Universidad de la República
Facultad de Ingeniería
Instituto de Computación
Uruguay

Intérprete Funcional para OCL
HaskellOCL

Leticia Vaz

Gonzalo Sintas

Proyecto de Grado
Ingeniería en Computación
Universidad de la República

Montevideo, Uruguay, Diciembre de 2018

Supervisor: Dr. Ing. Daniel Calegari
Dr. Ing. Marcos Viera
Universidad de la República

Resumen

El paradigma de Ingeniería Dirigida por Modelos (MDE por sus siglas en inglés) propone la construcción de software basado en una abstracción de su complejidad a través de la definición de modelos y en un proceso de construcción (semi)automático guiado por transformaciones de estos modelos. El Object Constraint Language (OCL), un lenguaje formal que permite expresar restricciones que se deben cumplir para asegurar la corrección semántica de un modelo. El lenguaje se define como un lenguaje sin efectos secundarios que combina aspectos funcionales (ej. composición de funciones) y orientados a modelos (ej. herencia de tipos).

Sus intérpretes se enfocan principalmente en aspectos orientados a modelos, proveyendo una representación directa de construcciones como herencia y navegación a través de propiedades de los elementos de los modelos. Sin embargo, en los últimos años se han propuesto diversas extensiones funcionales al lenguaje, por ejemplo: *pattern matching*, expresiones lambda y evaluación perezosa. En un trabajo previo se propuso la construcción de un intérprete de OCL basado en el paradigma funcional (usando Haskell), que incluye además la interpretación de modelos, metamodelos y transformaciones.

El objetivo de este proyecto es continuar desarrollando las capacidades del intérprete, particularmente definiendo una transformación que permita generar la infraestructura funcional necesaria para interpretar expresiones OCL (OCL2Haskell), así como adaptar un editor de OCL (en Eclipse) para permitir ejecutar automáticamente la transformación y realizar la validación de dichas expresiones OCL haciendo que la infraestructura funcional sea transparente para el usuario final.

Palabras clave: MDE, OCL, Haskell, Eclipse

Contenido

1 Introducción

El paradigma de la ingeniería dirigida por modelos (MDE - *Model Driven Engineering*, [1]), propone elevar el nivel de abstracción de un sistema mediante la construcción de modelos y transformaciones automáticas de estos a texto u otros modelos. Un modelo es una simplificación de la realidad. Una transformación es básicamente la generación automática de un modelo (o texto) de destino a partir de un modelo (o texto) de origen, de acuerdo a una especificación.

Para representar restricciones no estructurales sobre los modelos, varias propuestas utilizan el lenguaje *Object Constraint Language* (OCL, [2]). OCL es considerado un lenguaje que combina aspectos funcionales y, al mismo tiempo, orientados a modelos. Dicha combinación no es sencilla, ya que la brecha entre ambos aspectos es muy grande. De hecho, los intérpretes de OCL que existen actualmente, por ejemplo Eclipse OCL [3], están enfocados en proveer representaciones de los aspectos orientados a modelos, sin tener en cuenta los aspectos funcionales.

En un trabajo previo [4], se propuso la construcción de un intérprete de OCL basado en el paradigma funcional (basado en Haskell [5]), que incluye la interpretación de modelos, metamodelos y transformaciones, y se realizó un prototipo funcional para validar la propuesta. Además, se propuso la extensión del lenguaje con algunas características del paradigma funcional: *pattern matching*, expresiones lambda y evaluación perezosa.

El objetivo general de este proyecto es extender la infraestructura funcional existente, definida en el trabajo previo, para la interpretación del lenguaje OCL y su integración a herramientas de modelado existentes. Los objetivos específicos del proyecto son:

1. Obtener un conocimiento base sobre MDE
2. Extender las capacidades del intérprete existente a través de su implementación en Haskell

3. Integrar el intérprete a una herramienta de modelado a través de la definición de una transformación de modelo a texto

4. Evaluar las capacidades del intérprete en relación a otros existentes a partir de casos de estudio

Los resultados obtenidos en este proyecto sirvieron como base para la redacción del trabajo de investigación *"Model-Driven Development of an Interpreter for the Object Constraint Language"* [6] que fue presentado en conjunto con los tutores de este proyecto.

El informe está organizado de la siguiente manera. En la Sección 2 se presentan las áreas conceptuales que forman parte de la propuesta y se introducen las herramientas utilizadas para la implementación de la solución. La Sección 2 también provee una lista de los trabajos relacionados a la interpretación funcional de OCL y propuestas de extensión del mismo haciendo uso del paradigma funcional. En la Sección 3 se introducen las ideas presentadas en [4] como punto de partida para el resto del proyecto. En las secciones 4 y 5 se presenta la implementación de la representación de modelos en código Haskell y la transformación de modelos, respectivamente. La Sección 6 introduce la herramienta construida para validar las invariantes OCL sobre un modelo (HaskellOCL) y la Sección 7 incluye un caso de estudio como ejemplo que emplea los conceptos anteriores. Finalmente, en la Sección 8 se listan las conclusiones y el trabajo futuro.

2 Marco Teórico

En esta sección se presentan las áreas conceptuales que forman la base de la propuesta: Ingeniería Dirigida por Modelos y Programación Funcional. Además, se introducen las herramientas que serán la base de la construcción de la solución. Finalmente, se presentan diversos trabajos relacionados a la interpretación funcional del lenguaje OCL.

2.1 Ingeniería dirigida por modelos

El principio básico en el que se basa la ingeniería dirigida por modelos (MDE - *Model Driven Engineering* en inglés), es que "Todo es un modelo" [7]. En términos generales, es posible definir un modelo como "una abstracción simplificada de un sistema o concepto del mundo real" [8], entendiendo como concepto un objeto de la realidad o una idea. Para que un modelo sea útil, en términos de ingeniería de software, debe ser abstracto, comprensible, preciso, predictivo y rentable [9].

Si bien los modelos en ingeniería de software se consideran básicamente como parte de la documentación o como un artefacto de comunicación, pueden tener una mayor jerarquía dentro del ciclo de vida del software, por ejemplo, realizar simulaciones, generar código fuente y realizar diversos análisis a nivel de modelo.

MDE es "un paradigma de ingeniería de software donde los modelos juegan un papel clave en todas las actividades de la ingeniería" [10]. Propone elevar el nivel de abstracción de un sistema mediante la construcción de modelos y realizar transformaciones automáticas a partir de modelos, a otros modelos o a texto como código fuente [11] [12].

El objetivo de MDE es abstraer la especificación de los programas y aumentar la automatización de la implementación de los mismos. Además, las herramientas MDE imponen restricciones específicas del dominio y realizan validaciones del modelo que pueden detectar y prevenir muchos errores desde el principio del ciclo de vida.

En la Figura 2.1 se muestran los componentes esenciales de MDE: modelos y metamodelos. A grandes rasgos, un modelo es una abstracción del sistema que está siendo estudiado, el cual puede que ya exista o que esté planificado implementar en un futuro. A su vez, un metamodelo es un modelo que define las partes y reglas necesarias para crear modelos válidos.

Figure 2.1: Diagrama de componentes MDE

2.1.1 Modelos y Metamodelos

Como se mencionó en la sección anterior, los modelos son una abstracción de la realidad. La definición de los modelos es realizada partiendo de la base de un metamodelo. Son estos metamodelos los que definen la sintaxis y semántica para un conjunto de modelos.

La Figura 2.2 muestra la creación de un modelo que conforma la sintaxis y semántica de un metamodelo dado. El modelo de ejemplo presenta una realidad con dos personas, Juan

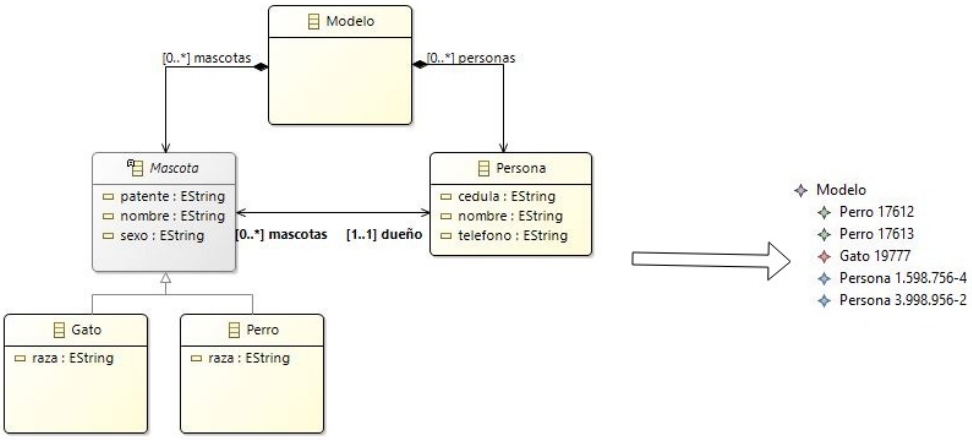


Figure 2.2: Modelo generado a partir de un metamodelo

Ruffino (CI: 1.598.756-4, Teléfono: 98 563 236) y Valeria Rodríguez (CI: 3.998.956-2, Teléfono: 94 410 012). Juan tiene dos mascotas, ambos perros labradores, Cacique (Patente: 17612) y Cacho (Patente: 17613), mientras que Valeria tiene una mascota, una gata persa de nombre Misifus (Patente: 19777).

Basta con analizar metamodelo y modelo de la Figura 2.2 para ver la relación que se de a entre ambos y con la realidad a representar. Por ejemplo, para poder representar a los dueños de las mascotas, Juan y Valeria, a nivel de metamodelo se define una clase `Persona`, que va a representar a los dueños. Esta clase contará con información en forma de atributos (Nombre, Cédula y Teléfono), y también asociará a cada `Mascota` de la que una persona es dueño mediante una relación.

Al generar un modelo, y definir una instancia de `Persona` para representar, por ejemplo, a Juan Ruffino, se le asocian los datos correspondientes en sus atributos (Nombre: Juan Ruffino, CI: 1.598.756-4, Teléfono: 94 410 012), y se le asocian las instancias de `Mascotas` que representan a sus dos perros, Cacique y Cacho.

Para este y ejemplos futuros, los metamodelos definidos utilizarán la notación de diagrama de clases UML.

2.1.2 Transformaciones de Modelos

En el núcleo de MDE se encuentra la transformación de modelos [13]. "Una transformación es una operación que recibe como entrada un conjunto de modelos y devuelve como salida un conjunto de modelos objetivo (M2M - *Model To Model Transformation*) o texto (M2T - *Model To Text Transformation*)" [10].

Generalmente se adopta un enfoque orientado a objetos para representar y manipular los modelos, debido a que modelos orientados a objetos facilitan la visualización del problema mediante diagramas, cuya representación es más cercana a la realidad del problema. Con las transformaciones M2M es posible generar nuevos modelos con diferentes niveles de abstracción y en diferentes lenguajes. Las transformaciones M2T permiten la transformación de modelos a artefactos textuales, tales como código, informes o documentación. Por lo tanto, la transformación de modelos puede aportar una amplia gama de tareas entre las que se incluyen: refinamiento, síntesis, abstracción, consultas, traducción, migración, análisis, refactorización, normalización, optimización, fusión y depuración [13].

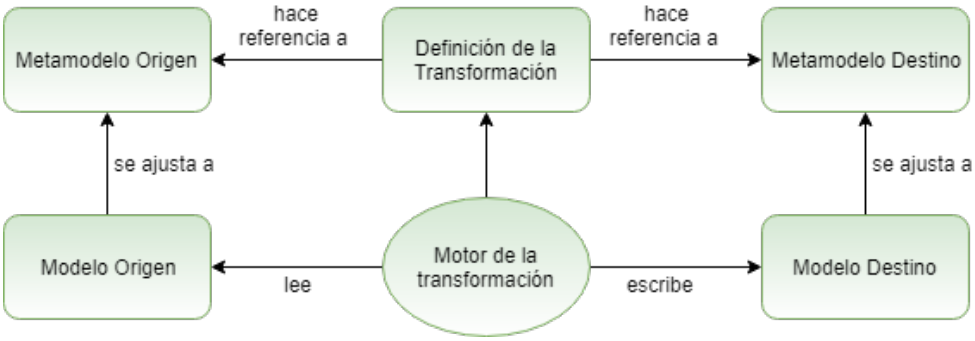


Figure 2.3: Componentes básicos de la Transformación de Modelos [14]

En la Figura 2.3 se pueden apreciar los componentes básicos que forman parte de la transformación de modelos y la relación entre ellos. Esta figura representa un ejemplo sencillo en el que solo existe un modelo como entrada (Modelo Origen) y una salida (Modelo Destino). Tanto el modelo de entrada como el de salida, conforman con sus respectivos metamodelos. Las figuras 2.4 y 2.5 mustran como, a partir de transformaciones, podría generarse un modelo MER (*Model to Model*) o un archivo xml (*Model to Text*).

En general, una transformación puede tener múltiples modelos de origen y destino, y es definida como un conjunto de reglas que describen la correspondencia entre elementos de los metamodelos Origen y Destino. Por último, el motor de la transformación, interpreta las reglas definidas por la transformación y las ejecuta sobre el modelo Origen.

2.1.3 OCL

Si bien los lenguajes gráficos de modelado ayudan a entender mejor los sistemas y los llevan al nivel de abstracción que predica MDE, en la algunos casos su falta de precisión no refleja todos los aspectos de la realidad. Es posible enriquecer los modelos agregando información

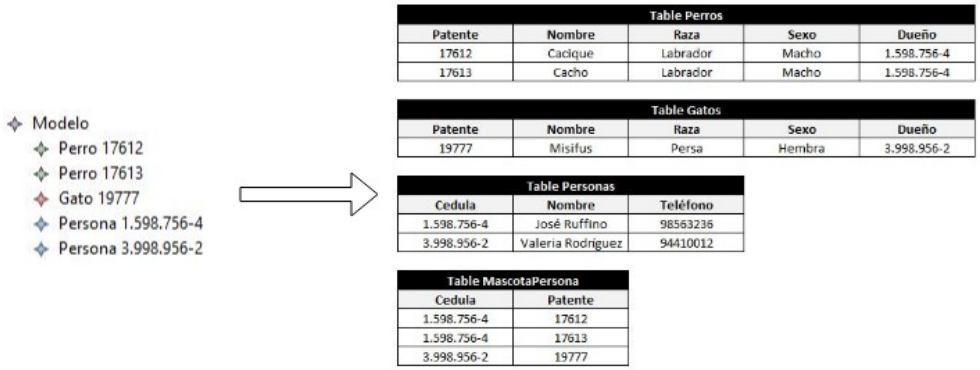


Figure 2.4: Transformación M2M - de modelo Ecore a Modelo Entidad Relación

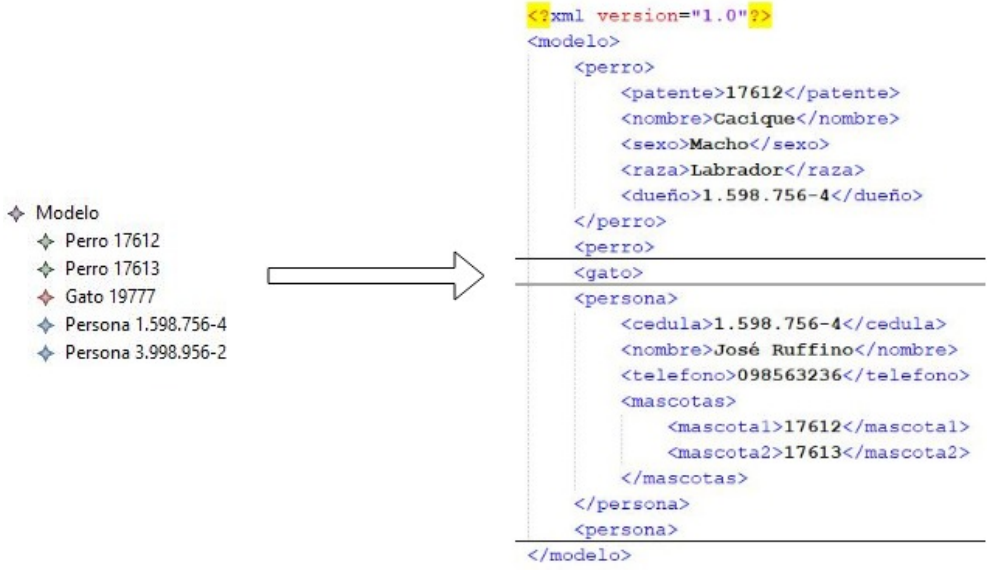


Figure 2.5: Transformación M2T - de representación Ecore a representación XML

adicional o restricciones, dando la posibilidad de diseñar modelos precisos, completos y sin ambigüedad [8].

En el metamodelo de la Figura 2.6, que a partir de ahora tomaremos como ejemplo de referencia, representa la organización de reuniones de equipo, específicamente de sus miembros, y las características de las mismas. A modo de ejemplo de la aplicación de OCL sobre el mismo, para garantizar la semántica del modelo correspondería imponer una restricción para que el fin de un *Meeting* sea posterior al inicio.

Para definir estas restricciones, y que sea posible chequear su validez en los modelos, es necesario utilizar lenguajes formales. En el caso de UML [15], se utiliza el lenguaje OCL (Object Constraint Language). Como se deine en [8], "OCL es un lenguaje textual, con base formal, y que además posee mecanismos y conceptos muy cercanos a los de UML, lo cual facilita su uso por parte de los modeladores".

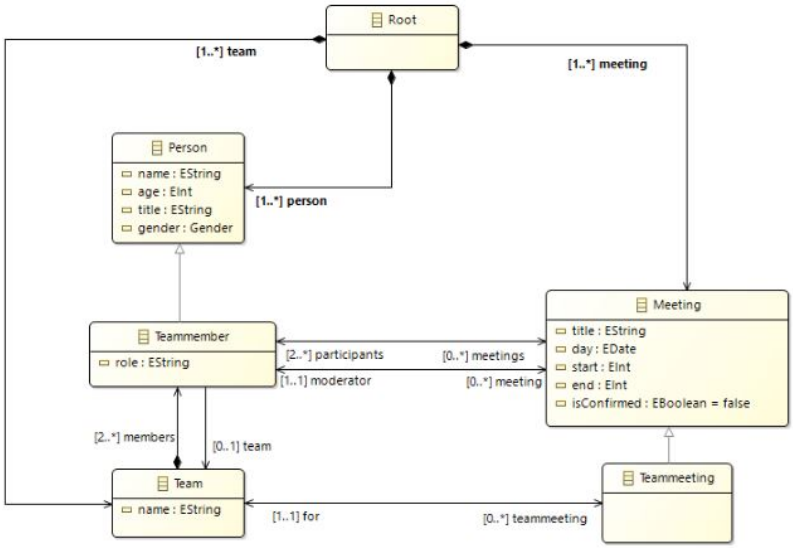


Figure 2.6: Metamodelo TeamMeeting

La siguiente expresión muestra cómo se representa en OCL la restricción antes mencionada (el fin de una *Meeting* debe ser posterior al inicio de la misma).

```
context Meeting
  inv:  self.end > self.start
```

Para especificar restricciones en OCL se utilizan invariantes. Las invariantes representan reglas que deben ser verdaderas para todas las instancias del modelo, en todo momento [16].

```
context <classifier>
inv [<constraint name>]: <Boolean OCL expression>
```

Figure 2.7: Sintaxis de una invariante [17]

Las invariantes deben respetar la sintaxis de la Figura 2.7. La etiqueta ***context*** seguida del nombre del elemento al que hace referencia la expresión, identifica el contexto en el que se lleva a cabo la restricción. En el ejemplo anterior, el contexto es *Meeting*.

La etiqueta ***inv*** declara que la restricción es una invariante y a continuación se escribe la restricción. Opcionalmente, se puede escribir el nombre de la invariante a continuación de la palabra clave ***inv***.

Cada expresión se escribe en el contexto de una instancia de un tipo específico. En una expresión OCL, la palabra reservada ***self*** se utiliza para referirse a la instancia contextual, es decir, es el objeto sobre el cual se se está evaluando la expresión [2]. En el ejemplo, el contexto es *Meeting*, entonces ***self*** hace referencia a una instancia de *Meeting*.

Cuando el contexto es claro, la palabra ***self*** se puede descartar y definir un nombre diferente que la reemplace. Para el caso del ejemplo, la misma invariante podría definirse de la siguiente manera:

```
context m:Meeting
      inv:  m.end > m.start
```

Además de especificar restricciones en clases y tipos en las clases de un modelo UML, OCL tiene otros propósitos [2]:

- lenguaje de consulta para la especificación de patrones en reglas de transformación
- para especificar pre y post condiciones en operaciones y métodos
- para describir guardas en máquinas de estado
- para especificar conjuntos de destinatarios de mensajes y acciones
- para especificar restricciones en operaciones
- para especificar reglas de derivación para atributos para cualquier expresión sobre un modelo UML

OCL es un lenguaje fuertemente tipado, es decir, toda expresión OCL es de un determinado tipo. Al ser compatible con UML, comparten un conjunto de tipos básicos. Dicho conjunto está predefinido de forma tal que los tipos son independientes de cualquier modelo y, además, forman parte de la definición de OCL [2]. En la Tabla 2.1 se enumeran algunos tipos básicos y sus respectivos valores. Además de los tipos enumerados en dicha tabla, se encuentran los tipos correspondientes a colecciones: *Collection*, *Set*, *Bag*, *Sequence* y *Tuple*.

Para los tipos predefinidos se definen operaciones. En la Tabla 2.2 se muestran algunos ejemplos de estas operaciones.

Los valores de los tipos básicos están ordenados en una jerarquía de tipos, como se puede ver en la Tabla 2.3, la cual determina la conformidad de los diferentes tipos entre sí. Por ejemplo, no es posible comparar un objeto de tipo *Integer* con uno de tipo *Boolean* o *String*. Para que una expresión OCL sea válida, todos los elementos deben respetar las reglas de conformidad de tipos. En caso de no respetar alguna de estas reglas, es considerada una expresión inválida.

Tipo	Valores
OclInvalid	invalid
OclVoid	null, invalid
Boolean	true, false
Integer	1, -5, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be ...'
Unlimited Natural	0, 1, 2, 42, ..., *

Table 2.1: Tipos básicos de OCL y sus valores [2]

Tipo	Operaciones
Integer	*, +, -, /, abs()
Real	*, +, -, /, floor()
Boolean	and, or, xor, not, implies, if-then-else
String	concat(), size(), substring()
UnlimitedNatural	*, +, /

Table 2.2: Ejemplos de operaciones en los tipos predefinidos [2]

Tipo	Conforma / es subtipo de	Condición
Set(T1)	Collection(T2)	si T1 es subtipo de T2
Sequence	Collection(T2)	si T1 es subtipo de T2
Bag(T1)	Collection(T2)	si T1 es subtipo de T2
OrderedSet(T1)	Collection(T2)	si T1 es subtipo de T2
Integer	Real	
UnlimitedNatural	Integer	

Table 2.3: Ejemplos de operaciones en los tipos predefinidos [2]

Se dice que un tipo "tipo1" es un subtipo de un tipo "tipo2" cuando una instancia de "tipo1" puede ser sustituida en cada lugar donde se esperaba una instancia de "tipo2". Por ejemplo, un *Integer* es un subtipo de *Real*.

Las reglas de conformidad de tipos para un diagrama de clases son:

- Cada elemento es un subtipo de cada uno de sus supertipos.
- La conformidad de tipos es transitiva: si el "tipo1" es un subtipo del "tipo2" y el "tipo2" es un subtipo del "tipo3", entonces el "tipo1" es un subtipo del "tipo3".

2.2 Programación Funcional

La programación funcional, como lo indica su nombre, está basada en el uso de funciones, entendiendo como función a la función matemática, la cual describe una relación entre una entrada y una salida. Este paradigma comprende aquellos lenguajes de programación donde las variables no tienen estado, es decir, no presentan cambios a lo largo del tiempo y son inmutables.

Un programa funcional es una expresión que es ejecutada mediante su evaluación y está constituido en su totalidad por funciones. El programa principal es una función que toma como argumento la entrada al programa y genera la salida del programa como resultado. La función principal se define en términos de otras funciones y estas, a su vez, en términos de otras funciones hasta llegar a funciones predefinidas o primitivas. Como todo se procesa recursivamente y usando funciones de alto orden, en los lenguajes funcionales no existen las instrucciones cíclicas como por ejemplo *For* y *While*.

Una de las herramientas más potentes que brinda la programación funcional es el *Pattern Matching*. Es un mecanismo que permite comprobar un valor comparándolo con un patrón. Ampliando el concepto, es una secuencia de alternativas, donde cada alternativa implementa un patrón y una o más expresiones que serán evaluadas en caso de que el patrón se cumpla. Esta funcionalidad es importante ya que favorece el código declarativo.

Otra de las características importantes que ofrece la programación funcional son las expresiones lambda. Las expresiones lambda se utilizan para crear funciones anónimas, que son funciones que carecen de un nombre y pueden ser invocadas desde cualquier contexto. Estas funciones pueden ser pasadas como parámetro a otras funciones de orden superior.

A diferencia de otros paradigmas, como el imperativo, los lenguajes funcionales poseen las siguientes características que los hacen especiales:

- **No hay ciclos**
  La programación funcional se basa en la recursividad, lo cual suele ser una ventaja al momento de expresar problemas.
- **Inmutabilidad**
  No hay varibles o asignaciones, esto significa que una vez que un valor ha sido establecido y almacenado, no puede ser cambiado a lo largo de la ejecución del bloque del programa en el cual fue definido.
- **Evita los efectos colaterales**
  Al no tener estados, se garantiza que al llamar una función múltiples veces con las mismas entradas, siempre devolverá los mismos resultados, por ende, estos no se verán influenciados por condiciones externas o estados almacenados previamente. En caso de tener efectos los representa explícitamente en el tipado.

- **Concurrencia**
  Al tener múltiples procesos e hilos ejecutando simultáneamente, el hecho de que las variables sean inmutables y no depender de estados, garantiza que no se van a tener problemas de concurrencia.

Como se puede apreciar en la Figura 2.8, es posible escribir la misma función mediante un lenguaje imperativo y uno funcional, llegando al mismo resultado, pero con muchas menos líneas de código. Esto se debe a la pureza que forma parte de la identidad de los lenguajes funcionales y que aporta a mejorar el testing y el mantenimiento de los programas. El concepto de pureza dentro del paradigma funcional hace referencia a la falta de estados, es decir que, dada la misma entrada para una función, siempre retorna el mismo valor. Las funciones puras son totalmente independientes ya que no dependen de estados externos del programa.

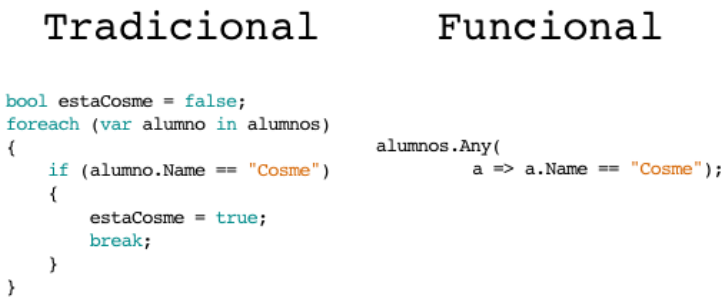


Figure 2.8: Lenguaje imperativo vs. lenguaje funcional

Los programas funcionales dan un gran salto hacia un modelo más elevado de programación. Son más fáciles de diseñar, escribir y mantener, pero el lenguaje ofrece al programador un menor control de la máquina [5].

2.3 Herramientas

2.3.1 Eclipse Modeling Framework

El proyecto EMF [18] es un framework de modelado y generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurados. A partir de la especificación de un modelo, descrita en XMI, EMF provee herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java que sean correspondientes al modelo, junto con un conjunto de clases de adaptadores que permiten la visualización y edición, basada en comandos, del modelo. [19]

El flujo básico de EMF resulta muy práctico; se crea y define un modelo en formato Ecore, el cual es básicamente un lenguaje de metamodelado. Luego, como se muestra en la Figura 2.9, a partir de un modelo Ecore es posible generar código Java.

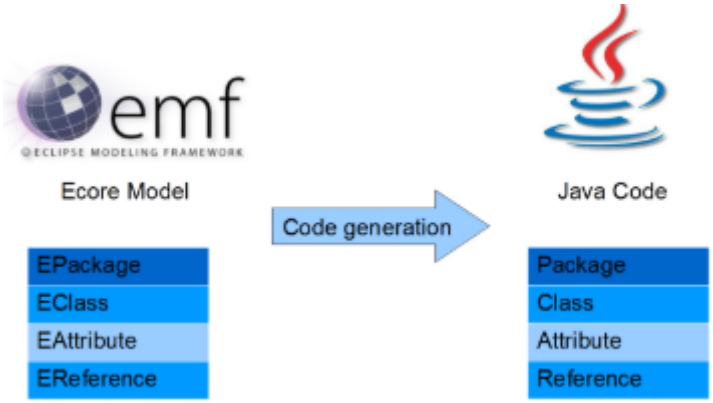


Figure 2.9: Transformación modelo Ecore a código Java

El componente **Ecore Tools** provee un ambiente completo para crear, editar y mantener los modelos Ecore. Este componente facilita el manejo de dichos modelos mediante un editor gráfico y, sumado a otras herramientas que provee Ecore, permite realizar acciones de validación, comparación y generación. El editor gráfico implementa, entre otros, compatibilidad con múltiples diagramas, una vista de propiedades con pestañas personalizadas, retroalimentación de validaciones y capacidades de refactorización.

