no la soporta directamente, introduciendo una complejidad adicional en la transformación. Asumamos la sentencia OCL que define la operación *receiveResult()* de *Client*:

```
context Client::receiveResult():void
pre: self.resultPage = ResultPageType::nullPage
post: self.resultPage = ResultPageType::homePage
```

Para evaluar esta expresión, se requieren dos estados consecutivos: uno para representar el estado antes la ejecución de la operación (precondición), y otro para el estado posterior (postcondición). El estándar OCL especifica formalmente el ambiente en que se evalúan ambas. Si esta sentencia fuera traducida directamente a Alloy, produciría este resultado:

```
pred receiveResult(act:Client){
        act.resultPage = nullPage
        act.resultPage = homePage }
```

Sin embargo, esta especificación Alloy lleva a un modelo inconsistente, asignando los valores *nullPage* y *homePage* al campo *resultPage* al mismo tiempo, generando una inconsistencia lógica, dado que no pueden ser ambas sentencias verdaderas.
Una solución propuesta es introducir la noción de *estado* al nivel de modelo. Esta es una forma estándar de modelar sistemas dinámicos en Alloy, permitiendo poseer dos estados consecutivos, evaluando las precondiciones de cada operación en el primer estado y la postcondición de una operación en el estado siguiente.

4.6.1.4 Herramientas de Traducción

Entre las herramientas de traducción más conocidas de UML/OCL a Alloy, contamos con *UML2Alloy* [35], que transforma los modelos que involucran diagramas de clases UML y

OCL al lenguaje Alloy. Luego, se utiliza Alloy para analizar el modelo e identificar inconsistencias en el diseño, produciendo contraejemplos que ayuden a revelar la fuente de las mismas. UML2Alloy fue exitosamente aplicado a varios dominios, incluyendo la manufactura ágil, la seguridad y el control de acceso.

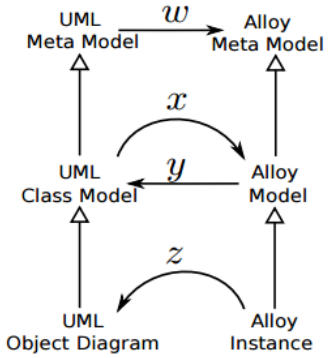


Figura 4.11 Múltiples capas en UML2Alloy. Las flechas horizontales representan transformación, y las verticales muestran relación de instancias.

UML2Alloy se basa en la definición de un mapeo entre el metamodelo UML y el metamodelo Alloy (*w* en la figura 4.11). La herramienta trabaja convirtiendo el modelo UML en un modelo equivalente en el lenguaje Alloy (*x* en la figura 4.11), y utilizando la API del Alloy Analyzer para llevar a cabo el análisis, producido parte de él en la forma de instancias Alloy, una forma en la cual el usuario no está muy familiarizado. Las herramientas UML no suelen soportar un análisis nativo, pero pueden ser usadas para ver y modificar el análisis producido externamente, donde es una forma UML estándar. Utilizando esta herramienta y una conversión de las instancias Alloy, el análisis puede luego ser utilizado como parte del proceso de desarrollo normal.
En este proyecto, utilizaremos la herramienta ***AlloyMDA***[52], a introducir y ejemplificar luego en el capítulo 7.

4.6.1.5 Conclusión

Tras aplicar las reglas de transformación de modelos desde UML a Alloy en el ejemplo presentado, se chequeó en [33] el modelo Alloy producido, utilizando el *Alloy Analyzer*. La aserción que se debe validar es que, si el atacante obtiene la clave de sesión secreta, el acuerdo SSL debería fallar siempre. Esta aserción se especifica en OCL de la siguiente manera:

```
context Client
inv sameKeySuccess: Client.allInstances() -> forAll(ac:Client |
        ac.loginAborted = ResultType::r_false implies (
        ac.cKey = SessionKeyType::symmKey and
        ac.at.sKey = SessionKeyType::symmKey
        and ac.at.aKey <> SessionKeyType::symmKey))
```

Esta sentencia OCL se tradujo automáticamente a la siguiente aserción Alloy:

```
assert sameKeySuccess{ all ac:Client | ac.loginAborted = r_false
implies (ac.cKey = symmKey && ac.at.lm.sKey = symmKey &&
ac.at.aKey != symmKey) }
```

Esta aserción se chequeó con un *scope* (alcance) de seis, esto es: el Alloy Analyzer intentará encontrar una instancia que viole la aserción, utilizando a lo sumo seis instancias

de cada entidad definida en el diagrama de clases visto (Client, Attacker, Server, por ejemplo). La aserción no produjo ningún contraejemplo, significando que es válida para el *scope* dado.

4.6.2 Alloy a UML/OCL

Para cumplir la visión de eliminar el puente existente entre el diseño y el análisis formal, hay una necesidad clara de transferir el resultado del análisis nuevamente al espacio de diseño. En otras palabras, un modelo de diseño *debe* ser transformado a un lenguaje adecuado para el análisis, y, tras conducir el mismo, el resultado *debe* ser transformado de vuelta al espacio de diseño. De allí, el diseñador puede tanto producir un modelo en el espacio de diseño como recibir el feedback del análisis en ese espacio. Al utilizar la herramienta mencionada *UML2Alloy* [35], si se encuentra un contraejemplo en el análisis, debe ser transferido a un diagrama de objetos UML, que sea una instancia del diagrama de clases en el espacio de diseño y que represente una violación de una propiedad del sistema; de otra forma, el desarrollador debe ser un experto en ambos lenguajes Alloy y OCL.
Ha sido demostrado que definir una transformación de Alloy a UML es una tarea muy desafiante. Primeramente, las transformaciones de UML a Alloy y viceversa no son independientes; la transformación desde instancias de modelos Alloy a UML debe resultar en una instancia del modelo UML original. Segundo, cualquiera podría pensar que la transformación de modelos bidireccional solucionaría el problema; pero este no es el caso, ya que la segunda transformación es entre las instancias de los modelos que fueron transformadas por la transformación original de UML a Alloy (en otras palabras, la primera se lleva a cabo en la capa M2 de la jerarquía MOF, mientras que la segunda lo hace en la capa M1).

La dificultad principal de convertir instancias de Alloy nuevamente en instancias UML (*z* en la figura 4.11) se basa en las diferencias semánticas inherentes entre ambos lenguajes. Las instancias en Alloy, naturalmente, no son instancias del modelo UML original; se pierde alguna información en el proceso de transformación. Por ejemplo, los atributos y asociaciones del modelo UML se convierten en campos Alloy, por lo que convertir la instancia Alloy nuevamente al modelo UML original requeriría conocer precisamente cómo fueron estos convertidos.
Si bien sería posible inferir el mapeo manualmente (vía inspección), el proceso sería muy largo, tedioso y susceptible a errores. Las instancias producidas por Alloy podrían ser tanto largas como numerosas, no adecuadas para la conversión manual. De ahí surge la necesidad de automatizar la conversión de las instancias Alloy en UML. Dado que los metamodelos (Modelo de clases UML y modelo Alloy de la figura 4.11) pueden cambiar, la transformación entre ellos debería idealmente ser generada automáticamente, basándose en la transformación UML2Alloy.
Otro problema con la conversión manual, o incluso con una transformación de modelos creada manualmente, es la exactitud de la conversión. En los modelos convertidos a mano, la malinterpretación de la transformación original puede significar instancias convertidas incorrectamente. Similarmente, es susceptible a un error del desarrollador, resultando en muchas instancias de análisis mal convertidas. En [36], se presenta una solución creando una instancia de análisis de transformación automáticamente, basada en el rastro de ejecución de la transformación UML2Alloy original, llevando a un mayor grado de confianza de que hay una consistencia entre las dos transformaciones.
En la figura 4.12 podemos observar los pasos enumerados de la solución propuesta para convertir instancias Alloy en instancias UML. El paso *w* de la figura 4.11 es “UML2Alloy” y *z* es el conversor de instancia “Alloy instance converter” de la figura 4.12. La solución se

centra en una transformación inicial en UML2Alloy, donde dado un modelo UML, se producirá un modelo Alloy (paso 1 en 4.12). El modelo Alloy resultante puede ser analizado automáticamente, con parte de este análisis producido como instancias Alloy (paso 2 en 4.12). Utilizando el rastro de la primera transformación UML2Alloy, creamos otra transformación de modelos (paso 3 en 4.12). Esta segunda transformación se utiliza sobre instancias Alloy, para convertirlas en instancias UML (paso 4 en 4.12).

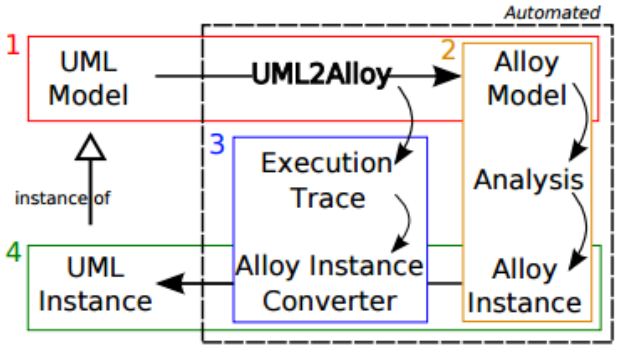


Figura 4.12 Pasos para convertir el modelo Alloy en uno UML

Paso 1: la primera transformación UML2Alloy se ejecuta en el modelo UML, para producir un modelo Alloy y un rastro de transformación. Las propiedades del modelo Alloy resultante pueden evaluarse automáticamente.
Paso 2: en este paso, UML2Alloy utiliza la API del Alloy Analyzer para analizar automáticamente el modelo Alloy. Para producir instancias, se realizan dos tipos generales de análisis: simulación (produce una instancia arbitraria conforme a las restricciones del modelo) y chequeo de aserciones (permitirá al usuario verificar una propiedad del modelo, con una instancia producida (contraejemplo) si la propiedad no se cumple).
Paso 3: el siguiente paso es crear el conversor de instancia de Alloy a UML, utilizando técnicas de MDA, creando una transformación. Se utiliza el rastro de la transformación de modelos UML a Alloy original (ejecutada en el paso 1) como especificación para crear la segunda instancia de transformación (paso 4). Cada instancia del rastro, que ha grabado una conversión en la primera transformación, se convierte en una regla de la segunda transformación.
Paso 4: la segunda transformación (paso 4 de 4.12) puede ejecutarse, convirtiendo instancias Alloy en instancias UML. La instancia resultante puede luego usarse en el soporte de herramientas UML estándar.

Consideremos el diagrama de clases UML con restricciones OCL de la figura 4.13a. Utilizando la transformación de modelos UML2Alloy en este modelo, un modelo Alloy es producido, mostrado en la figura b. Por ejemplo, la clase UML *Person* se convierte a la *signature* de Alloy *Person* (línea 1, figura b). Los atributos de clase se vuelven campos de la *signature*, por ejemplo el atributo *age* (edad) de *Person* se vuelve el campo *age* de la *signature* (línea 2, figura b). La asociación UML navegable entre *Person* y *BankAccount* se transforma en los campos y hechos de las líneas 3, 6, 8-10 del modelo Alloy. La conversión de asociaciones y atributos a campos resalta una mayor diferencia semántica entre los formalismos. Finalmente, la restricción OCL del modelo se convierte en los *facts* (hechos) del modelo Alloy (línea 12).

El modelo Alloy puede ser simulado utilizando el Alloy Analyzer para producir una (generalmente muchas más y más largas) instancia Alloy como la de la figura 4.13c. La transformación UML2Alloy produce un rastro, creado en este caso cuando la clase *Person* se convierte en el *sig Person*. El rastro de la clase a *sig* también se refiere a otro rastro: el atributo *age* al campo *age*.
Utilizando esta información, la transformación *Trace2MT* (trazo a MT) creará la instancia de transformación (paso 3 de 4.12). Basándose en ese trazo, la primera regla creada es *PersonSig2PersonClass*, que toma una instancia de la *signature Person* de Alloy y la convierte en una instancia de la clase UML *Person*. Esta regla se creará para invocar otra regla: *AgeField2AgeAttribute*, quien convierte instancias del campo *age* de Alloy en instancias del atributo *age* de UML. El resto de la transformación de modelos (paso 4 de 4.12) es creada repitiendo el proceso para cada trazo (producido en el paso 1 de 4.12). Una vez que la transformación fue creada, las instancias Alloy pueden ser automáticamente convertidas en instancias UML.

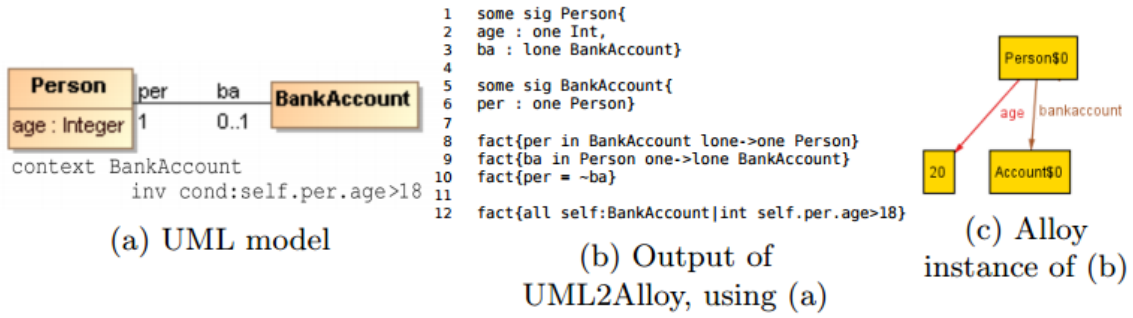


Figura 4.13 Transformación de modelos multi-nivel, con ejemplos

4.7 Resumen

En el capítulo, hemos mencionado la definición de un lenguaje formal, su utilidad, su aplicación en la industria actual, y específicamente nos enfocamos en OCL y Alloy, comparando a los mismos, y mostrando sus ventajas y desventajas. Como criterios de evaluación de los mismos, se tomaron en cuenta la expresividad, la frecuencia de errores, la redundancia, la localidad de los cambios, la reusabilidad y el soporte/documentación.
Como conclusión, podemos decir que las especificaciones formales en el modelado son tanto deseables como posibles. La formalización es **deseable** porque revela errores en modelos conceptuales, incluso cuando estos fueron pensados para ser de alta calidad. Además, es **posible**, pero sólo cuando se da un soporte extensivo en la forma de guías y herramientas de software (de ahí que nos quedaremos con OCL, quien cuenta al día de la fecha con mayor disponibilidad de las mismas). La relación estructural cercana entre los modelos formales e informales hace posible el brindado de un soporte extensivo para un gran número de aspectos del proceso de formalización.
Por otro lado, además de las conclusiones positivas de la utilización de los lenguajes formales, existe una conclusión negativa, revelada en las dificultades del reutilización de fragmentos de modelos. A veces, las interpretaciones ambiguas de fragmentos de modelos informales ponen en peligro a los fragmentos de los modelos, tanto formales como informales.
En los capítulos siguientes, utilizaremos a los lenguajes OCL y Alloy en la práctica, llegando a un mejor conocimiento de los mismos.

5. *Transformaciones de modelos*

Ya metidos en el mundo de la especificación formal y en las herramientas con las que contamos para ello, debemos comenzar a ver cómo se traducen tales especificaciones automáticamente a código. Esto implica revisar desde lo más abstracto, investigando sobre las reglas de traducción automática existentes y a tener en cuenta; hasta lo más específico, como seleccionar el tipo de herramienta para lograr el objetivo. Pero una transformación implica más que sólo meramente generar código automáticamente, pudiendo referirnos por ejemplo a transformación entre modelos de capas superiores.
El objetivo de este capítulo es inicialmente definir las transformaciones y sus tipos, para luego especificar las reglas que las mismas poseen (específicamente, para la derivación de casos de prueba), ejemplificar uno de sus tipos, y observar cómo se lleva a cabo el testing de estos procesos.

5.1 Transformaciones

En el capítulo 2, hablamos del proceso de MDD y mencionamos los roles de varios modelos, tales como PIM y PSM. Una parte clave del MDD es la noción de transformaciones de modelos automatizadas, donde los modelos son refinados en modelos más detallados, y, eventualmente, en código. Existen herramientas que soportan MDD y toman un PIM como entrada y lo transforman a un PSM; esas mismas herramientas u otras, tomarán al PSM y lo transformarán ahora a código. Estas transformaciones son esenciales en el proceso de desarrollo de MDD, conformando su esencia. En la figura 5.1 observamos la herramienta de transformación como una caja negra, que toma un modelo de entrada y produce otro como salida.

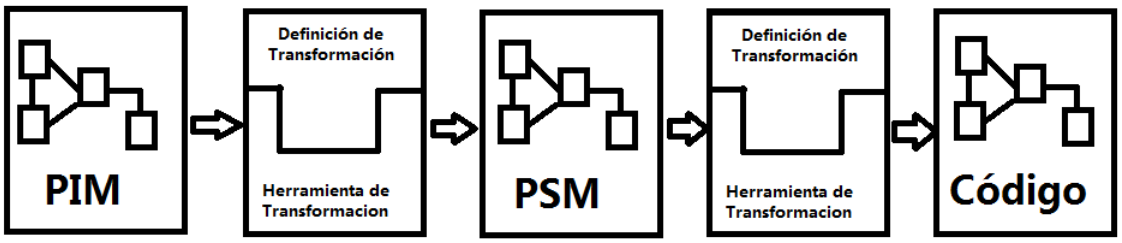


Figura 5.1 Definiciones de transformaciones dentro de las herramientas de transformación

Si inspeccionáramos la herramienta de transformación por dentro, veríamos los elementos involucrados en la ejecución de la transformación. En algún lugar dentro de ella hay una definición que describe cómo transformar el modelo fuente para producir el modelo destino. A esto se conoce como la **definición de la transformación**. Lo que vemos en la imagen es su **estructura**. Existe una diferencia entre la transformación misma, que es el proceso de generar un nuevo modelo a partir de otro, y su definición.
Para especificar la transformación (que se aplicará muchas veces, independientemente del modelo fuente al que se aplicará), se relacionan construcciones de un lenguaje fuente en construcciones de un lenguaje destino. En nuestro caso, definiremos una transformación que relaciona elementos de UML/OCL a elementos Java, describiendo cómo los elementos Java pueden generarse a partir de los elementos UML/OCL.
Definimos entonces a una transformación como una *colección de reglas, las cuales son específicamente no ambiguas de las formas en que un modelo (o parte de él) puede ser*

usado para crear otro modelo (o parte de él)[1]. Según [40], es un programa que mapea modelos de entrada conformes a un metamodelo fuente, a modelos de salida conformes a un metamodelo objetivo. La mayoría de estas transformaciones son Turing-completas, pudiendo ser utilizadas para resolver cualquier problema computacional.
La correctitud de una transformación es esencial para el éxito de MDD, y, mientras varias investigaciones se han centrado en la verificación formal, como vimos en el capítulo previo, el testeo sigue siendo el método más eficiente de validación, aunque el testeo de las transformaciones es distinto al del código, presentando nuevos desafíos (ver 5.1.3).