El metamodelo EMF consiste en dos partes: por un lado el archivo de descripción *ecore* y, por otro lado, el archivo de descripción *genmodel*. El archivo *ecore* contiene información acerca de las clases definidas, mientras que el archivo *genmodel* contiene información adicional para la generación del código, en la que se incluyen los parámetros de control acerca 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.
- **EAttribute:** representa un atributo, el cual tiene un nombre y un tipo.
- **EReference:** representa el extremo de una asociación entre dos clases.
- **EDataType:** representa el tipo de un atributo.

En la Figura 2.9 se muestra la correspondencia que se genera entre los elementos de los modelos Ecore y el código Java que se genera.

La Figura 2.10 muestra un fragmento del código generado por la herramienta Ecore Tools para representar el metamodelo de ejemplo *TeamMeeting*. Se puede apreciar el uso de las palabras reservadas ***package***, ***class***, ***attribute*** y ***property*** para definir los componentes *EPackage*, *EClass*, *EAttribute* y *EReference* del modelo. Como ejemplo de cada caso:

- `teamMeeting` es un *EPackage*
- `Person` es un *EClass*
- `name` es un *EAttribute* dentro de `Person`
- `members` es un *EReference* dentro de `Team`

```
package teamMeeting : teamMeeting = 'http://www.example.org/teamMeeting'
{
    class Person
    {
        attribute name : String[?];
        attribute age : ecore::EInt[1];
        attribute title : String[?];
        attribute gender : String[?];
    }
    class Teammember extends Person
    {
        attribute role : String[?];
        property team : Team[?];
        property meetings#participants : Meeting[*|1] { ordered };
        property meeting#moderator : Meeting[*|1] { ordered };
    }
    class Team
    {
        attribute name : String[?];
        property members : Teammember[2..*|1] { ordered composes };
        property teammeeting#for : Teammeeting[*|1] { ordered };
    }
```

Figure 2.10: Código para metamodelo de ejemplo *TeamMeeting*

2.3.2 Sirius

Sirius es un proyecto de Eclipse que permite al usuario crear su propio *workbench*[1] de modelado gráfico, aprovechando las tecnologías de modelado de Eclipse, entre las que se incluye EMF.

Esta herramienta proporciona un *workbench* genércio para la ingeniería de arquitectura basada en modelos que se puede adaptar fácilmente a necesidades específicas.

Un *workbench* de modelado creado con Sirius provee un conjunto de editores de Eclipse que permite a los usuarios crear, editar y visualizar modelos EMF. Hay tres tipo de editores principales: diagramas, tablas y árboles.

En el marco de este proyecto solo se utilizó el editor de diagramas. Cuando el diagrama se sincroniza con el modelo EMF, se actualiza automáticamente para reflejar las modificaciones que corresponden con los objetos mostrados, estas pueden ser creaciones, supresiones o actualizaciones de los mismos. [21]

2.3.3 Acceleo

Acceleo [22] es un lenguaje y un entorno para generar texto a partir de modelos. Es una implementación de MOFM2T (MOF Model to Text Transformation Language) definida por OMG. El lenguaje Accceleo está compuesto por dos tipos de estructuras principales dentro de un mismo modelo: consultas y plantillas.

Se diferencia del resto de los generadores de código en que la mayoría generan, o un tipo de tecnología específico (por ejemplo Java), o usan solo un tipo de modelos (por ejemplo UML). Acceleo por su parte no restringe al usuario a un dominio específico, permitiendo generar código a partir de:

- Modelos UML
- Modelos Ecore
- Metamodelos en el *workspace* con instancias dinámicas (".xmi")
- Metamodelos personalizados
- Metamodelos personalizados con dependencias a UML
- Modelos UML con perfiles
- Varios modelos personalizados con dependencias entre ellos

[1] *Workbench:* "Refiere al ambiente de desarrollo de escritorio. Tiene como objetivo lograr una integración perfecta de las herramientas y una apertura controlada, proporcionando un paradigma común para la creación, gestión y navegación de los recursos del espacio de trabajo." [20]

Dado que Acceleo está basado en EMF, es posible crear un modelo con cualquier otra herramienta que esté basada en EMF y usarlo como entrada de la generación.

Con el enfoque basado en plantillas es posible definir el tipo de código que se genera y puede ser personalizado para que respete su propio estilo de codificación. Entre los tipos más comunes se encuentran: Java, Javascript, HTML y Phyton. Para este proyecto, se definió que el código de salida es un código funcional personalizado que pueda ser ejecutado usando el compilador de Haskell.

Un generador está compuesto por varios archivos llamados módulos. Un módulo se parametriza mediante las URIs de los metamodelos, creando instancias de los modelos a partir de los cuales se quiere generar el código. A su vez, un módulo se puede extender para tener acceso a sus elementos públicos y privados. Como se mencionó anteriormente, los módulos están compuestos por dos tipos de estructuras: plantillas, que generan el código y las estructuras, y consultas, que se utilizan para encapsular expresiones complejas.

Los tipos de los parámetros que se definen en las plantillas son únicamente los definidos en el metamodelo o los básicos de OCL (por ejemplo *String*, *Boolean*, *Integer*). Dentro de una plantilla es posible usar dos tipos de expresiones para generar el código: expresiones estáticas, que se generan sin transformaciones, y expresiones de Acceleo, que manipularán elementos del modelo para calcular el texto generado.

Un servicio importante que brinda esta herramienta son los *Java Wrappers*, mediante los que permite invocar un método de una clase Java utilizando una operación llamada "invoke". La importancia de este servicio radica en que simplifica enormemente el código del programa, ya que, si bien Acceleo es un lenguaje sencillo de utilizar, a veces se tornan engorrosas las sentencias y, de esta forma, pueden implementarse métodos en Java que resuelvan problemas complejos.

Acceleo también permite iterar dentro de cada uno de los elementos de un metamodelo y aplicar, para cada uno de ellos, ciertas reglas que definen cómo serán traducidos a texto plano. Fácilmente se pueden generar reglas que tomen como entrada todas las clases de un modelo y apliquen reglas que terminen generando el texto plano resultante de la transformación.

Las reglas de generación se definen en los módulos. Un módulo en Acceleo, es un archivo *.mtl*, que contiene *templates* (para generar el código) y consultas (para extraer información de los modelos que se están manipulando).

En la segunda línea de la Figura 2.11 se muestra la definición del módulo, representado con el comando *module*. En dicho comando, se define el nombre del módulo, en este caso *generate*, y se indica que se trabajará con un metamodelo Ecore.

La siguiente sentencia representa la definición de un *template*. Como se mencionó anteriormente, los *template* son los encargados de generar el código. Es posible definir varios *template* e invocarlos dentro del propio módulo o desde módulos diferentes. En este caso, el *template* es *public*, esto significa que puede ser invocado desde otros módulos. Dentro

*generate.mtl

```
1  [comment encoding = UTF-8 /]
2  [module generate('http://www.eclipse.org/emf/2002/Ecore')]
3
4
5  [template public generateElement(anEClass : EClass)]
6  [comment @main/]
7  [file (anEClass.name, false, 'UTF-8')]
8
9  [/file]
10 [/template]
11
```

Figure 2.11: Plantilla mtl

del comando además se definen el nombre del *template* y los parámetros que puede recibir. Opcionalmente se pueden agregar otros parámetros, como precondiciones e inicializaciones de variables [22]. En el ejemplo recibe un único parámetro de tipo *EClass*.

Dentro del *template* también se utiliza el comando *file*. Este comando indica que el texto generado a partir del código contenido en dicha sección se escriba en un archivo. Los parámetros que tiene este comando en el ejemplo son:

- **anEClass.name** que corresponde al nombre del archivo
- **false** que indica que el archivo se regenerará en cada ejecución
- **UTF-8** que corresponde a la codificación del archivo

En la Figura 2.12 se muestra el código Acceleo y el archivo de texto que se genera a partir de la la ejecución del mismo para el modelo del ejemplo de las figuras 2.2 y 2.4.

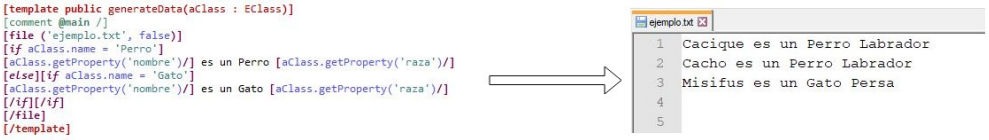


Figure 2.12: *Template* Acceleo generando archivo de texto

2.3.4 Haskell

Haskell [5] es un lenguaje funcional, estático, implícitamente tipado y perezoso, que utiliza conceptos de alto nivel, lo que lo hace un lenguaje elegante, poderoso y general.

Posee una sintaxis expresiva, una amplia variedad de tipos primitivos (enteros, racionales, punto flotante, booleanos) y hay un gran conjunto de bibliotecas que pueden ser importadas.

El compilador más popular, y el que fue usado para la realización de este proyecto, es el Glasgow Haskell Compiler (GHC). Al instalarlo, se obtienen los siguientes programas:

- **ghc:** es el encargado de compilar bibliotecas y aplicaciones escritas en Haskell a código binario
- **ghci:** intérprete que permite cargar módulos, escribir código Haskell y obtener un resultado inmediato

Como se explicó en la sección anterior, los lenguajes funcionales se basan en el uso de funciones, por lo que es común necesitar que las funciones sean ejecutadas en secuencia. Para resolver este problema Haskell provee las mónadas. Una mónada es una estructura que representa cálculos definidos como una secuencia de pasos. Los detalles de las mismas dicen exactamente cómo las acciones deberían ser secuenciadas. Además, es posible almacenar información, que es leída y escrita mientras se llevan a cabo dichas acciones. El uso de mónadas permite entregar pureza a expresiones que parecen no ser puras.

Dentro de las principales propiedades y beneficios que brinda Haskell se encuentran los siguientes:

- **Puro:** El resultado de una función está determinado únicamente por su entrada y como consecuencia no hay efectos colaterales, lo que permite que las expresiones sean evaluadas en cualquier orden. Además, las variables en Haskell, al contrario de los lenguajes orientados a objetos, no cambian su valor, es decir, son inmutables; esta propiedad se denomina integridad referencial.
- **Perezoso:** No se evalúa nada hasta que deba ser evaluado. Esto significa que, a menos que se indique lo contrario, no ejecutará funciones ni calculará resultados hasta que sea necesario. Esta propiedad permite, por ejemplo, que sea posible manipular estructuras de datos infinitas. Además, la pureza del código Haskell hace que sea fácil fusionar cadenas de funciones juntas, permitiendo mejoras en el rendimiento.
- **Fuertemente Tipado:** Cada expresión tiene un tipo que se determina en tiempo de compilación. Los tipos de todos los elementos de una función deben respetarse, de no hacerlo el compilador rechazará el programa. Haskell utiliza un sistema de tipos que posee inferencia de tipos. Esto significa que no es necesario etiquetar cada pedazo de función explícitamente con un tipo porque el sistema de tipos lo puede deducir en tiempo de compilación. La inferencia de tipos también permite que el código sea más general, si por ejemplo, se crea una función para operar con enteros y no se escribe explícitamente el tipo, puede usarse también para operar con otros tipos de números. De esta manera, los tipos no solo se convierten en una forma de garantía, sino en un lenguaje para expresar la construcción de programas.
- **Elegante:** Es un lenguaje conciso que utiliza conceptos de alto nivel. Los programas son normalmente más cortos que en los lenguajes imperativos, por lo que son más fáciles de mantener en comparación a los programas largos. Además, al codificar en

un nivel de abstracción superior, dejando los detalles para el compilador, es más difícil que se cometan errores.

- **Memoria administrada:** El *Garbage Collector* de Haskell es el encargado de gestionar la memoria de forma eficiente, por lo que los programadores solo deben preocuparse por la implementación del algoritmo.

- **Modular:** Haskell ofrece formas más potentes de componer programas a partir de módulos ya desarrollados, en consecuencia, es posible generar programas más modulares. Los pequeños módulos ayudan a que la programación sea más fácil y rápida. Además, es posible probar los módulos de forma independiente, ayudando a reducir los tiempos de depuración. "La capacidad para descomponer problemas en partes depende directamente de nuestra capacidad para unir las soluciones. Para ayudar a la programación modular, un lenguaje deber proveer buen pegamento" [23]. Haskell, y en general los lenguajes funcionales, proveen dos formas de pegamento: funciones de alto orden y evaluación perezosa. Las funciones de alto orden son aquellas que admiten funciones como parámetro y a la vez pueden retornar funciones como resultado de la evaluación. Dichas funciones son indispensables para definir cálculos que eviten especificar pasos que cambien algún estado y luego tener que aplicar bucles sobre estos [24]. De esta manera, es posible modularizar problemas, algo que sería imposible utilizando los lenguajes de programación imperativos.

2.4 Trabajos Relacionados

La extensión del lenguaje OCL incluyendo aspectos de la programación funcional, ha sido objeto de varias investigaciones. En [25] el autor aborda el problema de la proliferación de expresiones de navegación, que ocurre cuando se expresan predicados sobre objetos. Para resolver este problema, propone incluir *pattern matching* a OCL. Otro ejemplo de la inclusión de aspectos funcionales puede encontrarse en [26], donde los autores argumentan que una semántica de evaluación perezosa (*lazy evaluation*) en expresiones OCL aumentaría el rendimiento de las evaluaciones y simplificaría la definición en el caso de las consultas. Otros textos proponen el uso de cálculo monoide[2] para la verificación eficiente de la integridad de los invariantes sobre modelos de datos [27], la extensión del lenguaje OCL con funciones [28], argumentando que el uso de funciones aumenta el nivel de abstracción y modularización, y el uso de expresiones lambda [29].

Se han propuesto varias herramientas que definen una semántica para OCL, de forma tal que el concepto de conformidad de metamodelo restringido por OCL cumpla con el estándar de MOF (MOMENT2 [30], Maude[31]). Complementariamente, se han propuesto herramientas para transformar metamodelos en lógica de primer orden (por ejemplo, LAMBDES [32]). Si bien estas propuestas proporcionan una semántica y un entorno formal para la verificación, ninguna incluye los aspectos de interpretación funcional de OCL.

En [33] se describe un sistema para traducir automáticamente especificaciones de software formales en lenguaje natural por medio de la construcción de árboles sintácticos abstractos del lenguaje funcional.

Una idea más cercana a los fundamentos en los que se basa este trabajo se encuentra en [34]. En dicho paper se plantea abordar las diferentes interpretaciones a las que se presta la semántica de OCL, mediante la formalización del núcleo de dicho lenguaje, usando Isabelle/HOL. Este trabajo se concentra en la implementación de un subconjunto central de OCL, particularmente en formalizar las consecuencias de una lógica de cuatro valores (*true*, *false*, *invalid*, *null*). Debido las similitudes que posee HOL con Haskell, Isabelle/HOL puede ser considerado un lenguaje funcional. Por otra parte, en [35] se propone la representación de los elementos de MDE haciendo uso de Gramáticas de Atributos (AG), donde se utiliza Haskell para especificar las mismas.

Finalmente, el trabajo en el que se basa el desarrollo de este proyecto es [4], donde los autores proponen el uso de Haskell como una alternativa para la interpretación funcional de OCL. En la siguiente sección se desarrollan las ideas presentadas en este trabajo.

[2] proporciona una notación uniforme para colecciones, listas y conjuntos

3 Interpretación Funcional de OCL

El trabajo a realizar en este proyecto parte de las ideas establecidas en el *paper "On the Functional Interpretation of OCL"* [4]. En dicho artículo se presenta la idea de poder representar modelos utilizando programación funcional y utilizar esta representación para realizar una validación no estructural del modelo.

3.1 On the Functional Interpretation of OCL

El término *Interpretación Funcional de OCL* refiere a la representación de los elementos y características de OCL, por ejemplo herencia de tipos y navegación, en un lenguaje funcional, como puede ser Haskell. En este caso, la representación de OCL se limita a las invariantes que se utilizan para validar las restricciones no estructurales de los modelos, dejando de lado características como la descripción de pre y post condiciones sobre operadores.

Si bien la interpretación de un lenguaje a otro no es directa, ya que OCL está orientado a la representación de modelos orientados a objetos y los lenguajes de programación funcional usan funciones y tipos algebraicos, hay algunas características de OCL que tienen una representación directa en lenguajes de programación funcional: *pattern matching*, expresiones lambda y evaluación perezosa.

En el artículo se propone el uso de Haskell para representar la interpretación de la validación de las invariantes en los modelos. Para que un modelo sea una instancia válida de su correspondiente metamodelo, además de cumplir con las invariantes, debe cumplir con las reglas estructurales del metamodelo. Para representar los modelos y metamodelos en código Haskell, se propone implementar una transformación de modelo a texto, de forma tal que la transformación de un lenguaje a otro sea automática. En el artículo los autores proveen los lineamientos para implementar las características de UML que se listan a continuación, demostrando su aplicación con un modelo y metamodelo de ejemplo.

- Clases y jerarquías
- *Datatypes* y enumeraciones
- Propiedades y multiplicidades
- Modelos
- Navegación y propiedades heredadas

Una vez adquiridos los conocimientos generales para implementar estas características, es posible ampliar el lenguaje para poder incluir los conceptos de OCL. Esto hace posible definir tipos y valores definidos por el usuario, representar operadores primitivos (ejemplo: <, > para enteros), la navegación entre propiedades y funciones sobre operaciones, entre otras. A partir de las definiciones y representaciones de tipos, operadores y conectores, se propone implementar una biblioteca que contenga todas las funciones necesarias para poder traducir modelos, metamodelos e invariantes. Dicha biblioteca incluye un núcleo básico de características de OCL, que luego será ampliado en el contexto de este proyecto.

Para representar la realidad las invariantes deben ser aplicadas sobre un modelo. Para representar el modelo se define un *datatype Model* y para la representación de elementos de OCL se define la mónada *OCL Model (Val a)*. Por ejemplo, las invariantes suelen retornar un booleano en el contexto de OCL, por lo que sus definiciones devolverán una computación de

tipo *OCL Model (Val Bool)*. En la Figura 3.1 se muestra la implementación de esta mónada basándose en la mónada *Reader*. Dicho tipo no solo permite representar los cuatro valores de retorno de la validación de invariantes (*True*, *False*, *Invalid* y *Null*), sino que además permite retornar tipos intermedios que devuelven las consultas OCL, ya que puede tomar valores de cualquier tipo (polimorfismo sobre el tipo a).

$$\textbf{type } OCL\ m\ a = Reader\ m\ a$$
$$\textbf{data } Val\ a \quad = Null \mid Inv \mid Val\ a$$

Figure 3.1: Definición de tipos
[4]

En la biblioteca también se incluyen funciones que corresponden a los operadores de navegación entre objetos y navegación entre colecciones, las cuales hacen uso explícito de la clase mónada definida anteriormente. Además, se definen funciones para operadores de colecciones en OCL, por ejemplo *iterate* y *collect*. Los mismos pueden ser traducidos de forma casi directa de las operaciones *fold* y *map* de la biblioteca estándar de Haskell. Se considera casi directa porque todas las computaciones son monádicas, por lo cual se definen funciones intermedias que utilizan la lógica de las mónadas. Para completar la traducción se define la función *pureOCL*, la cual asegura que solo es posible invocar a una función si el valor que se recibe es un valor válido. En el la Figura 3.2 se muestran las definiciones descritas anteriormente.

```
iterate :: (Val b -> Val a -> OCL m (Val b)) -> Val b -> Val [Val a] -> OCL m (Val b)
iterate f b = pureOCL (foldM f b)

collect :: (Val a -> OCL m (Val b)) -> Val [Val a] -> OCL m (Val [Val b])
collect f = pureOCL (\l -> mapM f l >>= oclVal)

pureOCL :: (a -> OCL m (Val b)) -> Val a -> OCL m (Val b)
pureOCL f (Val x) = f x
pureOCL _ Inv     = oclInv
pureOCL _ Null    = oclNull
```

Figure 3.2: Implementación de *iterate* y *collect* en [4]

En base a los conocimientos de OCL adquiridos en el Marco Teórico, es necesario identificar el contexto en el que la invariante va a ser ejecutada. Con este fin, se define una función *context*, la cual verifica una lista de invariantes definidas para un contexto dado.La Figura 3.3 muestra la implementación en Haskell de la función *context*-

Para la representación de los metamodelos, los autores se basaron en la estructura Zipper [36]. Esta estructura resuelve el problema de recorrer árboles, pudiendo ir en cualquier sentido, desde arriba hacia abajo y viceversa. Los elementos se representan como tipos y se definen como el conjunto {elemento, supertipo inmediato}, donde el supertipo inmediato corresponde al nivel siguiente en la jerarquía. Para que la representación de todos los ele-

```
context :: (OCLModel m e, Cast m e a) => Val a -> [Val a-> OCL m (Val Bool)] -> OCL m (Val Bool)
context self inv = return self |.| allInstances |->| forAll (mapInvs inv)

mapInvs :: [Val a -> OCL m (Val Bool)] ->Val  a -> OCL m (Val Bool)
mapInvs is e = andOCL <$> mapM ($ e) is
```

Figure 3.3: Implementación de *context* en [4]

mentos sea válida, en el caso del elemento con mayor rango en la jerarquía, es decir, que no tiene ningún padre, se define el *datatype* Top.

Luego, los elementos del metamodelo se representan creando un *datatype* para cada uno de ellos, asociándoles la información necesaria para lograr la representación de atributos, relaciones y herencia. Una explicación más detallada de cómo se definen estos *datatypes* y las funciones de acceso a las propiedades del elemento se puede encuentra en el artículo.

Por último, los autores presentan los lineamientos para la implementación de un intérprete automático y entre los trabajos futuros planteados se encuentra la implementación del mismo en el contexto de Eclipse. La bases de la explicación y el desarrollo en el que se enmarca este proyecto parten de esos lineamientos.

3.2 Interpretación Funcional de un Metamodelo

El primer desafío que presenta la definición de modelos utilizando programación funcional es lograr definir las distintas partes del metamodelo asociado de modo de que (a) se pueda definir el modelo en base a cómo fuera definido el metamodelo, (b) se pueda utilizar a la hora de la evaluación de restricciones no funcionales y (c) permita emular el comportamiento del modelo necesario para la evaluación.

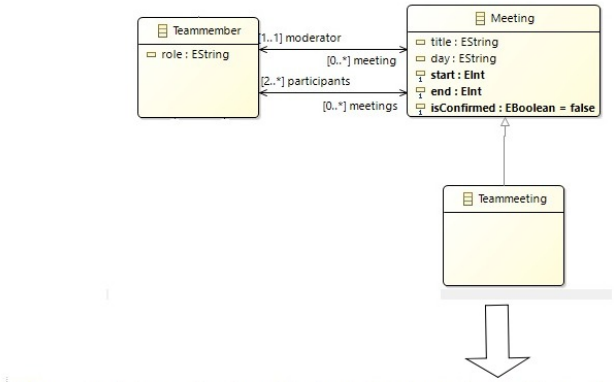
La interpretación funcional del metamodelo debe resolver cómo implementar cinco aspectos de UML para definir un modelo: clases, atributos, referencias, herencias e invariantes.

Definición funcional de clases
Cada una de las clases del metamodelo será representada en Haskell con un *datatype* equivalente, manteniendo su nombre, y agregando como campos del único constructor la información de sus atributos, referencias y herencias.

La Figura 3.4 muestra como, a partir de la clase *Meeting* definida en el metamodelo de ejemplo[1], se genera un *datatype* en Haskell que corresponde a su representación.

Los **atributos** serán representados con su correspondiente tipo. Estos atributos luego, a partir de su posición en el constructor, tendrán asociada una función que permita interactuar con cada uno de ellos (en el siguiente punto se detalla el funcionamiento de estas funciones).

[1] A partir de esta sección, los ejemplos utilizan el mismo modelo de ejemplo, *TeamMeeting*, que es utilizado en [6]




```
data    Meeting = Meeting (String) (String) (Int) (Int) (Bool) ([Int]) ([Int]) (MeetingChild)
        deriving (Eq, Show)

data    MeetingChild = TeammeetingCh (Teammeeting)
        deriving (Eq, Show)
```

Figure 3.4: Definición de una clase en Haskell

Para aquellas clases que cuenten con **referencias** las mismas serán representadas en este punto como listas de enteros. De forma muy similar a los atributos, las referencias también tendrán definidas funciones que permitan acceder a su contenido. Cada entero dentro de la lista que representa la referencia corresponderá con el índice único asociado a cada uno de los elementos del modelo. Para el caso de referencias cuya cardinalidad sea 1, la interpretación para a ser directamente utilizando un entero (que representará el índice del elemento referenciado en el modelo), evitando así la necesidad de tener que trabajar con listas para esos casos particulares.

Por último, el manejo de **herencias** se hace a través de la definición de un nuevo *datatype*. Este nuevo *datatype* podrá tener más de un constructor en caso de que exista más de una clase en la herencia. Para cada hijo de la clase en el metamodelo se generará un nuevo constructor en el *datatype*, permitiendo así que se pueda establecer a qué hijo se está haciendo referencia en la herencia. En la Figura 3.4 se puede ver un caso de definición de herencia con un solo hijo.

Definición funcional de atributos y referencias

En la definición de clases se menciona, la representación que se hace de los atributos y referencias. A nivel del *datatype*, la estructura generada que contiene la información de una instancia de una clase del modelo, los atributos están simplemente representados por un campo del constructor. Esto mismo ocurre para las referencias.

Para poder tener una interpretación completa de OCL resulta vital contar con una forma de acceder al valor de dichos atributos y relaciones de forma directa. Por ejemplo, como

se muestra en la Figura 3.5, para la clase *Team*, va a ser necesario contar con una función que permita obtener el valor del atributo *name*, y los valores de las referencias *members* y *teammeetings*.

```
name :: Cast Model UMLModelElement_ a => Val a -> OCL Model (Val String)
name a = upCast _UMLModelElement a >>= pureOCL( \ (UMLModelElement _ _ x _,_) -> return (Val x))
```

Figure 3.5: Definición de funciones para interactuar con valores de atributos y referencias

A nivel de la interpretación funcional será necesario generar estas funciones para cada uno de los atributos y referencias de una clase.

1. Referenciar a un atributo de la clase, donde se obtiene el valor de dicho atributo (Figura 3.6).

```
name :: Cast Model Person_ a => Val a -> OCL Model (Val String)
name a = upCast _Person a >>= pureOCL( \ (Person x _ _ _ _ , _) -> return (Val x))
```

Figure 3.6: Representación Funcional de un Atributo

2. Referenciar a una relación entre clases, en cuyo caso se obtiene la lista de clases en dicha relación (Figura 3.7).

```
elements :: Cast Model Package_ a => Val a -> OCL Model (Val [Val Classifier_])
elements a = upCast _Package a >>= pureOCL (\ (Package x,_)
                                   -> mapM (lookupM _Classifier) x >>= oclVal)
```

Figure 3.7: Representación Funcional de una Relación

3. Referenciar a una relación entre clases donde la misma tiene cardinalidad 1 y, por lo tanto, retornará siempre un único elemento (Figura 3.8).

```
owner :: Cast Model Attribute_ a => Val a -> OCL Model (Val Class_)
owner a = upCast _Attribute a >>= pureOCL (\ (Attribute _ x,_)
                                      -> lookupM _Class x)
```

Figure 3.8: Representación Funcional de una Relación con cardinalidad 1

Definición funcional de herencias

En la definición de clases ya queda establecida la herencia para una clase y sus posibles hijos. El desafío a nivel funcional con la herencia no pasa tanto por la implementación de la misma, sino por cómo lograr recorrer el árbol de herencias y determinar si una clase hereda de otra.

Pensándolo a nivel de OCL, esto va a ser necesario para manejar consultas del estilo *oclIsTypeOf* y *oclIsKindOf*, entre otras.

En definitiva, resulta necesario contar con una forma funcional de recorrer la jerarquía del modelo. Para ello, se define en Haskell una clase *Cast* que permite establecer una relación de jerarquía entre dos *datatypes* previamente definidos y cuya implementación se puede ver en la Figura 3.9.

```
class Cast m a b where
 downCast :: Val b -> Val a -> OCL m (Val b)
 upCast   :: Val a -> Val b -> OCL m (Val a)


instance Cast m a a where
 downCast _ a = return a
 upCast   _ a = return a


instance {-# OVERLAPS #-} Cast m a b where
 downCast _ _ = return Inv
 upCast   _ _ = return Inv
```

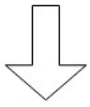
Figure 3.9: Definición clase *Cast*

Definición funcional de invariantes

Hasta ahora se vio cómo se definen los componentes estructurales del metamodelo, generando así una representación del mismo utilizando un lenguaje de programación funcional. Para complementar faltaría contar con una forma de representar los elementos no estructurales del modelo, específicamente, las invariantes.

A partir de la definición funcional de OCL (ver 4.3) y la definición de las estructuras del modelo vistas anteriormente, se transforman las invariantes asociadas al modelo, desde su definición en OCL, al equivalente en la interpretación funcional, como se puede apreciar en el ejemplo de la Figura 3.10.

```
context Meeting inv: self.end > self.start
```



```
invariant1 = context _Meeting [inv1]
inv1 self = (ocl self |.| end) |>| (ocl self |.| start)
```

Figure 3.10: Definición de una invariante en Haskell

3.3 Interpretación Funcional de un Modelo

Un modelo puede ser visto como un conjunto de elementos que se corresponden con la definición de uno de los elementos del metamodelo asociado. Estos estarán relacionados entre si, en base a las distintas relaciones establecidas en el metamodelo.

En [4] se presenta una idea de cómo realizar la representación de un modelo a nivel funcional, donde se puedan mantener las estructuras de identidad y relacionamiento necesarias. Esta representación tiene dos pilares fundamentales, (1) almacenar la información de cada entidad del modelo dentro de una colección y (2) poder identificar cada elemento a partir de un número identificador.

1. **Modelo como colección**
   A nivel funcional las colecciones de elementos tienen que ser todas de un mismo tipo. El problema a resolver, para poder definir un modelo como una colección de sus elementos, se reduce a definir un supertipo, o tipo raíz, que represente cualquier elemento del modelo.

2. **Identificación de elementos**
   La propuesta es sencilla, es necesario tener un valor único que identifique cada uno de los elementos del modelo, de modo que puedan ser referenciados. La forma simple de hacer esto es mediante un número de identificación que se le asigna a cada entidad.

   En la Figura 3.11 se puede apreciar cómo, en la definición de *UMLModelElement*, supertipo del modelo Ecore, se definió un primer atributo de tipo entero, que representará el número identificador único de cada elemento del modelo.

   Este identificador es la razón por la que, al definir la representación funcional de relaciones, se hace mediante una lista de enteros. Estos números serán los identificadores de los elementos del modelo parte de la relación.

```
Model [ UMLModelElement 1 "Persistent" "Pack1"  (UMLMECPck $ Package [3,4,6,7])
      , UMLModelElement 2 "Persistent" "Pack2"  (UMLMECPck $ Package [5,8])
      , UMLModelElement 3 "Persistent" "String" (UMLMECCla $ Classifier 1
                                                (Just $ ClassifierChPri  PrimitiveDataType))
      , UMLModelElement 4 "Persistent" "PDT1"   (UMLMECCla $ Classifier 1
                                                (Just $ ClassifierChPri  PrimitiveDataType))
      , UMLModelElement 5 "Persistent" "PDT2"   (UMLMECCla $ Classifier 2
                                                (Just $ ClassifierChPri  PrimitiveDataType))
      , UMLModelElement 6 "Persistent" "Class1" (UMLMECCla $ Classifier 1
                                                (Just $ ClassifierChCla (Class [9,10])))
      , UMLModelElement 7 "Persistent" "Class2" (UMLMECCla $ Classifier 1
                                                (Just $ ClassifierChCla (Class [11])))
      , UMLModelElement 8 "Persistent" "Class3" (UMLMECCla $ Classifier 2
                                                (Just $ ClassifierChCla (Class [12])))
      , UMLModelElement 9 "Persistent" "Att1"   (UMLMECAtt $ Attribute  4 6)
      , UMLModelElement 10 "Persistent" "Att2"  (UMLMECAtt $ Attribute  3 6)
      , UMLModelElement 11 "Persistent" "Att3"  (UMLMECAtt $ Attribute  4 7)
      , UMLModelElement 12 "Persistent" "Att4"  (UMLMECAtt $ Attribute  5 8)
      ]
```

Figure 3.11: Definición de un Modelo

4 Interpretación de Modelos

El objetivo de este trabajo consiste en establecer una vía alternativa para la evaluación automática de restricciones no estructurales en modelos, haciendo uso del paradigma funcional. Se basa en el *paper "On the Functional Interpretation of OCL"* [4], donde se presenta la idea de poder representar modelos utilizando programación funcional. Este trabajo extiende dichas definiciones procurando lograr una interpretación de modelos en Haskell y, a partir de esta definición, transformar los modelos de entrada y utilizar el código generado para validar el cumplimiento de sus invariantes OCL. Los resultados obtenidos en este proyecto formaron parte del trabajo *"Model-Driven Development of an Interpreter for the Object Constraint Language"* [6], que fuera presentado en la Conferencia de Latinoamericana de Informática de 2018.

En [4] se presentan las bases para crear una interpretación funcional de modelos e invariantes OCL utilizando Haskell. Partiendo de esa base, la primera etapa de este trabajo consistió en extender estos conceptos para lograr una interpretación lo más completa posible.

El código Haskell generado correspondiente a la interpretación realizada de los modelos se dividirá en 2 partes. La primera tendrá la interpretación del modelo a procesar, conteniendo toda la información del mismo, así como de su metamodelo. En este caso, el código de la representación dependerá directamente del modelo a procesar, por lo que será generado utilizando una transformación M2T a la hora de la evaluación.

Para la segunda parte resulta necesario contar con una definición del lenguaje OCL que brinde soporte a la evaluación de los invariantes asociados al modelo, objetivo central de este trabajo. Esta definición de OCL escrita en Haskell es independiente del modelo a procesar, convirtiéndolo en un activo estático del proyecto.

4.1 Interpretación Funcional de un Metamodelo

Manteniendo presente el poder definir las distintas partes del metamodelo asociado a un modelo de modo de que (a) se pueda definir el modelo en base a cómo fuera definido el metamodelo, (b) se pueda utilizar a la hora de la evaluación de restricciones no funcionales y (c) permita emular el comportamiento del modelo necesario para la evaluación, el objetivo en esta sección es extender el trabajo previo a partir del que se generó este proyecto.

Con la idea básica para la definición del metamodelo en lenguaje funcional ya diseñada, simplemente se tuvieron que hacer los ajustes necesarios para soportar funcionalidades previamente no consideradas (por ejemplo, manejo de colecciones o clases abstractas).

Definición funcional de clases
Para cada clase del metamodelo se mantiene la representación en Haskell a través de un *datatype*, con la información de sus atributos, referencias y herencia como campos del constructor.

A nivel de **herencias**, una variante a resolver es el de las **clases instanciables**. Hasta ahora, en las herencias se consideraba que siempre debían ser representadas a través de uno de sus hijos, asumiendo que la misma fuera una **clase abstracta**. Sin embargo, para el caso de clases instanciables, podría ocurrir que no hubiera necesidad de detallar el tipo de hijo en los casos donde se estuviera instanciando la propia clase. Para poder representar este comportamiento se recurre al tipo *Maybe*.

```
data Meeting    = Meeting (String) (String) (Int) (Int) (Bool) ([Int]) ([Int]) (Maybe MeetingChild)
        deriving (Eq, Show)

data MeetingChild   = TeamMeetingCh (TeamMeeting)
        deriving (Eq, Show)
```

Figure 4.1: Ejemplo de herencia con *Maybe*

Como se puede apreciar en la Figura 4.1, *Maybe* es agregada como parte de la definición de una herencia donde la clase padre es instanciable (no abstracta), permitiendo definir el tipo del hijo, en caso de que hubiera, (constructor *Just*) o indicar que no se trata de un hijo, sino que se está queriendo instanciar directamente la clase (constructor *Nothing*).

Definición funcional de atributos y referencias
Partiendo de las ideas en [4], se mantienen las tres posibles invocaciones a atributos y referencias:

1. Referenciar a un atributo de la clase, donde se obtiene el valor de dicho atributo.
2. Referenciar a una relación entre clases, en cuyo caso se obtiene la colección de clases en dicha relación.

3. Referenciar a una relación entre clases donde la misma tiene cardinalidad 1 y, por lo tanto, retornará siempre un único elemento.

Sin embargo, con el agregado de soporte a colecciones implementado en este trabajo, el manejo de referencias a clases cuya cardinalidad no está fija en 1 se modificó, ya que ahora se retornará un elemento de tipo `Collection a`, en lugar de una lista de Haskell (`[a]`) como se sugería en el trabajo original.

La Figura 4.2 muestra un ejemplo de la nueva implementación de la función para referenciar una relación entre clases, donde ahora el tipo de retorno será una colección. La implementación de la búsqueda se mantiene, simplemente se agrega una transformación de lista a colección para reflejar el cambio en el tipo de retorno.

```
account :: Cast Model Membership_ a => Val a -> OCL Model (Val (Collection LoyaltyAccount_))
account a = liftM (toCollection) (upCast _Membership a >>= pureOCL( \ (Membership _ _ _ x _ , _)
                                    -> mapM (lookupM _LoyaltyAccount) x >>= oclVal))
```

Figure 4.2: Ejemplo de referencia con colecciones

Definición funcional de herencias
Estudiando el uso de la clase *Cast* previamente definida, y tomando en cuenta el agregado para manejo de clases instanciables en la herencia (uso del tipo *Maybe*), se detecta la necesidad de contar con una tercera regla para recorrer el árbol de jerarquía, además de las funciones *upCast* y *downCast*. El comportamiento que faltaba modelar era justamente la variante en una clase instanciable para que, dependiendo de si tiene herencia (constructor *Just*) o no (constructor *Nothing*) se pueda determinar si es una hoja del árbol. Este comportamiento es el modelado por medio de la función *directInstance*.

La Figura 4.3 muestra la nueva definición para la clase *Cast* con el agregado de la función *directInstance*.

Para poder generar una definición completa, que abarque todos los escenarios de la representación de las jerarquías de un modelo utilizando instancias de la clase *Cast*, se establecen cuatro reglas.

1. **Identidad**
   Toda clase del modelo es instancia de si misma (Figura 4.4).

2. **Herencia Directa**
   Toda relación directa de herencia entre 2 clases del modelo será representada por una instancia de *Cast* (Figura 4.5).

3. **Transitiva**
   Si existe una instancia de *Cast* que define una herencia entre los tipos A y B, y existe otra instancia de *Cast* que define una herencia entre los tipos B y C, entonces se debe definir una nueva regla de herencia entre A y C (Figura 4.6).

```
class Cast m a b where
    downCast        :: Val b -> Val a -> OCL m (Val b)
    directInstance  :: Val b -> Val a -> OCL m (Val b)
    upCast          :: Val a -> Val b -> OCL m (Val a)

instance {-# OVERLAPS #-} Cast m a a where
    downCast        _ a = return a
    directInstance  _ _ = return Inv
    upCast          _ a = return a

instance {-# OVERLAPS #-} Cast m a b where
    downCast        _ _ = return Inv
    directInstance  _ _ = return Inv
    upCast          _ _ = return Inv
```

Figure 4.3: Definición clase *Cast*

```
instance Cast Model Teammeeting_ Teammeeting_ where
    downCast        _ a = return a
    directInstance  _ _ = return Inv
    upCast          _ a = return a
```

Figure 4.4: *Cast*, regla Identidad

```
instance Cast Model Person_ Teammember_ where
    downCast _ (Val e@(Person _ _ _ _ (TeammemberCh x), _ModelElement))
                                                = return $ Val (x,e)
    downCast _ _ = return Inv
    upCast   _ (Val (_,e)) = return (Val e)
```

Figure 4.5: *Cast*, regla Herencia Directa

```
instance Cast Model ModelElement_ Meeting_ where
    downCast _ (Val e@(ModelElement _ (MeetingCh x), Top)) = return $ Val (x,e)
    downCast _ _ = return Inv
    upCast   _ (Val (_,e)) = return (Val e)

instance Cast Model Meeting_ Teammeeting_ where
    downCast _ (Val e@(Meeting _ _ _ _ _ _ _ (TeammeetingCh x), _ModelElement))
                                                = return $ Val (x,e)
    downCast _ _ = return Inv
    upCast   _ (Val (_,e)) = return (Val e)

instance Cast Model Meeting_ Teammeeting_ => Cast Model ModelElement_ Teammeeting_ where
    downCast t (Val e@(ModelElement _ (MeetingCh x), Top)) = downCast t (Val (x,e))
    downCast _ _ = return Inv
    upCast   t (Val (_,e)) = upCast t (Val e)
```

Figure 4.6: *Cast*, regla Transitiva

4. **Herencia de ModelElement**
   Para toda clase A, existirá una relación de herencia definida en una instancia de *Cast* desde ModelElement hacia A (Figura 4.7).

```
instance Cast Model ModelElement_ Team_ where
    downCast _ (Val e@(ModelElement _ (TeamCh x), Top)) = return $ Val (x,e)
    downCast _ _ = return Inv
    upCast   _ (Val (_,e)) = return (Val e)
```

Figure 4.7: *Cast*, regla Herencia de *ModelElement*

Definición funcional de invariantes
La forma de definición de invariantes mantiene la estructura definida en [4] y presentada en la Sección 3.2, ya que dicha representación permite, inclusive con los cambios y nuevas funcionalidades implementadas, seguir realizando la evaluación de invariantes OCL para los modelos.

En este punto, los cambios se dieron en la extensión del alcance de OCL del que se logró una implementación en Haskell. Ese tema será abordado en profundidad en la Sección 4.3.

4.2 Interpretación Funcional de un Modelo

En la sección 3.3 se describe la idea presentada en [4] para la representación de un modelo a nivel funcional. En ella, las dos ideas principales eran:

- almacenar la información de cada entidad del modelo dentro de una colección.
- poder identificar cada elemento a partir de un número identificador.

Para ello ello se establecía la idea de representar un modelo como una **colección**.

1. **Modelo como colección**
   El concepto sugiere considerar el modelo como una lista de elementos del supertipo del modelo en la definición Ecore. El problema con esta idea era poder reconocer cuál es dicho supertipo. Para no tener que lidiar con este problema, se plantea la idea de que la propia interpretación cree un supertipo fijo.

   La solución planteada sugiere crear un *datatype* extra a las clases del metamodelo que se generan durante su interpretación (ver sección Definición funcional de clases en 4.1). Es aquí que surge el concepto de *ModelElement* como elemento raíz que puede representar a cualquier elemento del modelo. La definición en Haskell para el caso de ejemplo se puede ver en la Figura 4.8, donde se establece el *datatype ModelElement*

que puede luego definirse en base a cualquiera de las clases del metamodelo, dependiendo del constructor que se utilice para definir *ModelElementCh*.

```
data    ModelElement    = ModelElement (Int) (ModelElementChild)
    deriving (Eq, Show)

data    ModelElementChild   = PersonCh (Person)
                            | TeamCh (Team)
                            | MeetingCh (Meeting)
                            | RootCh (Root)
    deriving (Eq, Show)
```

Figure 4.8: Definición del supertipo *ModelElement*

2. **Identificación de elementos**
   Manteniendo la idea de tener un número identificador para cada elemento del modelo, para así poder individualizarlos, es que se agrega a la definición de *ModelElement* un atributo de tipo *Int* que servirá como identificador para todos los elementos.

4.3 Interpretación Funcional de OCL

Hasta ahora se vio cómo lograr la representación de un modelo UML en lenguaje funcional. Tener la definición del modelo resultará necesario para poder obtener la información del mismo a la hora de realizar las validación de invariantes OCL. Lo que está faltando es definir cómo se implementaron, utilizando Haskell, los distintos componentes de OCL para lograr que las invariantes puedan ser evaluadas.

En esta sección se mostrará la idea general para la definición del lenguaje OCL y, por lo tanto, de invariantes en base al mismo. Las mismas parten de las ideas establecidas en [4], que son adaptadas y extendidas para lograr abarcar el máximo alcance posible del lenguaje. Se entrará en detalle para algunas definiciones importantes, pero no se recorrerá el alcance total. La Tabla 4.1 muestra la evolución del alcance de OCL cubierto, partiendo de las bases establecidas en [4].

4.3.1 Un primer acercamiento

La forma más sencilla de presentar el enfoque para crear la representción de OCL basada en Haskell es mediante un ejemplo. La Figura 4.9 muestra la definición de una invariante sobre el modelo que se viene manejando en ejemplos anteriores. La invariante *moreThanOneParticipant* establece que toda *Meeting* del modelo deberá tener al menos dos *Participants* asociados.

```
context Meeting
    inv moreThanOneParticipant : self.participants->size() >= 2
```

Figure 4.9: Ejemplo invariante en OCL

Es una invariante muy sencilla, pero que servirá como ejemplo para definir las estructuras bases necesarias para la representación en Haskell. En la Figura 4.10 se puede ver cómo quedaría la representación de ese mismo invariante en la versión funcional de OCL desarrollada.

```
invariant1 = context _Meeting [moreThanOneParticipant]
moreThanOneParticipant self = ((ocl self |.| participants |->| size)) |>=| (oclInt 2)
```

Figure 4.10: Ejemplo invariante OCL en representación funcional

A simple vista es fácil notar la similitud que existe entre la sentencia OCL original y el equivalente en la representación funcional. Uno de los objetivos trazados originalmente fue poder tener una representación que, no solo emulara el funcionamiento, sino que también fuera lo más semejante posible, de modo que la equivalencia fuera sencilla de comprender.

	Trabajo Original [4]	HaskellOCL
Constructores OCL		
context Especifica el contexto para expresiones OCL	✓	✓
inv Establece una condición que siempre debe ser cumplida por todas las instancias de un contexto	✓	✓
init Establece el valor inicial de un atributo o una función de asociación	✗	✓
derive Establece el valor de un atributo derivado o una función de asociación	✗	✗
def Introduce un nuevo atributo o una operación de consulta	✗	✓
Expresiones OCL		
self Denota la instancia contextual	✓	✓
Navigation Navegación a través de atributos, fines de asociación, clases de asociación y asociaciones calificadas	−	−
if-then-else expression Expresión condicional con una condición y dos expresiones	✓	✓
let-in expression Expresión con variables locales	✗	✓
Biblioteca estándar de OCL		
Boolean Type Valores: `True` y `False`	✓	✓
Boolean operations Operaciones: `or, and, xor, not, =, <>, implies`	−	✓
Integer/Real Type Valores: `-10, 0, 10, ..., -1.5, 3.14, ...`	−	✓
Integer/Real Operations Operaciones: `=, <>, <, >, +, -, *, /, mod, div, max, round, ...`	−	✓
String Type Valores: `"value"`	✗	✓
OCLAny Supertipo de todos los tipos de UML y OCL	✗	✗
OCLAny Operations Operaciones definidas para cualquier tipo: `=, <>, oclIsNew, oclAsType, T::allInstances, ...`	−	−
OCLVoid Tipo con una única instancia (`undefined`) que conforma con los otros tipos	−	−
Tuple Type Una tupla consiste en partes nombradas, cada una de las cuales puede tener un tipo distinto	✗	✓
Collection Types Cuatro tipos de colecciones: `Set`, `OrederedSet`, `Bag` y `Sequence`	−	✓
Collection operations Operaciones: `any, append, asBag, count, collect, excludes, exists, first, ...`	−	−

Table 4.1: Alcance

Algunos de los elementos del lenguaje OCL presentes en el ejemplo y cuya definición es esencial para poder soportar la representación:

- *context*
- *self*
- navegación (*"."* y *"->"*)
- operaciones primitivas básicas (*">="*)
- literales (en este caso, un entero)
- operaciones sobre colecciones (*"size"*)

En el Anexo D está detallada la implementación funcional para cada una de estas funcionalidades, así como el resto de las operaciones que están en el alcance. Lo importante a tener en cuenta analizando esta lista es que, a grandes rasgos, tenemos un caso que cuenta con ejemplos de todas las estructuras que precisamos.

- ***context***
  Palabra reservada para determinar sobre qué tipo de elemento del metamodelo se está aplicando la invariante. La representación Haskell creará una función que permita definir una clase del metamodelo, a partir de su definición en Haskell vista en 4.1, y que limite la ejecución del control para la invariante a elementos del modelo que sean de este tipo.

  Toda invariante requiere de la definición de un *context* sobre el que operar, haciendo de su definición en Haskell una obligación.

- ***self***
  Como lo indica su nombre, esta representación sirve para referirse al elemento sobre el que se está evaluando. Usualmente las invariantes terminarán en algún momento utilizando información acerca del elemento sobre el que opera para determinar su resultando, por lo que contar con este tipo de acceso es vital.

- **navegación**
  La navegación OCL viene esencialmente en dos formas. Está el operador "." que, a partir de un elemento único, permite aplicarle, o bien una de las funciones para obtener atributos/relaciones que se definieron en 4.1, o aplicarle alguna de las funciones de OCL al elemento.

  En el ejemplo se puede ver un caso de obtención de información de una relación en base a la función para obtener datos que se definió al crear la representación del metamodelo (*self.participants*).

  La otra alternativa de navegación es "->". La diferencia con "." es que esta navegación es utilizada con una colección como entrada, aplicando una de las funciones para operar

sobre la colección y/o sus elementos.

En el ejemplo se puede apreciar el uso de la función *size* para obtener el tamaño de la colección generada por *participants*.

- **operaciones primitivas básicas**
  En el ejemplo hay una condición de mayor o igual que se debe cumplir para que la invariante sea válida. En general, se brinda soporte a todo tipo de operaciones aritméticas y sobre booleanos.

  La representación soporta las siguientes operaciones:

  ```
  +, -, *, /, div

  =, <>, <, <=, >, >=

  and, or, not

  implies
  ```

- **literales**
  La representación de OCL en lenguaje funcional tiene soporte para los siguientes tipos primitivos:

  ```
  Boolean

  Integer

  Real

  String
  ```

- **colecciones**
  El soporte a colecciones, específicamente *Bags*, *Sets*, *Sequences* y *OrderedSets*, permite extender las posibilidades de sentencias a procesar, por ejemplo, permitiendo manipular la información obtenida de una relación para un tipo de elemento dado. Justamente ese es el ejemplo que se puede apreciar en la invariante de la Figura 4.10, donde, dada la colección de *participants* obtenida, se ejecuta la operación *size* sobre la misma.

  En el Anexo D se puede encontrar toda la lista de funciones OCL sobre colecciones cuya implementación se pudo soportar utilizando Haskell. Por otro lado, más adelante en esta sección se analizará la implementación de la representación de colecciones, entrando más en detalle sobre la misma.

4.3.2 *OCLLibrary* - La biblioteca de OCL en Haskell

En las secciones anteriores el código Haskell generado en la interpretación tiene que ser procesado en el momento a partir de la información obtenida de modelo y metamodelo. Para el caso de la interpretación OCL, todo el trabajo de implementar las funcionalidades y soportar las distintas estructuras es común a todo proyecto, es por esto que ya en [4] se instala la idea de crear una biblioteca en Haskell que contenga todas aquellas definiciones de representaciones que no dependan del modelo y cuyas definiciones sean fácilmente consumidas desde cualquier proyecto.

La definición de toda estructura OCL mencionada anteriormente como parte de la representación estará contenida en esta biblioteca, así como toda funcionalidad que no esté definida en esta biblioteca implicará que la misma no está soportada.

Estructuras centrales de OCL, como las vistas en el ejemplo en 4.3.1 (*context*, navegación, literales y colecciones), serán definidas dentro de esta biblioteca, denominada ***OCLLibrary***, tal como muestran los respectivos ejemplos de las figuras 4.11, 4.12, 4.13 y 4.14.

```
context :: (OCLModel m e, Cast m e a)
        => Val a -> [Val a -> OCL m (Val Bool)] -> OCL m (Val Bool)
context self inv = return self |.| allInstances |->| forAll (mapInvs inv)

mapInvs :: [Val a -> OCL m (Val Bool)] -> Val a -> OCL m (Val Bool)
mapInvs is e = andOCL <$> mapM ($ e) is
```

Figure 4.11: Implementación de *context* en *OCLLibrary*

```
(|.|) :: OCL m (Val a) -> (Val a -> OCL m (Val b)) -> OCL m (Val b)
(|.|) = (>>=)
```

Figure 4.12: Implementación de operador *"."* en *OCLLibrary*

```
oclInt :: Int -> OCL m (Val Int)
oclInt = oclVal

oclVal :: a -> OCL m (Val a)
oclVal = return . Val
```

Figure 4.13: Implementación de *Integers* en *OCLLibrary*

Esta biblioteca fue creada partiendo de las bases establecidas por Daniel Calegari y Marcos Viera en su proyecto original, luego fue creciendo a medida que nuevas funcionalidades iban logrando ser implementadas, y puede seguir fácilmente extendiéndose con nuevas definiciones para aumentar el alcance de OCL representado.

```
size :: Val (Collection a) -> OCL m (Val Int)
size list = (oclVal . length) (collectionToList list)
```

Figure 4.14: Implementación de *size* en *OCLLibrary*

4.3.3 Navegación

La Figura 4.12 muestra la implementación de uno de los casos de navegación dentro de estructuras OCL. En ella, a partir de un elemento de tipo `OCL m (Val a)` y una función aplicada sobre el mismo (`((Val a -> OCL m (Val b))`), retorna el valor obtenido de dicha función.

Se puede navegar aplicando dos tipos distintos de funciones sobre el objeto, dependiendo de qué tipo de objeto se trate. Si se trata de un elemento del modelo, entonces las funciones con las que se navegará corresponderán con la obtención de información del mismo, por ejemplo invocando uno de sus atributos (ver Figura 4.15), o invocando una de sus relaciones (ver Figura 4.16.

```
ocl self |.| end
```

Figure 4.15: Navegación sobre un elemento del modelo hacia un atributo

```
ocl self |.| participants
```

Figure 4.16: Navegación sobre un elemento del modelo hacia una relación

La otra alternativa posible es que el elemento sea un literal sobre el que ya no habrán atributos o relaciones por los que navegar, pero si se le podrán aplicar métodos de OCL ya definidos. Un ejemplo de esto se puede ver en la Figura 4.17 donde, a un literal de tipo *Real* se le aplica la función *floor*.

```
oclDouble 6.5 |.| floorOCL
```

Figure 4.17: Navegación sobre un literal para aplicarle una función

Navegación en colecciones
Para la navegación sobre colecciones en OCL se utiliza la función de navegación *"->"*. La implementación funcional de OCL tiene, en la función `"|->|"` una función de navegación que reproduce ese comportamiento.

La Figura 4.18 muestra la implementación de `"|->|"`, que es análoga a la de `"|.|"` vista

```
(|->|) :: OCL m (Val (Collection a))
       -> (Val (Collection a) -> OCL m (Val b))
       -> OCL m (Val b)
(|->|) = (>>=)
```

Figure 4.18: Implementación de *"->"* en *OCLLibrary*

en la Figura 4.12, con la salvedad de que, en lugar de tomar como entrada un elemento OCL, ya sea del modelo o un literal, la entrada es de tipo *Collection*.

La navegación sobre colecciones puede utilizarse tanto para operar sobre relaciones de elementos, que retornan colecciones de enteros, o directamente sobre colecciones dadas de forma explícita. Las figuras 4.19 y 4.20 muestran ejemplos del uso de la navegación para ambos casos.

```
ocl self |.| participants |->| size
```

Figure 4.19: Navegación sobre una relación obtenida de un elemento

```
oclVal (Sequence [Val "E", Val "S", Val "T"])  |->| prepend (oclVal "X")
```

Figure 4.20: Navegación sobre una colección aplicándole una función

Shorthands

Hasta ahora la implementación vista de las funcionalidades de navegación, salvo algunos retoques, proviene de la investigación desarrollada en [4]. OCL tiene algunas variantes de navegación que dan mayor ductilidad al lenguaje. Esto son los *Shorthands*.

- El *shorthand* de navegación *"."* realiza de forma implícita un collect de una propiedad u operación sobre una colección.

  `aSet.name` es un *shorthand* para `aSet->collect(name)`

  La forma en Haskell de realizar sobrecarga de funciones, que es lo que OCL realiza con el operador *"."*, es mediante el uso de *Type Classes*. Hacer uso de *Type Classes* para el operador *"."* resulta en modificar la implementación anterior. Es por ello que se decide crear una segunda función en *oclLibrary* que represente este comportamiento. La Figura 4.21 muestra la implementación de la función `"|..|"`, que simula el comportamiento del *shorthand*, tomando como entrada una colección, y aplicando *collect* sobre la misma con la función de entrada como parámetro.

- El *shorthand* de navegación *"->"* realiza de forma implícita una conversión a *set* del objeto.

```
(|..|) :: OCL m (Val (Collection a)) -> (Val a -> OCL m (Val b))
          -> OCL m (Val (Collection b))
(|..|) xs f = xs |->| (collect f)
```

Figure 4.21: Implementación *shorthand "."*

`anObject->union(aSet)` es un *shorthand* para
`anObject.oclAsSet()->union(aSet)`

Nuevamente, para no tener que recurrir a la sobrecarga de funciones de Haskell utilizando *Type Classes*, se crea una función alternativa en *oclLibrary* que represente este comportamiento. La Figura 4.22 muestra la implementación de la función `"|->|"`, que simula el comportamiento de este *shorthand*, tomando como entrada un elemento, y aplicando *oclAsSet* sobre el mismo, previo a aplicar la función.

```
(|-->|) :: OCL m (Val a)
           -> (Val (Collection a) -> OCL m (Val b))
           -> OCL m (Val b)
(|-->|) o f = (o |.| oclAsSet) |->| f
```

Figure 4.22: Implementación *shorthand "->"*

	Objeto como entrada	**Colección como entrada**
.	Operador de Navegación sobre objetos. Implementado en *oclLibrary* como `"|.|"`	*Shorthand* con aplicación de *collect* implícito. Implementado en *oclLibrary* como `"|..|"`
->	*Shorthand* con conversión a Set implícita. Implementado en *oclLibrary* como `"|->|"`	Operador de Navegación sobre colecciones. Implementado en *oclLibrary* como `"|-->|"`

4.3.4 Colecciones

OCL soporta cuatro tipos de colecciones, *Bags*, *Sets*, *Sequences* y *OrderedSets*. Como la mayoría de las funciones OCL que toman una colección como entrada lo hacen de forma indistinta entre los cuatro tipos o un subcojunto de ellos, para no necesitar varias funciones en Haskell para implementar la misma función OCL, se decide implementar las colecciones como un único tipo en Haskell que luego pueda transformarse en una de las cuatro variantes. Para ello se definen las colecciones utilizando el *datatype* de la Figura 4.23.