Existen distintas intenciones al realizar una transformación de modelos, que aparecen repetidamente en la mayoría de los esfuerzos del MDD. Es útil para sus desarrolladores estar al tanto de estas intenciones. Por ejemplo, la intención de una de ellas puede ser extraer distintas visiones de un modelo (query), agregar o remover detalles (refinamiento o abstracción), traducir el modelo a otro lenguaje de modelado (traducción), ejecutar el modelo (simulación), reestructurar el mismo para mejorar ciertos atributos de calidad (refactoring), componer modelos (composición), o reconciliar la información en distintos modelos (sincronización).
Cada uno de estos intentos posee su propio conjunto de atributos y propiedades. La efectividad de una transformación en base a su intención depende de cuán bien la misma respete a esta. Por ejemplo, las queries deberían producir información contenida en el modelo en alguna forma, las traducciones y el refactoring deberían preservar las semánticas del modelo, y el refinamiento debería agregar información.
Toda transformación se lleva a cabo entonces elaborando un conjunto de reglas y restricciones que luego deben ser respetadas.

5.2 Tipos y Elección de Transformaciones de Modelos

Al momento de decidir el enfoque de transformación de modelos (desde aquí, MT) más apropiado para abordar un problema determinado, existen un conjunto de preguntas que deben ser respondidas [56]. Basándose en estas respuestas, el programador podrá luego elegir el tipo de MT más adecuado para sus necesidades.

1) ¿Qué necesita ser transformado en qué?
La primera pregunta importante es ¿Qué necesita ser transformado en qué?, refiriéndose a la fuente y los artefactos objetivo de la MT.
Transformación de programa y modelo: El primer aspecto a considerar es el tipo de artefacto que se está transformando. Si este es un programa (código fuente, código de bytes, código de máquina), se utiliza el término *transformación de programa*; si en cambio se trata de modelos, hablamos de MT, que puede ir de un modelo abstracto a uno concreto (por ejemplo, de diseño a código) y viceversa (por ejemplo, en un contexto de ingeniería inversa), y es vital en herramientas comunes como generadores de código y *parseadores*.
Transformación Endógena vs. Exógena: para transformar modelos, estos deben estar expresados en algún lenguaje de modelado (por ejemplo, UML para modelos de diseño, y lenguajes de programación para modelos de código fuente), cuya sintaxis y semántica está expresada por un metamodelo (el UML, entre otros).
Basándose en el lenguaje en que el modelo fuente y el modelo objetivo de una MT son expresados, se puede hacer una distinción entre MT endógenas y exógenas:

- Una MT **endógena** es una MT entre modelos expresados en el mismo lenguaje, realizadas para:
    - Optimizar, preservando la semántica del software.

  - *Refactorizar*, cambiando la estructura interna del software para mejorar su calidad, sin alterar su comportamiento observable.
  - Simplificar y normalizar
- Una MT **exógena** es una MT entre modelos expresados utilizando lenguajes distintos, presentes por ejemplo en:
  - Síntesis de una especificación de alto nivel, abstracta (análisis, modelo de diseño) en otra más concreta de bajo nivel (modelo de programa Java). La transformación típica de este tipo es la generación de código, donde el código fuente se traduce a un *bytecode* que corre en una máquina virtual, o a un código ejecutable. En base a esto, podemos decir que en nuestro proyecto, llevaremos a cabo una MT exógena.
  - Ingeniería inversa.

MT horizontales vs verticales: una MT horizontal es aquella en la que los modelos fuente y objetivo están especificados en el mismo nivel de abstracción, como en el *refactoring*; en una MT vertical, los modelos residen en distintos niveles, como en el refinamiento.
En base a estas dos tipificaciones de las MT, contamos con la siguiente tabla:

	Horizontal	**Vertical**
Endógena	Refactoring	Refinamiento formal
Exógena	Migración de lenguaje	Generación de codigo

Tabla 5.1 Transformaciónes de modelos: Endógena vs. Exógena, Horizontal vs Vertical

Espacio tecnológico: al traducir por ejemplo documentos XML a diagramas UML, uno puede elegir ejecutar la transformación real tanto en el espacio XML (utilizando XQuery o XSLT) como en el MDA (requiriendo un metamodelo MOF para XML).

2) ¿Cuáles son las características importantes de una MT?
Nivel de automatización: distinción entre las MT automatizables y las que deben realizarse manualmente (o requieren al menos cierto nivel de intervención). El segundo caso se presenta al traducir por ejemplo un documento de requerimientos a un modelo de análisis, debiendo resolver el usuario ambigüedades, incompletitud e inconsistencia en los requerimientos expresados parcialmente en lenguaje natural manualmente.
Complejidad de la transformación: algunas MT, como el *refactoring*, son consideradas pequeñas, mientras que otras son considerablemente tareas pesadas, como *parseadores*, compiladores y generadores de código. La diferencia entre estos dos es tan grande que prácticamente requieren un conjunto de técnicas y herramientas totalmente distinto.
Preservación: si bien existe un amplio rango de tipos de MT útiles durante el MDD, cada una de estas preserva ciertos aspectos del modelo fuente en el modelo transformado; estos difieren significativamente dependiendo del tipo de MT: en *refactoring*, por ejemplo, el comportamiento externo necesita ser preservado, mientras que la estructura es modificada; en el *refinamiento*, la correctitud del programa requiere ser preservada.

3) ¿Cuáles son los criterios de éxito para una herramienta o lenguaje de transformación?
Al mismo nivel de importancia que las características de las MT o de los modelos siendo transformados, aparecen las características de las herramientas o lenguajes de transformación.
Habilidad de crear/leer/actualizar/eliminar transformaciones (CRUD): si bien este es un requerimiento trivial para un lenguaje de transformación, no es tan obvio para una herramienta de transformación. Si consideramos una herramienta típica de *refactorización* de un programa, esta incluye un conjunto de transformaciones predefinidas que pueden ser aplicadas, pero no suele contar con una manera de definir nuevas transformaciones o

adaptar las existentes a las necesidades de usuario; tener la posibilidad de realizar tales acciones es un criterio importante para asegurar la extensibilidad y elasticidad de una herramienta de transformación.
Habilidad de sugerir cuándo aplicar transformaciones: para ciertos escenarios, herramientas dedicadas pueden crearse para sugerir al usuario la MT más apropiada en un contexto determinado.
Habilidad de adaptar o reutilizar transformaciones: por ejemplo, utilizar el mecanismo de herencia para este propósito en un lenguaje de transformación orientado a objetos.
Habilidad para garantizar la correctitud de las transformaciones: la noción más simple es la de *correctitud sintáctica*: dado un modelo fuente bien formado, ¿podemos garantizar que el modelo producido por la transformación también está bien formado? Una noción bastante más compleja es la de *correctitud semántica*: ¿posee el modelo producido las propiedades semánticas esperadas?
Habilidad de manejar modelos incompletos o incosnistentes: es importante ser capaz de transformar modelos tempranamente en el ciclo de vida del MDD, cuando los requerimientos pueden no haber sido totalmente comprendidos o descriptos en lenguaje natural. Esto suele llevar a modelos ambiguos, incompletos o incosnistentes, lo que implica que requerimos mecanismos para el manejo de inconsistencias; estos pueden utilizarse para detectar inconsistencias en las transformaciones en sí, o en los modelos siendo transformados.
Habilidad de agrupar, componer y descomponer transformaciones: útil para incrementar la legibilidad, modularidad y mantenibilidad de un lenguaje de transformación.
Habilidad de testear, validar y verificar transformaciones: necesitamos aplicar técnicas de validación y verificación y asegurar que poseen el comportamiento deseado.

4) ¿Qué mecanismos pueden ser utilizados para MT?
Para especificar y aplicar una transformación, pueden utilizarse ideas de cualquiera de los paradigmas de programación. Uno puede decidir utilizar un enfoque funcional, lógico, procedural u orientado a objetos, o uno híbrido combinando cualquiera de los anteriores.
La mayor diferencia entre los mecanismos de transformación es si estos se basan en u enfoque *declarativo* u *operacional* (o *imperativo*). Los enfoques declarativos se enfocan en el **qué** (por ejemplo, en qué necesita ser transformado en qué); por su parte, los enfoques operacionales se enfocan en el **cómo** (por ejemplo, en cómo la transformación en si necesita ser realizada).
Desde un punto de vista teórico, el enfoque declarativo parece ser el más prometedor, fundado formalmente, ofreciendo bidireccionalidad, y lo más importante, ofrece un modelo de semántica más simple para entender y especificar MT. Por ejemplo, el orden de aplicación de reglas y la creación de modelos finales son implícitos, dándole mantenibilidad y haciéndolo compacto. Este enfoque incluye la programación funcional y lógica, y la transformación de grafos.
Por otro lado, el enfoque operacional puede adaptarse mejor al considerar MT que son requeridas para actualizar incrementalmente un modelo. Gracias a sus nociones de secuencia, selección e iteración, este enfoque mejora el control del orden de aplicación de un conjunto de MTs.

5.3 Testing de Transformaciones de Modelos
Las transformaciones de modelos son los artefactos primarios del MDD, son su corazón y esencia, y deben ser consideradas de esa forma al aplicar técnicas de MDD. Como cualquier otra pieza de software, estas transformaciones deben ser diseñadas, implementadas y testeadas. Al no contar con técnicas y métodos estándar para testearlas

(especialmente en el caso de las transformaciones modelo-a-texto, como el nuestro), se proponen diferentes enfoques para tal objetivo, uno de ellos descripto en [39], donde se testea una transformación de modelo que mapea modelos diseñados en UML con perfiles a aplicaciones de escritorio de Java.
Testear estas transformaciones es un problema más complejo que el testeo de código. Hay varios factores a considerar al evaluar la complejidad del problema, como lo son:

- Existen muchos lenguajes de transformación de modelos, algunos de ellos de propósito general (Java, C++), otros diseñados para tareas específicas como transformaciones modelo-a-modelo M2M (ATL, QVT) y traducción de modelo-a-código M2C (Acceleo, Jet). Esta heterogeneidad debe ser tomada en cuenta especialmente en la selección/definición de técnicas de testing de caja blanca.
- Construir un buen conjunto de modelos a utilizar como entrada de una transformación para el propósito de testear transformaciones es un problema difícil. Por ejemplo, si los modelos de entrada son instancias de un metamodelo particular (UML, UML con perfiles) que pueden ser restringidos por una serie de reglas bien formadas, deberían contener al menos una instancia para cada clase del metamodelo. Claramente, esto no es viable, pero es posible crear automática o manualmente un conjunto de modelos de entrada, lo más pequeños posibles, pero al mismo tiempo siendo una buena representación de todo el espacio de entrada.
- Escribir casos de prueba y manejar modelos de entrada y salida, metamodelos y programas de transformaciones de modelos requiere un soporte de herramientas, especialmente en el caso de una cadena de transformaciones, donde varios lenguajes de transformación de modelos pueden ser utilizados. Todas estas herramientas deberían integrarse lo máximo posible, minimizando los problemas de interoperabilidad.

Actualmente, no existen estándares o propuestas bien establecidas disponibles para el testeo de transformaciones, especialmente en el caso de M2C y cadenas de transformaciones. Esto puede deberse al hecho de que es un problema difícil de resolver.
En [39], se introduce el concepto de *MeDMT*, refiriendo a “Métodos para desarrollar transformaciones de modelos”. Este especifica:

- Cómo definir los requerimientos de transformación de modelos, especificando:
    - El dominio de la transformación, como una clase específica de modelos UML bien formados escritos utilizando un perfil específico, y conforme a un conjunto de reglas bien formadas dada
    - El codominio de la transformación, como artefactos textuales estructurados, por ejemplo, un conjunto de archivos de texto organizados en una jerarquía de directorios específica
    - La definición informal de la correspondencia entre los elementos del dominio y el codominio
- Cómo diseñar la transformación de modelos, siguiendo una arquitectura específica y utilizando una notación específica.

6. Herramientas para el modelado, traducción y verificación en Eclipse

Dado que el modelo y el diagrama de clases a realizar, las restricciones a especificar en estos, y el código que generará automáticamente los casos de prueba en base a tal modelo se escribirán todos utilizando Eclipse, debemos contar con una introducción de las herramientas que el mismo nos provee para poder especificar cada uno de estos.

6.1 Eclipse Modeling Framework (EMF)[37]

Set de plug-ins que pueden utilizarse para modelar un modelo de datos y generar código u otra salida basada en tal modelo. Posee una distinción entre el metamodelo y el modelo real: el metamodelo describe la estructura del modelo; un modelo es una instancia concreta de tal metamodelo. Provee un framework que puede incluirse para almacenar la información del modelo, que utiliza por defecto un formato de datos llamado XMI (*intercambio de metadatos xml*) para persistir los datos del modelo. Permite al desarrollador crear metamodelos por diferentes vías, por ejemplo: XMI, anotaciones Java, UML, schemas XML, etc.

La información almacenada en un modelo EMF puede utilizarse para generar una salida derivada. Un típico uso de EMF es el de especificar metadatos que representen el modelo de dominio de nuestra aplicación y utilizar funcionalidad EMF para generar las clases de implementación Java correspondientes a partir de ese modelo. El framework EMF alega que el código generado puede ser extendido sin peligros a mano.

EMF posee un metamodelo, formado por dos partes: los archivos de descripción *ecore* y *genmodel*. El archivo *ecore* contiene la información acerca de las clases definidas. El archivo *genmodel* contiene información adicional para la generación de código, como el path y la información del archivo; además, contiene el parámetro de control de cómo el código debe ser generado.

El archivo *ecore* permite definir los siguientes elementos:

- *Eclass*: representa una clase, con cero o más atributos y cero o más referencias.
- *EAtribute*: representa un atributo que posee un nombre y un tipo.
- *EReference*: representa un final de una asociación entre dos clases.
- *EDataType*: representa el tipo de un atributo (int, float, Date).

Con EMF uno define su modelo de dominio explícitamente, ayudando a proveer una vista clara del modelo. El generador de código de los modelos EMF puede ajustarse. Provee funcionalidad de notificación de cambios al modelo en caso de que el mismo cambie. EMF genera interfaces y una fábrica para crear nuestros objetos, ayudando a mantener la aplicación limpia de clases de implementación individuales. Otra ventaja es que uno puede regenerar el código Java desde el modelo en cualquier momento.

6.2 Herramientas para el Desarrollo de Modelos MDT

Eclipse cuenta además con un conjunto de herramientas para el desarrollo de modelos, incluidas en lo que se denomina *Model Development Tools* (desde ahora, MDT) [38]. MDT es un proyecto que se enfoca en el modelado, cuyo objetivo es:

- Proveer una implementación de los metamodelos estándares de la industria

- Proveer herramientas ejemplares para desarrollar modelos basados en tales metamodelos

A continuación, destacamos sus herramientas principales.

BPMN2**:** componente de código abierto del subproyecto MDT, que provee una implementación de metamodelos basada en la especificación OMG del Modelado de Procesos de Negocios y Notación (BPMN) 2.0. Los objetivos de sus componentes son los de proveer:

- Una implementación de referencia de código abierto de la especificación de BPMN 2.0
- Una fundación basada en EMF en la que las herramientas de modelado de procesos de negocios puedan ser construidas
- Una base para integrar e intercambiar artefactos entre herramientas de modelado de procesos de negocios
- Un foro para unir a la comunidad en validación de la especificación de BPMN 2.0
- Una oportunidad para incrementar la colaboración entre Eclipse y OMG

IMM**:** componente de código abierto del subproyecto MDT que provee implementaciones de metamodelos y perfiles, basado en la especificación OMG IMM (metamodelo de manejo de información). Sus objetivos son los de proveer:

- Una implementación de referencia de código abierto de la especificación IMM
- Una fundación basada en EMF en la que las herramientas de modelado para el manejo de información puedan ser construidas
- Una base para integrar e intercambiar artefactos entre herramientas de modelado del manejo de información

MoDisco**:** provee un framework para desarrollar herramientas dirigidas por modelos para la modernización del software. Su objetivo es soportar casos de uso como aseguramiento de calidad (verificar si un sistema existente cumple la calidad requerida), documentación (extraer información de un sistema existente para ayudar a entender un aspecto de tal, como la estructura, comportamiento, persistencia, impacto del cambio), mejora (transformación de un sistema existente para integrar mejores normas de codificado o patrones de diseño) y migración (transformar un sistema existente para cambiar un componente, el framework, el lenguaje o su arquitectura). Sus objetivos principales son los de proveer:

- Metamodelos para describir sistemas existentes
- Descubridores para crear modelos de estos sistemas automáticamente
- Herramientas genéricas para entender y transformar modelos complejos creados a partir de sistemas existentes
- Casos de uso que ilustren cómo MoDisco puede soportar el proceso de modernización

OCL (a utilizar): implementación del estándar OCL de OMG para modelos basados en EMF. Provee las siguientes capacidades para soportar integración de OCL:

- Define APIs para parsear y evaluar restricciones OCL y queries en Ecore o modelos UML
- Definir implementaciones Ecore y UML del modelo de sintaxis abstracto OCL, incluyendo soporte para la serialización de las expresiones OCL parseadas.
- Proveer una API de visitantes para analizar/transformar el modelo AST en expresiones OCL

- Proveer una API extensible para clientes para personalizar el ambiente de parseo y evaluación utilizados por el parseador.

Papyrus (a utilizar): componente del subproyecto MDT que busca proveer un ambiente integrado y consumible por el usuario para editar cualquier tipo de modelo EMF, y particularmente soportar UML y los lenguajes de modelado relacionados tales como MARTE. Papyrus provee editores de diagramas para los lenguajes de modelado basados en EMF como UML2, y la chance de integrar estos editores (sean basados en GMF o no) con otras herramientas. También ofrece un soporte muy avanzado de perfiles UML que permite al usuario definir editores para DSLs basados en el estándar UML2 y sus mecanismos de extensión.
Su característica principal, en relación al último punto mencionado, es un set de mecanismos de personalización muy poderosos que pueden ser utilizados para crear perspectivas de Papyrus definidas por el usuario, dándole el mismo aspecto y sintiéndose como un editor DSL.

6.3 Acceleo

Acceleo es un proyecto de código abierto, licenciado bajo la licencia pública de Eclipse (EPL), gratuito. Fue diseñada para los desarrolladores de las tecnologías MDA con el objetivo de incrementar su productividad de desarrollo de software. Permite la generación de archivos utilizando UML, MOF y módulos EMF. Posee una integración completa tanto con Eclipse como con el framework EMF, sincronización de código y modelo, generación incremental, fácil actualización y manejo de templates, coloreo de sintaxis, auto-completado y detección de errores. Claramente, requiere poseer un conocimiento previo tanto en Java, como en modelado (por ejemplo, UML) y en la plataforma Eclipse.
Tras instalar la herramienta en Eclipse, uno debe crear un proyecto con el tipo de transformación de modelo a código Acceleo. En este proyecto, se crearán las plantillas de generación de código, en el formato *.mtl*. Este tipo de proyectos incluye un *builder* para compilar este tipo de archivos, que luego serán utilizados para generar el código (un error común es que no se encontró el archivo *"mtl"*, debiéndose esto a que el *builder* no compiló al mismo).

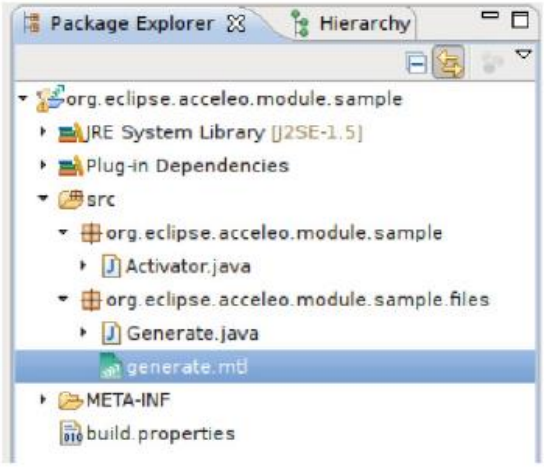


Figura 6.1 La herramienta crea su propio árbol, con un template por defecto (generate.mtl) en el lenguaje de templates Acceleo

Luego de generar el proyecto (que genera la estructura de directorios observada en 5.4), contamos con una perspectiva especial de Eclipse para ver el mismo. En el archivo *.mtl* generado, definiremos las reglas de generación que Acceleo utilizará para generar código.

El contenido de tal archivo es sólo una propuesta, teniendo la libertad de modificarlo a gusto. Sin embargo, antes de continuar, debemos entender qué significan y representan sus elementos.

Figura 6.2 Código por defecto del archivo generate.mtl

Hagamos un análisis línea por línea de este archivo:

1. Primero debemos notar que los comandos de generación en Acceleo comienzan con el símbolo "[" y terminan con "/]" o bien con "[/nombrecomando]", esto es análogo a xml pero con un símbolo distinto. En la primera línea vemos el uso del comando "comment", el cual nos permite poner comentarios que el compilador ignorará. Sin embargo en este caso se usa para un propósito distinto que es el de especificar el encoding en el cual el parser de Acceleo interpretará el archivo mtl.
2. En la siguiente linea vemos la definición del módulo, con su nombre "generate" y la indicación de que usaremos el meta-modelo de uml. En Acceleo los módulos son un mecanismo para organizar los trabajos complejos de generación en "módulos" mas pequeños. En este trabajo, utilizaremos sólo un módulo, el cual corresponde al archivo "generare.mtl"
3. Luego vemos la definición de un *template*. Los *templates* son los que realmente hacen el trabajo de generar código. Podemos definir todos los *templates* que queramos e invocarlos desde nuestro *módulo* o desde un *módulo* diferente. En este caso tenemos un *template* que es "public" es decir, se puede invocar desde otros *templates*. Luego vemos el nombre del template "generateElement" y sus parámetros, en este caso es un solo parámetro y es de tipo "Class" y se llama "aClass".
    1. Notamos también que se está usando el comando "comment" dentro de este template. Ese comentario le dice al generador que ese es el primer template que debe invocar. Además el "Launcher" de eclipse que se usa para realizar la generación de código es generado dinámicamente para cada módulo que tenga este comentario.
    2. En la siguiente linea vemos el comando "file" el cual genera un archivo en donde se guardará el texto generado, este comando recibe los siguientes comandos:
        1. "aClass.name": nombre del archivo. Notarán que será justamente el nombre de la clase.
        2. "false": el archivo no se abrirá en modo "append" para cada generación, es decir, en cada generación de código los contenidos anteriores se perderán.
        3. "UTF-8": el encoding del archivo.

Para utilizar este código, debemos crear el modelo. El modelo es donde se encuentran los datos que utilizaremos para generar el código. En nuestro caso el modelo será un modelo UML con restricciones OCL, escrito utilizando la herramienta Papyrus.
Tras elaborar el modelo, debemos ingresar los datos que queremos usar para la generación. Lo primero es crear un “Package” que actuará como padre de las “Classes”. Estas últimas serán las que se transformaran en código generado.
Por último, contamos con el Launcher de Acceleo, quien es el encargado de preparar el ambiente de generación e invocar al template con el atributo “main”. Tras correr al mismo, se genera el código resultante.

7. Implementación de la solución

Ya teniendo en claro el poder y el alcance de los lenguajes de especificación formal y su aplicación en los tradicionales modelos de datos, procederemos a su utilización en nuestro caso de estudio. Como mencionamos previamente, el lenguaje formal a utilizar será OCL, por la gran disponibilidad de herramientas que el mismo posee.
En este capítulo, llevaremos a cabo la traducción de código automática a partir de un modelo de datos conteniendo restricciones formales en el lenguaje OCL, generando las clases correspondientes junto a sus casos de prueba, y un archivo OCL que contenga todas estas restricciones de manera centralizada. Dadas las ventajas explicadas de Alloy con respecto a OCL, se utilizará además una herramienta que permite traducir el código OCL realizado a su código Alloy correspondiente, para luego analizar este formalmente.
Para esto, definiremos primero nuestro modelo de datos. El mismo ha de ser desarrollado con la herramienta de Eclipse Papyrus, previamente mencionada.

7.1 Creando el diagrama de clases con Papyrus

La herramienta de modelado Papyrus es la herramienta estándar de Eclipse para este propósito. Como en cualquier otro ambiente, los estándares han proveído tradicionalmente mayores estímulos al progreso tecnológico, permitiendo además una gran independencia. La página oficial de esta herramienta se encuentra en [44].
Una vez instalado Papyrus, tenemos la chance de crear un modelo de datos, como vemos en la figura 7.1:

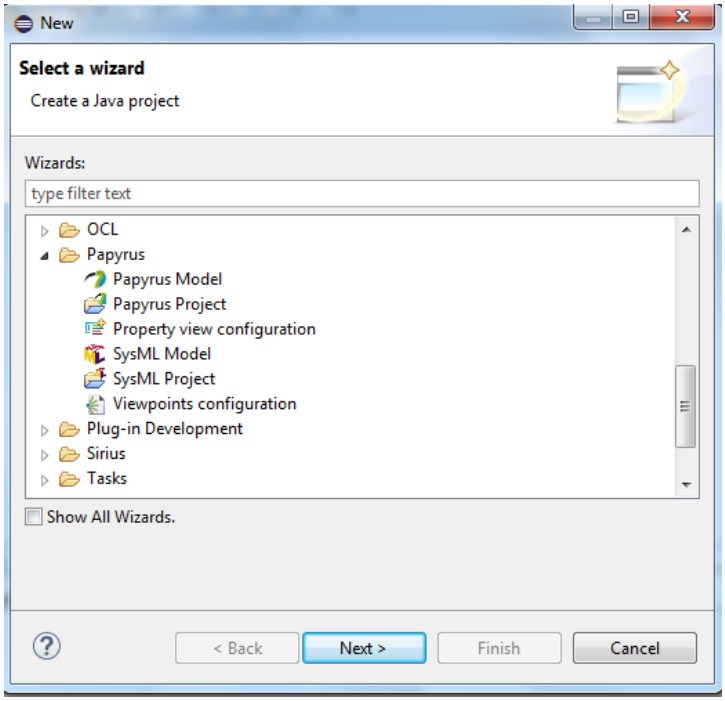


Figura 7.1 Creando un modelo Papyrus

En nuestro caso, crearemos un proyecto Papyrus, que nos crea por defecto un diagrama de clases UML en tres formatos:

- Visual: vista de modelo tradicional
- Anotaciones XML

- Árbol de directorios.

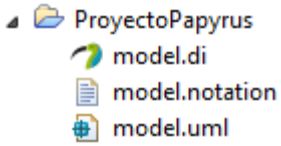


Figura 7.2 Proyecto Papyrus

Nuestro foco estará en el archivo con extensión .di, desde donde podremos crear un diagrama de clases tradicional. Cabe aclarar que la herramienta permite también generar otros tipos de diagramas, como diagramas de secuencia, casos de uso, etc., pero en nuestro caso, nos conformaremos con el diagrama de clases.

Una vez que nos situamos en el archivo .di, se nos ofrece una paleta de elementos que podemos incluir en el diagrama, visibles en la figura 7.3, situada en la derecha. Vemos que estos elementos se dividen en dos grandes categorías, tomando al modelo como un grafo:

- Nodos: elementos principales del modelo, tales como clases, propiedades (atributos), definición de tipos de datos, restricciones, interfaces, etc.
- Aristas: permiten relacionar los nodos de diversas formas, con asociaciones que permiten recibir distintas cardinalidades, generalizaciones, dependencias, etc.

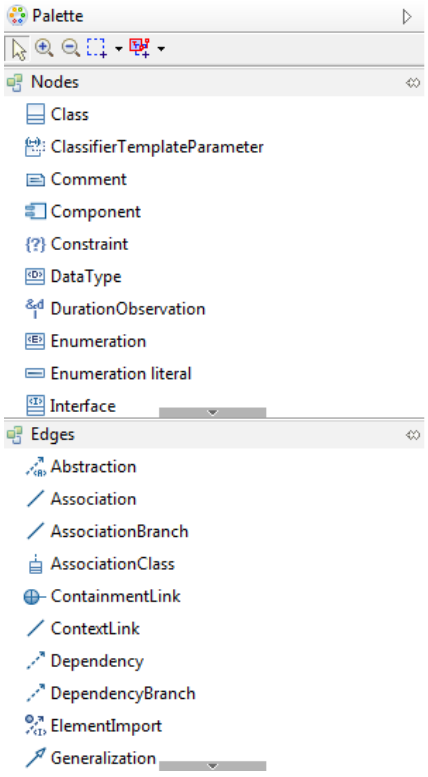


Figura 7.3 Paleta Papyrus

Procederemos entonces a definir nuestro modelo de datos. Este representará las clases constituyentes de una institución universitaria, conteniendo entre otros a Estudiantes, Profesores, Materias, Carreras y Planes de carreras. En la figura 7.4 podemos ver la realización del modelo.

Además de lo mencionado, observamos las restricciones OCL (elemento *Constraint* de la paleta), que representan invariantes, asociadas a clases particulares. Por ejemplo, este modelo no permite a un estudiante estar inscripto en más de una carrera, reflejado en la siguiente invariante OCL:

Context Student
self.careers->size()<=1

Por otro lado, y representando una invariante un poco más compleja, tenemos que para que un profesor dicte una materia, el mismo debe contar con la especialidad en el área en que esta se encuentra, representado con la siguiente invariante:

Context Subject
self.teachers->forAll(o | o.specialties->includes(self.area))

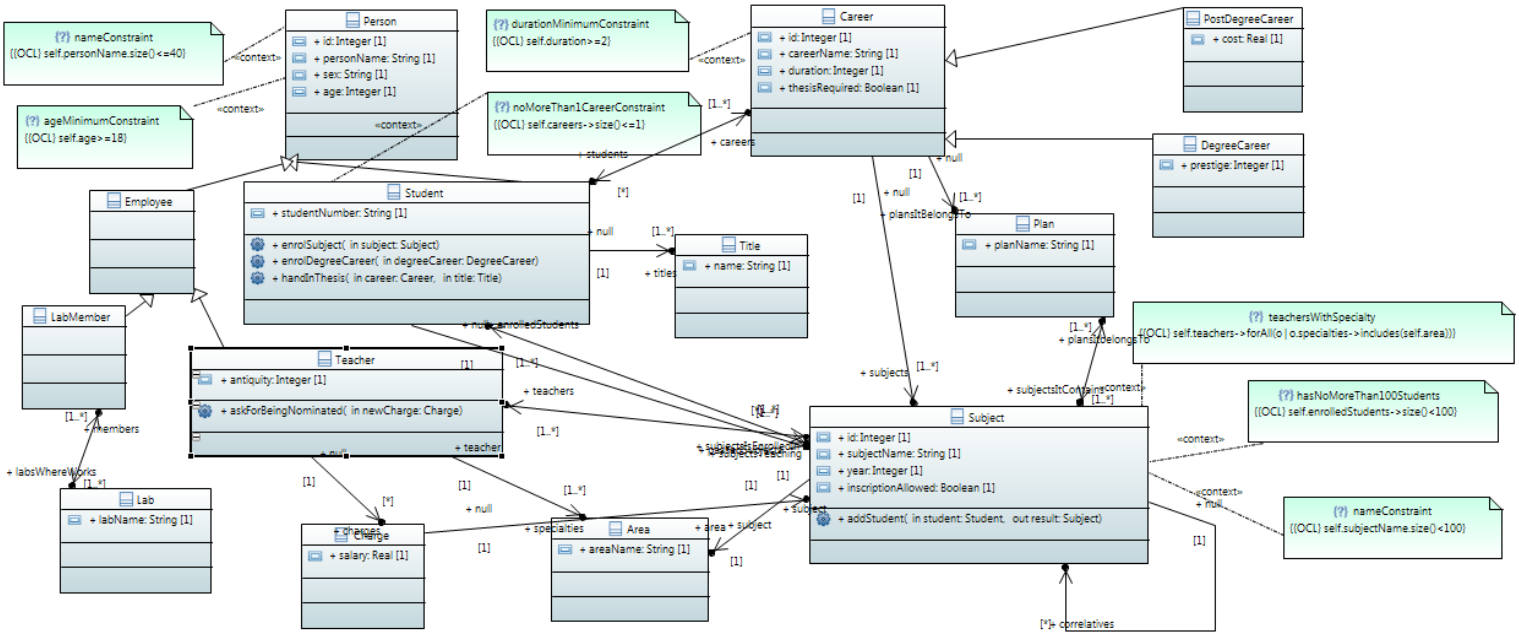


Figura 7.4 Diagrama de clases del caso de estudio

En el modelo tan sólo podemos observar el cuerpo de las restricciones mencionadas. Para especificar el contexto de un invariante en Papyrus, asociamos a la misma a una clase del modelo mediante una flecha punteada, como se puede observar. Dependerá del futuro código a generar que estos invariantes se cumplan en todo momento, o se informe al menos de lo contrario.
Existen otras restricciones OCL en el modelo, no visibles a simple vista, las cuales representan las pre y post condiciones de las operaciones definidas en él. Por ejemplo, para la operación *enrolSubject(subject)* de la clase Student, la cual inscribe al alumno a una materia, existe una precondición OCL denominada *hasPassedItsCorrelatives* (ha aprobado las correlativas), que restringe el atributo *subjectsIsEnrolledIn* (materias en las cuales está inscripto) de la clase y se vale de su propiedad *passedSubjects* (materias que ha aprobado), especificando lo siguiente:

self.passedSubjects->includesAll(subject.correlatives)

Especificamos entonces que para que un estudiante se inscriba en una materia, el mismo debe haber aprobado antes todas sus correlativas. Además se define otra precondición que chequea que la inscripción a tal materia esté habilitada, llamada *inscriptionAllowed*:

subject.inscriptionAllowed=true

Luego, definimos dos postcondiciones para el método. La primera se denomina *isEnrolledInSubject*, quien chequea que la materia especificada realmente se haya agregado a la colección, con el siguiente cuerpo:

self.subjectsIsEnrolledIn->includes(subject)

La segunda, denominada *adds1Subject*, especifica:

self.subjectsIsEnrolledIn->size()=self.subjectsIsEnrolledIn@pre->size()+1

Esta última chequea que el tamaño de la colección de materias inscriptas se incremente en uno. La expresión *self.subjectsIsEnrolledIn@pre* representa a la colección del objeto previa a su modificación.
Nuevamente, dependerá de la calidad del código resultante chequear que se cumplan tanto las pre como postcondiciones al ejecutar los métodos.

7.2 Código de Traducción Acceleo

7.2.1 Introducción

Ya contando con nuestro modelo de datos, lo que resta es crear el código de traducción necesario para traducir el diagrama de clases mencionado a código Java. Para esto, nos valdremos de otra de las herramientas mencionadas previamente: Acceleo.
Acceleo es una herramienta diseñada para los desarrolladores de tecnologías MDA, con el objetivo de incrementar su productividad en el desarrollo de software. Está integrada con el entorno de Eclipse y el framework EMF. Su referencia de operaciones se encuentra en [45].
Una vez instalado, tenemos distintas chances de creación Acceleo, visibles en la figura 7.5. En nuestro caso, crearemos un proyecto Acceleo, debiendo elegir el Tipo de Meta-Modelo que usaremos (en este caso, UML). Notar que también se deben elegir las opciones “Generate file” y “Main template”. Esto nos creará la estructura de directorios visible en la figura 7.6, conteniendo:

- 2 clases Java: *Activator.java* y *Generate.java*. Estos archivos son de configuración, especificando librerías incluidas entre otras cosas, y no deben ser modificadas por el usuario, a menos que este sea un usuario avanzado. En nuestro caso, dejaremos sus valores por defecto.
- Un módulo Acceleo denominado *generate.mtl*: en este archivo es donde explayaremos el código de traducción. En un principio, el código por defecto de este archivo es el observado en la figura 7.7. La lógica de su sintaxis ya fue analizada y explicada en el capítulo previo.