Esta definición permitirá a funciones sobre colecciones tomar como tipo de entrada el *datatype* definido y luego, en la propia implementación de la función, se determina el comportamiento

```
data Collection a    = Bag [Val a]
                     | Sequence [Val a]
                     | Set [Val a]
                     | OrderedSet [Val a]
    deriving (Eq, Show)
```

Figure 4.23: Definición de *Collection*

dependiendo de qué tipo de colección es. En la Figura 4.24 se puede apreciar como la función *including* es implementada y, utilizando *pattern matching*, se determina cuál va a ser el comportamiento dependiendo del tipo de la colección de entrada.

```
including :: Eq a => OCL m (Val a) -> Val (Collection a)
                                   -> OCL m (Val (Collection a))
including elem col = liftM2 (oclIncluding) elem (ocl col)

oclIncluding x  (Val (Set c))           = if (elem x c) then (Val (Set c))
                                          else oclAppend(Val (Set c)) x
oclIncluding x  (Val (OrderedSet c))    = if (elem x c) then (Val (Set c))
                                          else oclAppend (Val (OrderedSet c)) x
oclIncluding x  (Val (Bag c))           = oclAppend (Val (Bag c)) x
oclIncluding x  (Val (Sequence c))      = oclAppend (Val (Sequence c)) x
```

Figure 4.24: Implementación de *including*

4.4 Limitaciones y pendientes

Este trabajo representa una primera aproximación en la definición de modelos y del lenguaje OCL utilizando programación funcional. Como toda primera aproximación, si bien logra demostrar que es posible implementar una interpretación y utilizarla para validar modelos, se restringe a estudiar una parte del problema; las invariantes, ya que se considera el pilar fundamental sobre el que luego se podrá ampliar el estudio.

En esta sección se detallarán las restricciones que tiene el estudio, específicamente qué secciones de OCL fueron consideradas. y para las cuales se validó esta forma de evaluar modelos es posible de realizar, cuáles no se tuvieron en cuenta para este primer análisis, y quedan para futuras extensiones de este estudio, y qué limitaciones se le encontraron a lo que fue analizado, que llevó a descartar la posibilidad de representar ciertas estructuras utilizando lenguaje funcional.

Generalidades de lo que abarca el trabajo se vieron en las anteriores secciones de este capítulo y se detallan en el anexo D, quedando por presentar los componentes de OCL que no están contemplados. Estos se categorizan en dos tipos:

- **Limitaciones**
  Aquellas estructuras de OCL que hayan sido parte del análisis, pero para las cuales se encontraron inconvenientes a la hora de generar una implementación funcional que emulara su comportamiento, lo que llevó a descartar su inclusión o restringir su uso a aquel comportamiento que sí se pudo modelar.
- **Pendientes**
  Componentes de OCL utilizados para validaciones OCL que no entraron dentro del alcance del proyecto y que se espera se estudie añadirlos en futuras iteraciones.

4.4.1 Limitaciones

Las siguientes funciones de OCL se encontraban dentro del alcance inicial del proyecto pero no fueron incluidas en la versión final porque se detectó que no era posible su implementación utilizando Haskell:

- ***flatten()* - operaciones sobre colecciones**
  La función *flatten*, aplicada sobre una colección de colecciones de elementos de tipo `a`, retorna una colección de elementos de tipo `a` con todos los elementos de las colecciones anidadas, como se muestra en el diagrama de la Figura 4.25.

  Emular este comportamiento utilizando programación funcional resulta sencillo en un ejemplo simple como el anterior, donde hay un único nivel que bajar. Basta con tomar

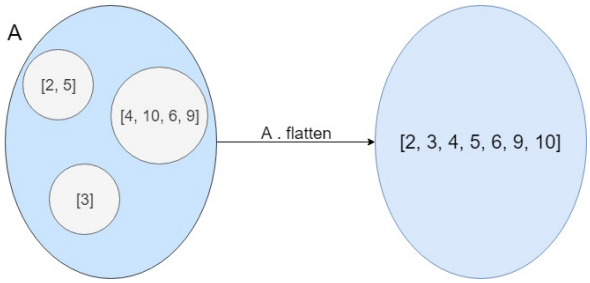


Figure 4.25: Aplicar *flatten* a una colección de colecciones de enteros

los elementos de cada una de las colecciones internas y agregarlos a una única colección (en el caso del ejemplo, *Bag*). La Figura 4.26 muestra una posible implementación válida de flatten para el ejemplo anterior.

```
flatten :: [[a]] -> [a]
flatten [] = []
flatten [x] = x
flatten (x:xs) = x ++ (flatten xs)
```

Figure 4.26: Código Haskell para resolver *flatten* de un nivel

La solución anterior es para un nivel de colecciones anidadas, el problema se comienza a complejizar cuando la cantidad de niveles de colecciones que hay que recorrer empieza a crecer. La Figura 4.27 muestra un ejemplo con 3 niveles de colecciones, donde la entrada es una colección de colecciones que contienen colecciones de colecciones de enteros.

Ya la solución inicial no aplica para este nuevo caso. Se necesita tener una función recursiva que recorra cada nivel de colecciones y vaya realizando el flatten de las mismas. Esto no representaría gran complejidad, realizar una recursión en Haskell es natural, pero la complejidad empieza a generarse a nivel del tipado de la función.

En el primer ejemplo teníamos una función que tenía el siguiente tipo en Haskell:

```
flatten ::  Collection (Collection Int) -> Collection Int
```

Mientras que para el ejemplo mostrado en la Figura 4.27 se podría definir una implementación de *flatten* con el siguiente cabezal.

```
flatten ::  Collection (Collection (Collection
     (Collection Int))) -> Collection Int
```

Si a partir de los dos ejemplos anteriores se quisiera encontrar una definición general

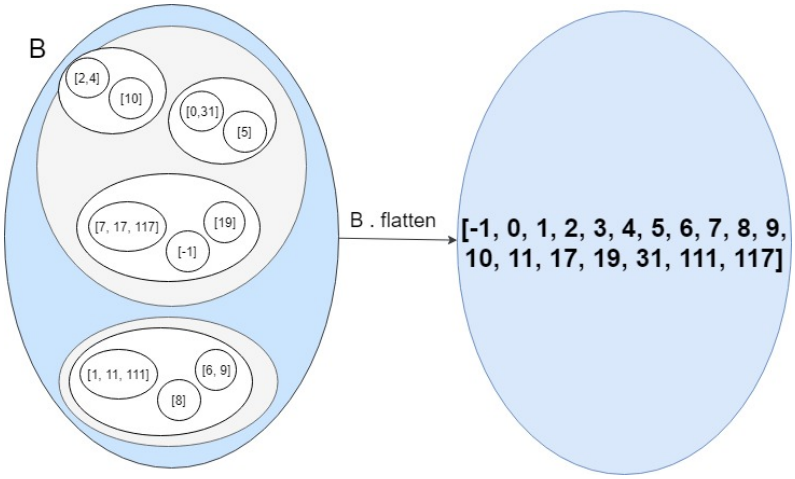


Figure 4.27: Aplicar *flatten* a una colección de colecciones que contienen colecciones de colecciones de enteros

para las colecciones de entrada, se podría pensar como una forma viable la siguiente definición:

```
flatten ::  Collection (Collection a) -> Collection b
```

Y acá comienzan los problemas de tipos. Para el primer caso, `a` representaría `Int`, pero para el segundo caso, `a` es de tipo `Collection (Collection Int)`. Pero generalizando el tema tipado, el problema pasa por no poder establecer una relación entre `b` y `a`, más específicamente, entre `b` y el tipo del que serán los elementos de las colecciones anidadas definidas en `a`.

Este problema de tipado se extiende también a la recursión. Se tendría que invocar *flatten* para cada una de las colecciones dentro de una colección. Sin embargo, si se hace eso, se está asumiendo que esas colecciones también pueden ser definidas como `Collection (Collection a)`, algo que no se aplica si la recursión es de un solo nivel, como en el primer caso presentado, donde los elementos dentro de la primera colección tienen tipo `Collection Int`.

Identificado este problema de tipado, se decide quitar *flatten* del alcance, ya que no se logró implementar una solución en programación funcional que permita emular completamente el comportamiento de la función y mantener la representación de modelos elegida.

A futuro se entiende podría agregarse al alcance una definición parcial de la función *flatten*, para un único nivel de colecciones anidadas, con una implementación similar a la que se muestra en la Figura 4.26.

- ***collect(expr)* - operaciones de iteración**
  La función *collect*, aplicada a una colección de elementos de tipo `a`, retorna una colección conteniendo el resultado de aplicar la expresión OCL *expr* a cada uno de los elementos de la colección original, previa aplicación de *flatten* sobre la misma, como se muestra en el ejemplo de la Figura 4.28.

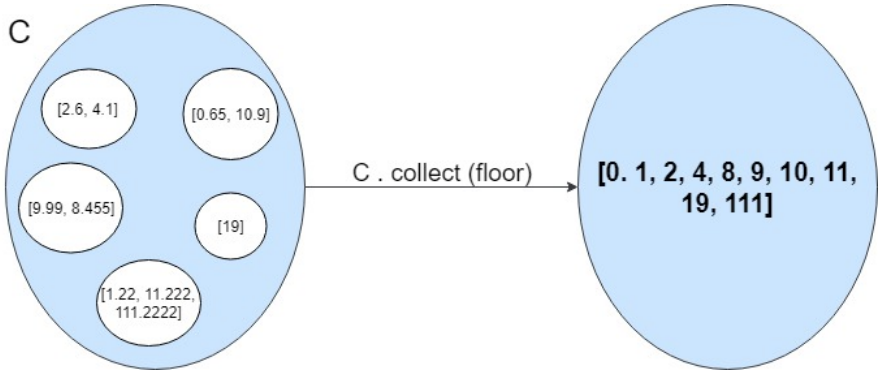


Figure 4.28: Ejemplo de uso de *collect*

La propia definición de la función ya deja entrever el problema de implementación que habrá. Para realizar un *collect* sobre una colección es necesario realizar previamente un *flatten* de la misma, operación cuyo comportamiento, como se mencionó anteriormente, no fue posible reproducir en programación funcional.

Pese al anterior inconveniente, el proyecto contiene una implementación de *collect* reducida. La misma obvia el uso de *flatten* sobre la colección de entrada y simplemente aplica la expresión OCL *expr* a cada uno de los elementos de la colección.

Esta versión limitada permite mantener la implementación de *collect* para colecciones que no contienen colecciones como elementos. Un ejemplo de esto se puede apreciar en la Figura 4.29

A futuro, si se agregara una implementación, aunque fuera reducida, de la operación *flatten*, podría extenderse la implementación de *collect* para esos casos que sí puedan ser resueltos con *flatten*, ya que el resto de la lógica de la función es la misma si toma como entrada una colección que no tiene colecciones dentro.

- ***collectNested(expr)* - operaciones de iteración**
  La función *collectNested*, aplicada a una colección de elementos de tipo `a`, retorna una colección conteniendo el resultado de aplicar la expresión OCL *expr* a cada uno de los elementos, como se muestra en el ejemplo de la Figura 4.30.

  El comportamiento de *collectNested* es el mismo que tiene la implementación reducida de *collect* descrita anteriormente. Para no redundar en dos funciones con el

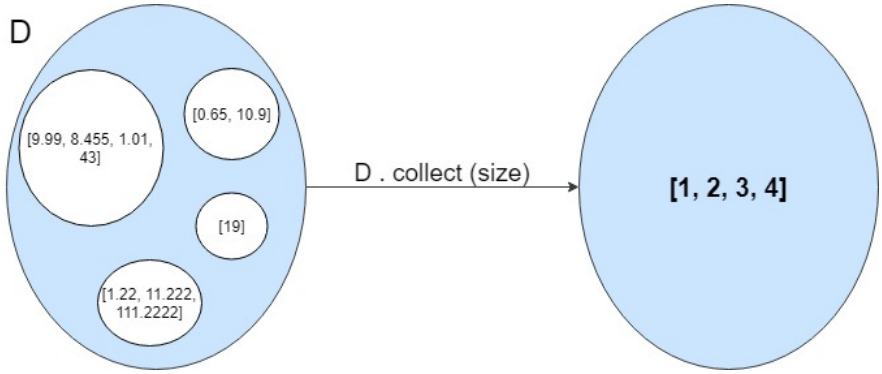


Figure 4.29: Ejemplo en el que se podría aplicar la implementación reducida de *collect*

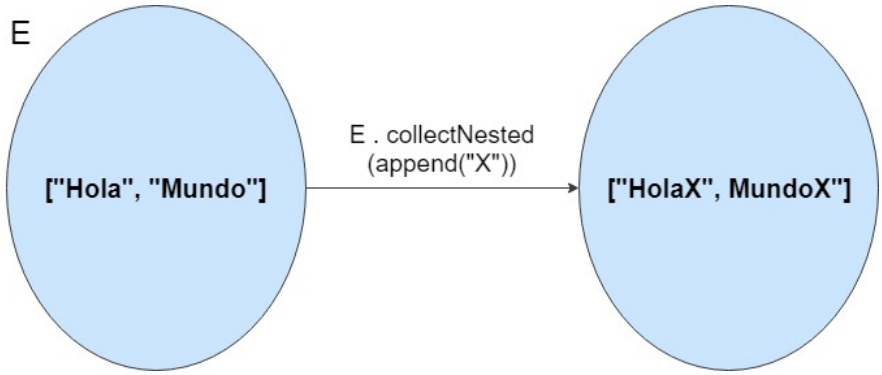


Figure 4.30: Ejemplo de uso de *collectNested*

mismo comportamiento en una prueba de concepto, se decidió omitir agregar la implementación de esta función, aclarando que se podría lograr el mismo comportamiento utilizando la implementación actual de *collect*.

Para agregar la funcionalidad al alcance del proyecto bastaría con copiar la implementación actual de *collect*, que es la que se muestra en la Figura 4.31.

```
collect :: (Val a -> OCL m (Val b)) -> Val (Collection a)
                    -> OCL m (Val (Collection b))
collect f list = liftM (listToBag) (mapM f (collectionToList list))
```

Figure 4.31: Implementación reducida de *collect* en Haskell que tiene el mismo comportamiento que *collectNested*

4.4.2 Pendientes

Como no era el objetivo del proyecto abarcar la totalidad del lenguaje OCL, distintos elementos del mismo quedaron pendientes de estudio para ser agregados al cubrimiento de OCL que tiene la herramienta.

La lista a continuación representa aquellos elementos del lenguaje OCL que se entiende pueden ser añadidos a la implementación, o al menos estudiarse la posibilidad de hacerlo.

Cabe aclarar que existen aspectos de OCL que no tiene sentido considerar y por ello no son parte de la lista. Para la validación a realizar, que es puramente estática, contar con funcionalidades que apuntan a un comportamiento dinámico de la evaluación (como ser *oclMessage*, *init*, *pre* y *post*) no tendría sentido y por ello no entran en consideración.

- ***derive***
  La palabra reservada *derive* en OCL permite definir una regla para establecer el valor de un atributo o una relación. A nivel de validación de modelo, esto implicaría asegurar que el valor dado en el modelo para un atributo o relación es el mismo que el indicado por la regla de *derive*. Un ejemplo del uso de *derive* se puede ver en la Figura 4.32.

  Si bien tiene bastante similitud definir una derivación a definir un invariante, la principal diferencia radica en que *derive* indica el valor que debe tener un atributo o relación, mientras que con invariantes solo fijamos reglas que los mismos deben cumplir. Pese a esta diferencia, podemos considerar a las derivaciones muy similares a los invariantes y entonces armar estructuras que puedan procesarlas y ejecutarlas como parte del proceso de validación en futuras versiones.

```
derive: if underAge
        then parents.income->sum() * 1%
        else job.salary
        endif
```

Figure 4.32: Ejemplo de uso de *derive*

- ***let***
  La operación *let* en OCL permite definir una variable para que represente una subexpresión dentro de otra expresión OCL (por ejemplo, dentro de una invariante, como se puede ver en la Figura 4.33). Haskell también cuenta con una implementación análoga para *let* haciendo que definir la interpretación de este elemento en lenguaje funcional resulte trivial.

  Se entiende que no sería necesario implementar nuevas funciones que representen este comportamiento en *oclLibrary*, siendo que para agregarlo se utilizan operaciones ya

definidas de Haskell y bastaría con que se generara el código a partir de la información del modelo.

```
context Person inv:
  let income : Integer = self.job.salary->sum() in
  if isUnemployed then
    income < 100
  else
    income >= 100
  endif
```

Figure 4.33: Ejemplo de uso de *let*

- ***def***
  OCL permite definir variables globales a todo el metamodelo utilizando la palabra reservada *def*. A diferencia de *let* que permite definir una variable para ser utilizada dentro de una expresión OCL, *def* permite establecer variables globales a ser utilizadas en cualquier expresión dentro de un contexto, como se puede apreciar en el ejemplo de la Figura 4.34.

  Inicialmente se podría considerar que, al definir variables de uso global utilizando *def*, bastaría con utilizar la definición de variables de Haskell para emular su comportamiento, pero al estar su definición restringida a un contexto dado, la misma no puede ser definida de forma global para todo el archivo.

  La solución a considerar estaría ligada a la implementación de *let*, entendiendo que perfectamente podría considerarse una definición de una variable dentro de un contexto como que existiera una entrada de *let* para cada expresión en dicho contexto.

```
context Person
def: income : Integer = self.job.salary->sum()
def: nickname : String = 'Little Red Rooster'
def: hasTitle(t : String) : Boolean = self.job->exists(title = t)
```

Figure 4.34: Ejemplo de uso de *def*

- ***enumerations***
  Las enumeraciones en OCL son la representación de los *datatypes* en UML. Un ejemplo de esto se puede ver en la Figura 4.35. En definitiva, a nivel del modelo UML se está definiendo un *datatype* y existe una sintaxis en OCL para operar con dicha estructura.

A nivel de la programación funcional podría entenderse que este comportamiento es similar al que tiene un tipo de datos algebraico. Podría transformarse cada uno de los *datatypes* definidos a nivel UML en un tipo de datos algebraico de Haskell, donde los constructores serían los distintos valores que puede tomar el *datatype* definido. La Figura 4.36 muestra cómo sería la definición del tipo de datos algebraico para el ejemplo en la Figura 4.35.

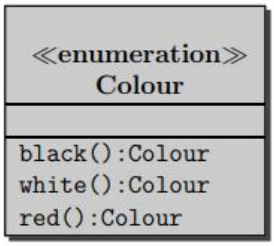


Figure 4.35: Ejemplo de uso de enumeraciones

```
data Colour = Black | White | Red
```

Figure 4.36: Tipo de datos algebraico en Haskell equivalente al ejemplo en la Figura 4.35

- ***tuplas***
  La Figura 4.37 muestra cómo se puede definir una tupla en OCL. Durante el estudio de posibles implementaciones de este comportamiento en programación funcional surge la idea de considerar las tuplas como tipo de datos algebraico, donde se tiene un único constructor y cada uno de los elementos del tipo de datos se pone como parámetro de ese constructor.

  Esta forma de implementación es muy similar a como se definen usando *algebraic datatypes* las distintas clases de un modelo. Incluso se puede establecer otra similitud entre ambos, ya que para las clases se define una función para obtener el valor de cada atributo/relación en la misma, algo que también es necesario tener para las tuplas y poder así acceder a cada uno de sus valores.

  La imagen de la Figura 4.38 muestra un intento sencillo de definir funcionalmente la tupla detallada en la Figura 4.37.

- ***sortedBy(expr)* - operaciones de iteración**
  La función ***sortedBy***, aplicada a una colección de elementos de tipo `a`, retorna una

```
Tuples{name: String, age: Integer}
```

Figure 4.37: Ejemplo de uso de tuplas

```
data Tuple_nameString_ageInteger = Tupla (Val String) (Val Int)

_name :: Val Tuple_nameString_ageInteger -> OCL m (Val String)
_name (Val (Tupla a _)) = ocl a

_age :: Val Tuple_nameString_ageInteger -> OCL m (Val Int)
_age (Val (Tupla _ b)) = ocl b
```

Figure 4.38: Implementación funcional de tuplas en Haskell

colección conteniendo todos los elementos de dicha colección ordenados según el resultado de evaluar la expresión OCL *expr* en cada uno de ellos.

La necesidad de establecer un orden para los elementos del tipo de retorno de la función *expr* llevó a no agregar la implementación de ***sortedBy*** a *ocliLibrary*, entendiendo que era necesario estudiar mejor el caso para no incurrir en errores por tipos que no puedan ser ordenados.

- ***notEmpty***
  Los anteriores ejemplos presentaban funcionalidades de OCL que eran parte del alcance al inicio del proyecto, pero finalmente no formaron parte del alcance de OCL que cubre la herramienta desarrollada. Además de estas estructuras, existen algunos métodos de OCL para ser utilizados en invariantes que se considera relevante sean agregados por ser muy común su uso.

  Uno de estos casos es utilizar ***notEmpty*** sobre colecciones, permitiendo validar si una colección es vacía o no. Afortunadamente para la herramienta de validación desarrollada existe una alternativa a esta consulta que sí se encuentra implementada, que es consultar si el tamaño (*size*) de una colección es mayor a cero para saber que una colección no es vacía.

  Teniendo esto último en cuenta resulta bastante trivial lograr agregar una definición de *notEmpty* simplemente invocando *size* y realizando el chequeo antes descrito.

- **'+' en *Strings*** (Concatenación de *Strings*)
  Otro método a agregar, que también tiene un método equivalente, es la suma de *Strings*. Para este caso existe la implementación del método *concat* para *Strings* que representa la misma funcionalidad, concatenación de dos *Strings*.

Correspondería entonces agregar su implementación a la biblioteca de funcionalidades OCL soportadas y, que no podrá ser la misma operación `|+|` ya implementada, porque la misma está definida para números (enteros y reales), lo que agrega la complejidad de tener nuevamente más de una función para implementar una única funcionalidad que OCL si puede sobrecargar.

- **Chequeos de Cardinalidad**
  Una de las validaciones que no se realiza normalmente a nivel OCL, pero que sí se realiza en el propio modelo, es chequear que las relaciones en un modelo mantengan las cardinalidades establecidas en el metamodelo.

  Siendo que la cardinalidad de relaciones, salvo aquellas que tienen cardinalidad 1, se pierde en la implementación funcional del metamodelo, agregar validaciones que permitan asegurar estas cardinalidades se estén cumpliendo sería un agregado de valor para la validación realizada por la herramienta, y no representa gran complejidad que sean agregadas.

  Estos controles pueden expresarse fácilmente como invariantes y ser validados de forma muy similar. La Figura 4.39 muestra un ejemplo de chequeo de cumplimiento de cardinalidad implementado en Haskell, donde se valida que un *Team* tenga al menos dos integrantes.

```
restriction1 = context _Team [res1]
res1 self = (ocl self |.| members |->| size) |>=| oclVal 2
```

Figure 4.39: Implementación funcional de un chequeo de cardinalidad

4.5 Interpretación vs. Eclipse OCL

Uno de los puntos de interés en el inicio de este trabajo era poder determinar qué tanto cubrimiento de los aspectos necesarios para la validación de modelos se podía abarcar con la interpretación definida, cuya orientación está puesta en los aspectos funcionales de la evaluación de modelos, en comparación con otras herramientas similares, pero que hacen enfoque en los aspectos de orientación a objetos que tiene la evaluación.

La Tabla 4.2 muestra una comparación del alcance del proyecto, y aquello que pudo ser implementado, en comparación con las funcionalidades de Eclipse OCL, una herramienta de validación con énfasis en los aspectos orientados a objetos.

	Haskell OCL	Eclipse OCL
Constructores OCL		
context Especifica el contexto para expresiones OCL	✓	✓
inv Establece una condición que siempre debe ser cumplida por todas las instancias de un contexto	✓	✓
init Establece el valor inicial de un atributo o una función de asociación	✓	✓
derive Establece el valor de un atributo derivado o una función de asociación	✗	✓
def Introduce un nuevo atributo o una operación de consulta	✓	✓
Expresiones OCL		
self Denota la instancia contextual	✓	✓
Navigation Navegación a través de atributos, fines de asociación, clases de asociación y asociaciones calificadas	−	✓
if-then-else expression Expresión condicional con una condición y dos expresiones	✓	✓
let-in expression Expresión con variables locales	✓	✓
Biblioteca estándar de OCL		
Boolean Type Valores: `True` y `False`	✓	✓
Boolean operations Operaciones: `or, and, xor, not, =, <>, implies`	✓	✓
Integer/Real Type Valores: `-10, 0, 10, ..., -1.5, 3.14, ...`	✓	✓
Integer/Real Operations Operaciones: `=, <>, <, >, +, -, *, /, mod, div, max, round, ...`	✓	✓
String Type Valores: `"value"`	✓	✓
OCLAny Supertipo de todos los tipos de UML y OCL	✗	✓
OCLAny Operations Operaciones definidas para cualquier tipo: `=, <>, oclIsNew, oclAsType, T::allInstances, ...`	−	−
OCLVoid Tipo con una única instancia (`undefined`) que conforma con los otros tipos	−	✓
Tuple Type Una tupla consiste en partes nombradas, cada una de las cuales puede tener un tipo distinto	✓	✓
Collection Types Cuatro tipos de colecciones: `Set`, `OrederedSet`, `Bag` y `Sequence`	✓	✓
Collection operations Operaciones: `any, append, asBag, count, collect, excludes, exists, first, ...`	−	✓

Table 4.2: HaskellOCL vs EclipseOCL

5 Transformación de Modelos

Una vez resuelta la interpretación funcional de modelos y OCL, uno de los objetivos iniciales trazados para este proyecto era poder integrar la interpretación funcional generada con una herramienta de modelado por medio de la definición de transformaciones de Modelo a Texto (M2T).

La entrada del proceso de transformación será un modelo, definido en Ecore, y, mediante el uso de transformaciones definidas en Acceleo, se creará un archivo con el código funcional correspondiente, que en definitiva es un archivo de texto plano.

Del mismo modo que al analizar las interpretaciones funcionales definidas se dividió el trabajo en metamodelo, modelo e invariantes, en la etapa de transformación también es conveniente verlo desde esa perspectiva y analizar por separado los 3 casos.

5.1 Transformación de un Metamodelo

Como se explica en la sección 2.3.3, iterando dentro de las clases de un metamodelo, se puede ir obteniendo la información necesaria para cada estructura que hay que definir.

- **Transformación de clases**
  Para transformar cada clase del metamodelo se define la transformación M2T de la Figura 5.1, en la que de cada clase se obtiene mediante Acceleo información de la misma (nombre, atributos, relaciones, herencia, etc.) y se crea la equivalente representación funcional.

```
[template public generateData(aClass : EClass)]
[comment @main /]
[file ('ACCELEO.hs', true)]
data    [aClass.name/] = [aClass.name/]
          [for (attr : EAttribute | aClass.getAttributes())]([toHaskellType(attr.eType.name)/]) [/for]
          [for (ref : EReference | aClass.getReferences())]
            [if (ref.lowerBound = 1 and ref.upperBound = 1)](Int)[else]['([Int])'/][/if]
          [/for]
          [let allClasses : Sequence(EClass) = aClass.ancestors().eAllContents("EClass")]
            [if (aClass.hasChildren(allClasses))]
              [if (aClass.abstract)]
                ([aClass.name/]Child)
              [else]
                (Maybe [aClass.name/]Child)
              [/if]
            [/if]
          [/let]
        deriving (Eq, Show)

[/file]
[/template]
```

Figure 5.1: Transformación de clases

La transformación de la Figura 5.1 se puede dividir en tres etapas. Una primera donde obtiene la lista de atributos de una clase del metamodelo e itera dentro de la lista obtenida en el llamado a `getAttributes()`, obteniendo el tipo del atributo en su correspondiente versión de Haskell. Como segunda etapa se efectúa un proceso similar, pero esta vez recorriendo las referencias que tiene la clase a través de la función `getReferences()`. Las referencias son representadas utilizando listas de enteros, o, en caso de que se trate de una referencia con cardinalidad 1, directamente como un entero, por lo que la transformación chequea la cardinalidad de la referencia para agregar el tipo correspondiente. Por último se realiza el manejo de herencia. Por medio del *wrapper* `hasChildren()` se chequea si la clase tiene hijos en el metamodelo. En caso de tenerlo, se agrega la estructura de herencia y el uso de *Maybe* en caso de que la clase a procesar sea abstracta.

El resultado de aplicar esta transformación sobre la clase *Meeting* resulta en su implementación en Haskell tal como se vio en la Figura 3.4.