Lógicamente, debemos elegir un modelo UML a partir del cual generar las clases correspondientes, por lo que en la configuración de Acceleo, seleccionamos al modelo creado previamente como modelo origen. Lo que resta definir ahora es el código en sí.

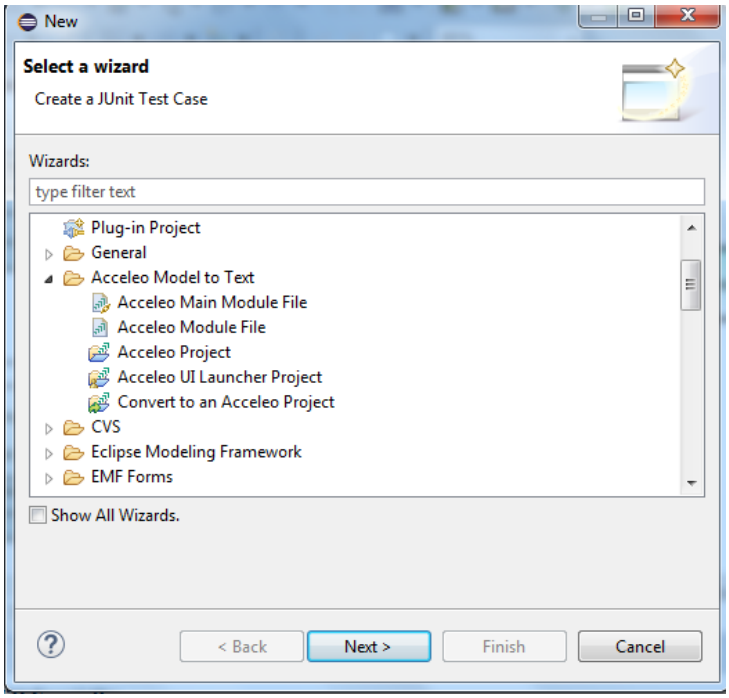


Figura 7.5 Creando un módulo Acceleo

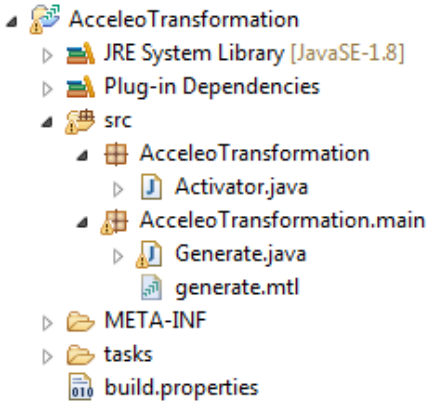


Figura 7.6 Estructura de directorios de un proyecto Acceleo

generate.mtl

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/uml2/3.0.0/UML')/]

[template public generate(aClass : Class)]

    [comment @main /]
    [file (aClass.name, false, 'UTF-8')]
    [aClass.name/]
    [/file]

[/template]
```

Figura 7.7 Código por defecto del archivo generate.mtl

7.2.2 Código Acceleo Final

Dado que el código generado es largo y no es el objetivo del trabajo analizarlo en detalle por completo, nos enfocaremos tan sólo en algunas partes del mismo, pero sobre todo en su análisis. Cabe destacar que el mismo se adjuntará en formato digital junto con este informe.

Su funcionamiento, a grandes rasgos, se basa en recorrer todas las clases del modelo UML tomado como origen, y para cada una crear una clase .java con su nombre, y otra clase TestNombreDeLaClase.java que la testee. Además, se creará el Test de integración, quien corre en un solo paso a todos los Tests generados y devuelve su veredicto, y el archivo University.ocl, quien cuenta con las restricciones OCL definidas asociadas a su contexto, centralizadamente. Dado que Acceleo sólo permite elegir una carpeta como destino, todos los archivos resultantes se incluirán en la misma carpeta/paquete results.

En la figura 7.8 podemos observar el inicio del código. La plantilla *generateElement* se ejecutará para cada clase del modelo; en cada una de ellas, se crea un archivo NombreClase.java, especificado por la palabra clave file. En base a la clase, se define si esta es abstracta, subclase o regular. Luego, vemos cómo se invoca a otro método dentro de Acceleo, llamando a *generarDefAtributos*, quien, definido como otro template, definirá los atributos de la clase, con el modificador de acceso protected en todos los casos, para permitir acceder a estos desde todo el paquete y así poder utilizarlos libremente desde sus tests.

```
[template public generateElement(aClass : Class)]
[comment @main/]
[file (aClass.name+'.java', false, 'UTF-8')]
package results;
import java.util.ArrayList;
import java.util.Collection;

[if (aClass.isAbstract)]
	public abstract class [aClass.name/] {
[else]
	[if (aClass.isLeaf)]
		public class [aClass.name/] extends [aClass.superClass.name/]{
	[else]
		public class [aClass.name/] {
	[/if]
[/if]

	[generarDefAtributos(aClass)/]
```

Figura 7.8 Inicio del código Acceleo

Para cada clase, se generará una clase interna, que representará un chequeador de la primera. Esta Consiste en dos métodos, denominados *cumpleInvariantes(objetoDeLaClase)* y *cumpleCondicion(condición).* El objetivo de este chequeador es utilizarlo ante el pedido de actualización de una instancia de la clase, para asegurar que sus invariantes sigan valiendo.

```
[if (aClass.isLeaf)]  public class [aClass.name/]Checker extends [dameClaseRaiz(aClass)/]Checker{
[else]   public class [aClass.name/]Checker{
[/if]
	public [aClass.name/]Checker(){}

	[if (aClass.isLeaf)]
	public boolean cumpleInvariantes([dameClaseRaiz(aClass)/] [aClass.name.toLowerFirst()/]In){
		[aClass.name/] [aClass.name.toLowerFirst()/] =
([aClass.name/])[aClass.name.toLowerFirst()/]In;
	[else]
	public boolean cumpleInvariantes([aClass.name/] [aClass.name.toLowerFirst()/]){
	[/if]
	/** metodo que define si una instancia de la clase respeta sus invariantes **/
```

```
		[if (aClass.ownedRule->asSet()->union(aClass.inheritedMember-
>selectByType(Constraint))->isEmpty())]
		return true; //NO POSEE INVARIANTES
		[else]
		try{
			[if (aClass.ownedRule->asSet()->isEmpty())]
			if([for (c: Constraint | aClass.inheritedMember->selectByType(Constraint))
separator('&&')]
			([OCLInvariant2Java(c.specification.eGet('body')-
>first(),aClass.name.toLowerFirst())/])
			[/for]
			[else]
			if([for (c: Constraint | aClass.ownedRule->asSequence()) separator('&&')]
				([OCLInvariant2Java(c.specification.eGet('body')-
>first(),aClass.name.toLowerFirst())/])
			[/for]
			[for (c: Constraint | aClass.inheritedMember->selectByType(Constraint))
separator('')]
			&& ([OCLInvariant2Java(c.specification.eGet('body')-
>first(),aClass.name.toLowerFirst())/])
			[/for]
			[/if]
			) return true;
			else return false;
		}catch(NullPointerException e){
			return false;
		}
		[/if]
	}

	public boolean cumpleCondicion(boolean condicion){
		return condicion;
	}
```

Figura 7.9 Generación del Chequeador

La utilidad del método *cumpleCondicion* se verá luego.
Luego, se define el constructor de la clase, recibiendo como argumento a todos los atributos de la misma. Si asignáramos los mismos sin previa verificación, podríamos violar alguno de los invariantes de este contexto; por ejemplo, si creamos un estudiante con una colección de carreras que contiene dos o más de estas, estaríamos violando el invariante de que un estudiante puede estar inscripto en a lo sumo una carrera. En cambio, lo que se hace es asignarle todos estos atributos a un objeto de prueba, y chequear si el mismo cumple los invariantes a través del chequeador; en caso positivo, le asignamos entonces los atributos al objeto; caso negativo, devolvemos un objeto con los valores de sus atributos por defecto (valiéndonos de otro método creado automáticamente aquí, denominado *setearValoresPorDefecto*).
Después, se podrán observar dos métodos que servirán de auxiliares para utilizar sobre todo en las pruebas, y por qué no, en un futuro por el usuario del sistema. Estos son *guardarEstado* y *restituirEstado*, quienes sirven para resguardar el estado actual de una instancia de la clase en una copia, y para devolverle a una instancia un estado previo, correspondientemente.
A continuación, definimos los getters y setters de los atributos (esto es, los métodos que permiten accederlos o establecerlos de manera correcta). En cuanto a los primeros, no representan modificación alguna al getter tradicional; pero en cuanto a los segundos, volvemos al problema de que si se ejecutan libremente, podemos llegar a violar los invariantes de la clase. Es por ello que antes de ejecutarlos y en cada uno, guardamos el estado corriente del objeto en una copia; luego ejecutamos el cuerpo del setter, y tras este, chequeamos que el objeto aún respete sus invariantes; en caso negativo, devolvemos al objeto su estado anterior, con *restituirEstado*.
Tras esto, definimos los métodos que fueron definidos en el modelo para esta clase.

```
[template public generarMetodos(aClass : Class) ]
         [for (o: Operation | aClass.ownedOperation) separator('\n')]
        public void [o.name/]([escribirParametros(o)/]) {
                  [aClass.name/] previo = this.guardarEstado();
                  [if (o.precondition->isEmpty()._not())]
                  if(this.getChecker().cumpleCondicion(
                  [for (c: Constraint | o.precondition->asSequence()) separator('&&')]
                        ([OCLPrePost2Java(c.specification.eGet('body')->first(),aClass.name)/])
                  [/for])){
                  [/if]
                  [OCLBody2Java(o.bodyCondition.specification.eGet('body')->first())/];
                         if(!(this.getChecker().cumpleInvariantes(this))){
                           this.restituirEstado(previo);
                         }
                  [if (o.precondition->isEmpty()._not())]}[/if]
        }
   [/for]
[/template]
```

Figura 7.10 Generando los métodos de las clases

En la especificación del método, antes del cuerpo en sí, se genera en primera instancia una copia del objeto, denominada previo. Luego, chequeamos que se cumplan las precondiciones del mismo; caso contrario, se termina su ejecución, sin alterar las propiedades del objeto. Si se cumplen estas precondiciones, se ejecuta el método (el método *OCLBody2Java* traduce el cuerpo OCL a su código Java correspondiente), y tras esto, se chequea que este no haya hecho que el objeto viole sus invariantes; en caso de que las viole, se devuelve al objeto su estado previo, valiéndonos de la copia creada, sin tener efecto el método.

En cuanto a los Tests de cada clase, estos también representan clases java, que extienden de la clase especial *TestCase*, lo que permite que sus métodos sean testeados a través de la librería *JUnit* (cabe aclarar que este proyecto debe incluir las librerías necesarias para extender esta clase y utilizar luego *Mockito* para mockear objetos).

```
[file ('Test'+aClass.name+'.java', false, 'UTF-8')]
package results;

import java.util.*;
import static org.mockito.Mockito.*;
import results.*;
import static org.junit.Assert.assertTrue;
import org.junit.*;
import junit.framework.TestCase;

public class Test[aClass.name/] extends TestCase{
         [aClass.name/] [aClass.name.toLowerFirst()/], result;
         [aClass.name/].[aClass.name/]Checker checker;

         @Before
         public void setUp() throws Exception{
                   super.setUp();
                   [aClass.name.toLowerFirst()/] = new [aClass.name/]();
                   checker = mock([aClass.name/].[aClass.name/]Checker.class);
            }

         @Test
         public void testInitialize() {
                  assertTrue([aClass.name.toLowerFirst()/] != null);
                  assertTrue(checker != null);
         }

         /** Idea general al testear los métodos:
         Se le asocia a la instancia de la clase un mock de un chequeador, al que le exigiremos distintos
comportamientos.
         1) Establecemos que el objeto cumple las precondiciones del método, y las invariantes de la
clase siempre (esto incluye, tras ejecutar el cuerpo del método), a través de su chequeador; entonces,
deben cumplirse las postcondiciones.
```

```
        2) Establecemos que el objeto cumple las precondiciones del método, pero nunca las invariantes
de la clase (lo que incluye, tras ejecutar el cuerpo del método), a través de su chequeador; entonces, NO
deben cumplirse las postcondiciones.
        3) Establecemos que el objeto NO cumple las precondiciones del método; cumpla o no las
invariantes de la clase tras ejecutar         su cuerpo, NO deben cumplirse las postcondiciones. **/
        [generarMetodosPrueba(aClass)/]


        @After
        protected void tearDown() throws Exception {
                super.tearDown();
                [aClass.name.toLowerFirst()/] = null;
                checker=null;
        }

}
[/file]
```

Figura 7.11 Generando los casos de prueba

Estas clases definen primero dos instancias de la clase a testear, uno con su nombre y otro llamado *result*, y luego una instancia del chequeador de la clase a prueba. Luego, define el método *setUp*, quien inicializa al objeto con el nombre de la clase con sus valores por defecto, y al chequeador como un mock de esa clase de chequeadores; esto nos permitirá luego exigirle un comportamiento determinado. El método *setUp* es el primero en ser ejecutado, procedente a cualquier test de la clase; esto se especifica con la anotación *@Before* previa a su definición.
A esto le sucede el primer caso de prueba de la clase, denominado *TestInitialize*, el cual testea que los atributos definidos no sean nulos. Es un método auxiliar no muy útil. Como cualquier método de prueba a utilizar con JUnit, el mismo posee previo a su definición la anotación *@Test,* que le permite al framework reconocer que debe ser testeado.
Luego, pasamos a los tests de los métodos definidos en el modelo para cada clase. Dado que este código es largo, procederemos a explicarlo. Estos métodos de prueba se dividen en dos submétodos: uno que testea que el método se ejecute correctamente, y otro que chequea que el mismo no se ejecute por no cumplir alguna condición.
En principio, en ambos submétodos, se crea una instancia utilizable de cada parámetro del método. Luego, se reinicializa a *result*, a quien luego se le aplicará el método, cuyas postcondiciones serán chequeadas comparando al objeto tras ser o no modificado por el método (*result*) con el objeto previo a tal posible cambio (el inicializado en el *setUp* con el nombre de la clase). A *result* le asignamos como chequeador a nuestro chequeador definido previamente.

En el primer submétodo, si queremos que el método se ejecute correctamente, debemos asegurarnos que se cumplan sus precondiciones, y que tras su cuerpo los invariantes no sean violados, con la siguiente estructura:

Asegurar que se cumplan las precondiciones
Asegurar que se cumplan los invariantes tras el cuerpo del método
Ejecutar el método
Verificar que se cumplan las postcondiciones

Aquí surge la necesidad de los mocks. Un mock es un objeto *"tonto"* representando una clase existente, al que se le exige un determinado comportamiento, y el mismo lo cumplirá al solicitárselo. A través del mock del chequeador de la clase creado, podemos pedirle al mismo que cuando se chequeen determinadas precondiciones de un método, este devuelva que se cumplen; lo mismo para los invariantes de la clase. Para ello, utilizamos la siguiente sintaxis:

when(checker.cumpleCondicion(precondiciones)).thenReturn(true/false);
when(checker.cumpleInvariantes(result)).thenReturn(true/false);

Aquí entendemos la necesidad del método *cumpleCondicion*, quien nos permite establecer si las precondiciones del método se cumplirán en un momento determinado o no.
Tras especificar lo anterior, una vez que preguntemos por esa condición, obtendremos que se cumple o no en base a lo escrito tras el *thenReturn*.
Luego, invocamos el método a testear sobre result. Resta después chequear que este satisfaga las postcondiciones. Para esto, contamos con la sintaxis especial de *JUnit*:

assertTrue(condición); // assertFalse(condición);

Esta sentencia es la que determinará si el test pasa o no.
En el segundo caso, tenemos dos chequeos, con la siguiente estructura:

Parte 1
Asegurar que se cumplan las precondiciones
Asegurar que NO se cumplan los invariantes tras el cuerpo del método
Ejecutar el método
Verificar que NO se cumplan las postcondiciones

Parte 2
Asegurar que NO se cumplan las precondiciones
Verificar que NO se cumplan las postcondiciones

Como vemos, en el segundo caso no interesa si se cumplen o no las invariantes; si las precondiciones no se satisfacen, entonces el método no debe ejecutarse.
Por último, se define el método *tearDown*, quien marca como *null* a los objetos previamente creados. Este método se ejecutará último, marcado así con la anotación *@After*.

Ya con el código definido y analizado, el siguiente paso es su ejecución. Para ello, realizamos un click derecho sobre el archivo generate.mtl, y seleccionamos la opción *Run As > Launch Acceleo Application*. Esto nos generará las clases y los casos de prueba que

especificamos en el código, y en la carpeta que especificamos, que en este caso se denomina "results", y se encuentra en el proyecto FinalCode.

Figura 7.12 Archivos generados tras ejecutar el código Acceleo

7.3 Análisis de los resultados obtenidos

Como vemos, tras la ejecución del archivo generate.mtl, se generaron las clases .java correspondientes más el archivo *.ocl,* visibles en la figura 7.12.

Una vez que cumplimos nuestro objetivo de generar las clases junto a sus casos de prueba correspondientes, el siguiente paso es chequear que tales casos de prueba funcionan. Para ello, corremos a través de JUnit el Test de Integración (quien, como recordemos, ejecuta en un solo paso a todos los otros Tests), y observamos su veredicto.

```
package results;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
import junit.framework.TestCase;

@RunWith(Suite.class)
@SuiteClasses({TestEmployee.class,TestPlan.class,TestLabMember.class,
               TestTeacher.class,TestSubject.class, TestPerson.class,
         TestLab.class,TestCharge.class,TestDegreeCareer.class,
         TestStudent.class,TestPostDegreeCareer.class,
         TestCareer.class,TestTitle.class,TestArea.class})
public class TestIntegracion extends TestCase{

}
```

Figura 7.13 Código del Test de Integración

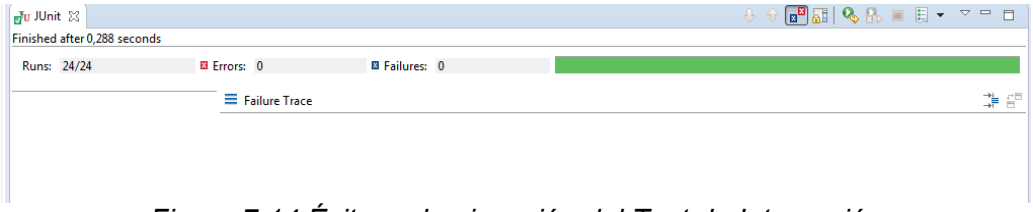


Figura 7.14 Éxito en la ejecución del Test de Integración

Para cerrar, veremos el código generado para la clase Student junto a su Test, de manera de ilustrar el resultado de la traducción acceleo en Java.

```
package results;
import java.util.ArrayList;
import java.util.Collection;
public class Student extends Person{
         protected ArrayList<Career> careers;
         protected ArrayList<Subject> passedSubjects;
         protected String studentNumber;
         protected ArrayList<Subject> subjectsIsEnrolledIn;
         protected ArrayList<Title> titles;

         public class StudentChecker extends PersonChecker{
```

```
	public StudentChecker(){}

	public boolean cumpleInvariantes(Person studentIn){
		Student student = (Student)studentIn;
		/** metodo que define si una instancia de la clase respeta sus invariantes **/
		try{
			if((student.careers.size()<=1) &&
				(student.personName.length()<=40) &&
				 (student.age>=18))
				 return true;
			else return false;
		}catch(NullPointerException e){
			return false;
		}
	}

	public boolean cumpleCondicion(boolean condicion){
		return condicion;
	}
}

public Student (int age, ArrayList<Career> careers, int id, ArrayList<Subject> passedSubjects, String
personName, String sex, String studentNumber, ArrayList<Subject> subjectsIsEnrolledIn, ArrayList<Title>
titles){
	/** metodo que crea una instancia de la clase con los parametros recibidos, en caso de que estos
	hagan que la misma cumpla los invariantes de la clase; caso contrario, todos los atributos se
	inicializan por defecto **/
	Student prueba = new Student();
	prueba.age=age;
	prueba.careers=careers;
	prueba.id=id;
	prueba.passedSubjects=passedSubjects;
	prueba.personName=personName;
	prueba.sex=sex;
	prueba.studentNumber=studentNumber;
	prueba.subjectsIsEnrolledIn=subjectsIsEnrolledIn;
	prueba.titles=titles;
	StudentChecker checker = new StudentChecker();
	if(checker.cumpleInvariantes(prueba)){
		this.age=prueba.getAge();
		this.careers=prueba.getCareers();
		this.id=prueba.getId();
		this.passedSubjects=prueba.getPassedSubjects();
		this.personName=prueba.getPersonName();
		this.sex=prueba.getSex();
		this.studentNumber=prueba.getStudentNumber();
		this.subjectsIsEnrolledIn=prueba.getSubjectsIsEnrolledIn();
		this.titles=prueba.getTitles();
		this.checker=checker;
	} else this.setearValoresPorDefecto();
}

public Student() {
	  this.setearValoresPorDefecto();
}

public Student guardarEstado(){
	/** metodo que copia el estado actual de un objeto en otro **/
	Student previo = new Student();
	previo.age = this.getAge();
	previo.careers = new ArrayList<Career>();
	for (Career career: this.getCareers())
		previo.careers.add(career);
	previo.id = this.getId();
	previo.passedSubjects = new ArrayList<Subject>();
	for (Subject subject: this.getPassedSubjects())
		previo.passedSubjects.add(subject);
	previo.personName = this.getPersonName();
	previo.sex = this.getSex();
	previo.studentNumber = this.getStudentNumber();
	previo.subjectsIsEnrolledIn = new ArrayList<Subject>();
	for (Subject subject: this.getSubjectsIsEnrolledIn())
		previo.subjectsIsEnrolledIn.add(subject);
	previo.titles = new ArrayList<Title>();
	for (Title title: this.getTitles())
		previo.titles.add(title);
	return previo;
}
public void restituirEstado(Student previo){
```

```
                /** metodo que restituye el estado de un objeto previo en el invocador **/
                this.titles=previo.getTitles();
                this.passedSubjects=previo.getPassedSubjects();
                this.age=previo.getAge();
                this.id=previo.getId();
                this.studentNumber=previo.getStudentNumber();
                this.sex=previo.getSex();
                this.subjectsIsEnrolledIn=previo.getSubjectsIsEnrolledIn();
                this.careers=previo.getCareers();
                this.personName=previo.getPersonName();
        }

        public void setearValoresPorDefecto(){
                /** metodo que setea los atributos de la instancia con sus valores por defecto **/
                super.age = 0;
                super.id = 0;
                super.personName = "";
                super.sex = "";
                this.careers = new ArrayList<Career>();
                this.passedSubjects = new ArrayList<Subject>();
                this.studentNumber = "";
                this.subjectsIsEnrolledIn = new ArrayList<Subject>();
                this.titles = new ArrayList<Title>();
                this.checker = new StudentChecker();
        }

        public ArrayList<Career> getCareers() {
                   return this.careers;
        }

        [.. RESTO DE LOS GETTERS ..]

        /** en cada setter, se chequea que tras setear la propiedad, se respeten los invariantes;
        caso contrario, no se modifica tal propiedad **/
        public void setCareers(ArrayList<Career> careers) {
                 Student previo = this.guardarEstado();
                 this.careers = careers;
                 if(!(this.getChecker().cumpleInvariantes(this)))
                         this.restituirEstado(previo);
        }

        [.. RESTO DE LOS SETTERS ..]


        public void enrolSubject(Subject subject) {
                  Student previo = this.guardarEstado();
                  if(this.getChecker().cumpleCondicion(
                          (this.passedSubjects.containsAll(subject.correlatives)) &&
                          (subject.inscriptionAllowed==true)
                   )){
                          this.subjectsIsEnrolledIn.add(subject);
                          if(!(this.getChecker().cumpleInvariantes(this)))
                            this.restituirEstado(previo);
                   }
        }

        public void enrolDegreeCareer(DegreeCareer degreeCareer) {
                  Student previo = this.guardarEstado();
                  this.careers.add(degreeCareer);
                  if(!(this.getChecker().cumpleInvariantes(this)))
                          this.restituirEstado(previo);
        }

        public void handInThesis(Career career,Title title) {
                  Student previo = this.guardarEstado();
                  if(this.getChecker().cumpleCondicion(
                           ((career.subjects.stream().filter(o ->
                           this.passedSubjects.contains(o)).count()==career.subjects.size())) &&
                           (career.thesisRequired==true)))
                 {
                          this.titles.add(title)      ;
                          if(!(this.getChecker().cumpleInvariantes(this)))
                                   this.restituirEstado(previo);
                   }
        }

}
```

Figura 7.15 Clase Student generada

```
package results;

import java.util.*;
import static org.mockito.Mockito.*;
import results.*;
import static org.junit.Assert.assertTrue;
import org.junit.*;
import junit.framework.TestCase;

public class TestStudent extends TestCase{
	Student student, result;
	Student.StudentChecker checker;

	@Before
	public void setUp() throws Exception{
		/** metodo que mockea cada atributo de tipo no primitivo de la clase,
				mas una instancia de la clase conteniendo estos mocks**/
		  super.setUp();
		  student = new Student();
		  checker = mock(Student.StudentChecker.class);
	}

	@Test
	public void testInitialize() {
		assertTrue(student != null);
		assertTrue(checker != null);
	}

	/** Idea general al testear los metodos:
	Se le asocia a la instancia de la clase un mock de un chequeador, al que le exigiremos distintos
comportamientos.
	1) Establecemos que el objeto cumple las precondiciones del metodo, y las invariantes de la clase siempre
(esto incluye, tras ejecutar el cuerpo del metodo), a traves de su chequeador; entonces, deben cumplirse las
postcondiciones.
	2) Establecemos que el objeto cumple las precondiciones del metodo, pero nunca las invariantes de la
clase (lo que incluye, tras ejecutar el cuerpo del metodo), a traves de su chequeador; entonces, NO deben
cumplirse las postcondiciones.
	3) Establecemos que el objeto NO cumple las precondiciones del metodo; cumpla o no las invariantes de la
clase tras ejecutar su cuerpo, NO deben cumplirse las postcondiciones. **/


	@Test
	public void testTrueEnrolSubject() {
		Subject subject = new Subject();
		result = new Student();
		result.setChecker(checker);

		/** PARTE 1 **/
		/** CUMPLE PRECONDICIONES **/
		when(checker.cumpleCondicion(
			(result.passedSubjects.containsAll(subject.correlatives)	) &&
				(subject.inscriptionAllowed==true)
		)).thenReturn(true);

		/** CUMPLE INVARIANTES SIEMPRE **/
		when(checker.cumpleInvariantes(result)).thenReturn(true);

		/** EJECUTO EL METODO A TESTEAR SOBRE EL OBJETO **/
		result.enrolSubject(subject);

		/** DEBE CUMPLIR LAS POSTCONDICIONES **/
		assertTrue((result.subjectsIsEnrolledIn.size()==student.subjectsIsEnrolledIn.size()+1) &&
			(result.subjectsIsEnrolledIn.contains(subject)));
	}

	@Test
	public void testFalseEnrolSubject() {
		Subject subject = new Subject();
		result = new Student();
		result.setChecker(checker);

		/** PARTE 2 **/
		/** CUMPLE PRECONDICIONES **/
		when(checker.cumpleCondicion(
			(result.passedSubjects.containsAll(subject.correlatives)	) &&
				(subject.inscriptionAllowed==true)
		)).thenReturn(true);

		/** NO CUMPLE INVARIANTES NUNCA **/
```

```
                when(checker.cumpleInvariantes(result)).thenReturn(false);

                /** EJECUTO EL METODO A TESTEAR SOBRE EL OBJETO **/
                result.enrolSubject(subject);

                /** NO DEBE CUMPLIR LAS POSTCONDICIONES **/
                assertFalse((result.subjectsIsEnrolledIn.size()==student.subjectsIsEnrolledIn.size()+1) &&
                (result.subjectsIsEnrolledIn.contains(subject)));

                /** PARTE 3 **/
                result = new Student();
                result.setChecker(checker);

                /** NO CUMPLE PRECONDICIONES **/
                when(checker.cumpleCondicion(
                         (result.passedSubjects.containsAll(subject.correlatives)        ) &&
                                  (subject.inscriptionAllowed==true)
                )).thenReturn(false);

                /** EJECUTO EL METODO A TESTEAR SOBRE EL OBJETO **/
                result.enrolSubject(subject);

                /** NO DEBE CUMPLIR LAS POSTCONDICIONES **/
                assertFalse((result.subjectsIsEnrolledIn.size()==student.subjectsIsEnrolledIn.size()+1) &&
                (result.subjectsIsEnrolledIn.contains(subject)));
        }


        [.. RESTO DE LOS TESTTRUE Y TESTFALSE DE LOS MÉTODOS ..]

        @After
        protected void tearDown() throws Exception {
                super.tearDown();
                student = null;
                checker=null;
        }
}
```

Figura 7.16 Clase de prueba TestStudent generada

7.4 Traduciendo de OCL a Alloy – Verificación formal

Como vimos, nuestro modelo contaba con restricciones OCL, las cuales fueron traducidas a su respectivo código Java; este código resultante provee las pruebas y métodos necesarios para testear y asegurar que el modelo se mantenga consistente con el paso del tiempo. Pero dada la investigación realizada, se cree necesaria la verificación formal del modelo, esto es, no a nivel Java sino a nivel OCL, y, por qué no, a nivel Alloy, quien permite generar contraejemplos que ayudan a revelar las inconsistencias del modelo.
Recordamos entonces que nuestra traducción generaba, además de las clases java, un archivo .ocl con todas las restricciones descriptas en este lenguaje en el modelo original. El contenido del mismo se puede observar en la figura 7.17.
Dado que Java incluye el manejo de este lenguaje en sus librerías, en este caso, al utilizar EMF, podemos realizar un chequeo en cuanto a la consistencia del modelo a nivel OCL, visible en las figuras 7.18 y 7.19. Pero en el caso de inconsistencias, el informe de los errores es de calidad pobre; además, este chequeo se basa específicamente en la sintaxis OCL, más que en la consistencia en sí.
Siendo que el objetivo del trabajo es contar con una verificación completa, tanto a nivel de código Java como formal, utilizaremos una herramienta denominada AlloyMDA [46], que permitirá traducir el código OCL con el que contamos a su respectivo código Alloy, a partir del cual podremos contar con el verificador de tal lenguaje, quien nos permite visualizar contraejemplos claros, y que presenta las ventajas mencionadas en capítulos anteriores ante OCL.

```
package results

                context Teacher::askForBeingNominated(newCharge:Charge)
                        pre: self.specialties->includes(newCharge.subject.area)
                        pre: self.antiquity>2
                        body: self.charges->union(Bag{newCharge})
                        post: self.charges->size()=self.charges@pre->size()+1
                        post: self.charges->includes(newCharge)
                context Subject
                        inv: (self.enrolledStudents->size()<100)
                        inv: (self.teachers->forAll(o | o.specialties->includes(self.area)))
                        inv: (self.subjectName.size()<100)
                context Subject::addStudent(student:Student)
                        pre: self.inscriptionAllowed=true
                        body: self.enrolledStudents->union(Bag{student})
                        post: self.enrolledStudents->size()=self.enrolledStudents@pre->size()+1
                context Person
                        inv: (self.personName.size()<=40)
                        inv: (self.age>=18)
                context Student
                        inv: (self.careers->size()<=1)
                context Student::enrolSubject(subject:Subject)
                        pre: self.passedSubjects->includesAll(subject.correlatives)
                        pre: subject.inscriptionAllowed=true
                        body: self.subjectsIsEnrolledIn->union(Bag{subject})
                        post: self.subjectsIsEnrolledIn->size()=self.subjectsIsEnrolledIn@pre->size()+1
                        post: self.subjectsIsEnrolledIn->includes(subject)
                context Student::enrolDegreeCareer(degreeCareer:DegreeCareer)
                        body: self.careers->union(Bag{degreeCareer})
                        post: self.careers->size()=self.careers@pre->size()+1
                        post: self.careers->includes(degreeCareer)
                context Student::handInThesis(career:Career,title:Title)
                        pre: career.subjects->forAll(o | self.passedSubjects->includes(o))
                        pre: career.thesisRequired=true
                        body: self.titles->union(Bag{title})
                        post: self.titles->size()=self.titles@pre->size()+1
                context Career
                        inv: (self.duration>=2)
endpackage
```

Figura 7.17 Código OCL centralizado generado

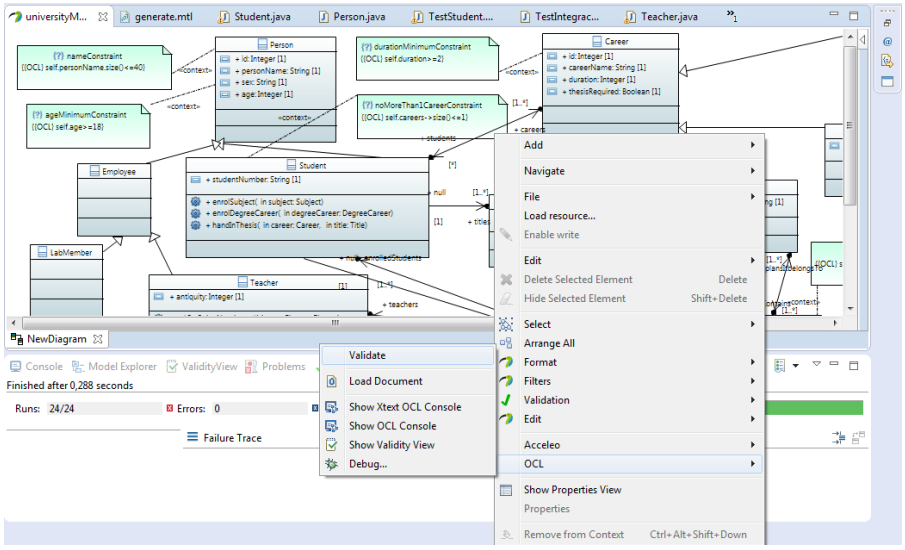


Figura 7.18 Validando el modelo a nivel OCL

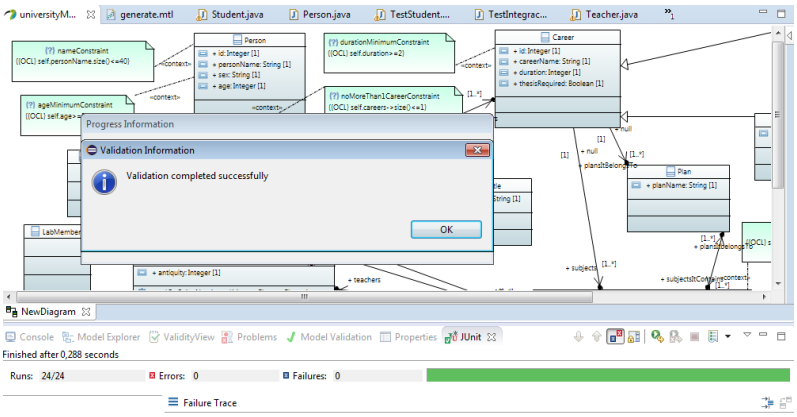


Figura 7.19 Éxito en la validación

La herramienta AlloyMDA puede descargarse de [46] gratuitamente. Está desarrollada en Haskell, y provee cuatro tipos de transformaciones posibles:

- Alloy a Diagrama de Clases
- Alloy a OCL
- Diagrama de Clases a Alloy
- OCL a Alloy

En nuestro caso, nos valdremos de la última opción para traducir nuestro código OCL a su correspondiente código Alloy.

Dado que la herramienta fue desarrollada como un prototipo para testear un enfoque de traducción, esta tiene varias limitaciones. Por ejemplo, el diagrama de clases a tomar como fuente debe estar muy simplificado; los tipos primitivos (salvo los booleanos) no son considerados, y en OCL también hay consideraciones similares. De acuerdo a los modelos que generamos previamente (*.ocl* y *.uml*), observamos en las figuras 7.20 y 7.21 parte del código del diagrama de clases simplificado y las especificaciones OCL permitidas.