- **Transformación de Atributos y Referencias**
  Para cada clase del metamodelo se definen las transformaciones M2T de las figuras 5.2

y 5.3. Para cada clase del metamodelo se toman todos los atributos y referencias y, de éstos, se obtiene la información (tipo, cardinalidad, etc.) para generar la representación en Haskell de las funciones para operar con cada uno de los atributos y referencias de dicha clase.

```
[template public generateProperties1(aClass : EClass)]
[comment @main /]
[file ('ACCELEO.hs', true)]
[for (attr : EAttribute | aClass.getAttributes())]
    [let attrName : String = getAttributeName(aClass, attr.name, aClass.ancestors().eAllContents("EClass"))]
    [let index : Integer = i]
        [attrName/] :: Cast Model [aClass.name/]_ a => Val a -> OCL Model (Val [attr.eAttributeType.name.toHaskellType()/])
        [attrName/] a = upCast _[aClass.name/] a >>= pureOCL( \ ([aClass.name/]
            [for (attr : EAttribute | aClass.getAttributes())]
                [if (index = i)]x [else]_ [/if]
            [/for]
            [for (ref : EReference | aClass.getReferences())]_ [/for]
            [let allClasses : Sequence(EClass) = aClass.ancestors().eAllContents("EClass")]
                [if (aClass.hasChildren(allClasses))]_ [/if]
            [/let]
        , _) -> return (Val x))
    [/let]
    [/let]
[/for]
[/file]
[/template]
```

Figure 5.2: Transformación de atributos

```
[template public generateProperties2(aClass : EClass)]
[comment @main /]
[file ('ACCELEO.hs', true)]
[for (ref : EReference | aClass.getReferences())]
    [let attrName : String = getAttributeName(aClass, ref.name, aClass.ancestors().eAllContents("EClass"))]
    [let index : Integer = i]
        [if (ref.lowerBound = 1 and ref.upperBound = 1)]
            [attrName/] :: Cast Model [aClass.name/]_ a => Val a -> OCL Model (Val [ref.eReferenceType.name/]_)
            [attrName/] a = upCast _[aClass.name/] a >>= pureOCL (\ ([aClass.name/]
                [for (attr : EAttribute | aClass.getAttributes())]_ [/for]
                [for (ref : EReference | aClass.getReferences())]
                    [if (index = i)]x [else]_ [/if]
                [/for]
                [let allClasses : Sequence(EClass) = aClass.ancestors().eAllContents("EClass")]
                    [if (aClass.hasChildren(allClasses))]_ [/if]
                [/let]
            , _) -> lookupM _[ref.eReferenceType.name/] x)

        [else]
            [attrName/] :: Cast Model [aClass.name/]_ a => Val a -> OCL Model (Val ['(Collection '/][ref.eReferenceType.name/]_))
            [attrName/] a = liftM (toCollection) (upCast _[aClass.name/] a >>= pureOCL( \ ([aClass.name/]
                [for (attr : EAttribute | aClass.getAttributes())]_ [/for]
                [for (ref : EReference | aClass.getReferences())]
                    [if (index = i)]x [else]_ [/if]
                [/for]
            [let allClasses : Sequence(EClass) = aClass.ancestors().eAllContents("EClass")]
                [if (aClass.hasChildren(allClasses))]_ [/if]
            [/let]
        , _) -> mapM (lookupM _[ref.eReferenceType.name/]) x >>= oclVal))
        [/if]
    [/let]
    [/let]
[/for]
[/file]
[/template]
```

Figure 5.3: Transformación de referencias

A modo de ejemplo, volviendo a la Figura 3.5, las transformaciones se encargan de completar con la información de cada uno de los atributos y cada una de las referencias de la clase, sustituyendo el nombre de la clase (`aClass.name`), el tipo de

retorno (`attr.eAtributeType` para atributos, `ref.eReferenceType` para referencias) y el nombre que tendrá la función para operar con cada atributo/referencia (`attr`).

El manejo del nombre no es tan directo como pasarle el nombre del atributo/referencia que se está procesando, ya que, al no poder definirse dos funciones con el mismo nombre en Haskell, pero si poder contar con dos atributos con el mismo nombre en un mismo metamodelo (distintas clases), es necesario asignar un nombre único a cada una de estas funciones. Es por ello que el valor de `attrName` se calcula al principio de la transformación mediante el Java wrapper `getAttributeName()`. En caso de que el nombre del atributo esté repetido, se agrega un apostrófe al final, lo que asegura que en el código funcional no van a existir funciones de igual nombre.

- **Transformación de Herencias**
  Para cada clase del metamodelo se define la transformación M2T de la Figura 5.4. Luego, para cada uno de sus hijos se genera una instancia de la clase *Cast*, que representa a nivel funcional la relación de herencia entre ambas clases.

```
[template public generateCast(aClass : EClass)]
[comment @main /]
[file ('ACCELEO.hs', true)]
[for (child : EClass | aClass.getChildren(aClass.ancestors().eAllContents("EClass")))]
    instance Cast Model [aClass.name/]_ [child.name/]_ => Cast Model
        [if (hasFather(aClass))][getFather(aClass).name/][else]ModelElement[/if]_ [child.name/]_ where
        downCast t (Val e@([if (hasFather(aClass))][let father : EClass = getFather(aClass)][father.name/]
            [for (attr : EAttribute | father.eAllContents("EAttribute"))]_ [/for]
            [for (ref : EReference | father.getReferences())]_ [/for]
            [if (not father.abstract)](Just [/if]
            ([aClass.name/]Ch x)
            [if (not father.abstract)])[/if]
            ,
        [/let]
        [else]
            ModelElement _ ([aClass.name/]Ch x),
        [/if]
        [if (hasFather(aClass) and hasFather(getFather(aClass)))]_[getFather(getFather(aClass)).name/]
        [elseif (hasFather(aClass))]_ModelElement
        [else]
            Top
        [/if]
        )) = downCast t (Val (x,e))
        downCast _ _ = return Inv
        directInstance  _ _ = return Inv
        upCast   t (Val (_,e)) = upCast t (Val e)
[/for]
[/file]
[/template]
```

Figure 5.4: Transformación de herencias

A su vez, si la clase a procesar no tiene un padre definido en el metamodelo, por cómo se definió previamente la representación funcional, corresponde definir una herencia a esa clase desde *ModelElement* y, de este modo, cumplir con que *ModelElement* sea padre de toda clase del metamodelo, como se puede ver en la transformación de la Figura 5.5.

```
[template public generateCast(aClass : EClass)]
[comment @main /]
[file ('ACCELEO.hs', true)]
instance Cast Model [if (hasFather(aClass))][getFather(aClass).name/][else]ModelElement[/if]_ [aClass.name/]_ where
    downCast _ (Val e@([if (hasFather(aClass))][let father : EClass = getFather(aClass)][father.name/]
        [for (attr : EAttribute | father.eAllContents("EAttribute"))]_ [/for]
        [for (ref : EReference | father.getReferences())]_ [/for]
        [if (not father.abstract)](Just [/if]
        ([aClass.name/]Ch x)
        [if (not father.abstract)])[/if]
        ,
    [/let]
    [else]
        ModelElement _ ([aClass.name/]Ch x),
    [/if]
    [if (hasFather(aClass) and hasFather(getFather(aClass)))]_[getFather(getFather(aClass)).name/]
    [elseif (hasFather(aClass))]_ModelElement
    [else]
        Top
    [/if]
        )) = return $ Val (x,e)
    downCast _ _ = return Inv
    directInstance  _ _ = return Inv
    upCast   _ (Val (_,e)) = return (Val e)
[/file]
[/template]
```

Figure 5.5: Transformación de herencias desde *ModelElement*

La transformación de la Herencia sigue las reglas que se mencionaron en la sección anterior y cuya representación se muestra en las figuras 4.4, 4.5, 4.6 y 4.7.

- **Transformación de Invariantes**
  Para cada clase del metamodelo se define la transformación M2T de la Figura 5.6. En ella, se obtienen las invariantes que tiene asociadas y, para cada una de esas invariantes, se obtiene la información necesaria (nombre y definición OCL) y se genera la representación funcional de la misma bajo el contexto de la clase del metamodelo que se está procesando. Un ejemplo del resulado de esta transformación se encuentra en la Figura 3.10.

```
[template public generateInvariants(aClass : EClass)]
[comment @main /]
[file ('ACCELEO.hs', true)]
[if (not aClass.getEAnnotation('http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot').oclIsUndefined())]
    [let invariants : OrderedSet(EStringToStringMapEntry)
        = aClass.getEAnnotation('http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot').details]
        [for (invariant : EStringToStringMapEntry | invariants)]
            [let key : String = addHaskellFunction(invariant.key)]
                invariant[addInvariant()/] = context _[aClass.name/] ['['/][key/][']'/]
                [key/] self = [aClass.parseInvariant(invariant.value, invariant.key)/]
            [/let]
        [/for]
    [/let]
[/if]
[/file]
[/template]
```

Figure 5.6: Transformación de invariantes

Para la transformación de la definición OCL de una invariante a su equivalente en la

representación Haskell se define el *Java wrapper* `parseInvariant` cuya implementación se profundiza más adelante en la sección 5.3.1.

5.2 Transformación de un Modelo

Del mismo modo que para la transformación del metamodelo Acceleo permitía recorrer las clases del mismo y aplicar transformaciones para cada una de ellas, para el caso del modelo existe la posibilidad de iterar dentro de sus elementos.

En este caso, iterando dentro de los objetos del modelo, se puede ir obteniendo la información necesaria para definir los objetos que lo comprenden y así crear cada uno de los *ModelElement* pertenecientes a la lista en Haskell que representa el modelo a evaluar.

Transformación de elementos
Para cada objeto del modelo se define la transformación M2T de la Figura 5.7. En ella, para cada objeto del modelo, se obtiene su información, que no es simplemente saber los valores que toman sus atributos y relaciones, sino también obtener toda la jerarquía de herencia del objeto, de modo de poder definir la cadena de descendencia desde *ModelElement* hasta llegar al *datatype* que representa el tipo del objeto que se está procesando (tal como fuera definido en 4.2).

```
[template public generate(aClass : EObject) post (trim())]
[comment @main /]
[file ('ACCELEO.hs', true)]
    [if (notFirst(aClass))], [else]  [/if]
    (ModelElement [addObject(aClass) /]
    [for (parent : EClass | aClass.eClass().getParents())]
        [if (i <> 1 and (not parent.abstract))](Just [/if]
        ([parent.name/]Ch ([parent.name/]
        [for (attrP : EAttribute | parent.getAttributes())][aClass.eGet(attrP.name)/][/for]
        [for (reference : EReference | parent.getReferences())]
            ['['/][for (elem : EObject | aClass.references(reference))][getID(elem)/][', '/][/for][']'/]
        [/for]
    [/for]
    [if (aClass.eClass().hasFather() and (not aClass.eClass().getFather().abstract))](Just [/if]
    ([aClass.eClass().name/]Ch ([aClass.eClass().name/]
        [for (attrP : EAttribute | aClass.eClass().getAttributes())][aClass.eGet(attrP.name)/][/for]
    [for (reference : EReference | aClass.eClass().getReferences())]
            ['['/][for (elem : EObject | aClass.references(reference))][getID(elem)/][', '/][/for][']'/]
    [/for]
    ))
[/file]
[/template]
```

Figure 5.7: Transformación de elementos del modelo

Si bien Acceleo permite fácilmente obtener toda la información necesaria del *EObject* a procesar, se presenta un desafío más complejo, **las referencias**.

Por la forma en que fue definida la representación funcional de los elementos, a cada uno de ellos se le asocia un ID único. Esto permite que las referencias se puedan representar simplemente como una lista de enteros, que en definitiva es la lista de los IDs de los objetos del modelo. El problema radica en cómo, al procesar una relación, se puede conocer el ID de un elemento referenciado.

Para solucionar este inconveniente es necesario mantener en memoria un mapeo entre cada *EObject* del modelo y su correspondiente ID, algo que a nivel de programación orientada a objetos resulta bastante sencillo, pero que no es posible implementar directamente sobre Acceleo.

Afortunadamente Acceleo brinda la posibilidad de definir ***Java service wrappers***. Estos *Java Wrappers* permiten invocar desde una transformación Acceleo métodos implementados en Java, lo que significa que podemos solucionar el mapeo *EObject* - ID utilizando la ayuda de la programación orientada a objetos.

Por lo tanto, se define un mapa donde, para cada *EObject* se tiene asociado su ID, y, cada vez que se procesa un *EObject* en la transformación, ya sea porque es el objeto a procesar en la iteración o porque se lo está referenciando mediante una relación, se invoca al *Java Wrapper* para obtener su ID correspondiente.

La Figura 5.8 muestra el proceso que sigue cada objeto del modelo para determinar su ID.

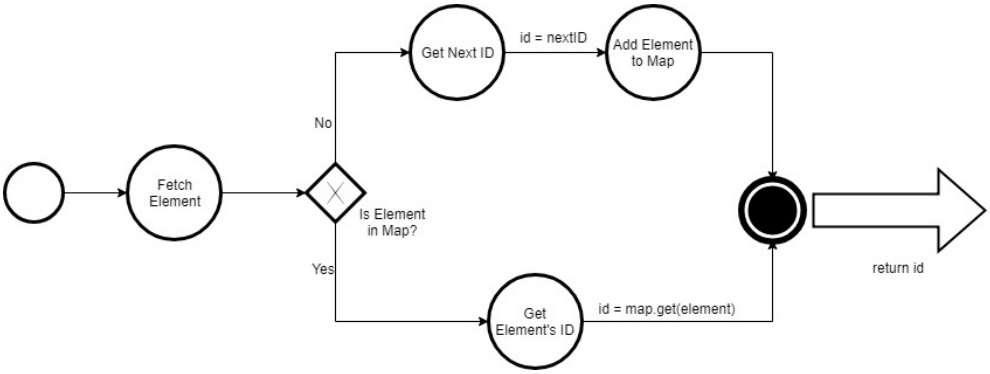


Figure 5.8: Diagrama de procesamiento de IDs usando *Java Service Wrapper*

5.3 Transformación de sintaxis OCL

La transformación de invariantes, especialmente el pasaje de la definición de una invariante en OCL a la definición de esa misma invariante en la interpretación funcional, fue sin dudas el mayor desafío en la etapa de transformación.

Si bien en la sección 5.1 se introduce la definición de una transformación M2T para invariantes utilizando Acceleo, se menciona también que la única información de relevancia que se obtiene es el nombre de la invariante y su definición OCL. Esto se debe a que la lógica de la transformación desde la definición OCL al código Haskell se realiza por fuera de Acceleo, utilizando un *Java Wrapper* (*parseInvariant*).

La decisión de no utilizar directamente funcionalidades de Acceleo para la transformación del texto de OCL, y hacerlo con la ayuda de Java, radica principalmente en la posibilidad de aprovechar el hecho de que Java ya cuenta con un intérprete de invariantes OCL que facilita su manipulación.

Este intérprete OCL definido en Java permite tomar una invariante y procesarlo, de modo de generar un árbol de sintaxis abstracta como el que se puede apreciar en la Figura 5.9, que no solo cuenta con toda la información de la invariante, sino que también la procesa respetando el orden de precedencia en que se evaluaría la expresión.

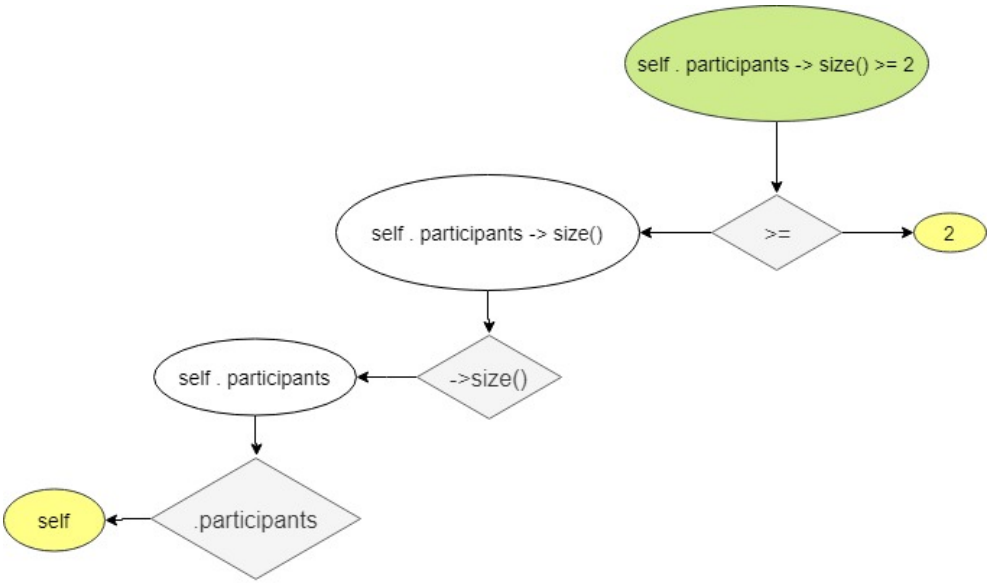


Figure 5.9: Ejemplo árbol de sintaxis abstracta parser OCL

Una vez generado el árbol a partir de la interpretación de la invariante, es posible recorrerlo y reconstruirla de forma textual. Si a esto además se le agrega que, para cada hoja del árbol, es posible transformar esa expresión OCL en su equivalente en la definición de OCL creada en lenguaje funcional, resulta bastante directa la transformación a realizar para procesar un invariante y obtener su definición en Haskell.

5.3.1 ParseInvariant

ParseInvariant es un *Java Wrapper* que permite tomar como entrada la definición de una invariante OCL, la procesa utilizando las funcionalidades de OCL disponibles en Java, y retorna la invariante transformada a lenguaje funcional.

Partiendo de que las funcionalidades de OCL para Java permiten fácilmente generar un árbol de sintaxis abstracta a partir de la definición de una invariante, como se vio en la Figura 5.9, se puede recorrer el árbol para ir transformando individualmente cada una de las hojas a su definición funcional, como se puede ver en la Figura 5.10

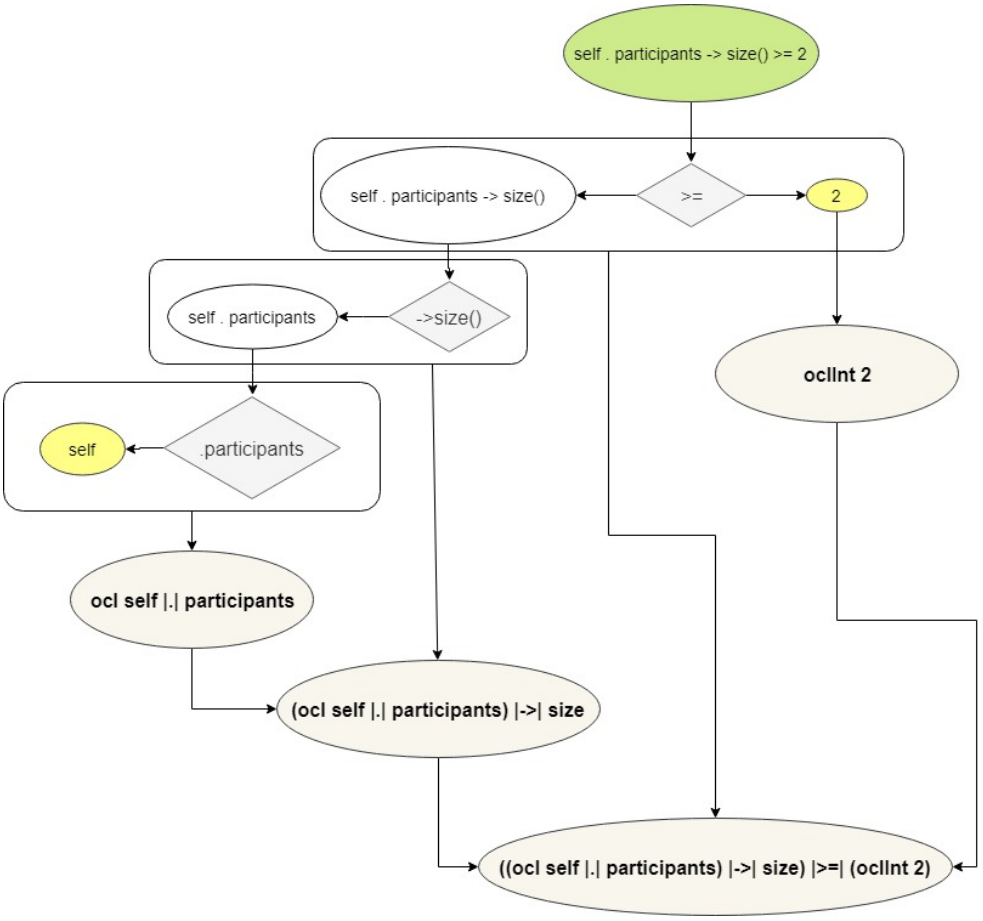


Figure 5.10: Ejemplo de transformación de una invariante

La forma de procesar una *OCLExpression* consiste en poder determinar qué tipo de expresión es para entender cómo corresponde interpretarla. Una *OCLExpression* puede ser de 10 tipos:

- ***OperationCallExp***
  Llamado a una operación. Ejemplos:

```
5 >= x
```

```
"Hola".append("Mundo").
```

- ***PropertyCallExp***
  Llamado a un atributo. Ejemplos:

```
self.participants
```

```
self.end
```

- ***IteratorExp***
  Operaciones de iteración sobre colecciones. Ejemplos:

```
->collect(name)
```

```
->forAll( p :  Person | p.age <= 65)))
```

- ***IterateExp***
  Operaciones de iteración sobre colecciones. Ejemplos:

```
->select(p | p.age > 50)
```

```
->exists(p :  Person | p.forename = 'Jack'))
```

- ***CollectionLiteralExp***
  Definición de una colección. Ejemplos:

```
Bag 1, 1, 2
```

```
Sequence 1, 1, 2, 3, 5
```

- ***StringLiteralExp***
  Literal de tipo *String*. Ejemplos:

```
"Hola"
```

```
"Mundo"
```

- ***IntegerLiteralExp***
  Literal de tipo entero. Ejemplos:

```
101

2
```

- ***RealLiteralExp***
  Literal de tipo real. Ejemplos:

```
10.1

0.95
```

- ***BooleanLiteralExp***
  Literal de tipo booleano. Ejemplos:

```
true

false
```

- ***VariableExp***
  Referencia a una variable. Ejemplos:

```
select(p | p.age > 50)

forAll( p :  Person | p.age <= 65)))
```

En la Figura 5.11 se puede apreciar el tipo de *OCLExpression* considerado por *parseInvariant* para cada nodo del árbol en base a la que se determina qué regla de transformación se debe aplicar, para así generar el código funcional correspondiente, acorde a los pasos vistos en la Figura 5.10.

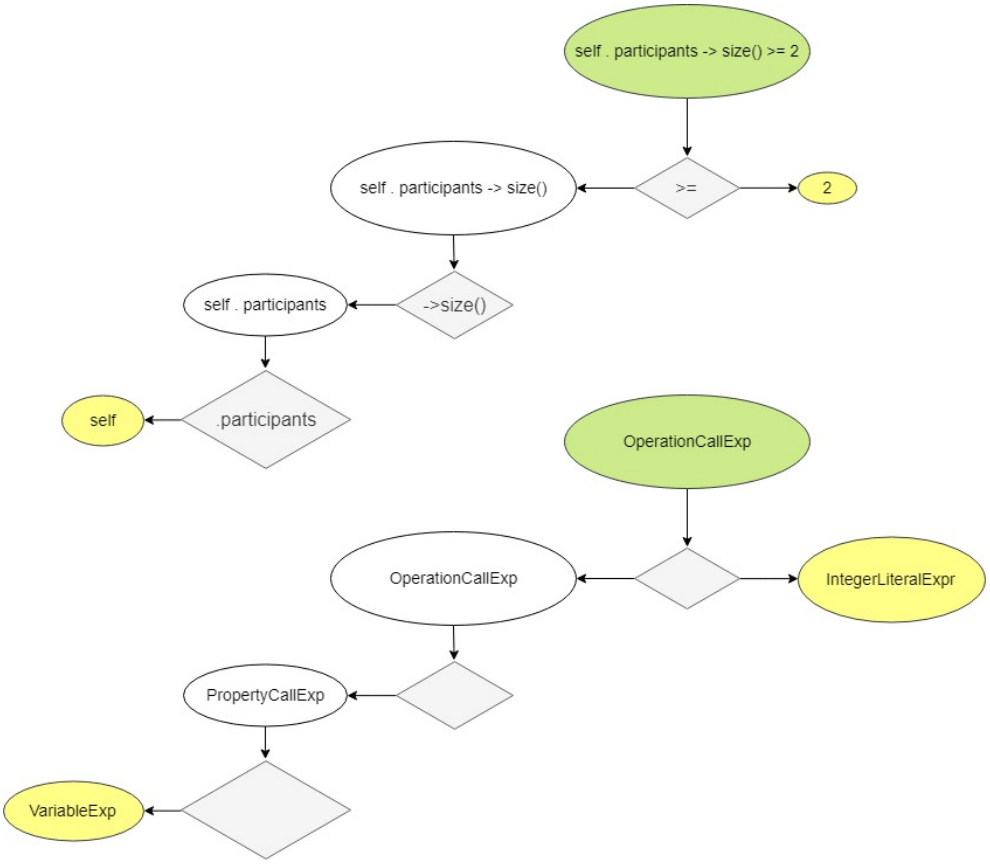


Figure 5.11: Proceso de transformación del árbol

5.4 Validación no estructural de Modelos

El objetivo principal de este trabajo es poder contar con una herramienta que valide si un modelo cumple con las restricciones OCL definidas para el mismo utilizando programación funcional. En las secciones anteriores se vio (1) cómo se puede definir un modelo utilizando lenguaje funcional, (2) cómo generar la interpretación funcional del modelo mediante transformaciones M2T y (3) la definición de *oclLibrary*, biblioteca Haskell que contiene la lógica de las operaciones OCL y de la cual dependerá el procesamiento a la hora de la validación. El resultado de las transformaciones es un archivo Haskell cuya ejecución va a validar el cumplimiento de las invariantes definidas.

```
main = do       putStrLn "MODEL EVALUATION:"
                let res = oclCheckInvariants example
                putStrLn "Checking Invariants:"
                putStrLn (show res)
```

Figure 5.12: Función *main* ejecutable

La función *main* de la Figura 5.12 procede a invocar la función *oclCheckInvariants*, pasándole como parámetro el modelo que se desea validar (*example*). Esta función procederá a evaluar, una a una, todas las invariantes definidas, retornando sus resultados.

Cabe recordar que las funciones que corresponden a la definición de invariantes vistas en 4.3 son evaluaciones que retornan, en principio, un valor booleano. Este valor booleano servirá para determinar si la restricción se cumple (*True*) o si no lo hace (*False*).

```
oclCheckInvariants m =  let     chk1 = runReaderT invariant1 m
                                chk2 = runReaderT invariant2 m
                                chk3 = runReaderT invariant3 m
                                chk4 = runReaderT invariant4 m
                                chk5 = runReaderT invariant5 m
                                chk6 = runReaderT invariant6 m
                                chk7 = runReaderT invariant7 m
                        in [chk1, chk2, chk3, chk4, chk5, chk6, chk7]
```

Figure 5.13: Función de ejecución de invariantes para evaluación

La función *oclCheckInvariants*, cuya implementación se puede apreciar en la Figura 5.13, es también generada utilizando las transformaciones, tomando como entrada la cantidad de invariantes definidas para generar una línea de ejecución para cada una de ellas.

6 Plugin HaskellOCL

Todo lo visto anteriormente acerca de la interpretación funcional de modelos en el capítulo 4 y la posterior definición de transformaciones para obtener dicha interpretación (Capítulo 5) sirve como base para la creación de una herramienta que permita, utilizando todo lo antes definido, realizar las validaciones de invariantes OCL sobre un modelo. Con ese objetivo en mente se crea **HaskellOCL**.

HaskellOCL es un *plugin* de Eclipse que permite al usuario evaluar un modelo definido en Ecore simplemente haciendo clic derecho sobre el mismo y utilizando la opción brindada por el *plugin* en el menú desplegable para realizar la evaluación.

En este capítulo se entrará en el detalle de la implementación del *plugin*, haciendo hincapié en el proceso de validación que realiza. Además, se darán instrucciones a desarrolladores para poder levantar y ejecutar el *plugin*, y a usuarios para que tengan una guía básica de cómo utilizarlo.

6.1 Implementación

Para el desarrollo del *plugin* se utilizó la herramienta PDE (*Plug-in Development Environment*) que provee Eclipse [37]. El diseño del mismo se basó en el actual *plugin* de OCL (Eclipse OCL), el cual ejecuta las validaciones de las invariantes a partir de un modelo y muestra los resultados en cascada en una ventana llamada *Validity View*.

En la Figura 6.1 se representa el proceso de validación que se realiza al ejecutar HaskellOCL. A partir de la definición del metamodelo, modelo y las invariantes OCL, se genera la estructura funcional en código Haskell, luego se evalúan las invariantes y finalmente se muestran los resultados.

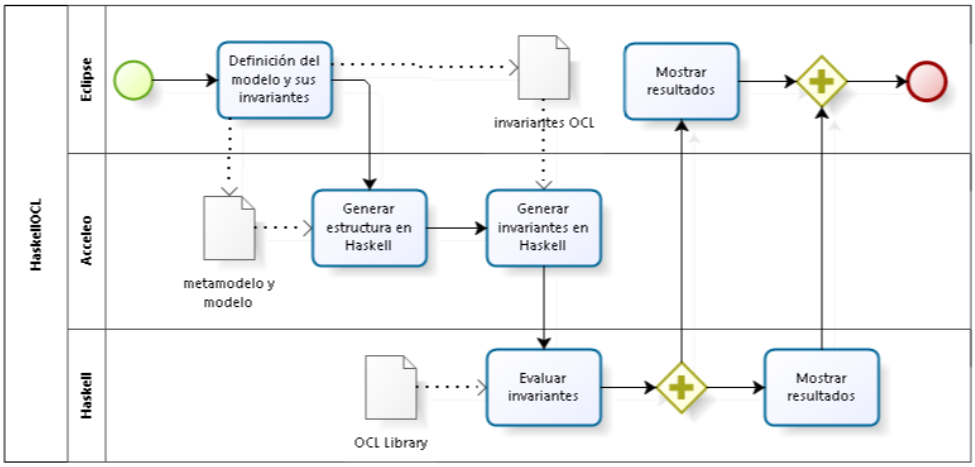


Figure 6.1: Pasos para validar un modelo usando HaskellOCL

El proceso de validación se puede dividir en cuatro etapas consecutivas, que serán desarrolladas en las siguientes subsecciones:

- Obtención de datos del modelo
- Transformación M2T a Haskell
- Validación funcional de invariantes
- Procesamiento y presentación de resultados

6.1.1 Obtención de datos del Modelo

El proceso de validación comienza con la obtención de lo datos del modelo seleccionado por el usuario, el modelo a validar. HaskellOCL, con ayuda de funcionalidades brindadas por las bibliotecas de Ecore, logra obtener información general del modelo que será de utilidad para la ejecución de los siguientes pasos.