```
package results
                context Teacher::askForBeingNominated(newCharge:Charge)
                         pre: self.specialties->includes(newCharge.subject.area)
                         post: self.charges->size()=self.charges@pre->size()+1
                         post: self.charges->includes(newCharge)
                context Subject
                         inv: (self.enrolledStudents->size()<100)
                context Subject
                         inv: (self.teachers->forAll(o | o.specialties->includes(self.area)))
                context Subject::addStudent(student:Student)
                         pre: self.inscriptionAllowed
                         post: self.enrolledStudents->size()=self.enrolledStudents@pre->size()+1
                context Student
                         inv: (self.careers->size()<=1)
                context Student::enrolSubject(subject:Subject)
                         pre: subject.inscriptionAllowed
                         post: self.subjectsIsEnrolledIn->size()=self.subjectsIsEnrolledIn@pre->size()+1
                         post: self.subjectsIsEnrolledIn->includes(subject)
                context Student::enrolDegreeCareer(degreeCareer:DegreeCareer)
                         post: self.careers->size()=self.careers@pre->size()+1
                         post: self.careers->includes(degreeCareer)
                context Student::handInThesis(career:Career,title:Title)
                         pre: career.subjects->forAll(o | self.passedSubjects->includes(o))
                         post: self.titles->size()=self.titles@pre->size()+1
endpackage
```

Figura 7.20 Modelo OCL simplificado a utilizar con AlloyMDA. Podemos notar la supresión de algunas expresiones, especialmente de aquellas que tratan con atributos de tipos primitivos como enteros y Strings

```
<?xml version='1.0' encoding='UTF-8'?>
 <uml:Package xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:uml="http://www.eclipse.org/uml2/3.0.0/UML" name='University'>

 <packagedElement xmi:type='uml:Class' xmi:id='Area' name='Area' isAbstract='false' >

 </packagedElement>

 <packagedElement xmi:type='uml:Class' xmi:id='Career' name='Career' isAbstract='false' >

          <ownedAttribute xmi:type='uml:Property' xmi:id='subjects' name='subjects' isReadOnly='true'>
                    <type xmi:idref="Subject"/>
                                        <upperValue xmi:type='uml:LiteralUnlimitedNatural' value='-1' />
                                        <lowerValue xmi:type='uml:LiteralInteger' value='1' />
           </ownedAttribute>

 </packagedElement>

 <packagedElement xmi:type='uml:Class' xmi:id='DegreeCareer' name='DegreeCareer' isAbstract='false' >

          <generalization xmi:type='uml:Generalization' general='Career' />

</packagedElement>

 <packagedElement xmi:type='uml:Class' xmi:id='Plan' name='Plan' isAbstract='false' >

 </packagedElement>

 <packagedElement xmi:type='uml:Class' xmi:id='Charge' name='Charge' isAbstract='false' >

           <ownedAttribute xmi:type='uml:Property' xmi:id='subject' name='subject' isReadOnly='true'>
                    <type xmi:idref="Subject"/>
                                        <upperValue xmi:type='uml:LiteralUnlimitedNatural' value='1' />
                                        <lowerValue xmi:type='uml:LiteralInteger' value='1' />
           </ownedAttribute>
 </packagedElement>

 <packagedElement xmi:type='uml:Class' xmi:id='Person' name='Person' isAbstract='false' >

 </packagedElement>
```

Figura 7.21 Porción del código del Modelo UML a utilizar con AlloyMDA. Vemos por ejemplo en la clase Career que se suprimieron la mayor parte de sus atributos presentes en el modelo original

Ya poseyendo nuestros dos archivos requeridos para generar el archivo *.als* (ejecutable por el chequeador de Alloy), procedemos a utilizar la herramienta AlloyMDA para generarlo. Para esto, tras localizarnos en la carpeta raíz de la herramienta y por consola, ejecutamos la siguiente línea:

$ runghc Tools/OCL2Alloy < OCL/University.ocl CD/University.uml

Esto resultará en el código Alloy que buscábamos, impreso por consola, como observamos en la figura 7.22. Una vez que contamos con este código, lo trasladamos a un archivo *.als* nuevo, denominado *University.als*. Analicemos un poco este código (en la figura 7.23 puede observarse con mayor grado de precisión).
Lo primero que podemos observar es la palabra clave *module*; esta describe el comienzo de un nuevo modelo.
Luego, vemos repetidamente la expresión *sig*, abreviatura de *signature*, quien representa a un conjunto o set de objetos (similar a una clase Java). Estas *signatures* pueden o no tener un conjunto de atributos; *Career*, por ejemplo, posee el atributo *subjects*, al que le sigue la expresión *some Subject*. Definimos aquí el significado de esta y otras palabras,

Figura 7.22 Corriendo AlloyMDA sobre nuestros archivos

```
module University
sig Time {}

sig Area {}

sig Career {subjects : some Subject}

sig DegreeCareer extends Career {}

sig Plan {}

sig Charge {subject : one Subject}

sig Person {}

sig Student extends Person {subjectsIsEnrolledIn : Subject some -> Time,passedSubjects : set Subject,titles :
                    set Title,careers : Career some -> Time}

sig Subject {plansItbelongsTo : some Plan,enrolledStudents : some Student,teachers : some  Teacher,correlatives : set
          Subject,area : one Area}

sig inscriptionAllowed in Subject {}
```

```
sig Employee {}

sig Teacher extends Employee {subjectsTeaching : some Subject,specialties : some Area,charges : Charge set -> Time}

sig Title {}

pred askForBeingNominated [self : Teacher,newCharge : Charge,t,t' : Time] {newCharge.subject.area in self.specialties
                                                    #self.(charges.t') = (#self.(charges.t) + 1)
                                                    newCharge in self.(charges.t')}

fact {all t : Time | all self : Subject | #self.enrolledStudents < 100}

fact {all t : Time | all self : Subject | all o : self.teachers | self.area in o.specialties}

pred addStudent [self : Subject,student : Student,t,t' : Time] {self in inscriptionAllowed
                                                #self.enrolledStudents = (#self.enrolledStudents + 1)}

fact {all t : Time | all self : Student | #self.(careers.t) <= 1}

pred enrolSubject [self : Student,subject : Subject,t,t' : Time] {subject in inscriptionAllowed
                                                #self.(subjectsIsEnrolledIn.t') = (#self.(subjectsIsEnrolledIn.t) + 1)
                                                subject in self.(subjectsIsEnrolledIn.t')}

pred enrolDegreeCareer [self : Student,degreeCareer : DegreeCareer,t,t' : Time]
                                                {#self.(careers.t') = (#self.(careers.t) + 1)
                                                degreeCareer in self.(careers.t')}

pred handInThesis [self : Student,career : Career,title : Title,t,t' : Time] {all o : career.subjects | o in self.passedSubjects
                                                                    #self.titles = (#self.titles + 1)}
```

Figura 7.23 Modelo Alloy pasado en limpio

que revelan distintos tipos de multiplicidades:

- *lone*: a lo sumo uno.
- *one*: exactamente uno.
- *none*: ninguno.
- *a in b:* subset o igual.
- *some*: al menos uno.
- *set*: cero o más.
- *all*: todos.

Tenemos entonces que cada carrera tiene al menos una materia, y que *inscriptionAllowed* (que en el modelo original representaba un booleano, pero aquí estos se toman de forma distinta) se declara como un subset de *Subject*.

La expresión *extends* no varía su significado del de Java, extendiendo otra *signature* (clase). A continuación, podemos observar en la *signature Student*, la expresión *subjectsIsEnrolledIn* : *Subject some -> Time*; esto se traduce como una colección de materias en las que el alumno está inscripto (una o más en base a *some*) en un momento determinado. ¿Por qué la aparición del tiempo aquí y no en todas las expresiones? Pues porque esta colección, al igual que otras, será luego alterada por algún método, debiendo luego accederla o solicitarla en sus distintos estados a lo largo de la variable tiempo.

Tras terminar de definir las *signatures*, comienza otro tipo de expresiones encabezadas por la palabra clave *pred*. *pred*, abreviando a *predicate* (predicado), representa la definición de propiedades, devolviendo luego *true* o *false* el analizador si se encuentran o no instancias que satisfagan el predicado. Vendrían a representar nuestros métodos originales, buscando luego instancias donde esos métodos se efectúen con éxito.

fact es nuestra próxima palabra a definir. “Hecho” en inglés, refiere a una restricción que se asume que siempre existirá (lo que significa, un invariante). El primero que vemos, *fact {all t : Time | all self : Subject | #self.enrolledStudents < 100}*, establece que una materia nunca puede superar los 99 alumnos; la expresión *all t:Time* le da ese carácter de invariante, manteniéndose firme en el tiempo.

Ahora que contamos con este código Alloy, podemos aprovechar el uso del Alloy Analyzer para verificar este "submodelo" de nuestro modelo original.
En el capítulo 4, explicamos brevemente cómo funciona esta herramienta. La misma fue desarrollada específicamente para soportar los métodos formales "livianos" (del inglés *lightweight*); como tal, su objetivo es el de proveer un análisis completamente automático, en contraste con las técnicas de testeo de teoremas comúnmente utilizadas en lenguajes de especificación similares. Funciona a través de una reducción a SAT (Problema de satisfactibilidad booleana, que busca determinar si existe una asignación que satisface una fórmula booleana dada), empleando lógica de primer orden que permite trasladar las especificaciones Alloy a expresiones booleanas muy largas que pueden analizarse automáticamente por un SAT solver; de ahí que dada una fórmula lógica en Alloy, su analizador puede intentar encontrar un modelo que la satisfaga. Claramente, la mayor utilidad de la herramienta es encontrar al menos un modelo que no la satisfaga, revelando errores; aquí debemos retomar la frase de Dijkstra mencionada previamente: "El testing revela la presencia de errores, pero no su ausencia". Si bien este analizador es muy potente, siempre existe la chance de que algún error o inconsistencia se siga cometiendo.
Procedemos entonces a utilizar este analizador. Para ello, lo descargamos gratis de [28] (su versión actual es la 4.2) en formato *.jar*, no requiriendo instalación sino simplemente ejecutarlo. Al abrirlo, deberíamos ver una pantalla como la de la figura 7.24.

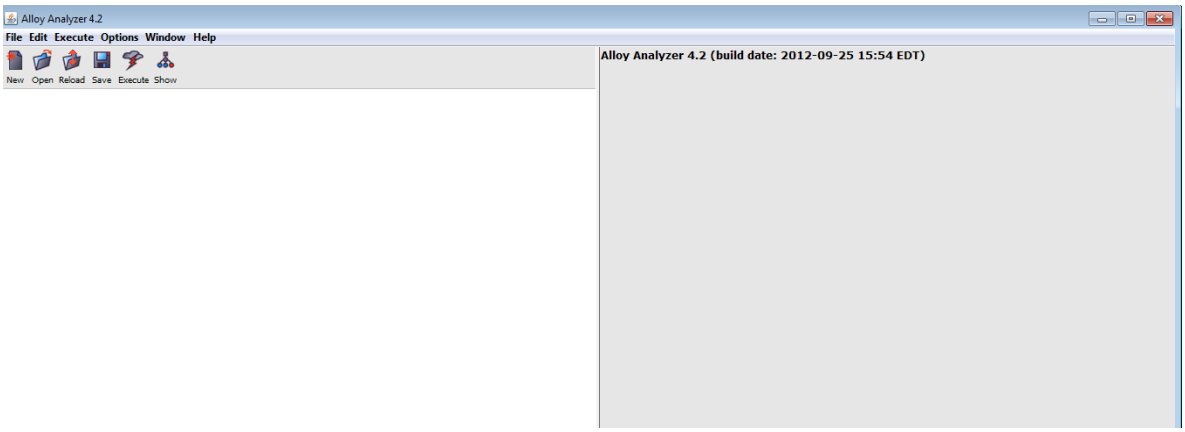


Figura 7.24 Pantalla principal del Alloy Analyzer

Una vez abierto, podemos configurarlo como deseemos, pudiendo elegir entre distintos tipos de SAT solvers, tamaño máximo de memoria y pila a utilizar, si las advertencias son fatales, etc.
Abrimos entonces nuestro archivo *.als*, y para correrlo debemos especificar con el comando especial run los predicados a correr junto con su *scope* (alcance, imponiendo un límite); los errores o ausencia de ellos que encontremos, ocurrirán dentro de este alcance, por lo que fuera de él podrían suceder situaciones distintas. Ejecutaremos el run para observar los ejemplos (si es que existe alguno) que el analizador encuentre para que el/los predicados sean verdaderos. Tenemos entonces que:

- Si se encuentra un ejemplo, el predicado es satisfactible.
- Si no se encuentra ningún ejemplo, el predicado será inválido (falso para todos los ejemplos posibles), o satisfactible, aunque no dentro del alcance especificado.

Especificamos entonces nuestro comando a ejecutar tras el código Alloy:

```
run enrolSubject for 4 but exactly 1 Student, exactly 1 Time
```

En este caso, estaremos testeando el predicado enrolSubject, con un alcance que limitará la búsqueda a aquellas instancias que posean a lo sumo 4 instancias de cada signature, exceptuando Student, quien contará con un solo objeto; además, lo ejecutaremos para sólo una instancia de tiempo, para evitar contar con un modelo resultante muy complejo.
Tras correr el programa Alloy, podemos observar los resultados. En principio, el mensaje devuelto por la consola de la herramienta, visible en la figura 7.25.

```
Starting the solver...

Warning #1
This variable is unused.

Warning #2
This variable is unused.

Note: There were 2 compilation warnings. Please scroll up to see them.

Executing "Run enrolSubject for 4 but exactly 1 Student, exactly 1 Time"
  Solver=minisat(jni) Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
  1674 vars. 214 primary vars. 2536 clauses. 16ms.
  Instance found. Predicate is consistent. 0ms.

Note: There were 2 compilation warnings. Please scroll up to see them.
```

Figura 7.25 Mensaje de la consola del Alloy Analyzer tras correr el programa

En él, podemos observar algunas advertencias irrelevantes, los datos de la configuración del analizador, y si se encontraron o no instancias, en qué tiempo se hizo y el veredicto sobre si el modelo es finalmente consistente o no. En este caso, lo es, y podemos ver la instancia generada clickeando en *Instance*, desplegando la figura 7.26.

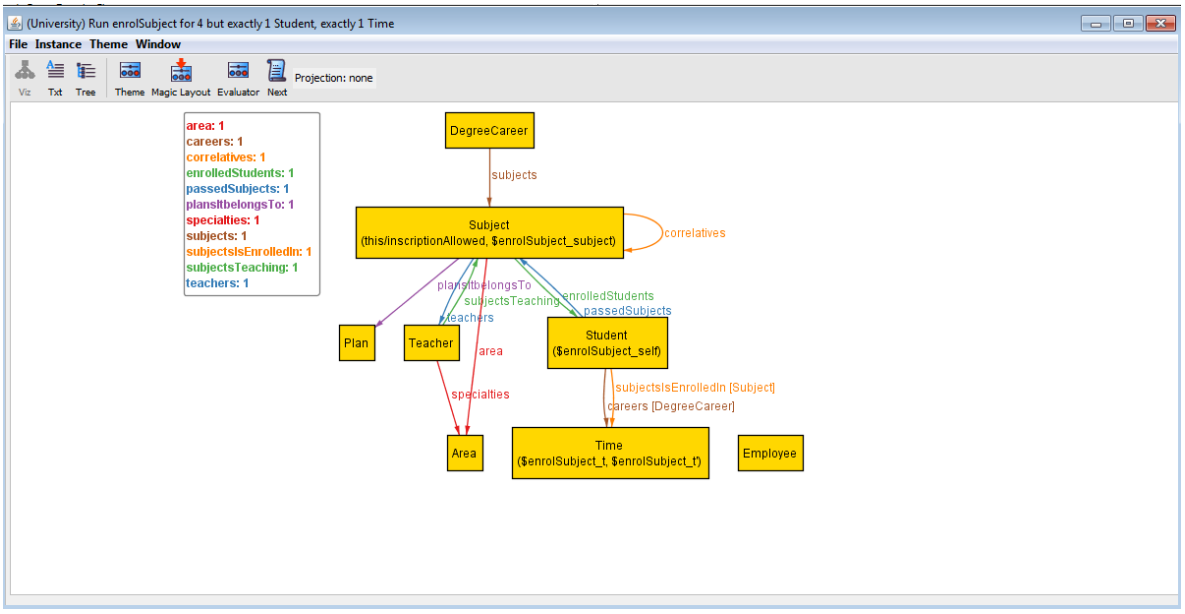


Figura 7.26 Modelo generado por el analizador

Queda a merced del usuario luego el uso de esta herramienta para testear todos y cada uno de los predicados, observando para cada uno los distintos modelos generados.

Buscando inconsistencias y contraejemplos

El verdadero sentido de permitir la especificación formal en nuestro modelo es, no sólo restringir cierto tipo de operaciones y estados, sino contar con un lenguaje lo suficientemente poderoso que permita detectar inconsistencias de manera temprana, segura y explicativa.

El código Java que generamos tras correr el código *Acceleo* asegura que desde su origen, el modelo no va a sufrir violaciones a los invariantes establecidos, y que respetando estos y cumpliendo las precondiciones de un método, se van a cumplir sus postcondiciones; a partir de un modelo fuente, el código sigue al pie de la letra lo que este especifica, pero no nos asegura que no sea inconsistente.

Para ello recurrimos a la herramienta *AlloyMDA*, logrando este objetivo con el *Alloy Analyzer*, quien además de detectar los errores de forma temprana (puede correrse en cualquier momento) y segura (sustentada por sus usuarios y publicaciones), lo hace de manera explicativa, valiéndose de los conocidos contraejemplos. Para ilustrar esto, procederemos a agregar una restricción en el modelo original que lo haga inconsistente.

El modelo fuente especificaba, entre otras cosas, que una materia o *Subject* no podía contar con más de 100 alumnos, especificado esto en el lenguaje OCL con la restricción aplicada a la clase *Subject* titulada *hasNoMoreThan100Students*:

self.enrolledStudents->size() < 100

Supongamos ahora que agregamos una nueva restricción, llamada *hasMoreThan100Students*, a la misma clase, especificando:

self.enrolledStudents->size() > 100

A simple vista se puede observar que esto haría al modelo inconsistente, siendo que ninguna instancia de *Subject* podría contener menos de 100 alumnos y al mismo tiempo más de 100.

Sin embargo, nuestro código Java generado seguirá creándose de la misma manera, y los tests seguirán teniendo éxito (no se chequea si los invariantes producen inconsistencias entre sí, sino que sólo se utilizan objetos mock, obligados a cumplir siempre/nunca sus invariantes), por lo que no es útil para asegurar la consistencia que buscamos.

En cambio, tras nuevamente traducir el código OCL a su código Alloy correspondiente, la situación se modifica. Si ejecutamos nuevamente este código, con la nueva restricción agregada, mediante el comando:

run enrolSubject for 4 but exactly 1 Student, exactly 1 Time

, lo que obtenemos es el resultado de la figura 7.27, en donde ninguna instancia del modelo fue encontrada, y se nos advierte que este puede ser inconsistente. Claro está, en un modelo de gran tamaño, encontrar esta inconsistencia podría costar un tiempo más que considerable.

Para detectar alguna instancia que viole la especificación del modelo con la que contamos, podemos utilizar el comando de *Alloy check*; este comando, dada una aserción, busca la existencia de contraejemplos que permitan observar cómo se viola/n determinado/s *facts*. En nuestro caso, los *facts* van a ser los dos invariantes que mencionamos (recordemos que en *Alloy* estos se traducen como *facts*), y la aserción va a ser una que creemos, denominada *noCollapsedSubjects*, y especificando:

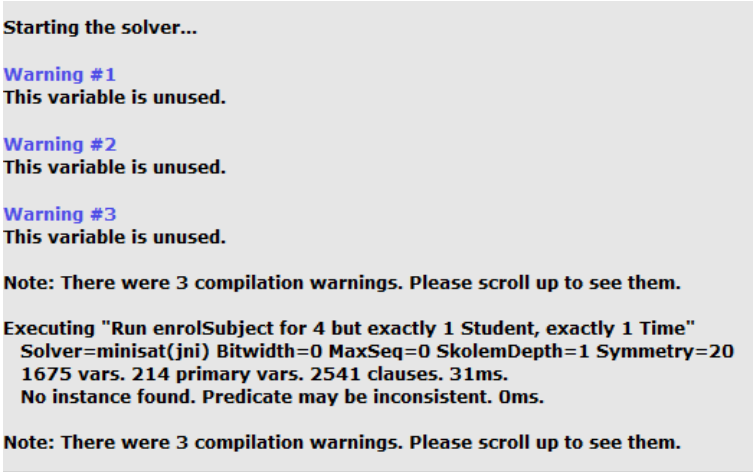


Figura 7.27 Sin instancias del modelo a mostrar