En particular en este punto, interesa poder conocer, a partir del modelo, cuál es su metamodelo asociado. Tomar conocimiento de las rutas absolutas donde se encuentran los archivos con la definición del modelo y metamodelo a procesar (ver Figura 6.2) es necesario para poder ejecutar las transformaciones del siguiente paso.

En este paso el *plugin* no estudia las estructuras dentro de modelo y metamodelo, eso será tarea de las transformaciones M2T de Acceleo cuando sean ejecutadas.

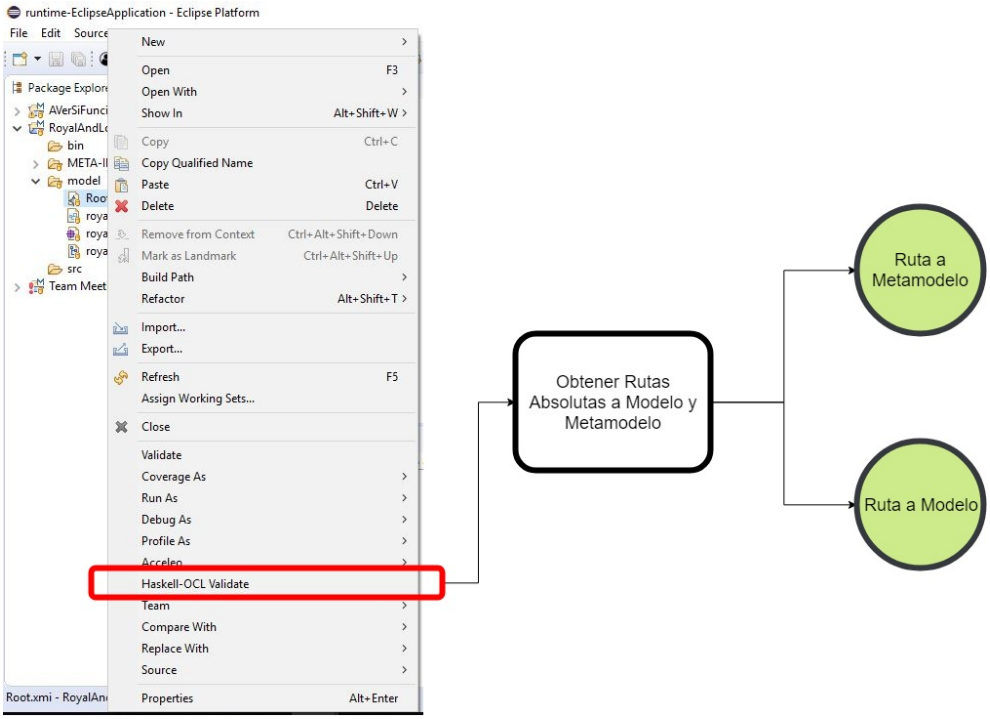


Figure 6.2: Obtención de rutas a modelo y metamodeo al inicio de la ejecución

Se podría considerar esta primera etapa como una preparación, bastante trivial, para el grueso de la ejecución lógica que se realizará luego.

6.1.2 Transformación M2T a Haskell

En el capítulo 5 se detalla el uso de Acceleo como herramienta para crear y ejecutar transformaciones *Model to Text* para generar el código funcional necesario, a partir de la información de modelo y metamodelo en Ecore.

Es aquí donde comienza a entrar en juego todo el desarrollo teórico presentado en este trabajo. Utilizando Acceleo se accede tanto al modelo como al metamodelo, para aplicarle las transformaciones definidas y así generar el archivo Haskell con la interpretación funcional (*Acceleo.hs*).

Este paso de la interpretación encapsula de forma práctica todo aquello que en la teoría se había definido en capítulos anteriores. Es aquí donde está la mayor complejidad en el funcionamiento del *plugin*.

6.1.3 Validación funcional de invariantes

Una vez aplicadas las transformaciones, el *plugin* genera un archivo Haskell ejecutable. Como se mencionó en la sección 5.4. este archivo contiene una función *main* incorporada que, al ejecutarla, realizará la validación de cada una de las invariantes del modelo.

Para realizar la validación funcional alcanza con ejecutar el método *main* definido en el archivo Haskell generado. Este proceso no presenta aún un resultado visible para el usuario, sino que generará un registro interno con los resultados de la validación de cada uno de las invariantes.

6.1.4 Procesamiento y presentación de resultados

Para hacer una presentación visualmente más amigable para el usuario, el resultado a mostrar de la evaluación de las invariantes va a ser una versión procesada de los resultados obtenidos en el paso anterior.

En la misma, en lugar de tener listados en consola los resultados en orden y sin ninguna referencia a la invariante al que corresponden (más allá de poder asociarlos por el orden en que son presentados), se toma cada una de las invariantes procesadas y se imprime en consola la información de la misma, junto con su resultado (`OK` para las invariantes que se cumplen y `FAIL` en caso de que la invariante no se cumpliera para algún elemento del modelo) como se muestra en la Figura 6.3.

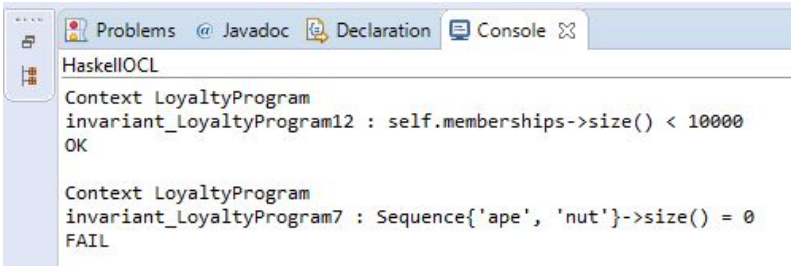


Figure 6.3: Resultado de la evaluación presentado en Consola

6.2 Mejoras pendientes

Siendo que el desarrollo de la herramienta que permitiera ejecutar la validación funcional de modelos no era el objetivo principal del proyecto, el desarrollo del *plugin* se redujo a dejar un prototipo mínimo que permitiera ejecutar el proceso. Es por esto que la versión entregada tiene aún muchas funcionalidades por desarrollar/mejorar para dejar un entregable completo y funcional que pueda ser publicado y utilizado por usuarios. También lograr la exportación del mismo, aunque fuera del prototipo, quedó pendiente efectuarla.

En esta sección se detallarán aquellos puntos donde se entiende se debe seguir trabajando para poder lograr una herramienta funcional que pueda ser distribuida y utilizada por usuarios.

- **Publicación del *plugin***
  Como se mencionó anteriormente, la herramienta desarrollada está más focalizada en generar la lógica de ejecución e implementar el proceso de transformación M2T, que en la creación de un *plugin* completamente funcional que pudiera ser distribuido e instalado en distintas versiones de Eclipse.

  Partiendo de la implementación actual se sugiere explorar los pasos a seguir para poder publicarlo. Esto requerirá estudiar cómo hacer correctamente la importación en un JAR del proyecto, y todas sus dependencias, para luego utilizar ese archivo JAR para realizar la instalación en otros entornos Eclipse.

  Al finalizar el desarrollo se estudió brevemente la posibilidad de realizar la publicación, siempre y cuando la misma no tuviera mucha complejidad. En esas horas dedicadas se concluyó que el proceso de exportación y manejo de dependencias sería el principal desafío, especialmente tomando en cuenta que la implementación de HaskellOCL está dividida en tres proyectos separados, cuyas interacciones también hay que manejar correctamente. Ante este panorama se decidió priorizar el cierre del proyecto y no continuar profundizando en la publicación del *plugin*.

- **Alternativas de ejecución**
  Originalmente, al definir lo que se esperaba el usuario pudiera hacer al utilizar HaskellOCL se definieron dos alternativas de ejecución. Una, que se encuentra implementada, es realizar el proceso de validación completo (tomar un modelo, transformarlo a código Haskell y correr la validación de las invariantes).

  La segunda alternativa era la ejecución parcial del proceso. Poder, a partir de un modelo, ejecutar únicamente la transformación M2T que genera la interpretación funcional, dejando al usuario la posibilidad de manipular directamente el modelo sobre el código Haskell y utilizarlo para realizar las validaciones. Esta alternativa podría ser de utilidad para usuarios avanzados con conocimientos de programación funcional que pudieran fácilmente entender e interactuar con la representación del modelo.

  Si bien no existe hoy en día la implementación correspondiente de esta segunda fun-

cionalidad, el proceso de validación completo deja disponible el archivo con la representación funcional del modelo. Esto sirve para tener en cuenta que agregar esta funcionalidad no va a representar un gran desafío, ya que se cuenta con todas las herramientas disponibles para hacerlo con lo que ya está implementado.

Además de las dos formas de ejecución discutidas al definir el *plugin*, el hecho de contar con ejecuciones parciales dentro de la implementación abre la puerta a considerar también hacer una ejecución parcial del final del proceso. Esto abarcaría tomar un archivo Haskell como entrada, procesarlo asumiendo que es la interpretación funcional de un modelo, y ejecutar la validación de las invariantes.

Esta funcionalidad permitiría complementar la primera ejecución parcial, que genera solamente la interpretación funcional del modelo, haciendo que no sea necesario para el usuario tener que pasar por el proceso de compilación y ejecución de Haskell para realizar la validación, ya que este proceso se haría internamente como parte de esta funcionalidad.

- **Integración con *plugin* Eclipse OCL**
  Otra funcionalidad que quedó pendiente en el desarrollo de la herramienta es la integración con el *plugin* Eclipse OCL. En particular, se definió que HaskellOCL debería poder invocar la validación que realiza Eclipse OCL y reportar sus resultados de la misma forma que lo hace cuando la validación se hace mediante el proceso de interpretación funcional.

  Al igual que la publicación del *plugin*, también se hizo un análisis de cómo implementar la integración con Eclipse OCL. En el proceso se descubrió que existe poca o nula documentación al respecto en cuanto al funcionamiento interno del código y, si bien se consiguió llegar a dar con lo que se entendió era la forma de invocar el proceso de validación, nunca fue posible reproducirlo correctamente.

- **Reporte de resultados**
  Un punto que quedó en el debe fue el reporte visual de los resultados de la evaluación. Actualmente, como se puede apreciar en la Figura B.2, el reporte de resultados es bastante simple y desplegado en la consola de Eclipse.

  Se entiende que este tipo de reportes se puede mejorar. Ya se cuenta con la información de las invariantes y el resultado de su validación, por lo que solo sería necesario trabajar en la forma de presentarlos.

  Una de las ideas iniciales, que va de la mano con la integración con Eclipse OCL, era utilizar la vista ***Validity View*** [38] para presentar los resultados, que es la vista utilizada por el Eclipse OCL. Sin embargo, al estudiar el uso de *Validity View*, se descubrió que la API de dicha vista quedó obsoleta, por lo que se desestimó tomarla como referencia.

  Se entiende que lo mejor en este caso sería desarrollar una nueva vista de Eclipse que

permita presentar los resultados y la información necesaria que retorna la validación. Especialmente considerando que, si se extiende la interpretación de modelos para realizar más validaciones, no solo el cumplimiento de las invariantes OCL, hay que tener la posibilidad de mostrar esos nuevos resultados.

- **Extender implementación para soportar agregados en interpretación funcional.**
  En la sección 4.4 se detallan conceptos de modelado que la interpretación funcional aún no abarca, pero que podría llegar a hacerlo en un futuro. Funcionalidades poco comunes de OCL, estructuras *let*, *def*, chequear las cardinalidades de las relaciones, etc. son todos agregados que podrían sumarse al *plugin* si se estudian e implementan.

  Llegado el caso, deberá poder extenderse el *plugin*, tanto las transformaciones como la validación, para soportar estos agregados a la interpretación.

- **Ejecución en sistemas operativos Unix**
  Si bien en líneas generales la implementación del *plugin* no utiliza recursos restringidos a un sistema operativo particular, HaskellOCL fue diseñado para funcionar en un ambiente Windows. Esto se puede ver principalmente cuando se deben considerar rutas a archivos, donde los *paths* utilizados siguen el formato Windows y no se considera la opción de una ruta Unix.

  La resolución en la implementación de cómo compilar y ejecutar los archivos Haskell con las representaciones se terminó resolviendo utilizando la clase **Runtime** [39]. El uso de esta clase invoca la línea de comandos, permitiendo, por código Java, ejecutar instrucciones. Esto permitió realizar la compilación y ejecución de los archivos directamente invocando al compilador *ghc* y leyendo su resultado de la salida de línea de comandos.

  El manejo de rutas para soportar ambos tipos de sistemas operativos es bastante sencillo de resolver, pero no es tan fácil con la compilación de código Haskell.

  Esta invocación en línea de comandos es esencialmente diferente en sistemas Unix, por lo que habría que resolver la compilación y ejecución en ese caso y contar con ambas alternativas dentro del *plugin*.

7 Caso de Estudio: Royal & Loyal

Para mostrar el plugin HaskellOCL en acción se utilizará el modelo *Royal and Loyal*, propuesto en el libro *The Object Constraint Language* [40]. Tomando dicho modelo como entrada, se mostrará en detalle el proceso de ejecución completo de la herramienta.

7.1 Modelo Royal & Loyal

7.1.1 Metamodelo

En base a la definición del metamodelo de *Royal & Loyal* se genera una representación del mismo en Ecore que satisface las condiciones para poder ser procesado por HaskellOCL.

La clase central del modelo es *LoyaltyProgram*. Un sistema que administra un único *loyalty program* contendrá únicamente una instancia de esta clase. Una compañía que ofrece membrecías (*memberships*) en un *loyalty program* es llamada *ProgramPartner*. Más de una compañía pueden ser parte del mismo programa. En ese caso, clientes que entran a un *loyalty program* podran hacer uso de los servicios de cualquiera de las compañías participantes.

Cada cliente de un *program partner* puede ingresar al mismo completando un formulario y obteniendo así una tarjeta de membresía. los objetos de la clase *Customer* representan a las personas que ingresaron al programa. La tarjeta de membresiía, representada por la clase *CustomerCard*, es emitida a una única persona. La mayoría de los *loyalty programas* permiten a los clientes acumular puntos. Cada *program partner* decide, de forma individual, cuando y cuantos puntos son obtenidos por una cierta compra. Los puntos acumulados pueden ser utilizados para canjear servicios específicos de uno de los *program partners*. Para llevar un conteo de los puntos obtenidos por un cliente, cada membresía es asociada con una *LoyaltyAccount*.

El modelo resultante se puede apreciar en la Figura 7.1.

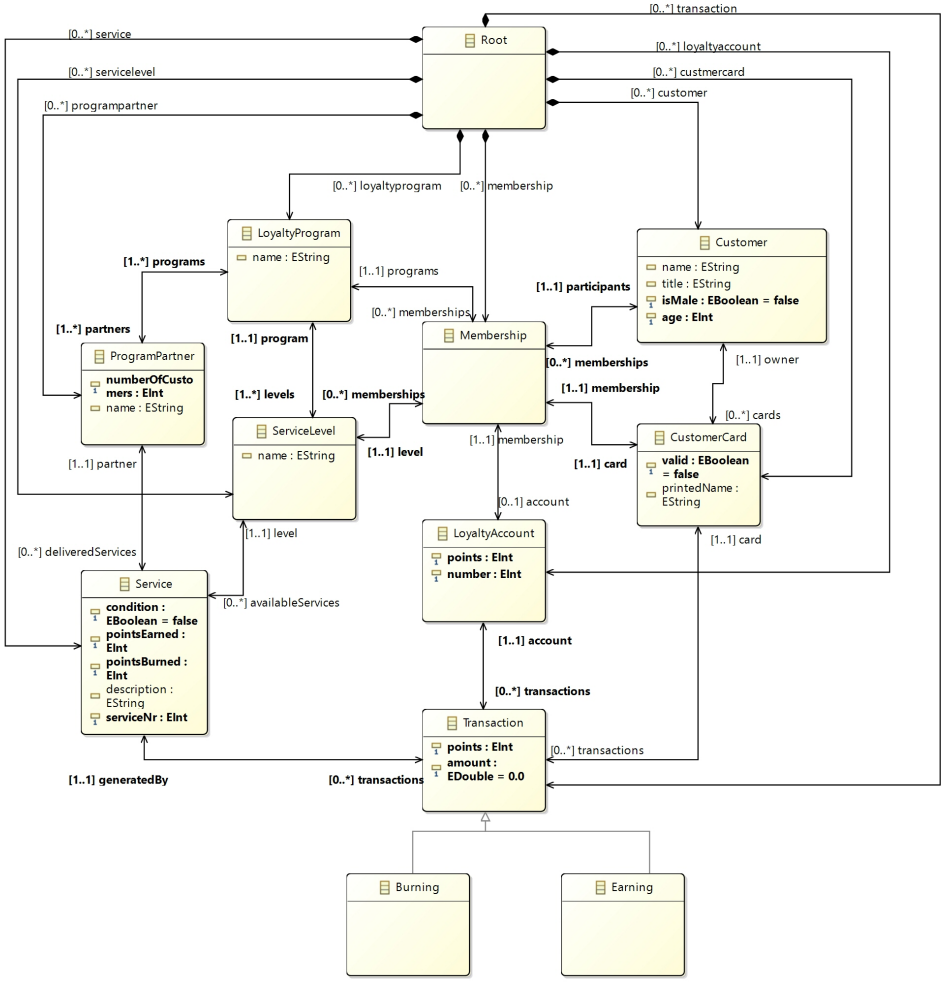


Figure 7.1: Metamodelo ejemplo *Royal & Loyal* implementado en Ecore

7.1.2 Invariantes OCL

A continuación se listan algunas de las invariantes que se validan en el ejemplo propuesto, las mismas son algunas de las propuestas en el ejemplo de [40]. En este conjunto de invariantes se encuentran ejemplos que hacen referencias a todas las clases del metamodelo y que muestran la representación de funciones que abarcan los distintos tipos; desde *Strings*, *Integers* y

Collections.

1. **Para todo *LoyaltyProgram*, el nombre de todos sus *ServiceLevels* es *"Silver"***
```
context LoyaltyProgram
inv:  self.levels->first.name = 'Silver'
```
2. **El título asociado a todos los clientes es *"Mr"***
```
context Customer
inv:  self.title = 'Mr'
```
3. **Para toda *CustomerCard*, el dueño es mayor de edad (más de 18 años)**
```
context CustomerCard
inv:  sel.owner.age >= 18
```
4. **Para toda *CustomerCard*, su dueño tiene al menos una membrecía**
```
context CustomerCard
inv:  self.owner.membershis->size() > 0
```
5. **Toda transacción es de clase *Earning***
```
context Transaction
inv:  self.oclIsKindOf(Earning) = true
```
6. **No todas las transacciones son de tipo *Burning***
```
context Transaction
inv:  self.oclIsTypeOf(Burning) = false
```
7. **El texto *"Anneke"*, pasado a mayúsculas, queda *"ANNEKE"***
```
context Service
inv:  'Anneke'.toUpperCase() = 'ANNEKE'
```
8. **Si al texto *"Anneke"* se le concatena *"and Jos"*, queda el texto *"Anneke and Jos"***
```
context Service
inv:  'Anneke'.concat('and Jos') = 'Anneke and Jos'
```

7.1.3 Modelo

Para utilizar como ejemplo en la validación se define un modelo que conforma con la definición del metamodelo de *Royal & Loyal* propuesto en la sección 7.1. En el modelo se define una única instancia de *LoyaltyProgram* de nombre Proyecto de Grado. Asociado a este *LoyaltyProgram* se encuentra un único *ProgramPartner*, InCo, que tiene 25.000 clientes, y ofrece un único servicio, Tutoría. Para este *LoyaltyProgram* se definen cuatro niveles de servicio (*ServiceLevel*), *Regular*, *Silver*, *Gold* y *Platinium*. El servicio de tutoría es parte del nivel de servicio *Gold*.

En el modelo se definen también dos clientes, Leticia Vaz y Gonzalo Sintas, donde ambos cuentan con una tarjeta de cliente y membresía asociada al nivel de servicio *Gold*. El primer

cliente, Leticia Vaz, tiene 29 años, es mujer y su título es *Project Manager*. La *CustomerCard* que tiene asociada es válida y el nombre que aparece en la misma coincide con el nombre del cliente. La membresía del servicio *Gold* tiene asociada una *LoyaltyAccount* de número 19890616, que ya tiene acumulados 800 puntos, producto de haber obtenido 1.000 tras una operación (*Earning*) y haber luego utilizado 200 de ellos para adquirir un servicio (*Burning*). El segundo cliente, Gonzalo Sintas, también tiene 29 años, es hombre y su título es *CTO*. La *CustomerCard* asociada también está vigente y el nombre que aparece en la misma coincide con el del cliente. La *Membership* asociada a la tarjeta cuenta con una *LoyaltyAccount* de número 19890603 que tiene un saldo de 600 puntos, producto de haber efectuado tres operaciones. Obtuvo 1.000 puntos tras una primera compra y luego gastó 400 de esos puntos adquiridos distribuidos en dos etapas, de 250 y 250 respectivamente.

La Figura 7.2 muestra un diagrama para hacer visualmente más entendible el modelo definido. Para este modelo se genera su correspondiente representación en Ecore, como se puede ver en la Figura 7.3

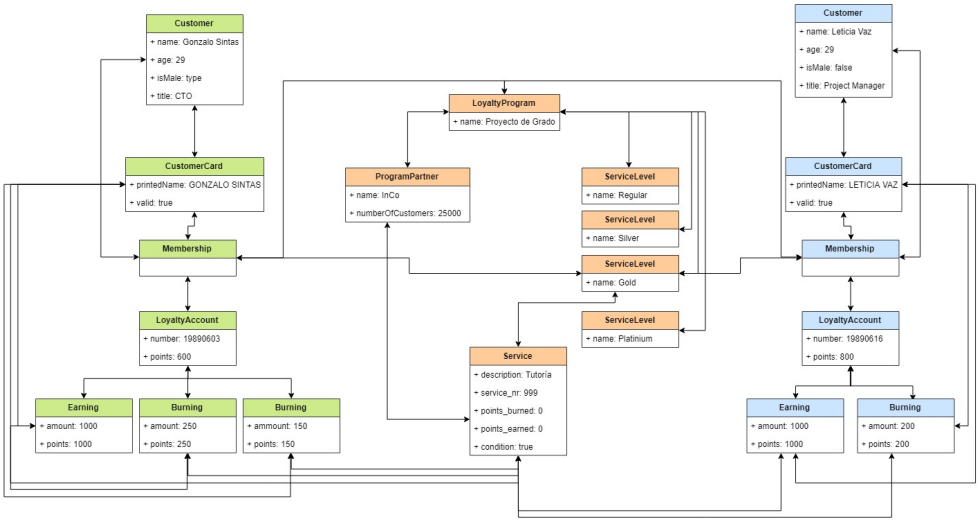


Figure 7.2: Diagrama definición de modelo sobre *Royal & Loyal*

7.2 Validación utilizando HaskellOCL

A partir de la implementación en Ecore de un modelo para *Royal & Loyal*, se procede a utilizar el *plugin* de Eclipse, HaskelOCL, para validar que dicho modelo cumpla con las restricciones no estructurales establecidas en los invariantes asociados al metamodelo.

El proceso de validación realizado al ejecutar HaskellOCL se divide en los cuatro pasos

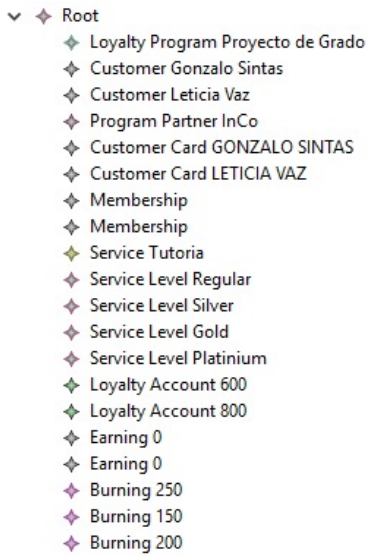


Figure 7.3: Modelo para ejemplo *Royal & Loyal*

descritos en la sección anterior en la Figura 6.1, obtener datos del modelo, transformar a Haskell, validar invariantes y, procesar y mostrar resultados.

En esta sección se verán estos cuatro pasos en la práctica, aplicándolo al modelo y metamodelo de ejemplo.

7.2.1 Paso 1: Obtener datos del modelo

El inicio del proceso de validación de un modelo utilizando HaskellOCL se produce cuando el usuario elige el modelo a validar, realiza clic con el botón secundario sobre dicho modelo y selecciona del menú desplegable la opción **HaskellOCL Validation**, como se muestra en la Figura 7.4.

Una vez seleccionado el modelo e iniciada la validación, el *plugin* toma toda la información necesaria del modelo que se utilizará luego al realizar la transformación. En particular, interesa para poder ejecutar las transformaciones conocer cuáles son el modelo y metamodelo a procesar (la ruta al mismo).

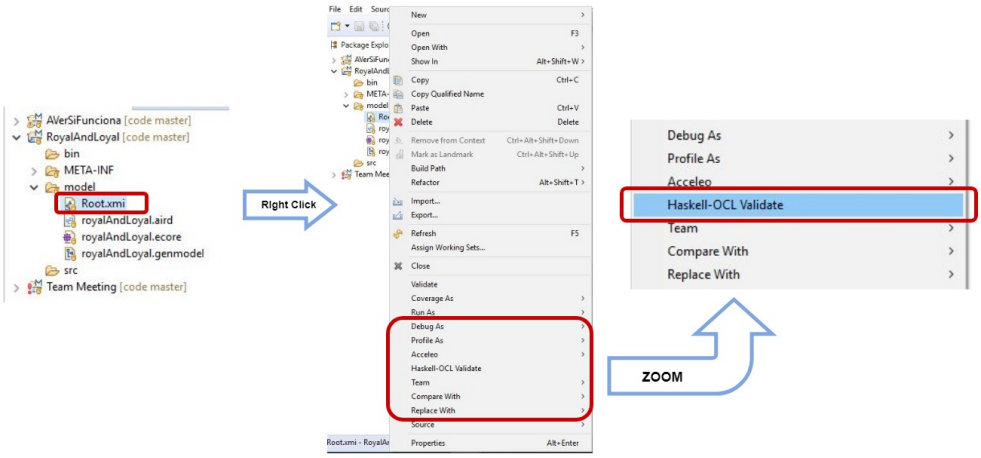


Figure 7.4: Ejecución del *plugin*

7.2.2 Paso 2: Transformar a Haskell

En el capítulo 5 se detalla el uso de Acceleo como herramienta para crear y ejecutar transformaciones M2T para generar el código funcional necesario a partir de la información del modelo y metamodelo en Ecore.

Al ejecutarse la totalidad de las transformaciones se genera el archivo Haskell, ACCELEO.hs, que contiene la interpretación funcional del modelo.

Las transformaciones M2T ejecutadas resuelven, por separado, la interpretación del metamodelo y el propio modelo.

Para el caso del metamodelo, cada transformación tomará como entrada cada una las clases del modelo y generará las estructuras Haskell necesarias para describirlas.

A modo de ejemplo, cuando se toma la clase *Transaction* (Figura 7.5) como entrada.

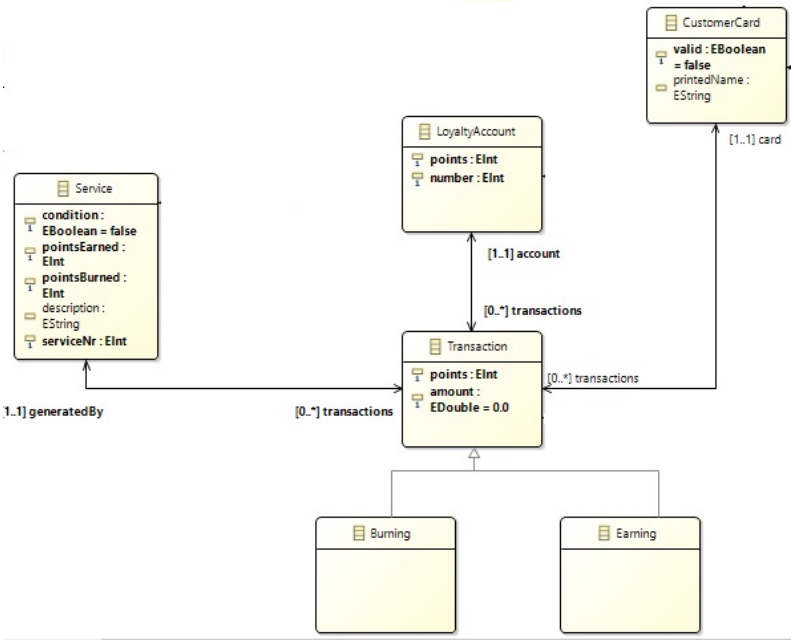


Figure 7.5: Definición clase *Transaction* en Metamodelo

Las transformaciones M2T aplicadas agregaran el código para definir la clase, su herencia, sus atributos, sus relaciones y sus invariantes asociados. Las Figuras 7.6, 7.7, 7.8, 7.9, 7.10 y 7.11 muestran la interpretación Haskell generada a partir de la información de la clase para cada uno de los distintos ítems anteriores.

Clase:

```
data    Transaction = Transaction (Int) (Double) (Int) (Int) (Int) (Maybe TransactionChild)
        deriving (Eq, Show)
```

Figure 7.6: Transformación de la clase *Transaction* a código Haskell

Herencia:

```
data    TransactionChild = EarningCh (Earning) | BurningCh (Burning)
        deriving (Eq, Show)
```

Figure 7.7: Representación de *Burnings* y *Earnings* como *datatypes* para representar la herencia

```
instance Cast Model ModelElement_ Transaction_ where
    downCast _ (Val e@(ModelElement _ (TransactionCh x), Top)) = return $ Val (x,e)
    downCast _ _ = return Inv
    directInstance  _ _ = return Inv
    upCast   _ (Val (_,e)) = return (Val e)

instance Cast Model Transaction_ Earning_ where
    downCast _ (Val e@(Transaction _ _ _ _ _ (Just (EarningCh x)), _ModelElement)) = return $ Val (x,e)
    downCast _ _ = return Inv
    directInstance  _ _ = return Inv
    upCast   _ (Val (_,e)) = return (Val e)

instance Cast Model Transaction_ Earning_ => Cast Model ModelElement_ Earning_ where
    downCast t (Val e@(ModelElement _ (TransactionCh x), Top)) = downCast t (Val (x,e))
    downCast _ _ = return Inv
    directInstance  _ _ = return Inv
    upCast   t (Val (_,e)) = upCast t (Val e)

instance Cast Model Transaction_ Burning_ where
    downCast _ (Val e@(Transaction _ _ _ _ _ (Just (BurningCh x)), _ModelElement)) = return $ Val (x,e)
    downCast _ _ = return Inv
    directInstance  _ _ = return Inv
    upCast   _ (Val (_,e)) = return (Val e)

instance Cast Model Transaction_ Burning_ => Cast Model ModelElement_ Burning_ where
    downCast t (Val e@(ModelElement _ (TransactionCh x), Top)) = downCast t (Val (x,e))
    downCast _ _ = return Inv
    directInstance  _ _ = return Inv
    upCast   t (Val (_,e)) = upCast t (Val e)
```

Figure 7.8: Función para castear desde niveles superiores a niveles inferiores en una herencia

Atributos:

```
points'' :: Cast Model Transaction_ a => Val a -> OCL Model (Val Int)
points'' a = upCast _Transaction a >>= pureOCL( \ (Transaction x _ _ _ _ _ , _) -> return (Val x))

amount' :: Cast Model Transaction_ a => Val a -> OCL Model (Val Double)
amount' a = upCast _Transaction a >>= pureOCL( \ (Transaction _ x _ _ _ _ , _) -> return (Val x))
```

Figure 7.9: Función para obtener el valor de los atributos de *Transaction*

Relaciones:

```
card :: Cast Model Transaction_ a => Val a -> OCL Model (Val CustomerCard_)
card a = upCast _Transaction a >>= pureOCL (\ (Transaction _ _ x _ _ _ , _) -> lookupM _CustomerCard x)

account :: Cast Model Transaction_ a => Val a -> OCL Model (Val LoyaltyAccount_)
account a = upCast _Transaction a >>= pureOCL (\ (Transaction _ _ _ x _ _ , _) -> lookupM _LoyaltyAccount x)

generatedBy' :: Cast Model Transaction_ a => Val a -> OCL Model (Val Service_)
generatedBy' a = upCast _Transaction a >>= pureOCL (\ (Transaction _ _ _ _ x _ , _) -> lookupM _Service x)
```

Figure 7.10: Función para obtener el valor de las relaciones de *Transaction*

Invariantes:

```
invariant1 = context _Transaction [invariant_Transaction1']
invariant_Transaction1' self = ((ocl self |.| (oclIsKindOf (_Transaction)))) |==| (oclVal True)

invariant2 = context _Transaction [invariant_Transaction3']
invariant_Transaction3' self = ((ocl self |.| (oclIsTypeOf (_Burning)))) |==| (oclVal False)

invariant3 = context _Transaction [invariant_Transaction2']
invariant_Transaction2' self = ((ocl self |.| (oclIsTypeOf (_Transaction)))) |==| (oclVal True)

invariant4 = context _Transaction [invariant_Transaction4']
invariant_Transaction4' self = ((ocl self |.| (oclIsKindOf (_Burning)))) |==| (oclVal False)
```

Figure 7.11: Invariantes asociadas a la clase *Transaction*

Al transformar el modelo, en lugar de tomar como entrada cada una de las clases, se toma como entrada para la transformación cada uno de los elementos del modelo. Tal es así que, para cada uno de los elementos del modelo, se va a generar una entrada dentro de la lista que representa el modelo en Haskell, con la información del elemento procesado. Las Figura 7.12 muestra la definición de un elemento de tipo *Earning* dentro del modelo en Ecore, mientras que en la Figura 7.15 se puede apreciar cómo queda la definición de ese mismo elemento en la interpretación funcional.

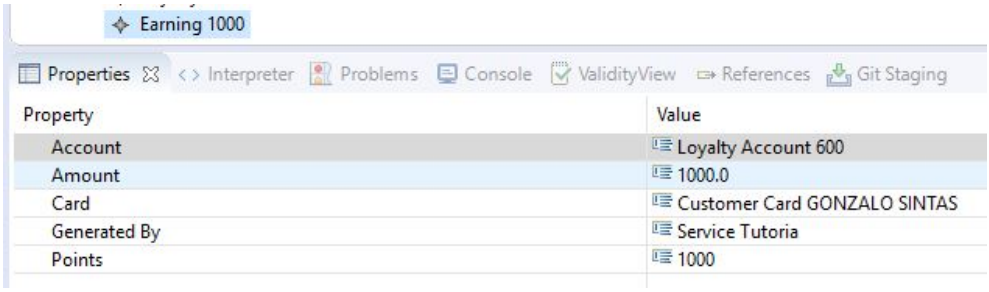


Figure 7.12: Propiedades de un elemento con la vista Ecore

```
(ModelElement 37 (TransactionCh (Transaction 0 1000.0  26 35 30
                                  (Just (EarningCh (Earning  ))))))
```

Figure 7.13: Representación de las propiedades de un elemento en código Haskell

7.2.3 Paso 3: Validar invariantes

Una vez aplicadas las transformaciones, el *plugin* generó un archivo Haskell ejecutable. Como se mencionó en la sección 5.4 este archivo ya viene con una función *main* incorporada que, al ejecutarla, realizará la validación de cada una de las invariantes asociadas al modelo.

Basta ahora con que el *plugin* compile y ejecute este archivo para realizar la validación. Este proceso no presenta un resultado visible para el usuario.

En definitiva, HaskellOCL ejecuta la validación corriendo el método *main* del archivo, generando una salida en Haskell con los resultados de cada una de las invariantes.

```
MODEL EVALUATION:
Checking Invariants:
[Identity (Val False), Identity (Val False), Identity (Val True), Identity (Val True), Identity (Val False),
 Identity (Val False), Identity (Val False), Identity (Val False), Identity (Val False), Identity (Val False),
 Identity (Val False), Identity (Val False), Identity (Val False), Identity (Val False), Identity (Val False),
 Identity (Val False), Identity (Val True), Identity (Val True), Identity (Val True), Identity (Val True),
 Identity (Val False), Identity (Val True), Identity (Val True), Identity (Val True), Identity (Val True),
 Identity (Val True), Identity (Val False), Identity (Val False), Identity (Val True), Identity (Val False),
 Identity (Val True), Identity (Val True), Identity (Val True), Identity (Val True), Identity (Val False),
 Identity (Val True), Identity (Val True), Identity (Val True), Identity (Val False), Identity (Val True),
 Identity (Val True), Identity (Val True), Identity (Val False), Identity (Val True), Identity (Val True),
 Identity (Val True), Identity (Val False), Identity (Val True), Identity (Val True), Identity (Val True),
 Identity (Val True), Identity (Val True), Identity (Val True), Identity (Val True), Identity (Val True),
 Identity (Val True)]
```

Figure 7.14: Resultado de evaluar modelo de ejemplo directamente sobre Haskell

La lista con los resultados retornada en la Figura 7.14 es una correspondencia, uno a uno, con los invariantes evaluados en el orden en que son procesados por el método *main*. La correspondencia de los resultados se lee de la siguiente manera:

```
invariant1   False
invariant2   False
invariant3   True
invariant4   True
invariant5   False
     .       .
     .       .
     .       .
invariant55  True
invariant56  True
```

7.2.4 Paso 4: Procesar y mostrar resultados

Para hacer una presentación visualmente más amigable para el usuario, el resultado a mostrar de la evaluación de las invariantes va a ser una versión procesada de los resultados obtenidos en el paso anterior.

En la misma, en lugar de tener listados en consola los resultados en orden y sin ninguna referencia a la invariante a la que corresponden (más allá de poder asociarlos por el orden en que son presentados), se toma cada una de las invariantes procesadas y se imprime en consola la información del mismo junto con su resultado (**OK** para las invariantes que se cumplen y **FAIL** en caso de que la invariante no se cumpliera para algún elemento del modelo)

```
Context Customer
invariant_Customer6 : (self.name = 'Edward') and self.title = 'Mr.'
FAIL

Context CustomerCard
invariant_CustomerCard4 : self.transactions->select( i_Transaction : Transaction | i_Transaction.points > 100 )->size() > 0
OK

Context CustomerCard
invariant_ofAge : self.owner.age >= 18
OK

Context CustomerCard
invariant_CustomerCard3 : self.owner.memberships->size() > 0
OK
```

Figure 7.15: Resultado de evaluar modelo aplicando HaskellOCL

7.3 Conclusiones

Mediante el uso de **HaskellOCL** se logró ejecutar la validación de invariantes OCL asociadas a un modelo, aplicando un enfoque funcional. Esta sección permite validar el cumplimiento

de los objetivos trazados inicialmente para este trabajo, aplicados a un modelo de ejemplo completo.

El análisis paso a paso de cada una de las etapas del *plugin* permite contar con un ejemplo práctico donde todos los conceptos teóricos vertidos anteriormente son aplicados, ayudando en la comprensión de los mismos y brindando una demostración de su uso.

8 Conclusiones y Trabajo Futuro

El trabajo de investigación realizado tenía cuatro objetivos bien marcados. El principal, determinar la posibilidad de lograr una representación de OCL utilizando programación funcional, buscando así poder realizar la validación de modelos UML mediante el paradigma funcional.

1. **Obtener un conocimiento base sobre MDE**
   Tanto la etapa de análisis e investigación, como la etapa de diseño e implementación de este trabajo hicieron fuerte foco en conceptos y herramientas relacionadas con el mundo de MDE. La etapa de análisis e investigación estuvo marcada por lograr un entendimiento completo de los conceptos de modelo y metamodelo, y especialmente adentrarse en lo que abarca el lenguaje OCL.

   La segunda etapa, diseño e implementación, ya estuvo más cargada de conocimientos prácticos. Fue necesario adentrarse en la creación de modelos utilizando la herramienta Ecore, aunque sin dudas la parte más compleja resultó crear las transformaciones M2T necesarias para la implementación del *plugin*. El uso de Acceleo como herramienta para esta etapa resultó en distintos problemas ante una carencia de documentación y explicaciones del lenguaje utilizado que llevaron a un escenario de "ensayo y error" para lograr encontrar cómo crear algunas de las transformaciones necesarias.

2. **Extender las capacidades del intérprete existente a través de su implementación en Haskell**
   Sobre este punto, se considera que el alcance obtenido permite concluir que la mayor parte del comportamiento de OCL puede ser representado utilizando funciones programadas en Haskell, haciendo viable considerarlo como una alternativa para la validación de invariantes OCL sobre modelos. Sin embargo, el trabajo llevó a encontrar algunos componentes cuyo comportamiento no fue posible reproducir. Quizás, con una investigación más dedicada, pueda lograr implementarse estas funcionalidades, así como el resto de las funcionalidades de OCL no contempladas en el alcance de la investigación,

y así tener un cubrimiento completo del lenguaje, pero, por el momento, la conclusión más exacta a la que se arriba es que el uso de programación funcional para la validación de modelos es una buena alternativa, aunque aún posee algunas restricciones.

3. **Integrar el intérprete a una herramienta de modelado a través de la definición de una transformación de modelo a texto**

   Para cumplir con este objetivo, se desarrolló el *plugin* HaskellOCL el cual es una primera aproximación a la herramienta que en un principio se deseaba obtener. Con mucho por mejorar aún, el *plugin* permite realizar el proceso completo de validación deseado, comenzando por un modelo con sus restricciones OCL, generando a partir de la información del modelo la representación funcional del mismo mediante transformaciones *M2T*, y ejecutando la evaluación del mismo, con posterior reporte de sus resultados.

   Una de las razones por las que se buscaba contar con una herramienta que pudiera realizar el proceso completo era para poder mostrar de forma práctica los resultados de la investigación teórica expuesta en [4] y que fuera profundizada en la primera parte de este trabajo. En ese sentido, el *plugin* de Eclipse desarrollado cumple el cometido, permitiendo mostrar cómo sería la aplicación de la idea fundamental del trabajo, validar modelos utilizando programación funcional.

4. **Evaluar las capacidades del intérprete en relación a otros existentes a partir de casos de estudio**
   Como se vio en la Sección 4.5, a nivel de alcance funcional, el *plugin* desarrollado se acerca mucho a las funcionalidades que tienen otras herramientas similares como Eclipse OCL. Ya en los aspectos de diseño, el prototipo de *plugin* creado aún tiene mucho por mejorar.

Así como en la representación OCL existen algunos puntos donde es necesario continuar con la investigación para poder contar con una implementación completa, el *plugin* también se da algunas licencias en su implementación, no considerando en su implementación algunas estructuras, con el fin de simplificar su desarrollo, manteniendo las bases necesarias para que la demostración práctica tenga sentido.

Por lo tanto, tanto del lado de la investigación como en el área de la implementación, existe espacio para mejorar lo presentado en un futuro. A nivel de investigación, lograr ampliar las estructuras de modelos y OCL que son parte del alcance resultará esencial para poder aplicar esta solución sobre cualquier tipo de modelo y, en caso de determinarse claramente que hay comportamientos que no es posible reproducir, detallar esas limitantes.

En lo que refiere a la implementación, HaskellOCL tiene mucho sobre lo que trabajar. Por un lado, a medida que crece el alcance de funcionalidades de OCL con representación funcional,

las transformaciones *M2T* en Acceleo deberán poder contemplarlas. Inclusive existen hoy en día algunas funcionalidades que están definidas en Haskell dentro de la investigación, pero que no están incluidas en el proyecto por no ser parte de estas transformaciones, como se mencionó en la sección 4.4. Extenderlas para que toda la investigación pueda reflejarse en la herramienta de validación permitirá que las demostraciones prácticas sean lo más completas posibles. Por otro lado, todo lo que rodea el funcionamiento de la herramienta es bastante precario. Desde la generación de los archivos Haskell con la implementación del modelo, cuya ruta es fija, hasta el reporte en consola de los resultados obtenidos, requieren un *upgrade* para hacer la herramienta lo más *user-friendly* posible. En la Sección 6.2 se detallan varios puntos a mejorar.

Referencias

[1] S. Kent, “Model-driven engineering,” in IFM, ser. Lecture Notes in Computer Science, vol. 2335. Springer, 2002, pp. 286–298

[2] OMG: Object Constraint Language. Sepc. V2.4, Object Managment Group (2014)

[3] Eclipse OCL
https://projects.eclipse.org/projects/modeling.mdt.ocl

[4] Daniel Calegari, Marcos Viera. *On the Functional Interpretation of OCL.* 16th International Workshop on OCL and Textual Modeling (OCL@MoDELS): 33-48. CEUR-WS.org, 2016.

[5] HaskellWiki
https://wiki.haskell.org

[6] Gonzalo Sintas, Leticia Vaz, Daniel Calegari and Marcos Viera. *Model-Driven Development of an Interpreter for the Object Constraint Language*, 2018.

[7] Jean Bézivin. On the unification power of models. Software and System Modeling, 4(2):171–188, 2005

[8] Francisco Durán, Javier Troya, Antonio Vallecillo (2013). Desarrollo de software dirigido por modelos. Universidad Oberta de Cataluya.

[9] Bran Selic. *The Pragmatics of Model-Driven Development*, 2008

[10] MDE Glossary - Modeling and model-driven engineering terms
https://modeling-languages.com/glossary-modeling-and-model-driven-engineering-terms/

[11] Parastoo Mohagheghi · Wasif Gilani · Alin Stefanescu · Miguel A. Fernandez · Bjørn Nordmoen · Mathias Fritzsche. Software and System Models, *Where does Model Driven engeneering help?*, pages 619-620, 2013.

[12] Notas del curso Taller de Ingeniería Dirigia por Modelos, Facultad de Inge-

niería, 2018.

[13] Nafiseh Kahani, Mojtaba Bagherzadeh, James Cordy, Juergen Dingel, Daniel Varró. Survey and classification of model transformation tools, 2017

[14] Krzysztof Czarnecki . Simon Helsen *Feature-based survey of model transformation approaches. IBM Systems Journal*, 2006

[15] OMG: Object Constraint Language. Spec. V2.4.1, OMG Unified Modeling Language Superestructure (2010)

[16] Claudia Pons . Roxana Giandini . Gabriela Perez Desarrollo de software dirigido por modelos: Conceptos teóricos y su aplicación práctica. McGraw-Hill, 2010

[17] Dr Birgit Demuth OCL By Example, Department of Computer Science, Institute for Software and Multimedia Technology

[18] Eclipse Modeling Project | The Eclipse Foundation
http://www.eclipse.org/emf

[19] Eclipse Modeling Framework (EMF) - Tutorial
http://www.vogella.com/tutorials/EclipseEMF/article.html

[20] Help - Eclipse Platform | Workbench User Guide > Concepts > Workbench
https://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fconcepts-2.htm

[21] Eclipse Sirius Documentation
http://www.eclipse.org/sirius/doc/

[22] Acceleo | The Eclipse Foundation
http://www.acceleo.org

[23] John Hughes. *Why Functional Programming Matters*

[24] Learn You a Haskell for Great Good!
http://learnyouahaskell.com

[25] Clark, T.: OCL pattern matching. In: Proc. OCL Workshop. Volume 1092 of CEUR Workshop Proceedings. (2013) 33–42

[26] Tisi, M., Douence, R., Wagelaar, D.: Lazy evaluation for OCL. In: Proc. 15th Intl. Workshop on OCL and Textual Modeling. Volume 1512 of CEUR Workshop Proceedings. (2015) 46–61

[27] Garcia, M.: Efficient integrity checking for essential MOF + OCL in software repositories. Journal of Object Technology 7(6) (2008) 101–119

[28] Brucker, A.D., Clark, T., Dania, C., Georg, G., Gogolla, M., Jouault, F., Teniente, E., Wolff, B.: Panel discussion: Proposals for improving OCL. In: Proc. of 14th Intl. Workshop on OCL and Textual Modelling. Volume 1285 of CEUR Workshop Proceedings. (2014) 83–99

[29] Willink, E.: Ocl 2.5 plans. Presentation in the 14th Intl. Workshop on OCL and Textual Modelling, 2014.

[30] Boronat, A., Meseguer, J.: Algebraic semantics of OCL-constrained metamodel specifications. In: TOOLS (47). Volume 33 of LNBIP., Springer (2009) 96–115

[31] Rivera, J.E., Durán, F., Vallecillo, A.: Formal specification and analysis of domain specific models using Maude. Simulation 85(11-12) (2009) 778–792

[32] Shan, L., Zhu, H.: Semantics of metamodels in UML. In: TASE, IEEE Computer Society (2009) 55–62

[33] Burke, D.A., Johannisson, K.: Translating formal software specifications to natural language. In: Proc. 5th Intl. Conf. Logical Aspects of Computational Linguistics. Volume 3492 of LNCS., Springer (2005) 51–66

[34] Brucker, A.D., Tuong, F., Wolff, B.: Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5. Archive of Formal Proofs 2014 (2014)

[35] Calegari, D., Viera, M.: Model-driven engineering based on attribute grammars. In: Proc. 19th Brazilian Symposium Programming Languages. Volume 9325 of LNCS., Springer (2015) 112–127

[36] Huet, G.: The zipper. J. Funct. Program. 7(5) (September 1997) 549–554

[37] PDE - Eclipsepedia
http://wiki.eclipse.org/PDE

[38] Help - Eclipse Platform | OCL Documentation > Users Guide > Validity View (new in Luna)
https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FValidityView.html

[39] Runtime (Java Platform SE 7)
https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html

[40] Jos Warmer and Anneke Kleppe. Analysis of model transformations via Alloy. *The Object Constraint Language, Second Edition, Getting your models ready for MDA*, pages 21–33, 2007.

Anexos

A Manual de Desarrollador

A.1 Manual Desarrollador: Cómo instalar HaskellOCL

Al momento de redactar este informe el *plugin* no ha alcanzado la fase de liberación, quedando en la etapa de ser un proyecto funcional cuya ejecución está aún vinculada a ejecutarlo localmente en modo de *debugging* utilizando PDE. Como trabajo futuro queda pendiente lograr exportar el *plugin*, de forma tal que sea posible importarlo en cualquier PC, y poder así ejecutarlo directamente sin necesidad de contar con el código fuente del mismo.

De momento, para poder ejecutar el *plugin* es necesario tener su código fuente. El mismo se encuentra en el repositorio GIT del proyecto[1] y alcanza con importar los proyectos: *HaskellOCL*, *AcceleoTest* y *OCLParser*. Una vez importados los tres proyectos en el *workspace* de Eclipse, abrir el archivo `MANIFEST` correspondiente al proyecto HaskellOCL y, en la pestaña *Overview*, utilizar la opción *"Launch an Eclipse application"* para iniciar la ejecución.

Esto abre una nueva instancia de Eclipse, la cual cuenta con el *plugin* instalado y permite utilizarlo. Todas las instrucciones detalladas en el manual de usuario harán referencia a esta nueva instancia de Eclipse.

Para la ejecución del *plugin* es necesario contar con un proyecto Ecore con un modelo sobre el que se pueda ejecutar. Este proyecto puede perfectamente ser creado por el usuario. Eclipse, al correr HaskellOCL y generar la nueva instancia para probarlo, va a importar los proyectos de Ecore que hayan en el *workspace* original, permitiendo tener ya cargados los proyectos y no tener que importarlos desde la instancia.

En el repositorio GIT del proyecto se encuentran ya creados algunos proyectos Ecore con modelos prontos para ser evaluados por la herramienta. Tanto *RoyalAndLoyal* como *TeamMeeting* pueden ser descargados e importados para ser utilizados como ejemplos.

[1] https://gitlab.fing.edu.uy/open-coal/haskellOCL

B

Manual de Usuario

B.1 Manual de Usuario: Cómo utilizar HaskellOCL

Para utilizar HaskellOCL lo primero es contar con un proyecto Ecore que tenga definido modelo y metamodelo que se desean evaluar. Por este motivo pasa a ser necesario contar con las funcionalidades de Ecore instaladas.

Para que la validación tenga sentido es necesario que el metamodelo tenga asociadas invariantes para que se verifique que los modelos derivados las cumplan. Ecore permite definir estas invariantes de dos maneras, (1) asociando un archivo `.ocl` que contenga las invariantes y (2) ingresando los invariantes en el código Ecore del metamodelo.

Utilizando la API de Ecore para obtener información del modelo, se descubrió que no era posible obtener la información de las invariantes desde los archivos `.ocl`, por lo que se decide aceptar como única forma de definir invariantes el hacerlo directamente en el código el metamodelo.

Para realizar la validación, posicionarse sobre el archivo donde se define el modelo (archivo `.xmi`), clic derecho para desplegar las opciones y seleccionar la acción **Validate HaskellOCL**, como se ve en la figura B.1. Esto ejecutará la validación no funcional, reportando los resultados obtenidos en la vista de consola de Eclipse, como se muestra en la figura B.2

Si bien el proceso parece ejecutar directamente la validación, y de forma transparente, además de reportar los resultados obtenidos, se genera también como parte del proceso el archivo **`ACCELEO.hs`**. Este archivo es la interpretación funcional generada del modelo. Usuarios avanzados podrán utilizar este archivo para hacer cambios y pruebas directamente en el código funcional.

El archivo queda disponible, junto con la biblioteca auxiliar `oclLibrary.hs`, dentro del

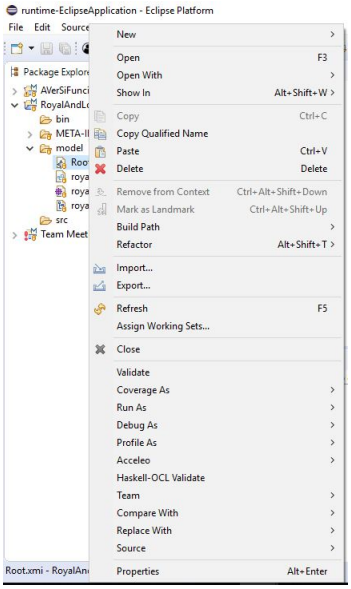


Figure B.1: Ejeuctar validación usando HaskellOCL

Figure B.2: Resultado de la evaluación presentado en consola

propio proyecto del *plugin*, en la carpeta **haskell**, como se puede ver en la siguiente Figura B.3

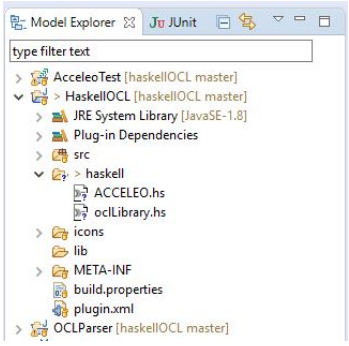


Figure B.3: Archivo `ACCELEO.hs` generado durante el proceso de evaluación

C Especificaciones Técnicas

C.1 Especificaciones técnicas

Especificaciones técnicas sobre las cuales se implementó el prototipo de *plugin* HaskellOCL que forma parte de este trabajo:

General	
Sistema Operativo	Windows 10 Pro (64-bit)
Memoria RAM	8 GB
Java version	Java 1.8.0 u 144
Haskell version	Haskell Platform 8.0.2-a
Eclipse	
Eclipse version	Eclipse IDE for Java Developers Oxygen.2 Release (4.7.2) (December 2017)
Acceleo version	3.7.1.201705121344
Eclipse Plug-in Development Environment version	3.13.2.v20171130-0510
Ecore Diagram Editor version	3.3.0.201706121316
EMF - Eclipse Modeling Framework version	2.13.0.v20180609-0928
OCL SDK version	6.3.0.v20180613-1432
Sirius version	5.0.1.201706290936

D

OCLLibrary

A continuación se listan las distintas funcionalidades de OCL implementadas y que se encuentran en la biblioteca **oclLibrary.hs** para ser utilizadas por cualquier representación de modelos.

La lista se encuentra dividida dependiendo del tipo sobre el que se aplican.