```
assert noCollapsedSubjects {
   no s:Subject | #s.enrolledStudents > 100
}
```

En lenguaje natural, esta aserción especifica que ninguna materia cuenta con más de 100 estudiantes; lo que buscaremos encontrar, dado el modelo *Alloy*, es un contraejemplo en donde alguna materia si los supere. Para chequear esto, debemos ejecutar entonces el comando *check*, invocando a la aserción mencionada:

```
check noCollapsedSubjects for 101 but exactly 1 Subject
```

Dado que esto genera 101 instancias de cada clase, menos de Subject de la cual genera una, el tiempo que le ocupa al analizador buscar contraejemplos es muy alto, por lo que se recomienda probarlo con 5 alumnos en lugar de 100, por ejemplo, como haremos a continuación.
Tras ejecutar entonces el analizador, obtenemos lo siguiente:

Figura 7.28 Se logró detectar un contraejemplo

Como vemos, el analizador logró encontrar un contraejemplo, significando que la aserción es inválida; clickeando en Counterexample, podemos visualizarlo:

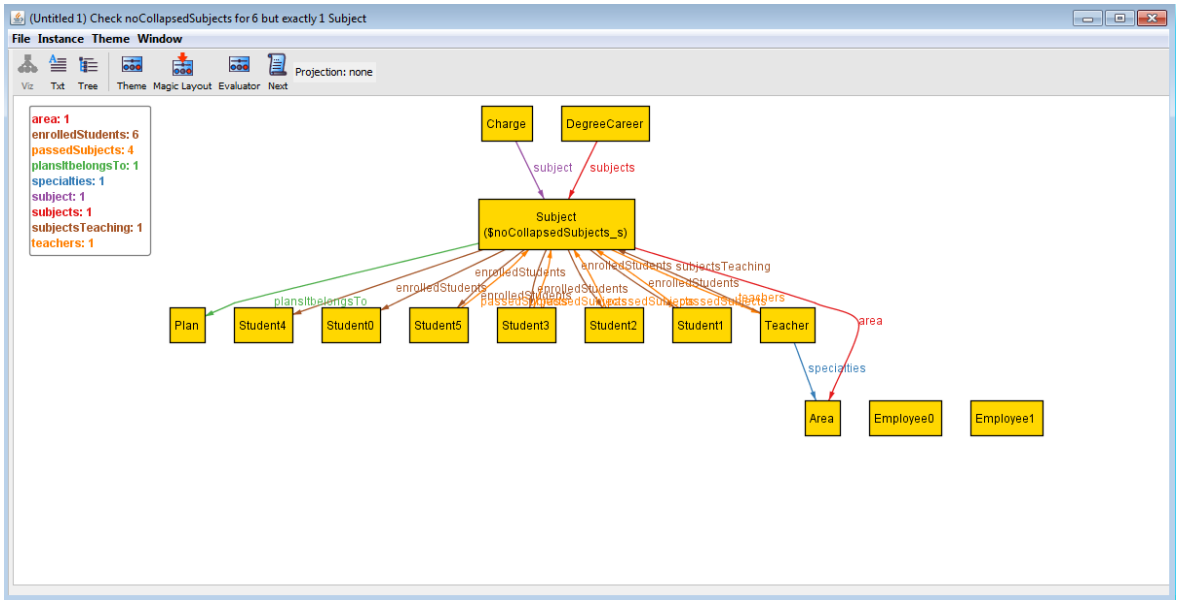


Figura 7.29 Contraejemplo encontrado

Como vemos, encontramos una instancia en donde una materia posee 6 estudiantes, siendo que los invariantes exigen que esta posea obligatoriamente más de 5 y menos de 5, más que la aserción especificaba que ninguna materia podía poseer más de 5; el contraejemplo es claro, y permitirá salvar las inconsistencias del modelo fuente.

7.5 Resumen

A lo largo de este capítulo, pudimos aplicar gran parte de la teoría vista a lo largo de los capítulos previos. Por un lado, definimos un modelo de datos simple como base; a este, le aplicamos restricciones en el lenguaje formal OCL, pasando desde invariantes hasta pre y post-condiciones y cuerpos de métodos. Aquí es donde finalizaría el desarrollo propio del usuario del trabajo realizado; con sólo correr el código Acceleo disponible y que definimos, este modelo se traduce a su código Java correspondiente, con casos de prueba y métodos en las clases que aseguran respetar las restricciones definidas en el modelo original, utilizable por cualquier usuario de esta plataforma y útil para iniciar un nuevo proyecto desde allí. A esto, le adicionamos la verificación formal (para el caso de que el/los usuarios cuenten con conocimiento en el tema y deseen llevar a cabo este tipo de certificación), generando el archivo *.ocl* centralizado y proveyendo las herramientas *AlloyMDA* y *Alloy Analyzer* para traducir este a su código Alloy correspondiente y luego descubrir si este representa un modelo consistente o no, y en caso negativo, poder determinar en qué situación este se hace inconsistente.
En síntesis, en una serie de pasos un usuario Java cualquiera con un leve conocimiento de OCL puede definir un modelo de datos y contar con las herramientas necesarias para testear si este modelo es consistente o no, más utilizarlo convenientemente en tal plataforma con un grado de código avanzado, reutilizable y escalable.

8. Trabajos Relacionados

Finalizado nuestro proyecto, procederemos a encuadrarlo en un contexto de trabajos relacionados que aportan al mismo objetivo de proveer más y mejores herramientas al mundo del MDD; en este caso, aportando a la generación de código automática dentro del entorno de Java y Eclipse, entorno más que ampliamente conocido y utilizado por la comunidad informática.
En este capítulo, hablaremos sobre diversos trabajos, que sirvieron como modelo para el desarrollo del propio y/o fueron realizados con propósitos similares.

8.1 TestEra

TestEra[47] es un framework de testing basado en especificación de programas Java. Para testear un método, se vale de la especificación de las precondiciones de los métodos con el fin de generar entradas de tests y la postcondición para chequear la correctitud de la salida. Su página oficial se puede encontrar en [48].
Este framework introdujo el testeo sistemático de caja negra de programas Java utilizando Alloy como una tecnología que permite un testeo exhaustivo limitado, donde un programa es testeado frente a todas las entradas no equivalentes dentro de un espacio de entrada determinado. Este testeo mostró una gran efectividad a la hora de encontrar bugs en varias aplicaciones que toman tests estructurados de forma compleja.
A diferencia del trabajo realizado en esta tesina, donde se parte pura y exclusivamente de un modelo de datos del cual luego se desprende el código Java, TestEra es un plugin (contando además con una librería requerida para su ejecución) que permite, a partir de código Java ya existente, generar tests más robustos que los existentes con Alloy. Veamos para ello un ejemplo.
Consideremos una clase lista con un método *agregarNodo*, que agrega un valor determinado a una lista de entrada. TestEra cuenta con un conjunto de anotaciones propias para clases y métodos; para una clase Java, los programadores pueden especificar sus invariantes de clase (de manera similar, en nuestro modelo de datos se permite restringir con invariantes OCL) quienes deben ser satisfechos por todas las instancias de la clase.
En la figura 8.1 podemos observar el código de la clase mencionada, especificando dos fórmulas; la primera, que le da la propiedad acíclica a la lista, y la segunda, asegura que su tamaño es igual al número de nodos alcanzables desde su cabezal.

```
@TestEra(invariant={
  "all l: LinkedList |
       all n: l.header.*next | n !in n.^next",
  "all l: LinkedList | l.size = #l.header.*next"} )
public class LinkedList {
  public ListNode header;
  public int size = 0;

  @TestEra(preCondition={"x >= 0",
    postCondition={"this.size' = this.size + 1"},
    runCommand="1 LinkedList, 3 ListNode, 3 int" )
  public void addNode(int x) {
    ListNode n = new ListNode();
    n.value = x;
    n.next = header;
    header = n;
    size++;
  }
}
public class ListNode {
  public int value;
  public ListNode next;
}
```

Figura 8.1 Programa Java de LinkedList

Para un método bajo prueba, TestEra requiere que los programadores especifiquen sus pre y post condiciones, y un límite de tamaño de entrada, utilizado en la ejecución del analizador Alloy. Para el método *addNode* del ejemplo, la precondición establece que el parámetro *x* no es negativo, y la postcondición declara que el tamaño de la lista luego de ejecutar el método se incrementa en uno (muy similar a las especificadas en nuestro trabajo). El límite, definido con la anotación *runCommand,* declara el alcance de

LinkedList, LinkedNode y los enteros primitivos utilizados en los tests a generar.
Basándose en el código Java y las anotaciones asociadas, TestEra puede enumerar automáticamente un conjunto de tests JUnit dentro del alcance especificado por el usuario; la figura 8.2 muestra el test generado para el método *addNode*.