D.1 Real

OCL	Implementación Funcional

round() : Integer	`roundOCL :: OCL m (Val Double) -> OCL m (Val Int)`
	`roundOCL e1 = liftM (_roundOCL) e1`
	
	`_roundOCL (Val a) = Val (round a)`
	`_roundOCL _ = Inv`

max(r : Real) : Real	`maxOCL' :: OCL m (Val Double) -> Val Double -> OCL m (Val Double)`
	`maxOCL' e1 e2 = liftM2 (_max') e1 (ocl e2)`
	
	`_max' (Val x) (Val y) = Val (max x y)`
	`_max' _ _ = Inv`

min(r:Real): Real	`minOCL :: OCL m (Val Int) -> Val Int -> OCL m (Val Int)`
	`minOCL e1 e2 = liftM2 (_min) e1 (ocl e2)`
	
	`_min (Val x) (Val y) = Val (max x y)`
	`_min _ _ = Inv`

toString(): String	`toString' :: OCL m (Val Double) -> OCL m (Val String)`
	`toString' e1 = liftM (toStr') e1`
	
	`toStr' (Val x) = Val (show x)`
	`toStr' _ = Inv`

floor(): Real	`floorOCL :: OCL m (Val Double) -> OCL m (Val Int)`
	`floorOCL e1 = liftM (_floorOCL) e1`
	
	`_floorOCL (Val a) = Val (floor a)`
	`_floorOCL _ = Inv`

/ (r : Real) : Real	`(|/|) :: (Eq b, Super a Double, Super b Double) => OCL m (Val a) -> OCL m (Val b) -> OCL m (Val Double)`
	`e1 |/| e2 = liftM2 (////) e1 e2`
	
	`_ //// Val 0 = Inv`
	`Val x //// Val y = Val ((toSuper x)/(toSuper y))`
	`_ //// _ = Inv`

D.2 Integer

OCL	Implementación Funcional

max(r : Real) : Real	`maxOCL :: OCL m (Val Int) -> Val Int -> OCL m (Val Int)`
	`maxOCL e1 e2 = liftM2 (_max) e1 (ocl e2)`
	
	`_max (Val x) (Val y) = Val (max x y)`
	`_max _ _ = Inv`

min(i : Integer) : Integer	`minOCL :: OCL m (Val Int) -> Val Int -> OCL m (Val Int)`
	`minOCL e1 e2 = liftM2 (_min) e1 (ocl e2)`
	
	`_min (Val x) (Val y) = Val (max x y`
	`_min _ _ = Inv`

toString() : String	`toString :: Val Int -> OCL m (Val String)`
	`toString e1 = liftM (toStr) (ocl e1)`
	
	`toStr (Val x) = Val (show x)`
	`toStr _ = Inv`

mod(i : Integer) : Integer	`modOCL :: OCL m (Val Int) -> Val Int -> OCL m (Val Int)`
	`modOCL e1 e2 = liftM2 (_modOCL) e1 (ocl e2)`
	
	`_modOCL (Val x) (Val y) = Val (mod y x)`
	`_modOCL _ _ = Inv`

div(i : Integer) : Integer	`oclDiv :: OCL m (Val Int) -> Val Int -> OCL m (Val Int)`
	`oclDiv e1 e2 = liftM2 (///) e1 (ocl e2)`
	
	`_ /// Val 0 = Inv Val y /// Val x = Val (round((fromIntegral x)/(fromIntegral y)))`
	`_ /// _ = Inv`

D.3 String

OCL	Implementación Funcional

substring(lower : Integer, upper : Integer) : String	`substring :: OCL m (Val Int) -> OCL m (Val Int) -> Val String -> OCL m (Val String)`
	`substring n1 n2 s = liftM3 (_substring) n1 n2 (ocl s)`
	
	`_substring (Val a) (Val b) (Val xs) = if (a < 1 || b > (length xs)) then Inv else Val (_substring' a b xs)`
	`_substring _ _ _ = Inv`
	
	`_substring' a b xs = reverse (take (b-a+1) (reverse (take b xs)))`

size() : Integer	`strSize :: Val String -> OCL m (Val Int)`
	`strSize s = liftM (_size) (ocl s)`
	
	`_size (Val s) = Val(length s)`
	`_size _ = Inv`

concat(s : String) : String	`concat :: OCL m (Val String) -> Val String -> OCL m (Val String)`
	`concat n s = liftM2 (_concat) n (ocl s)`
	
	`_concat (Val s1) (Val s2) = Val (s2 ++ s1)`
	`_concat _ _ = Inv`

toInteger() : Integer	`toInteger :: Val String -> OCL m (Val Int)`
	`toInteger s = liftM (_toInteger) (ocl s)`
	
	`_toInteger (Val s) = _toInteger' (reads s :: [(Int, String)])`
	
	`_toInteger _ = Inv`
	`_toInteger' ((a, []):[]) = (Val a)`
	`_toInteger' _ = Inv`

toReal() : Real	`toReal :: Val String -> OCL m (Val Double)`
	`toReal s = liftM (_toReal) (ocl s)`
	
	`_toReal (Val s) = _toReal' (reads s :: [(Double, String)])`
	`_toReal _ = Inv`
	
	`_toReal' ((a, []):[]) = (Val a)`
	`_toReal' _ = Inv`

toUpperCase() : String	`toUpperCase :: Val String -> OCL m (Val String)`
	`toUpperCase s = liftM (_toUpperCase) (ocl s)`
	
	`_toUpperCase (Val s) = Val (_toUpper s)`
	`_toUpperCase _ = Inv`
	
	`_toUpper [] = []`
	`_toUpper (x:[]) = [toUpper x]`
	`_toUpper (x:xs) = (toUpper x):(_toUpper xs)`

toLowerCase() : String	`toLowerCase :: Val String -> OCL m (Val String)`
	`toLowerCase s = liftM (_toLowerCase) (ocl s)`
	
	`_toLowerCase (Val s) = Val (_toLower s)`
	`_toLowerCase _ = Inv`
	
	`_toLower [] = []`
	`_toLower (x:[]) = [toLower x]`
	`_toLower (x:xs) = (toLower x):(_toLower xs)`

OCL	**Implementación Funcional**

indexOf(s : String) : Integer	`indexOf :: OCL m (Val String) -> Val String -> OCL m (Val Int)`
	`indexOf s1 s2 = liftM2 (_indexOf) (ocl s2) s1`
	
	`_indexOf (Val []) (Val sub) = Val 0`
	`_indexOf (Val ori) (Val []) = Val 1`
	`_indexOf (Val ori) (Val sub) = if (isInfixOf sub ori) then Val (_indexOf' ori sub 1) else Val 0`
	`_indexOf _ _ = Inv`
	
	`_indexOf' o s n = if ((_substring' n (n + length s - 1)) o == s) then n else (_indexOf' o s (n+1))`

toBoolean() : Boolean	`toBoolean :: Val String -> OCL m (Val Bool)`
	`toBoolean e1 = liftM (_strToBoolean) (ocl (_toUpperCase e1))`
	
	`_strToBoolean (Val "TRUE") = Val True`
	`_strToBoolean (Val "FALSE") = Val False`
	`_strToBoolean _ = Inv`

equalsIgnoreCase(s : String) : Boolean	`equalsIgnoreCase :: OCL m (Val String) -> Val String -> OCL m (Val Bool)`
	`equalsIgnoreCase e1 e2 = liftM2 (_equalsIgnoreCase) e1 (ocl e2)`
	
	`_equalsIgnoreCase (Val x) (Val y) = Val (_toUpper x == _toUpper y)`
	`_equalsIgnoreCase _ _ = Inv`

at(i : Integer) : String	`at :: OCL m (Val Int) -> Val String -> OCL m (Val String)` `at n s = substring n n s`

characters() : Sequence(String)	`characters :: Val String -> OCL m (Val (Collection String))`
	`characters s = liftM (toCollection) (liftM (_characters) (ocl s))`
	
	`_characters (Val s) = Val (_chars s)`
	`_characters _ = Inv`
	
	`_chars [] = []`
	`_chars (x:[]) = [Val [x]]`
	`_chars (x:xs) = (Val [x]):(_chars xs)`

D.4 Boolean

OCL	Implementación Funcional

or (b : Boolean) : Boolean	`(||||) :: OCL m (Val Bool) -> OCL m (Val Bool) -> OCL m (Val Bool)`
	`e1 |||| e2 = (|||) <$> e1 <*> e2`
	
	`Val True ||| _ = Val True`
	`_ ||| Val True = Val True`
	`Val False ||| Val False = Val False`
	`_ ||| _ = Inv`

xor (b : Boolean) : Boolean	`(|%%|) :: OCL m (Val Bool) -> OCL m (Val Bool) -> OCL m (Val Bool)`
	`e1 |%%| e2 = liftM2 (%%%) e1 e2`
	
	`Val True %%% Val False = Val True`
	`Val False %%% Val True = Val True`
	`Inv %%% _ = Inv`
	`_ %%% Inv = Inv`
	`_ %%% _ = Val False`

and (b : Boolean) : Boolean, blue	`(|&&|) :: OCL m (Val Bool) -> OCL m (Val Bool) -> OCL m (Val Bool)` `e1 |&&| e2 = liftM2 (&&&) e1 e2`

not : Boolean	`notOCL :: OCL m (Val Bool) -> OCL m (Val Bool)`
	`notOCL e1 = liftM (!!!) e1`
	
	`(!!!) (Val True) = Val False`
	`(!!!) (Val False) = Val True`
	`(!!!) _ = Inv`

implies (b : Boolean) : Boolean	`(|==>|) :: OCL m (Val Bool) -> OCL m (Val Bool) -> OCL m (Val Bool)`
	`e1 |==>| e2 = (==>) <$> e1 <*> e2`
	
	`(==>) :: Val Bool -> Val Bool -> Val Bool`
	`Val False ==> _ = Val True`
	`Val True ==> b = b`
	`_ ==> Val True = Val True`
	`_ ==> _ = Inv`

toString() : String	`boolToString :: Val Bool -> OCL m (Val String)`
	`boolToString e1 = liftM (btoStr) (ocl e1)`
	
	`btoStr (Val True) = Val "true"`
	`btoStr (Val False) = Val "false"`
	`btoStr _ = Inv`

D.5 Collection

OCL	Implementación Funcional
size() : Integer	`size :: Val (Collection a) -> OCL m (Val Int)` `size list = (oclVal . length) (collectionToList list)`
includes(object : T) : Boolean	`includes :: Eq a => Val a -> Val (Collection a) -> OCL m (Val Bool)` `includes e list = (oclVal . (elem e)) (collectionToList list)`

asBag () : Bag(T)	`asBag :: Val (Collection a) -> OCL m (Val (Collection a))`
	`asBag col = liftM (oclAsBag) (ocl col)`
	
	`oclAsBag (Val (Set c)) = Val (Bag c)`
	`oclAsBag (Val (OrderedSet c)) = Val (Bag c)`
	`oclAsBag (Val (Bag c)) = Val (Bag c)`
	`oclAsBag (Val (Sequence c)) = Val (Bag c)`
	`oclAsBag _ = Inv`

asSet () : Set(T)	`asSet :: Eq a => Val (Collection a) -> OCL m (Val (Collection a))`
	`asSet col = liftM (oclAsSett) (ocl col)`
	
	`oclAsSett (Val (Set c)) = Val (Set c)`
	`oclAsSett (Val (OrderedSet c)) = Val (Set c)`
	`oclAsSett (Val (Bag c)) = Val (Set (nub c))`
	`oclAsSett (Val (Sequence c)) = Val (Set (nub c))`
	`oclAsSett _ = Inv`

asOrderedSet () : OrderedSet(T)	`asOrderedSet :: Eq a => Val (Collection a) -> OCL m (Val (Collection a))`
	`asOrderedSet col = liftM (oclAsOrderedSet) (ocl col)`
	
	`oclAsOrderedSet (Val (Set c)) = Val (OrderedSet c)`
	`oclAsOrderedSet (Val (OrderedSet c)) = Val (OrderedSet c)`
	`oclAsOrderedSet (Val (Bag c)) = Val (OrderedSet (nub c))`
	`oclAsOrderedSet (Val (Sequence c)) = Val (OrderedSet (nub c))`
	`oclAsOrderedSet _ = Inv`

asSequence () : Sequence	`asSequence :: Val (Collection a) -> OCL m (Val (Collection a))`
	`asSequence col = liftM (oclAsSequence) (ocl col)`
	
	`oclAsSequence (Val (Set c)) = Val (Sequence c)`
	`oclAsSequence (Val (OrderedSet c)) = Val (Sequence c)`
	`oclAsSequence (Val (Bag c)) = Val (Sequence c)`
	`oclAsSequence (Val (Sequence c)) = Val (Sequence c)`
	`oclAsSequence _ = Inv`

including (object : T) : Collection(T)	`including :: Eq a => OCL m (Val a) -> Val (Collection a) -> OCL m (Val (Collection a))`
	`including elem col = liftM2 (oclIncluding) elem (ocl col)`
	
	`oclIncluding x (Val (Set c)) = if (elem x c) then (Val (Set c)) else oclAppend(Val (Set c)) x`
	`oclIncluding x (Val (OrderedSet c)) = if (elem x c) then (Val (Set c)) else oclAppend (Val (OrderedSet c)) x`
	`oclIncluding x (Val (Bag c)) = oclAppend (Val (Bag c)) x`
	`oclIncluding x (Val (Sequence c)) = oclAppend (Val (Sequence c)) x`

OCL	Implementación Funcional

excluding (object : T) : Collection(T)	`excluding :: Eq a => OCL m (Val a) -> Val (Collection a) -> OCL m (Val (Collection a))`
	`excluding e c = liftM2 (oclExcluding) (ocl c) e`
	
	`oclExcluding (Val (Set c)) x = Val (Set (delete x c))`
	`oclExcluding (Val (OrderedSet c)) x = Val (OrderedSet (delete x c))`
	`oclExcluding (Val (Bag c)) x = Val (Bag (deleteAll x c))`
	`oclExcluding (Val (Sequence c)) x = Val (Sequence (deleteAll x c))`
	`oclExcluding _ _ = Inv`
	
	`deleteAll :: Eq a => a -> [a] -> [a]`
	`deleteAll x c = if (elem x c) then (deleteAll x (delete x c)) else c`

any (expr : OclExpression) : T	`any :: (Val a -> OCL m (Val Bool)) -> Val (Collection a) -> OCL m (Val a)`
	`any p list = let l1 = (ocl list) |->| (collect p) |->| (select isInv)`
	`in oclIf ((l1 |->| size) |>| (oclVal 0)) oclInv ((ocl list) |->| (select p) |->| asSequence |->| first)`

= (coll : Collection(T)) : Boolean	`(||==||) :: Eq a => OCL m (Val (Collection a)) -> OCL m (Val (Collection a)) -> OCL m (Val Bool)`
	`e1 ||==|| e2 = liftM2 (oclCollCMP) e1 e2`
	
	`oclCollCMP (Val (Bag c1)) (Val (Bag c2)) = Val (((length c1) == (length c2)) && (compareList c1 c2)) oclCollCMP (Val (Sequence c1)) (Val (Sequence c2)) = Val (((length c1) == (length c2)) && (c1 == c2))`
	`oclCollCMP (Val (Set c1)) (Val (Set c2)) = Val (((length c1) == (length c2)) && (compareList c1 c2)) oclCollCMP (Val (OrderedSet c1)) (Val (OrderedSet c2)) = Val (((length c1) == (length c2)) && (c1 == c2))`
	`oclCollCMP (Val (OrderedSet c1)) (Val (Set c2)) = Val (((length c1) == (length c2)) && (compareList c1 c2)) oclCollCMP _`
	
	`compareList :: Eq a => [a] -> [a] -> Bool`
	`compareList xs ys = [length (elemIndices x xs) | x <- xs] == [length (elemIndices x ys) | x <- xs]`

<> (coll : Collection(T)) : Boolean	`(||<>||) :: Eq a => OCL m (Val (Collection a)) -> OCL m (Val (Collection a)) -> OCL m (Val Bool)` `e1 ||<>|| e2 = notOCL (liftM2 (oclCollCMP) e1 e2)`

'-' (coll : Collection(T)) : Collection(T)	`(||-||) :: Eq a => OCL m (Val (Collection a)) -> OCL m (Val (Collection a)) -> OCL m (Val (Collection a))`
	`e1 ||-|| e2 = liftM2 (|-'|) e1 e2`
	
	`(Val (Set c1)) |-'| (Val (Set c2)) = Val (Set (c1 \\c2))`
	`(Val (OrderedSet c1)) |-'| (Val (Set c2)) = Val (Set (c1 \\c2))`
	`_ |-'| _ = Inv`

append (object : T) : Collection(T)	`append :: Eq a => OCL m (Val a) -> Val (Collection a) -> OCL m (Val (Collection a))`
	`append e2 e1 = liftM2 (oclAppend) (ocl e1) e2`
	
	`oclAppend (Val (Sequence c1)) o = Val (Sequence (c1 ++ [o]))`
	`oclAppend (Val (OrderedSet c1)) o = if (elem o c1) then (Val (OrderedSet c1)) else (Val (OrderedSet (c1 ++ [o])))`
	`oclAppend _ _ = Inv`

OCL	Implementación Funcional

at (index : Integer) : T	`att :: OCL m (Val Int) -> Val (Collection a) -> OCL m (Val a)`
	`att i c = liftM2 (oclAt) i (ocl c)`
	
	`oclAt (Val x) (Val (OrderedSet c)) = if ((x >= 1) && (x <= (length c))) then (c!!(x-1)) else Inv`
	`oclAt (Val x) (Val (Sequence c)) = if ((x >= 1) && (x <= (length c))) then (c!!(x-1)) else Inv`
	`oclAt _ _ = Inv`

first () : T	`first :: Val (Collection a) -> OCL m (Val a)` `first = att (oclVal 1)`

indexOf (object : T) : Integer	`indexOff :: Eq a => OCL m (Val a) -> Val (Collection a) -> OCL m (Val Int)`
	`indexOff elem col = liftM2 (oclIndexOf) (ocl col) elem`
	
	`oclIndexOf (Val (OrderedSet c)) x = (liftJust (elemIndex x c)) +++ (Val (1::Int))`
	`oclIndexOf (Val (Sequence c)) x = (liftJust (elemIndex x c)) +++ (Val (1::Int))`
	`oclIndexOf _ _ = Inv`
	
	`liftJust (Just x) = Val x`
	`liftJust _ = Inv`

insertAt (index : Integer, object : T) : Collection(T)	`insertAt :: Eq a => OCL m (Val Int) -> OCL m (Val a) -> Val (Collection a) -> OCL m (Val (Collection a))`
	`insertAt index elem col = liftM3 (oclInsertAt) (ocl col) index elem`
	
	`oclInsertAt (Val (OrderedSet c)) (Val i) e = if (elem e c) then (Val (OrderedSet c)) else (`
	`if (i < 1 && ((i-1) > (length c))) then Inv else Val (OrderedSet ((take (i-1) c) ++ [e] ++ (drop (i-1) c))))`
	`oclInsertAt (Val (Sequence c)) (Val i) e = if (i < 1 && ((i-1) > (length c))) then Inv else Val (Sequence ((take (i-1) c) ++ [e] ++ (drop (i-1) c)))`
	`oclInsertAt _ _ _ = Inv`

intersection (coll : Collection(T)) : Colleccion(T)	`intersection :: Eq a => OCL m (Val (Collection a)) -> Val (Collection a) -> OCL m (Val (Collection a))`
	`intersection c1 c2 = liftM2 (oclIntersection) c1 (ocl c2)`
	
	`oclIntersection (Val (Bag c1)) (Val (Bag c2)) = Val (Bag (oclListIntersection c1 c2))`
	`oclIntersection (Val (Set c1)) (Val (Bag c2)) = Val (Set (oclListIntersection c1 c2))`
	`oclIntersection (Val (Set c1)) (Val (Set c2)) = Val (Set (intersect c1 c2))`
	`oclIntersection (Val (Bag c1)) (Val (Set c2)) = Val (Set (oclListIntersection c1 c2))`
	`oclIntersection _ _ = Inv`
	
	`oclListIntersection :: Eq a => [a] -> [a] -> [a]`
	`oclListIntersection xs ys = let list1 = [length (elemIndices x xs) | x <- nub xs]`
	`list2 = [length (elemIndices x ys) | x <- nub xs] list3 = minListByPosition list1 list2`
	`in oclIntersectionAux (nub xs) list3`
	`oclIntersectionAux [] _ = []`
	`oclIntersectionAux [x] [y] = replicate y x`
	`oclIntersectionAux (x:xs) (y:ys) = (replicate y x) ++ oclIntersectionAux xs ys`

OCL	Implementación Funcional

last () : T	`lastt :: Eq a => Val (Collection a) -> OCL m (Val a)`
	`lastt col = liftM (oclLast) (ocl col)`
	
	`oclLast (Val (OrderedSet c)) = if (length c == 0) then Inv else last c`
	`oclLast (Val (Sequence c)) = if (length c == 0) then Inv else last c`
	`oclLast _ = Inv`

prepend (object : T) : Collection(T)	`prepend :: Eq a => OCL m (Val a) -> Val (Collection a) -> OCL m (Val (Collection a))` `prepend = insertAt (oclVal 1)`

subOrderedSet (startIndex : Integer, endIndex : Integer) : OrderedSet(T)	`subOrderedSet :: Eq a => OCL m (Val Int) -> OCL m (Val Int) -> Val (Collection a) -> OCL m (Val (Collection a))`
	`subOrderedSet lower upper set = liftM3 (oclSubOrderedSet) (ocl set) lower upper`
	
	`oclSubOrderedSet (Val (OrderedSet c)) (Val l) (Val u) = if ((l < 1) || (u >= (length c)) || (u > l)) then Inv else Val (OrderedSet (subList c l u))`
	`oclSubOrderedSet _ _ _ = Inv`
	
	`subList :: [Val a] -> Int -> Int -> [Val a] subList xs l u = reverse (drop (length xs - u) (reverse (drop (l-1) xs)))`

subSequence (startIndex : Integer, endIndex : Integer) : Sequence(T)	`subSequence :: Eq a => OCL m (Val Int) -> OCL m (Val Int) -> Val (Collection a) -> OCL m (Val (Collection a))`
	`subSequence lower upper seq = liftM3 (oclSubSequence) (ocl seq) lower upper`
	
	`oclSubSequence (Val (Sequence c)) (Val l) (Val u) = if ((l < 1) || (u >= (length c)) || (u > l)) then Inv else Val (Sequence (subList c l u))`
	`oclSubSequence _ _ _ = Inv`

symmetricDifference (coll : Collection(T)) : Collection(T)	`symmetricDifference :: Eq a => OCL m (Val (Collection a)) -> Val (Collection a) -> OCL m (Val (Collection a))`
	`symmetricDifference s1 s2 = liftM2 (oclSymmetricDifference) s1 (ocl s2)`
	
	`oclSymmetricDifference (Val (Set s1)) (Val (Set s2)) = Val (Set ((nub (s1 ++ s2)) (intersect s1 s2)))`
	`oclSymmetricDifference _ _ = Inv`

union (coll : Collection(T)) : Collection(T)	`unionn :: Eq a => OCL m (Val (Collection a)) -> Val (Collection a) -> OCL m (Val (Collection a))`
	`unionn c1 c2 = liftM2 (oclUnion) c1 (ocl c2)`
	
	`oclUnion (Val (Set c1)) (Val (Set c2)) = Val (Set (union c2 c1))`
	`oclUnion (Val (Bag c1)) (Val (Set c2)) = Val (Bag (c2 ++ c1))`
	`oclUnion (Val (Bag c1)) (Val (Bag c2)) = Val (Bag (c2 ++ c1))`
	`oclUnion (Val (Set c1)) (Val (Bag c2)) = Val (Bag (c2 ++ c1))`
	`oclUnion (Val (Sequence c1)) (Val (Sequence c2)) = Val (Bag (c2 ++ c1))`
	`oclUnion (Val (Bag c1)) (Val (OrderedSet c2)) = Val (Bag (c2 ++ c1))`
	`oclUnion (Val (Set c1)) (Val (OrderedSet c2)) = Val (Set (union c2 c1))`
	`oclUnion _ _ = Inv`

collection->iterate(elem : Type; acc : Type = <expression> | expression-with-elem-and-acc)	`iterate :: (Val b -> Val a -> OCL m (Val b)) -> Val b -> Val (Collection a) -> OCL m (Val b)` `iterate f b col = foldM f b (collectionToList col)`

collection->forAll(v : Type | boolean-expression-with-v)	`forAll :: (Val a -> OCL m (Val Bool)) -> Val (Collection a) -> OCL m (Val Bool)`
	`forAll p = allM p`
	
	`allM p l = mapM p (collectionToList l) »= return . andOCLInv`

OCL	Implementación Funcional

collection->select(boolean-expression)	`select :: (Val a -> OCL m (Val Bool)) -> Val (Collection a) -> OCL m (Val (Collection a))`
	`select p (Val (Bag xs)) = liftM (listToBag) (filterM (\x -> p x >= return . (== Val True)) (collectionToList (Val (Bag xs))))`
	`select p (Val (Set xs)) = liftM (listToSet) (filterM (\x -> p x >= return . (== Val True)) (collectionToList (Val (Set xs))))`
	`select p (Val (Sequence xs)) = liftM (listToSequence) (filterM (\x -> p x >= return . (== Val True)) (collectionToList (Val (Sequence xs))))`
	`select p (Val (OrderedSet xs)) = liftM (listToOrderedSet) (filterM (\x -> p x >= return . (== Val True)) (collectionToList (Val (OrderedSet xs))))`

collection->collect(v : Type | expression-with-v)	`collect :: (Val a -> OCL m (Val b)) -> Val (Collection a) -> OCL m (Val (Collection b))` `collect f list = liftM (listToBag) (mapM f (collectionToList list))`

collection->exists(v : Type | boolean-expression-with-v)	`exists :: (Val a -> OCL m (Val Bool)) -> Val (Collection a) -> OCL m (Val Bool)`
	`exists p = anyM p`
	`anyM p l = mapM p (collectionToList l) >= return . orOCLInv`

collecton->any (boolean-expression)	`any :: (Val a -> OCL m (Val Bool)) -> Val (Collection a) -> OCL m (Val a)` `any p list = let l1 = (ocl list) |->| (collect p) |->| (select isInv) in oclIf ((l1 |->| size) |>| (oclVal 0)) oclInv ((ocl list) |->| (select p) |->| asSequence |->| first)`

collection->isUnique (boolean-expression)	`isUnique :: Eq b => (Val a -> OCL m (Val b)) -> Val (Collection a) -> OCL m (Val Bool) isUnique p list = let l1 = (ocl list) |->| (collect p) in oclIf ((l1 |->| (select isInv) |->| size) |>| (oclVal 0)) oclInv (liftM (oclIsUnique) l1)`
	
	`oclIsUnique (Val (Bag xs)) = Val (xs == (nub xs))`
	`oclIsUnique _ = Inv`

collection->one (boolean-expression)	`one :: (Val a -> OCL m (Val Bool)) -> Val (Collection a) -> OCL m (Val Bool)`
	`one p list = let l1 = (ocl list) |->| (collect p) |->| (select isInv)`
	`in oclIf ((l1 |->| size) |>| (oclVal 0)) oclInv (((ocl list) |->| (select p) |->| size) |==| (oclVal 1))`

collection->reject (boolean-expression)	`reject :: (Val a -> OCL m (Val Bool)) -> Val (Collection a) -> OCL m (Val (Collection a))` `reject p = select (notOCL . p)`
size()	`size :: Val (Collection a) -> OCL m (Val Int)` `size list = (oclVal . length) (collectionToList list)`
includes(object : T) : Boolean	`includes :: Eq a => Val a -> Val (Collection a) -> OCL m (Val Bool)` `includes e list = (oclVal . (elem e)) (collectionToList list)`

D.6 Funciones lógicas

OCL	Implementación Funcional

= : Boolean	`(|==|) :: Eq a => OCL m (Val a) -> OCL m (Val a) -> OCL m (Val Bool)`
	`e1 |==| e2 = oclCmp (==) <$> e1 <*> e2`
	
	`oclCmp _ Null Null = Val True`
	`oclCmp _ Inv _ = Inv`
	`oclCmp _ _ Inv = Inv`
	`oclCmp f x y = Val (f x y)`

<> : Boolean	`(|<>|) :: Eq a => OCL m (Val a) -> OCL m (Val a) -> OCL m (Val Bool)` `e1 |<>| e2 = oclCmp (/=) <$> e1 <*> e2`
<= : Boolean	`(|<=|) :: Ord a => OCL m (Val a) -> OCL m (Val a) -> OCL m (Val Bool)` `e1 |<=| e2 = oclCmp (<=) $> e1 <*> e2`
< : Boolean	`(|<|) :: Ord a => OCL m (Val a) -> OCL m (Val a) -> OCL m (Val Bool)` `e1 |<| e2 = oclCmp (<) <$> e1 <*> e2`
> : Boolean	`(|>|) :: Ord a => OCL m (Val a) -> OCL m (Val a) -> OCL m (Val Bool)` `e1 |>| e2 = oclCmp (>) <$> e1 <*> e2`
>= : Boolean	`(|>=|) :: Ord a => OCL m (Val a) -> OCL m (Val a) -> OCL m (Val Bool)` `e1 |>=| e2 = oclCmp (>=) <$> e1 <*> e2`

implies : Boolean	`(|==>|) :: OCL m (Val Bool) -> OCL m (Val Bool) -> OCL m (Val Bool)`
	`e1 |==>| e2 = (==>) <$> e1 <*> e2`
	
	`(==>) :: Val Bool -> Val Bool -> Val Bool`
	`Val False ==> _ = Val True`
	`Val True ==> b = b`
	`_ ==> Val True = Val True`
	`_ ==> _ = Inv`

D.7 Funciones aritméticas

OCL	Implementación Funcional

+ : Number	`(|+|) :: (Num a, Num b, Num c, Super a c, Super b c) => OCL m (Val a) -> OCL m (Val b) -> OCL m (Val c)`
	`e1 |+| e2 = liftM2 (+++) e1 e2`
	
	`Val x +++ Val y = Val ((toSuper x) + (toSuper y))`
	`_ +++ _ = Inv`

- : Number	`(|-|) :: (Num a, Num b, Num c, Super a c, Super b c) => OCL m (Val a) -> OCL m (Val b) -> OCL m (Val c)`
	`e1 |-| e2 = liftM2 (.--) e1 e2`
	
	`Val x .-- Val y = Val ((toSuper x) - (toSuper y))`
	`_ .-- _ = Inv`
	
	`(|'-|) :: Num a => OCL m (Val a) -> OCL m (Val a)`
	`(|'-|) e1 = liftM (neg) e1`
	
	`neg (Val x) = Val(negate x)`
	`neg _ = Inv`

abs : Integer	`absOCL :: Num a => Val a -> OCL m (Val a)`
	`absOCL e1 = liftM (_absOCL) (ocl e1)`
	
	`_absOCL (Val a) = Val (abs a)`
	`_absOCL _ = Inv`

D.8 Funciones de tipos

OCL	Implementación Funcional
oclAsType(type : Classifier) : T	`oclAsType :: (Cast m a b, Cast m b a) => Val a -> Val b -> OCL m (Val a)` `oclAsType t e = do c <- downCast t e case c of Val _ -> return c _ -> upCast t e`
oclIsKindOf(type : Classifier) : Boolean	`oclIsKindOf :: (OCLModel m e, Cast m e a, Cast m e b) => Val a -> Val b -> OCL m (Val Bool)` `oclIsKindOf t e = do m <- modelElem r <- upCast m e case r of Val _ -> do c <- downCast t r oclVal $ case c of Inv -> False _ -> True _ -> oclVal False`
oclIsTypeOf(type : Classifier) : Boolean	`oclIsTypeOf :: Cast m a b => Val b -> Val a -> OCL m (Val Bool)` `oclIsTypeOf t e = do r <- downCast t e case r of Val _ -> do c <- directInstance r e oclVal $ case c of Inv -> False _ -> True _ -> oclVal False`