```
@Test
public void test5() {
  // TestEra Auto-Comment: Initialization statements
  LinkedList LinkedList_0 = new LinkedList();
  LinkedList_0.size = 0;
  // TestEra Auto-Comment: Pre-state abstraction
  StateManager sm = new StateManager();
  sm.addToState("LinkedList_0", LinkedList_0);
  sm.generatePreState();
  // TestEra Auto-Comment: Invoke method under test
  LinkedList_0.addNode(0);
  // TestEra Auto-Comment: Post-state checking
  TestEra.checkPostState(
    sm, "dataStructures.list.LinkedList",
    "addNode", "LinkedList_0", "0");
}
```

Figura 8.2 Ejemplo de Test de JUnit

8.2 Modelo de base para UML y verificación OCL

Los lenguajes de modelado como UML y OCL son cada vez más usados en etapas tempranas en el diseño de un sistema. Estos lenguajes ofrecen un inmenso set de construcciones; como consecuencia, los motores de verificación existentes sólo soportan una parte restringida de ellas. En [49], se propone un enfoque usando transformaciones de modelos para unificar diferentes significados de descripciones en un modelo base. En el curso de esta transformación, se expresan construcciones en un lenguaje complejo con un pequeño grupo de los llamados elementos de *core*. Esta simplificación permite interactuar con un amplio rango de motores de verificación con potencias complementarias y debilidades.
El objetivo de los autores es que, guiados por un análisis estructural del modelo base, el desarrollador pueda elegir el motor de verificación más prometedor. La información de diversos diagramas UML se combina en uno solo, reduciendo el set de construcciones de lenguaje al mínimo. Este modelo base provee una interface entre descripciones arbitrarias de modelos UML/OCL y validación y herramientas de verificación.
En la figura 8.3 podemos ver el flujo de proceso al utilizar tal modelo base. La descripción del modelo fuente consiste en varios diagramas UML optimizados con expresiones OCL que especifican el sistema y su comportamiento. Estos diagramas en conjunto son transformados y combinados en un modelo base; utilizando a este, un análisis estructural puede proveer las pistas necesarias para seleccionar un *solver* apropiado. Al elegir a este, el modelo base es transformado en uno específico para ese *solver*, eliminando aquellas construcciones no soportadas explícitamente por este, reemplazándolas por representaciones más simples.
Por ejemplo, en la figura 8.4 vemos como una agregación del modelo fuente se transforma en una asociación con una invariante.

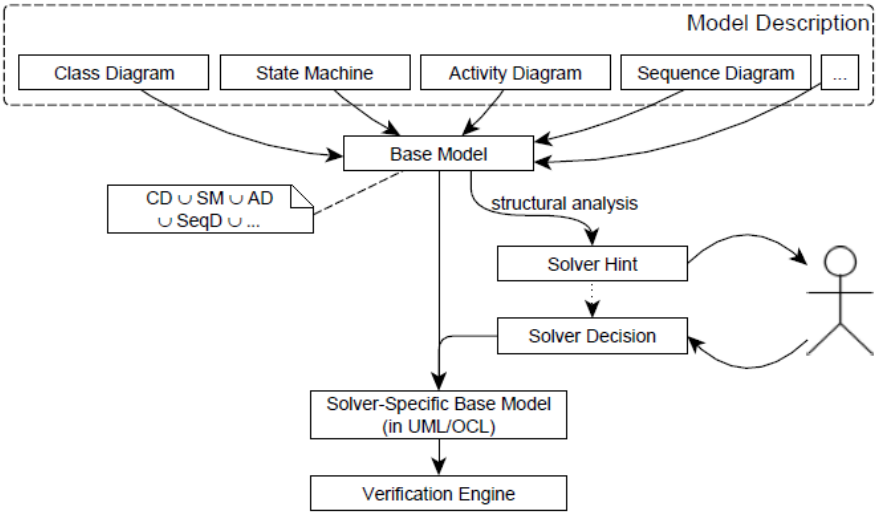


Figura 8.3 Flujo de proceso empleando el modelo base

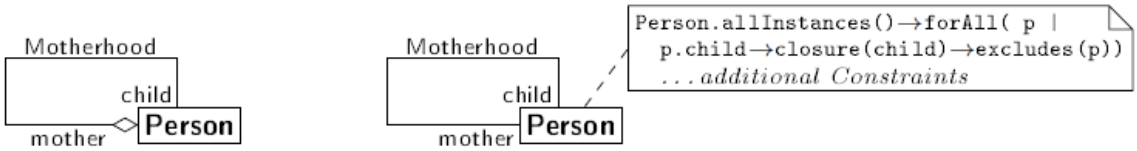


Figura 8.4 Transformación de una agregación en una asociación con una restricción

8.3 Generación de Tests basado en modelos para aplicaciones Web

En [50] se presenta una herramienta para filtrar/configurar los casos de prueba generados dentro del Proyecto de testing basado en modelos PBGT. Los modelos están escritos en un DSL llamado PARADIGM y se componen de patrones de test de UI (UITP) describiendo los objetivos de test. Para generar casos de prueba, el tester debe proveer datos de input de prueba a cada UITP en el modelo. Sin embargo, sin un ***filtro/configuración*** del algoritmo de generación de casos de prueba, el número de test cases puede ser tan grande que se vuelve inmanejable. Por lo que en el paper presentan un enfoque para definir parámetros para la generación de test cases para generar un número manejable de ellos, enfoque evaluado comparando las diferentes estrategias de test y midiendo la performance de la herramienta de modelado contra una herramienta de capture-replay usada para web testing.

Las apps web se están volviendo cada día más importantes. Debido a la estabilidad y seguridad frente a perder datos, la nube y las aps web basadas en ella progresivamente están ganando una base de usuarios más grande. Las apps web pueden manejar ahora tareas que antes sólo podían realizarse por apps desktop, como editar imágenes o crear documentos spreadsheet.

Aunque las apps web tengan tal relevancia en la comunidad, aún sufren de una falta de estándares y convenciones, a diferencia de las apps desktop y mobile, lo que significa que la misma tarea puede implementarse en muchas maneras distintas. Puro HTML, Js, Java y Flash son algunas tecnologías que pueden usarse para implementar una cierta característica en una app web. Esto hace más difícil el testing automático para la Web, inhibiendo el reuso de código de test.

Sin embargo, a los usuarios les gusta tener sentido de conformidad y facilidad, incluso alguna familiaridad, al usar sus apps. Para este propósito, los desarrolladores usan

elementos comunes, como ***patrones de UI***, solución recurrente con soluciones comunes de problemas de diseño. Cuando un user ve una instancia de un patrón UI, puede inferir fácilmente como se supone que la UI se usa. Un buen ejemplo es el patrón Login, usualmente compuesto por 2 text boxes y un botón para enviar el usuario y pass, y su función es permitir al usuario acceder datos privados.
Aunque tengan un comportamiento común, los patrones UI pueden diferir levemente en sus implementaciones en la web. Sin embargo, es posible definir estrategias de test genéricas y reutilizables para testearlos. Es necesario un proceso de configuración para adaptar estas estrategias a las diferentes aplicaciones posibles. Esta es la idea principal detrás del *Pattern-Based GUI Testing*, proyecto en el que se basó este proyecto. En el enfoque PBGT, el usuario crea un modelo de test conteniendo instancias de las mencionadas estrategias de test de UI Test Patterns, para testear ocurrencias de UI Patterns en apps web.
El objetivo de la investigación es proveer diferentes técnicas de generación de test cases desde modelos PBGT para testear apps web.

8.4 Fokus!MBT – Un entorno de modelado de Tests multiparadigma

Los entornos de modelado UML para crear testing basado en modelos suelen ser incómodos de usar y cargan de conocimiento sobre detalles internos de UML a los usuarios; respectivamente, lo hacen con los ingenieros de test. Pero estos últimos son expertos en UML, por lo que la ganancia que implican los enfoques de testing basados en modelos se reduce.
La herramienta *Fokus!MBT* [51], desarrollada por el centro de competencia MOTION de Fraunhofer FOKUS, es un ambiente de modelado de tests multi-padarigma basado en el perfil de testing UML, y una notación manejada por la industria y adoptada por la OMG para testing basado en modelos. *Fokus!MBT* simplifica la creación y autoría de los modelos de test con un soporte específico de metodología. Está creado sobre *Eclipse Papyrus*, un entorno de modelado UML de código abierto muy poderoso, que confía en el *Eclipse Modeling Framework EMF* y el *Graphical Modeling Framework*.
Este paper provee ideas profundas sobre los conceptos básicos y la realización técnica de *Fokus!MBT* así como sobre las lecciones que tuvieron que aprender sus creadores durante el desarrollo y la aplicación.

8.5 AlloyMDA

Alloy es un lenguaje de especificación formal con popularidad creciente y que soporta verificación automática. Pero desgraciadamente, su adopción industrial es retrasada por la falta de un ecosistema de herramientas MDE, como lo son los generadores de código. En [52], se presenta una transformación de modelos de Alloy a diagramas de clase anotados con OCL (UML+OCL), reflejando cómo una transformación existente de UML+OCL a Alloy puede mejorarse para manejar cuestiones dinámicas. La transformación bidireccional propuesta permite una integración de Alloy en los contextos de MDE actuales, permitiendo que especificaciones UML+OCL sean transformadas en Alloy para validarlas, verificarlas, corregirlas y posiblemente refinarlas dentro de Alloy, y luego traducirlas de vuelta a UML+OCL para compartir con los *stakeholders* o reutilizar herramientas de MDA para refinarlas.
El documento mencionado especifica las reglas de transformación entre un lenguaje y otro, y ofrece una herramienta gratuita que permite tal conversión. La misma fue utilizada en el capítulo previo para transformar el código OCL centralizado generado a partir del modelo de datos en su correspondiente código Alloy, para poder luego verificar a este formalmente con el verificador Alloy.

8.6 Casos de Prueba del Sistema Generados en el Contexto MDD/MDT

La investigación realizada en [54] se centra en el mismo objetivo que el de nuestro proyecto: generar modelos y herramientas de testing a partir de modelos descriptivos del sistema, aunque especificándose en modelos de casos de uso UML ante nuestros diagramas de clase UML. En el trabajo, se propone la generación de un diagrama de actividades (de testing del sistema, abreviados *DATS*) que especifique secuencias de ejecuciones funcionales para distintos tipos de usuarios, a ejecutar sobre un sistema dado; combinando esto con una herramienta que detalla para cada caso de uso una actividad específica, se generan tests de unidad y luego de integración robustos, como en nuestro proyecto.
El proceso, de manera similar al propio, cuenta con tres etapas bien definidas, como se puede observar en la tabla 8.1.

	Nuestro Proyecto	**Proyecto generador de DATS**
Etapa 1	Definición del Modelo de Datos: Diagrama de Clases UML (PIM).	Definición del Modelo de Casos de Uso (PIM).
Etapa 2	Transformación de 1 para obtener el código Java correspondiente y un archivo OCL centralizado.	Transformación de 1 para obtener el DATS utilizando un perfil UML.
Etapa 3	Nueva transformación para generar el archivo Alloy y luego poder completar la verificación formalmente a través de este.	Nueva transformación para generar un archivo XML con las posibles secuencias de ejecución, cubriendo todas las posibles funcionalidades.

Tabla 8.1 Comparando ambos proyectos

OCL también fue requerido en este proyecto, para especificar reglas de buena formación y restricciones en los estereotipos generados. A diferencia de nuestro trabajo, se definen y se hace uso de los perfiles UML.

8.7 Perfiles de testing aplicados a modelos de software

En esta tesina [55], también se desarrolla una herramienta que permite realizar transformaciones de forma automática; en este caso, de los modelos de prueba estructurales y de comportamiento a código JUnit, definiendo un lenguaje para modelar dominios de pruebas utilizando, como en 8.6, el Perfil de Pruebas UML, y las reglas formales de transformación de modelos U2TP a código de testing JUnit basadas en el lenguaje MOFScript. Nuestro proyecto puede entenderse como un avance de esta propuesta, en los siguientes puntos:

- Reemplazo de la herramienta *MOFScript* por *Acceleo*, quien representa una versión más moderna y útil.
- Inclusión en el modelo de restricciones *OCL* que dan la chance a aquellos usuarios especializados en lenguajes formales de participar activamente en la etapa de modelado, aportando a que el sistema sea robusto y confiable desde una etapa temprana.

- Reemplazo del framework *EasyMock* por su posterior versión *Mockito*, cuyas ventajas explicamos ya en 3.5.2.
- Al no definir perfiles UML ni estereotipos, se testea absolutamente toda la funcionalidad del modelo fuente.
- La utilización de la herramienta *Papyrus* para elaborar el modelo de datos, permitiendo una buena visualización y acorde al desarrollo tradicional de este tipo de esquemas, ante la vista jerárquica en forma de árbol del trabajo de la tesina mencionada.
- La complementación del código de pruebas Java utilizado para testear con el *Alloy Analyzer*, quien da garantías de verificación formal.

9. Conclusiones y Trabajos Futuros

Las organizaciones modernas son responsables de la producción de la mayoría de los servicios y productos avanzados que nunca antes se habían producido, buscando cotinuamente nuevas oportunidades para estos y respondiendo en tiempos de reacción que previamente hubieran sido inimaginables. Los procesos que solían tomar meses, ahora toman semanas o incluso días.
Todo esto se debe a los sistemas de información, y a la tecnología de información y comunicación o ICT. Las posibilidades incrementales permitidas por esta tecnología le permitieron a las organizaciones cambiar drásticamente en muchos aspectos.
Entre otros avances revolucionarios en el ámbito informático, surge la conocida **Cloud** o Nube, representando un cambio por sobre todo económico, permitiendo que un usuario común, que nunca podría afrontar los costos de comprar grandes estructuras de computación, aproveche estas tan solo por un pequeño tiempo y para un proyecto propio, a un precio razonable, y con el lujo de la comodidad.
Pero estas comodidades llevaron a un proceso de reorganización y de reflexión sobre ***qué*** era lo que se estaba haciendo y ***cómo*** se lo hacia. La eliminación de los límites que a capacidad y dinero referían implicó repensar cómo se programaba para aprovechar el poder de cómputo de un centro de datos, y dio origen a un conjunto de tecnologías que tienen como objetivo tratar con estos.
La tendencia en el avance de la tecnología ha abierto las puertas hacia un nuevo enfoque de entendimiento y toma de decisiones, utilizada para describir enormes cantidades de datos (siendo estos estructurados, no estructurados y/o semi estructurados), lo que tomaría demasiado tiempo y sería muy costoso si se los cargara en una base de datos relacional para su análisis. De aquí que surgió el concepto de **Big Data**, que aplica para toda aquella información que no puede ser procesada o analizada utilizando procesos o herramientas tradicionales.
Si nos enfocamos ahora en un sistema de información particular, siendo que en la actualidad toda empresa cuenta con sistemas incrementalmente complejos, observamos que a medida que estos son continuamente modificados, su complejidad, reflejando una estructura deteriorada, se incrementa a menos que se realice cierto trabajo para mantenerla o reducirla. La *Ley de Lehman* [5] establece que, a medida que el software se modifica a lo largo del tiempo, su estructura tiende a volverse más y más compleja, y modificar la misma implica efectos secundarios no deseados, volviéndose cada vez más costoso actualizar el sistema.
Dado que *refactorizar* un sistema siempre implica una mayor disponibilidad de tiempo y un costo alto, lo ideal sería crear estos sistemas de manera automática, respetando ciertas características desde su nacimiento y siguiendo estándares abstractos de buena formación que puedan aplicarse a cualquier tipo de sistema. Aquí es donde surgió el **MDD**, como mencionamos en los primeros capítulos.

En base a este pequeño raconto histórico y quedándonos con lo que a este trabajo compete, que es el surgimiento del MDD, pudimos aportar a la automatización de uno de los primeros pasos en el proceso del desarrollo de un sistema. Creamos una herramienta que permite traducir un modelo de datos con restricciones formales a su correspondiente código Java, automatizando la generación de casos de prueba robustos y especificados no sólo en este lenguaje sino también en dos lenguajes formales (OCL y Alloy), lo que le da un soporte confiable y verificable con diversas técnicas.

A través de herramientas de modelado no tan conocidas para usuarios de Eclipse ajenos al MDD, como lo son *Acceleo* y *Papyrus*, y de otras bien conocidas por el mundo Java, como *JUnit* y *EasyMock/Mockito*, logramos integrar distintos lenguajes, cada uno con su sintaxis propia, para concluir en un resultado que es más que utilizable por el usuario que así lo desee. Dado el poco tiempo en el mercado que tienen algunas de ellas, se proveyó además una buena documentación para aquel usuario interesado en iniciar su trabajo valiéndose de las mismas.

Contando ya entonces con nuestros resultados, proponemos al lector el desarrollo de trabajos futuros aprovechándolos, entre ellos:

- Permitir especificar las restricciones en el modelo fuente directamente en el lenguaje Alloy, ahorrándonos un paso en la etapa final del proceso.
- Lograr que, tras modificar el código obtenido en primera instancia y realizar luego alguna modificación en el modelo original, la herramienta *Acceleo* regenere el código sin alterar las actualizaciones realizadas o aquel texto que estuviera delimitado por separadores especiales.
- Contar con tests menos abstractos y desprenderse de a poco del uso de mocks, para generar tests más específicos y acordes a cada método que le den mayor confiabilidad a los casos de prueba.
- En caso de inconsistencias en el modelo fuente, generar contraejemplos en lenguaje natural/Java, de manera que usuarios ajenos a la verificación formal puedan entenderlas y aportar a su arreglo. Proponer además soluciones posibles.
- Realizar el mismo proceso permitiendo la selección de diversos lenguajes de programación destino para el código generado.
- Optimizar la herramienta *Acceleo*.

10. Referencias

[1] *Desarrollo de Software Dirigido por Modelos*. Claudia Pons, Roxana Giandini, Gabriela Perez. Universidad Nacional de La Plata. Editorial: McGraw-Hill Educación y Edulp. Marzo 2010.

[2] *A Survey of Requirements Specification in Model-Driven Development of Web Applications*. PEDRO VALDERAS and VICENTE PELECHANO, Universitat Politècnica de València. Mayo, 2011.

[3] *Desarrollo de Software Dirigido por Modelos*. Francisco Durán Muñoz, Javier Troya Castilla, Antonio Vallecillo Moreno. Universidad Abierta de Catalunya.

[4] *The Humble Programmer*. Edsger Dijkstra

[5] *Normalized Systems: Re-creating Information Technology based on laws for software evolvability*. Mannaert Herwig, Verelst Jan. 2009

[6] *Problems in Application Software Maintenance*. Bennet P. Lientz and E. Burton Swanson
UCLA

[7] *Software Maintenance Costs*. Jussi Koskinen, School of Computing, University of Eastern Finland, Joensuu, Finland. April 30, 2015

[8] *Practical Model Based Testing: A tools approach*. Mark Utting and Bruno Legeard. 2007

[9] Nato Software Conference. Rome, Italy, 27th to 31st October 1969

[10] IBM Research: https://www.research.ibm.com/haifa/projects/verification/mdt/

[11] Model Based Testing Tools: http://www.cs.tut.fi/tapahtumat/testaus08/Olli-Pekka.pdf

[12] MDT book: https://books.google.com.ar/books?hl=es&lr=&id=1jPh-EWwapMC&oi=fnd&pg=PA3&dq=model+driven+testing&ots=SSBPLpuTCK&sig=UO6jLzv3qh0CtKuGIutX1TCZQKU#v=onepage&q&f=true

[13] OMG: http://utp.omg.org/

[14] *The UML Testing Profile.* Ina Schieferdecker, Technical University Berlin/FOKUS Øystein Haugen, University of Oslo. June 2004

[15] *Automated Model-based Testing using the UML Testing Profile and QVT*. Beatriz Pérez Lamancha1, Pedro Reales Mateo[2], Ignacio Rodríguez de Guzmán[2], Macario Polo Usaola[2], Mario Piattini Velthius[2]. [1]Software Testing Centre (CES), School of Engineering, Republic University,Montevideo, Uruguay. [2]Alarcos Research Group, Information Systems and Technologies Department,University of Castilla-La Mancha, Ciudad Real, Spain.

[16] *Using Testing and JUnit Across The Curriculum***.** Michael Wick, Daniel Stevenson and Paul Wagner. Department of Computer Science, University of Wisconsin-Eau Claire. Eau Claire, WI 54701

[17] *JUnit 3.8 Documented Using Collaborations***.** Dirk Riehle, SAP Research, SAP Labs LLC. 3475 Deer Creek Rd, Palo Alto, CA, 94304, U.S.A.

[18] *Using Mock Objects Frameworks to teach object-oriented design principles***.** Jagadeesh Nandigam and Yonglei Tao, Computing and Information Systems, Grand Valley State University.

[19] *Test-Driven Development – A Practical Guide*. Astels, D.The Coad Series, Prentice Hall, 2003.

[20] Mock Objects: *http://www.mockobjects.com/*

[21] EasyMock: *http://easymock.org/*

[22] Revista Cubana Scielo: *http://scielo.sld.cu/*

[23] *Software Abstractions: Logic, Language, and Analysis*. Daniel Jackson, 2006.

[24] CICS Family: http://www-01.ibm.com/software/htp/cics/
[25] *Formal Modeling with Z: An Introduction*. Luca Viganò. Institut für Informatik, Albert-Ludwigs-Universität Freiburg. Spring 2002
[26] PL/1: http://groups.engin.umd.umich.edu/CIS/course.des/cis400/pl1/pl1.html
[27] *CS 290C: Formal Models for Web Software*. Tevfik Bultan
[28] Alloy: http://alloy.mit.edu/alloy/
[29] *Alloy as a Language for Domain Modeling*. Nesa Asoudeh, Ramtin Khosravi, School of Electrical and Computer Engineering, University of Tehran, Kargar Ave., Tehran, Iran.
[30] *A Guide To Alloy - Second Year Group Project.* Edward Yue Shung Wong, Michael Herrmann, Omar Tayeb; Supervisor: Dr. Krysia Broda.
[31] *Some Shortcomings of OCL, the Object Constraint Language of UML*. Mandana Vaziri and Daniel Jackson. MIT Laboratory for Computer science. December 7, 1999
[32] *Comparison of the Modeling Languages Alloy and UML*. Yujing He. Department of Computer Science, Portland State University. Portland, Oregon, USA
[33] *On Challenges of Model Transformation from UML to Alloy*. Kyriakos Anastasakis1, Behzad Bordbar1, Geri Georg2, Indrakshi Ray2. 1 School of Computer Science, University of Birmingham, Edgbaston, Birming-ham, UK. 2 Computer Science Department, Colorado State University, Fort Collins, Col-orado, USA
[34] *Manuel Wimmer and Gerhard Kramler. Bridging grammarware and modelware*. In Jean-Michel Bruel, editor, MoDELS Satellite Events, volume 3844 of Lecture Notes in Computer Science, pages 159–168.
[35] UML2Alloy: http://www.cs.bham.ac.uk/~bxb/UML2Alloy/
[36] *From UML to Alloy and Back Again*. Seyyed M.A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. School of Computer Science, The University of Birmingham, Edgbaston, B15 2TT. United Kingdom.
[37] Eclipse Modeling Framework EMF: https://eclipse.org/modeling/emf/
[38] Eclipse Model Development Tools MDT: https://www.eclipse.org/modeling/mdt/?ref=binfind.com/web.
[39] *Early Experiences on Model Transformation Testing***.** Alessandro Tiso, Gianna Reggio, Maurizio Leotta. Dipartimento interscuola di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS). Università di Genova, Italy
[40] *Model Transformation Testing: The State of the Art***.** Gehan M. K. Selim, Juergen Dingel. School of Computing, Queen's University, Kingston, ON, Canada
[41] *Towards a Model Transformation Intent Catalog*. Moussa Amrani, Jürgen Dingel, Leen Lambers, Levi Lúcio, Rick Salay, Gehan Selim, Eugene Syriani, Manuel Wimmer
[42] *Verification of Graph-based Model Transformations Using Alloy*. Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2014). Xiaoliang Wang, Fabian Büttner, Yngve Lamo
[43] Acceleo: https://www.eclipse.org/acceleo/
[44] Papyrus: https://eclipse.org/papyrus/
[45] Acceleo Operations Reference: https://wiki.eclipse.org/Acceleo/Acceleo_Operations_Reference
[46] AlloyMDA: http://sourceforge.net/p/alloymda/wiki/Home/
[47] *TestEra: A Tool for Testing Java Programs Using Alloy Specifications*. Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov†, Sarfraz Khurshid. Electrical and Computer Engineering, University of Texas at Austin
[48] TestEra: http://cs.txstate.edu/~g_y10/testera/
[49] *Towards a Base Model for UML and OCL Verification*. Frank Hilken, Philipp Niemann, Robert Wille, and Martin Gogolla. University of Bremen, Computer Science Department

[50] Model-based test case generation for Web Applications. Miguel Nabuco, Ana C. R. Paiva. Department of Informatics Engineering. Faculty of Engineering of University of Porto.
[51*] Fokus!MBT – A Multi-Paradigmatic Test Modeling Environment*. Marc-Florian Wendland, Andreas Hoffmann, Fraunhofer, Ina Schieferdecker.
[52] *Translating between Alloy Specifications and UML Class Diagrams Annotated with OCL.* Alcino Cunha, Ana Garis, Daniel Riesco. Journal of Software and Systems Modeling (SoSyM), Springer, ISSN 1619-1366, 2015.
[53] Mockito: *http://mockito.org/*
[54] *Casos de Prueba del Sistema Generados en el Contexto MDD/MDT.* Natalia Correa, Roxana Giandini. 13th Argentine Symposium on Software Engineering, ASSE 2012.
[55] *"Perfiles de testing aplicados a modelos de software".* Luis Fernando Palacios. Tesis de Magister en Ingeniería de Software. UNLP. Abril de 2010.
[56] *A Taxonomy of Model Transformations.* Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp.