View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

***Título**: Razonamiento sobre Esquemas Conceptuales UML con Operaciones*

***Volumen**: 1/1*
***Alumno**: Fernando Cardona Cardona*

***Director/Ponente**: Ernest Teniente López*
***Departamento**: Enginyeria de Serveis i Sistemes d'Informació*
***Fecha**: 1/07/2011*

DATOS DEL PROYECTO

Título del Proyecto: Razonamiento sobre Esquemas Conceptuales UML con Operaciones

Nombre del estudiante: Fernando Cardona Cardona
Titulación: Ingeniería Técnica en Informática de Gestión
Créditos: 22,5
Director/Ponente: Ernest Teniente López
Departamento: Enginyeria de Serveis i Sistemes d'Informació(ESSI)

MIEMBROS DEL TRIBUNAL *(nombre y firma)*

Presidente: Juan Antonio Pastor Collado

Vocal: José Antonio Lubary Martínez

Secretario: Ernest Teniente López

CALIFICACIÓN

Calificación numérica:
Calificación descriptiva:

Fecha:

Agradecimientos

Quiero agradecer especialmente la colaboración de las personas que me ayudaron con su tiempo y conocimientos a sacar este proyecto adelante.

A todos ellos:

- Xavier Oriol
- Lluis Miquel Munguia
- Albert Tort
- Guillem Rull

Muchas Gracias

Agradecer también a Ernest Teniente, la paciencia, apoyo y conocimientos que me ha aportado durante la realización del proyecto.

Contenido

1. Introducción 8
   1.1. Motivación y contexto del proyecto 8
   1.2. Objetivos 10
   1.3. Tecnología y entorno de trabajo 11
   1.4. Planificación 12
      1.4.1. Coste temporal 12
      1.4.2. Coste económico 13
2. Conceptos básicos 14
   2.1. UML 14
      2.1.1. Clase 15
      2.1.2. Método (operación) 15
      2.1.3. Generalización 16
      2.1.4. Asociación 16
   2.2. OCL 17
      2.2.1. Invariantes 17
      2.2.2. Contratos 17
      2.2.3. Propiedades 18
      2.2.4. Métodos estándar 18
      2.2.5. Navegación 18
   2.3. Lógica 19
      2.3.1. Constante 19
      2.3.2. Variable 19
      2.3.3. Término 19
      2.3.4. Predicado 20
      2.3.5. Átomo 20
      2.3.6. Literal 20
      2.3.7. Regla o cláusula normal 21
3. Definición de un esquema conceptual UML con operaciones 22
   3.1. ArgoUML 22
   3.2. Ampliación de ArgoUML 23
      3.2.1. Restricciones textuales OCL: 23
      3.2.2. Tipos de datos 23
      3.2.3. Operaciones 24
4. Conceptos y métodos previos 26
   4.1. Carga de un esquema conceptual 26

4.1.1. XMI (XML Metadata Interchange) 26
4.1.2. EinaGMC 27
4.1.3. XMI Parser 28
4.2. Traducción de operaciones OCL a Lógica 29
4.2.1. Interpretación formal 29
4.2.2. Identificación de creación y eliminación de instancias en OCL 31
4.2.3. Restricciones 34
4.3. Validación del esquema 35
4.3.1. Satisfactibilidad de un esquema 35
4.3.2. Otros test 36
5. Herramienta 37
5.1. Especificación 37
5.1.1. Visión general 37
5.1.2. Modelo de casos de uso 38
5.2. Diseño 40
5.2.1. Arquitectura de la herramienta 40
5.2.2. Capa de presentación 40
5.2.3. Capa de dominio 41
5.2.4. Capa de datos 42
5.3. Implementación 43
5.3.1. Visión general 43
5.3.2. Carga de un esquema conceptual 44
5.3.3. Traducción de operaciones OCL a lógica 46
5.3.4. Validación del esquema 52
5.3.5. Problemas de validación 54
6. Ejemplo 55
6.1. Traducción del esquema estructural 56
6.2. Traducción del esquema de comportamiento 57
6.3. Test y conclusiones 59
7. Conclusiones 61
7.1. Sobre la herramienta y trabajos futuros 61
7.2. Valoración del proyecto 61
8. Referencias y bibliografía 62

1. Introducción

1.1.Motivación y contexto del proyecto

Durante el proceso de análisis/especificación de software, la corrección anticipada de errores, suele ser menos costoso que durante las etapas de diseño o implementación. Por lo tanto, es importante detectar y corregir los errores lo antes posible, para garantizar una base solida del modelo que se va a desarrollar.

Las técnicas actuales de validación de un modelo conceptual, se centran solo en el esquema estructural, dejando a un lado el esquema de comportamiento. Podemos determinar si un esquema estructural es satisfactible, es decir, comprobar si en el modelo existen contradicciones o redundancias, y si todas las clases y asociaciones puedan ser instanciadas. Sin embargo, un esquema estructural satisfactible no implica necesariamente que el esquema conceptual en conjunto también lo sea, es decir, si tenemos en cuenta que el esquema de comportamiento es el que define los cambios permitidos sobre el sistema, puede suceder que los eventos de las operaciones no cumplan con las restricciones del esquema estructural.

Así pues, la motivación de este PFC, es crear una herramienta que permita validar un esquema conceptual UML en presencia de operaciones. La técnica que se propone está basada en las ideas expuestas en la tesis doctoral de Anna Queralt “Validation of UML conceptual schemas with OCL constraints and operations” [1] dirigida por Ernest Teniente. Dicha tesis describe un método para traducir un modelo conceptual con operaciones a una representación lógica, de tal manera que cualquier método de razonamiento puede ser utilizado para realizar las pruebas de validación.

Esta herramienta se enmarca dentro del proyecto Aurus [2], una aplicación implementada por Guillermo Lubary Fleta, que permite validar esquemas estructurales con restricciones, que al igual que este PFC, se basa en los conceptos propuestos en la tesis doctoral [1]. Por lo tanto, esta nueva herramienta se nutre de dicha aplicación, y consigue a portar un nuevo enfoque de validación, donde se tienen en cuenta las operaciones definidas en el esquema.

Durante el proceso de especificación, la primera pregunta que nos planteamos es, o debería ser, ***es correcto el esquema que estoy construyendo?***

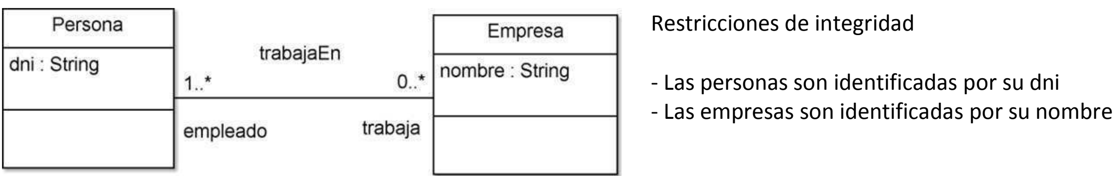
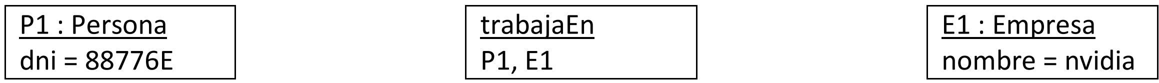


Figura 1 - Esquema estructural de la relación Persona, Empresa

```
Op: altaPersona(numero: String)
Pre:
Post: Se crea una nueva instancia de Persona con dni = numero
Op: nuevaEmpresa(emp: String)
Pre:
Post: Se crea una nueva instancia de Empresa con nombre = emp
```

Figura 2 - Esquema de comportamiento

El proceso de validación se refiere a la construcción de un estado correcto, es decir, comprobar si todas las clases y asociaciones pueden ser instanciadas. De acuerdo a esto, si utilizamos cualquiera de los métodos existentes para la validación de un modelo conceptual obtendríamos lo siguiente:



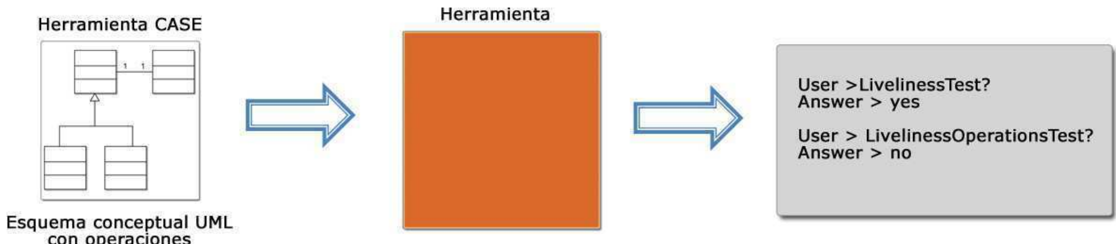
Por tanto, el esquema estructural es correcto, ya que se puede crear un estado que satisface todas las restricciones. Sin embargo, solo las operaciones del esquema de comportamiento son las que definen los cambios permitidos en el sistema, es decir, la creación y eliminación de instancias solo se puede dar como ocurrencia de una operación.

De esta forma, si tenemos en cuenta el esquema de comportamiento, ***podemos asegurar que el modelo conceptual es correcto?.*** En este caso no, ya que la operación *nuevaEmpresa* , viola la restricción de multiplicidad (1..*) al no crear una asociación entre la nueva *Empresa* y una *Persona*.

Gracias a este pequeño ejemplo, podemos entender el papel que juega la validación de un esquema de comportamiento en un modelo conceptual.

1.2.Objetivos

El propósito de este proyecto consiste en la implementación de una herramienta que permita la traducción de las operaciones (esquema de comportamiento) a una representación lógica. Y para la validación del esquema conceptual en conjunto, se parte desde la traducción y validación hecha por Aurus sobre el esquema estructural, y se complementa con la traducción de las operaciones para crear una traducción común, de esta forma y por medio de un método de razonamiento, crear un conjunto de pruebas para validar el esquema.



Desde una visión general del proyecto, podemos identificar tres grandes fases/objetivos a cumplir:

Carga de un esquema conceptual: Esta fase es la que da inicio al proceso de validación, es el mecanismo para cargar un esquema conceptual con operaciones dentro de nuestro entorno de programación

Traducción de operaciones OCL a lógica: El núcleo del proyecto, es quizás la fase más importante, a partir del esquema de comportamiento se analizan las operaciones y se identifican los eventos que generan, eventos que se traducen a predicados lógicos.

Validación del esquema: Con la traducción hecha por Aurus del esquema estructural y la aportación de este proyecto en la traducción de las operaciones, utilizaremos un razonador lógico que nos permitirá realizar un conjunto de pruebas para evaluar la (in)corrección del esquema.

Además de lo anterior, que representa el núcleo para la validación, no podemos olvidar que se necesita un mecanismo que nos facilite la interacción con la herramienta. Por esa razón, el último objetivo es la implementación de una interfaz grafica.

1.3.Tecnología y entorno de trabajo

La herramienta se ha implementado bajo dos entornos de programación, en lo que se refiere a la capa de dominio de la aplicación y la capa presentación.

Bajo la capa de dominio, se utilizo el lenguaje de programación Java. La decisión de su utilización, básicamente fue por dos motivos, dado que este proyecto parte de una base solida como lo es Aurus, y la finalidad es crear una herramienta conjunta, es sin lugar a dudas indispensable armonizar con el lenguaje de Aurus.
Además, una de las herramientas de apoyo que se necesita para cargar un modelo conceptual en nuestro entorno de programación, es la utilización de la librería java EinaGMC iniciativa del grupo GMC [4], que proporciona un entorno tecnológico para trabajar con esquemas conceptuales UML y OCL.

Por otro lado, para la implementación de la interfaz grafica, capa de presentación, hemos optado por llevarlo a la web. Esto conlleva a la utilización tanto de entorno (Tomcat) como herramientas de soporte (Struts) que ayuden a la utilización de java en la capa de dominio y que permitan la comunicación con la capa de presentación.

Tomcat es un servidor web, que funciona como contenedor para aplicaciones java. Y Struts es una herramienta de soporte para el desarrollo de aplicaciones web bajo el patrón MVC (modelo vista controlador). Ambos bajo el desarrollo de Apache Software Fundation. Gracias a estos elementos podemos implementar el núcleo de la aplicación en Java y crear una interfaz gráfica para la web, de una manera eficaz y simple.

1.4. Planificación

1.4.1. Coste temporal

Este proyecto se empezó el 14/02/2011, y según una primera valoración en el informe previo, se pretendía acabar con su desarrollo el 31/05/2011. Toda la planificación inicial se basaba en cuatro fases: estudio previo de los lenguajes y entorno, carga de un modelo, traducción esquema de comportamiento a lógica y finalmente la validación. Tanto para la fase de traducción como para la validación, al ser los núcleos de este proyecto se reservaron prácticamente un mes para cada una.

		Nombre de tarea	Duración	Comienzo	Fin
1		Estudio previo	10 días	lun 14/02/11	vie 25/02/11
2		**Carga de un esquema conceptual**	**17 días**	**lun 21/02/11**	**mar 15/03/11**
3		Estrategia	7 días	lun 21/02/11	mar 01/03/11
4		Implementación	10 días	mié 02/03/11	mar 15/03/11
5		Pruebas	7 días	lun 07/03/11	mar 15/03/11
6		**Traducción de operaciones OCL a Lógica**	**57 días**	**mié 16/03/11**	**jue 02/06/11**
7		Estrategia	9 días	mié 16/03/11	lun 28/03/11
8		Implmentación	29 días	lun 28/03/11	jue 05/05/11
9		Pruebas y correcciones	35 días	vie 15/04/11	jue 02/06/11
10		**Validación esquema**	**26 días**	**lun 09/05/11**	**lun 13/06/11**
11		Estrategia	5 días	lun 09/05/11	vie 13/05/11
12		Implementación	16 días	lun 16/05/11	lun 06/06/11
13		Pruebas y correcciones	5 días	mar 07/06/11	lun 13/06/11
14		Elaboración Memoria	32 días	lun 16/05/11	mar 28/06/11

Figura 3 - Planificación real del proyecto

Como se aprecia en la evolución real del proyecto, la fecha de finalización es el 28/06/2011, que dista mucho de la primera valoración. Las razones de esta desviación fueron los imprevistos que surgieron en las etapas de traducción y validación.

Una vez terminada una primera implementación de la traducción del esquema de comportamiento, se podían realizar validaciones de forma manual sobre el razonador SVTe. A partir de ese momento empezó el retraso sobre todo el proyecto, ya que los resultados que nos proporcionada el razonador, no concordaban con los esperados.

Después de realizar pruebas, buscar errores en la traducción y analizar detalladamente los resultados, se detectaron un par de problemas en SVTe. Estos problemas inducían a que los ejemplos no terminaran, ya que el análisis sobre la lógica generada, en algunos casos era interpretaba como ciclos recursivos sin posibilidad de reparar.

Id		Nombre de tarea	Duración	Comienzo	Fin
1		Estudio previo	10 días	lun 14/02/11	vie 25/02/11
2		**Carga de un esquema conceptual**	**17 días**	**lun 21/02/11**	**mar 15/03/11**
3		Estrategia	7 días	lun 21/02/11	mar 01/03/11
4		Implementación	10 días	mié 02/03/11	mar 15/03/11
5		Pruebas	7 días	lun 07/03/11	mar 15/03/11
6		**Traducción de operaciones OCL a Lógica**	**57 días**	**mié 16/03/11**	**jue 02/06/11**
7		Estrategia	9 días	mié 16/03/11	lun 28/03/11
8		Implmentación	29 días	lun 28/03/11	jue 05/05/11
9		Pruebas y correcciones	35 días	vie 15/04/11	jue 02/06/11
10		**Validación esquema**	**26 días**	**lun 09/05/11**	**lun 13/06/11**
11		Estrategia	5 días	lun 09/05/11	vie 13/05/11
12		Implementación	16 días	lun 16/05/11	lun 06/06/11
13		Pruebas y correcciones	5 días	mar 07/06/11	lun 13/06/11
14		Elaboración Memoria	32 días	lun 16/05/11	mar 28/06/11

Proyecto: PFC
Fecha inicio: 14/02/2011
Fecha final: 28/06/2011

Tarea	Hito externo	Informe de resumen manual
División	Tarea inactiva	Resumen manual
Hito	Hito inactivo	Sólo el comienzo
Resumen	Resumen inactivo	Sólo fin
Resumen del proyecto	Tarea manual	Progreso
Tareas externas	Sólo duración	Fecha límite

Planificación Real

Para finalizar con el análisis del coste temporal, se puede observar en el diagrama de Gantt, como las ultimas tareas se solaparon formando un pequeño caos. En principio hasta no estar terminada la fase de traducción no se debería empezar con la validación, pero realidad fue distinta. Debido a los problemas antes mencionados, hubo que estar en constante modificación sobre la implementación de la traducción a lógica. Finalmente, gracias a la ayuda de Ernest Teniente y Guillem Rull, pudimos detectar el problema y trasladar la solución a una nueva versión del SVTe.

1.4.2. Coste económico

En el estudio económico tendremos en cuenta únicamente los recursos humanos, ya que a nivel de desarrollo ningún software utilizado para la implementación de esta herramienta tuvo coste alguno, y como terminal para la implementación, aprovecharemos un ordenador personal que ya tenemos disponible.

La duración total del proyecto, teniendo en cuenta solo los días laborables durante el periodo del 14/02/2011 hasta el 28/06/2011, fueron 90 días, y la dedicación media al día, fue de 5horas. El presupuesto se hace en base al precio medio de coste por hora de un analista/programador, suponiendo una retribución media de 35€/hora.

Por lo tanto el coste total del proyecto se puede fijar en 450h * 35€/h = 15750€.

2. Conceptos básicos

Podemos ver el esquema estructural como una serie de restricciones graficas o textuales que definen las condiciones para cada una de las instancias del esquema conceptual, es decir, cada estado de la información base.

UML (Unified Modeling Language) es un lenguaje grafico, que sirve además para representar un esquema estructural por medio de un diagrama de clases, con sus restricciones graficas. Dado que los modelos gráficos no son suficientes para una especificación precisa, para las restricciones textuales (restricciones de integridad), nos apoyamos en un lenguaje textual, OCL (Object Constraint Language) que sirve para describir de forma formal expresiones en los modelos UML.

En cuanto al esquema de comportamiento, describe la dinámica del sistema, es decir, como el contenido de la información base cambia debido a la ejecución de las operaciones. Para formalizar dichas operaciones también utilizamos OCL como lenguaje de especificación.

A continuación se especifican los lenguajes antes comentados, sin embargo sus especificaciones formales se pueden encontrar en [6], [7] respectivamente. Además, dado que uno de los objetivos es obtener una representación lógica a partir del esquema de comportamiento, hemos de entender los conceptos de la lógica, para referirnos a los procesos de construcción y eventos que generan las operaciones.

2.1.UML

Lenguaje Unificado de Modelado (UML) es un estándar respaldado por el grupo OMG (Object Management Group) para el modelado de sistemas software. Es un lenguaje grafico para visualizar, especificar, construir y documentar un sistema.

UML cuenta con varios tipos de diagramas que muestran diferentes aspectos sobre los sistemas modelados. En este pfc nos centraremos en los diagramas de clases que representan los esquemas estructurales.

Los diagramas de clases enfatizan sobre los elementos que deben existir en el sistema y qué condiciones deben cumplir. Se representan por medio de clases, atributos y las relaciones entre ellos.

2.1.1. Clase

Una clase describe un conjunto de objetos (instancias) que existen en el mundo real. Estos objetos tienen las mismas propiedades (atributos) y comportamiento común. Representación visual de una clase con atributos y métodos en UML:

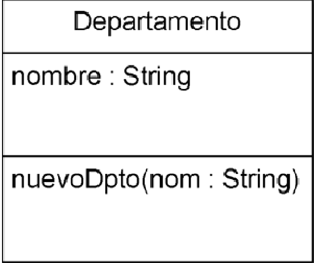


Figura 4 - Clase Departamento

Un atributo es una propiedad compartida por los objetos de la clase. Se muestran con un nombre, tipo, y otras propiedades como valor inicial o su privacidad.

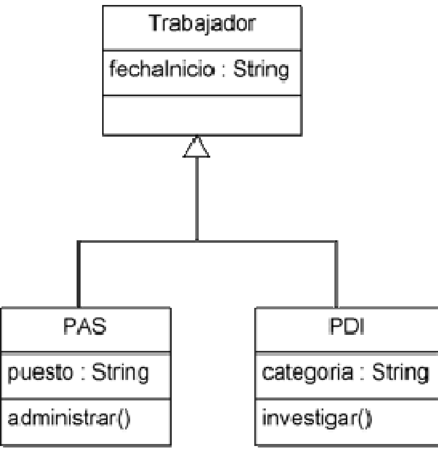
2.1.2. Método (operación)

En la etapa de especificación de software, las operaciones no se asignan a objetos concretos, forman parte del sistema como operaciones del tipo especial "sistema". Sin embargo como se explicará en el capítulo 3.2.3, las operaciones estarán presentes en las clases, solo como una forma rápida de modelado, sin que esto influya en las reglas de especificación.

En UML las operaciones también se muestran con al menos su nombre, parámetros y valores de retorno, y otras propiedades como pueden ser la privacidad. (ver Figura 4 *nuevoDpto*)

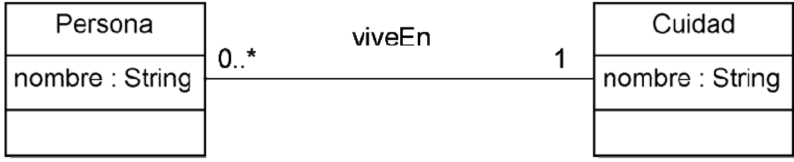
2.1.3. Generalización

Las generalizaciones (herencias, especialización) permiten que una clase (superclase) comparta atributos y métodos con una clase derivada de ella (subclase). Una subclase a su vez, puede modificar o añadir más atributos y métodos.



2.1.4. Asociación

Una asociación representa una relación entre clases, y aporta una semántica común. En otras palabras, son los mecanismos que permiten a los objetos de clases comunicarse entre sí.
Las asociaciones pueden tener un nombre que especifica el propósito así como la dirección de lectura. Cada extremo un la asociación tiene un nombre (rol) que aclara su papel dentro de la asociación, además tiene un valor de multiplicidad, que indica la cantidad de objetos que están relacionados en dicho extremo.



2.2.OCL

OCL (Object Constraint Language) en su versión actual 2.0, fue adoptado por el grupo OMG como parte de UML 2.0. Es un lenguaje de especificación formal de expresiones en modelos UML, permite representar expresiones como: invariantes, precondiciones, postcondiciones, inicializaciones, reglas de derivación, así como consultas a objetos para determinar sus condiciones de estado.

En este pfc nos centraremos únicamente en los contratos de las operaciones, y solo aquellas con carácter a modificar el estado del sistema. Sin embargo también es importante saber la estructura de un invariante, ya que nuestro punto de partida es un esquema conceptual con restricciones textuales (invariantes) validado por Aurus.

2.2.1. Invariantes

Un invariantes es una expresión a la que se le aplica el estereotipo estándar *invariant* (su abreviación es ***inv***) en el contexto de un clasificador, llamado tipo. La expresión es un invariante del tipo y debe satisfacerse para cada una de sus instancias.
Ejemplo:

```
context Persona
inv: self.edad >= 0
```

Este invariante se lee de la siguiente manera: "Todas las instancias de la clase ***Persona*** deben tener en el atributo ***edad*** un valor mayor o igual a cero".

2.2.2. Contratos

Un contrato es un documento que describe el comportamiento de una operación. El tipo de descripción es declarativo, con un neto énfasis en *qué* es lo que debe ocurrir y no en *cómo* se debe conseguir, además de describir la salida que proporciona cuando de invoca la operación. La esencia de un contrato es describir el estado que debe satisfacer una instancia (sobre la que se aplica la operación) *antes* de la ejecución de la operación (*precondiciones*), y describir el estado de la instancia *después* de la ejecución de la operación (postcondiciones).
Estructura formal de un contrato:

```
context Typename::operationName(param1 : Type1, … ) : ReturnType
pre: param1 > …
post: result =
```

Como se ha comentado en el punto 2.1.2, las operaciones se aplicarán a la especificación de las operaciones del sistema y por tanto *Typename* siempre será ***sistema***.

2.2.3. Propiedades

Es posible acceder a las propiedades de los objetos (los cuales pueden ser a su vez objetos) mediante el operador " . " seguido del nombre de la propiedad. Una propiedad es un atributo, un extremo de una asociación o un método sin efectos laterales (sobre el estado de la instancia contextual). Las propiedades de las colecciones son accedidas en forma similar pero usando el operador " -> ".

2.2.4. Métodos estándar

OCL cuenta con un amplio catalogo de métodos para trabajar con tipos como: boolean, string, collection, set, bag. Dado que este PFC se centra en la identificación de creación y eliminación de instancias, nos centraremos solo en los métodos necesarios.

Operación	Resultado
`allInstances`	Retorna el conjunto de todos los elementos del objeto
`exists(expression)`	expression es cierto para algún elemento?
`OclIsNew()`	Evalúa cierto si al ser usado en una postcondición el objeto es creado durante la ejecución de la operación
`OclIsTypeOf(object)`	Evalúa a cierto si self y el objeto son del mismo tipo
`includes(object)`	Cierto si el objeto pertenece a la colección
`excludes(object)`	Cierto si el objeto no pertenece a la colección

2.2.5. Navegación

A partir de un objeto específico es posible navegar a través de una asociación en un diagrama de clases para acceder a otros objetos y sus propiedades. Para ello se usa el nombre del *rol* del extremo opuesto de la asociación. En caso de no estar especificado el nombre del rol, se usa el nombre de la clase en minúsculas.

2.3.Lógica

Como ya se ha comentado, UML y OCL son lenguajes de especificación que nos permiten modelar la realidad. Directamente no podemos realizar ninguna validación sobre estos lenguajes, sin embargo, gracias a la lógica, también podemos realizar una representación formal del esquema UML con su parte de comportamiento.

Vamos a resumir algunos conceptos básicos de la lógica de primer orden, para comprender las expresiones que se utilizan a lo largo de este PFC, así como el metamodelo que representa todos estos conceptos, basado en el que se expone en [8].

2.3.1. Constante

Una constante es un valor que refiere a un dominio (entidad). Por ejemplo ("Pedro", "María", "Juan") que refieren a Personas. Una entidad no tiene que existir para que se pueda hablar de ella, de modo que la lógica de primer orden tampoco hace supuestos acerca de la existencia o no de las entidades a las que refieren sus constantes.

Las constantes se presentan con una letra minúscula {a,b,c, …} o bien pueden ser {1,2,3, …} si conocemos su valor exacto.

2.3.2. Variable

Una variable es una expresión cuya referencia no está determinada, representa un rango de valores pertenecientes a un dominio. Se representan con letras mayúsculas {A, B, C, …}.

2.3.3. Término

Un término T es una variable o constante. Así {A, a, "Pedro", 3, 5} son un conjunto de términos. Una barra horizontal sobre una variable o constante, representa un conjunto de términos de variables o constantes. Asi, si vemos una $\bar{U}$, sabemos que en realidad puede estar refiriéndose a {U,W,X, …}. Y si por el contrario vemos $\bar{a}$ puede significar {a,b,c, …}.

2.3.4. Predicado

Un predicado es una palabra o letra (Mayúscula) que identifica un concepto o dominio. Por ejemplo Persona, Cuidad o PersonaViveEnCuidad son predicados. Dentro de un esquema conceptual UML pueden referirse a las clases, asociaciones u operaciones.

2.3.5. Átomo

Sean T_1, …, T_n términos y P un predicado, entonces P(T_1, …, T_n) es un átomo. Por ejemplo Persona("Pedro"), Cuidad("Mallorca") o PersonaViveEnCuidad("Pedro", "Mallorca") también son átomos.

2.3.6. Literal

Hay dos tipos de literales:

- Literal ordinal: Es un átomo verdadero o falso. Por ejemplo
  Persona(X) y ¬*Persona(X)* son literales ordinales.

- Literal Built-In: Es una formula expresada de la siguiente forma: T_1 opComp T_2

donde T_1 y T_2 son términos y *opComp* es un operador de comparación <, >, =, <>, >=, <=

2.3.7. Regla o cláusula normal

Una regla de deducción o cláusula normal es una formula lógica de primer orden. De la forma:

$P(T_1, \ldots, T_n) \leftarrow L_1 \wedge L_2 \wedge \ldots \wedge L_n$

donde la parte izquierda (*head*) $P(T_1, \ldots, T_n)$ es un átomo que denota conclusión. El cuerpo de la regla, la parte derecha $L_1 \wedge L_2 \wedge \ldots \wedge L_n$ es una conjunción de literales representando una condición.

Se puede leer una regla de forma general de la siguiente manera:

"*Si todos los literales del cuerpo de la regla son cientos, entonces el literal head también lo es*".

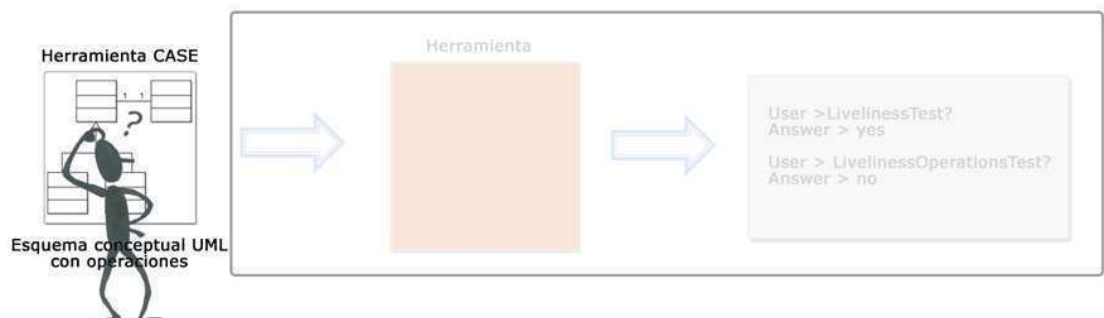
Si la regla no tiene cuerpo se dice que su *head* (átomo del head) es un "hecho", es decir, es cierto siempre. En cambio, si la regla no tiene *head*, se dice que la regla es un objetivo, utilizada normalmente para expresar condiciones.

Las reglas sin *head*, se expresan en términos de "*no puede ser que …*":

← *PersonaViveEnCuidad("Pedro", "Mallorca") ^ ¬ Cuidad("Mallorca")*

"*No puede ser que Pedro viva en Mallorca y que no exista la cuidad de Mallorca*"

3. Definición de un esquema conceptual UML con operaciones



El primer paso es crear nuestro esquema conceptual UML con operaciones, es el modelado, para tal propósito necesitamos una herramienta que nos facilite la tarea. En este punto, había que decidir entre dos soluciones: Utilizar una única herramienta para modelar tanto el esquema estructural como las operaciones, o crear una pequeña herramienta para el esquema de comportamiento. La gracia de crear una herramienta independiente para la operaciones, era generar las operaciones en base a los eventos que se querían efectuar en el sistema, y de esta forma controlar el formato de las operaciones y ajustarlo a nuestras necesidades. Sin embargo, limitar la forma de creación de las operaciones no es necesario, ya que lo que se pretende es traducir y validar exactamente lo que el diseñador especifica.

Por tanto, se utilizará una única herramienta para especificar tanto el esquema estructural como el esquema de comportamiento. Ahora bien, que herramienta utilizamos? de las diferentes herramienta existentes en el mercado, principalmente lo que buscamos durante todo el proyecto es la utilización de software libre, así que nos decantaos por ArgoUML, además de ser libre, es una herramienta que ya he utilizando durante algunas asignaturas de la carrera.

3.1.ArgoUML

ArgoUML es una aplicación de modelado para crear, visualizar y modificar diagramas UML. Para los ejemplos realizados en este PFC se ha trabajado con la versión 0.30.2

que nos ofrece ArgoUML?

Aurus en su comienzo utilizaba la herramienta Poseidon, que al igual que ArgoUML sirve para crear diagramas UML, sin embargo Poseidon tiene algunas limitaciones importantes que han hecho decantar la balanza sobre ArgoUML, por ejemplo, no se pueden crear asociaciones n-arias: las asociaciones solo pueden tener dos extremos. Estas limitaciones obligaban a diseñar todo lo que fuera posible con Poseidon, cargar el esquema en la EinaGMC y finalmente completar invocando las operaciones de la librería. Así pues, una de las mejoras introducidas, fue realizar el proceso de diseño utilizando ArgoUML.

3.2.Ampliación de ArgoUML

Dado que ArgoUML solo sirve para crear diagramas UML, debemos de representar de alguna manera las restricciones textuales OCL, así como también los contratos de las operaciones. Los convenios para modelar todos los elementos se describen a continuación:

3.2.1. Restricciones textuales OCL:

Para representar las restricciones textuales (invariantes), se utiliza el elemento "*comentario*".

Icono correspondiente a un comentario en la herramienta:

3.2.2. Tipos de datos

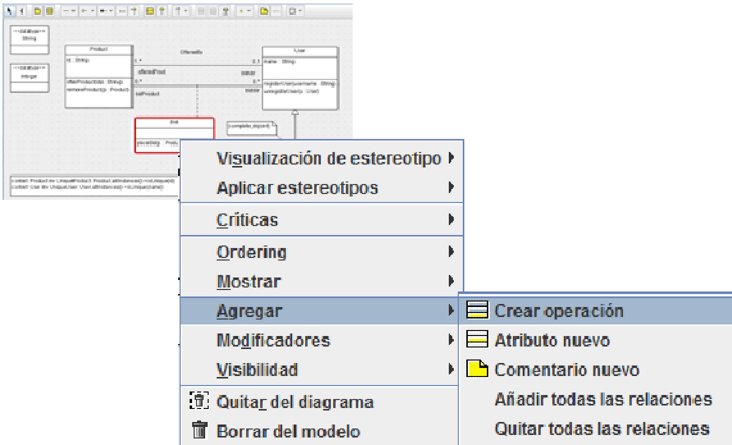
Todos los tipos de datos, tanto para los atributos del diagrama de clases como para los parámetros de las operaciones se han de modelar, es decir, se han de crear explícitamente en el modelo. Esto es debido a que, ArgoUML utiliza sus propios formatos de datos que impiden la extracción de la información.

Icono correspondiente para la creación de un nuevo tipo de datos en la herramienta:

3.2.3. Operaciones

Uno de los convenios para modelar el esquema de comportamiento, fue la representación de las operaciones dentro del diagrama de clases. Dado que las operaciones forman parte del tipo "*sistema*" no están sujetas a ninguna clase, y por tanto, no existe ningún tipo de restricción a la hora de modelarlas en alguna clase en concreto.

Para escribir los contratos de las operaciones, se crean utilizando las opciones que la herramienta nos proporciona, definiendo el nombre de la operación y los parámetros de entrada / salida.



Por otra parte, las precondiciones y postcondiciones se escriben en un campo en concreto de las propiedades de la operación. El campo se llama "***Especificación***", y dentro se utiliza el lenguaje formal OCL para especificar solo la pre y post, como se indica en el punto 2.2.2

A continuación, podemos ver un diagrama conceptual UML con restricciones de integridad y operaciones, modelado con ArgoUML.

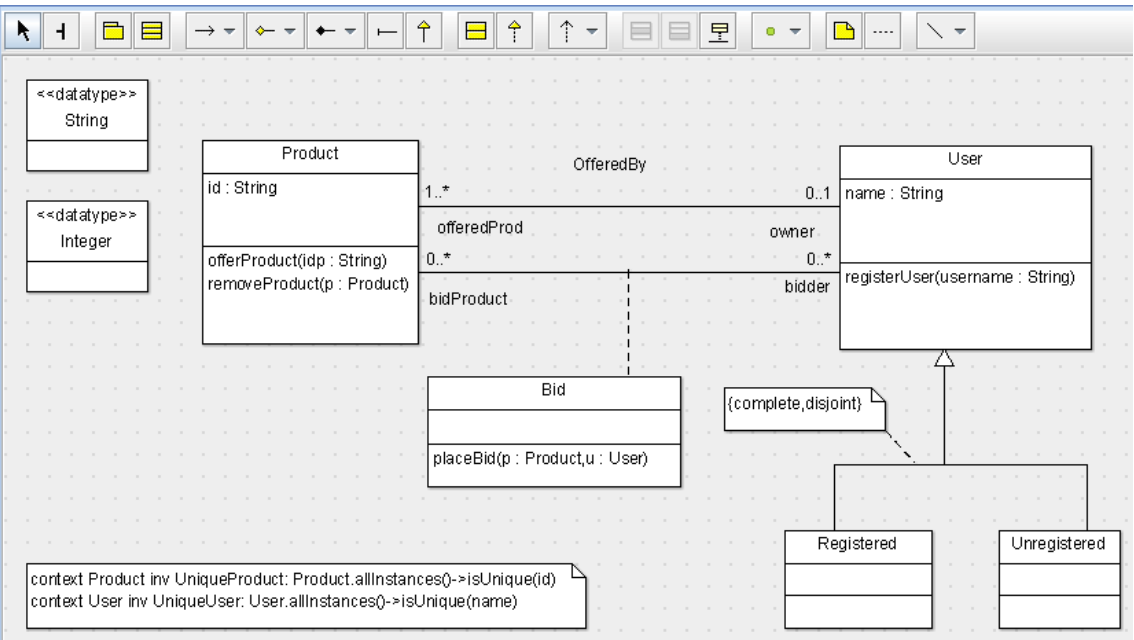


Figura 5- Esquema conceptual UML con operaciones

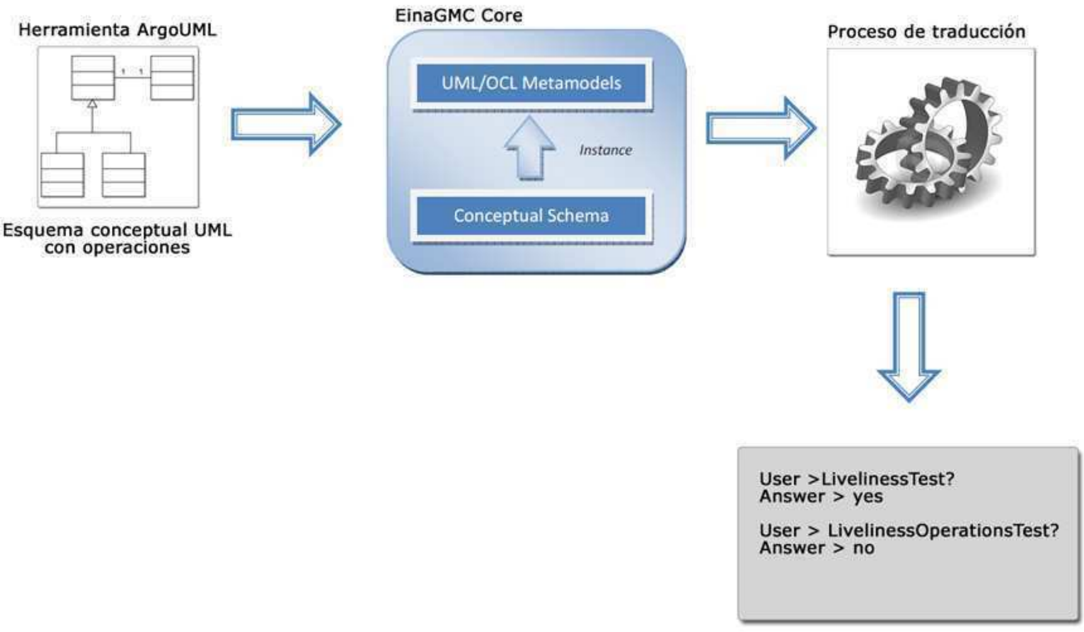
Los contratos de las operaciones son los siguientes:

```
Op: registerUser(username: String)
Pre:
Post: Registered.allInstances()-> exists(u | u.oclIsNew() and u.name =
username)

Op: placeBid(p: Product, u: User)
Pre: u.oclIsTypeOf(Registered)
Post: Bid.allInstances()-> exists(b | b.oclIsNew() and b.bidder = u and
b.bidProduct = p)

Op: offerProduct(idp: String)
Pre:
Post: Product.allInstances()->exists(p| p.oclIsNew() and p.id=idp)

Op: removeProduct(p: Product)
Pre:
Post: Product.allInstances()->excludes(p)
```

Este ejemplo será el que utilicemos durante toda la memoria para exponer la validación de un esquema conceptual UML con operaciones.

4. Conceptos y métodos previos

En este capítulo se describe de una forma detallada algunos componentes, herramientas y métodos de formalización que la herramienta utiliza y que nos ayudarán a comprender mejor su funcionamiento. Sin embargo, no es hasta el capitulo 5, que se describe realmente como es funcionamiento interno de la herramienta.



4.1.Carga de un esquema conceptual

Este es el primer reto que debe afrontar el proyecto para llegar a la validación del esquema conceptual UML con operaciones. Antes de poder acceder al modelo desde nuestro entorno de programación, se debe cargar en memoria.

4.1.1. XMI (XML Metadata Interchange)

La primera pregunta que nos planteamos es: ¿Cómo codifico mi esquema conceptual UML para cargarlo en memoria? Actualmente las diferentes herramientas para el modelado de diagramas permiten la exportación a un formato llamado XMI.

XMI (XML Metadata Interchange) es una especificación estándar respaldada por el grupo OMG (Object Management Group) basada en el lenguaje de tags para el intercambio de diagramas. En concreto la finalidad de XMI es ayudar a los programadores que utilizan el modelado (UML) con diferentes leguajes y herramientas de desarrollo para el intercambio de sus modelos de datos entre sí.

Gracias a las ventajas de este formato, establecemos XMI como entrada estándar a la herramienta, en el punto 4.1.3 se explica la forma en cómo se importa esta información.

4.1.2. EinaGMC

El proceso de cargar un esquema conceptual en memoria, es la vía que tenemos para la manipulación del esquema dentro de nuestro entorno de programación (Java). Así pues, la EinaGMC nos proporciona un conjunto de clases que implementan las metaclases de UML y metamodelos OCL, que nos permite crear instancias de esquemas conceptuales como un conjunto de objetos Java que se pueden acceder y manipular.

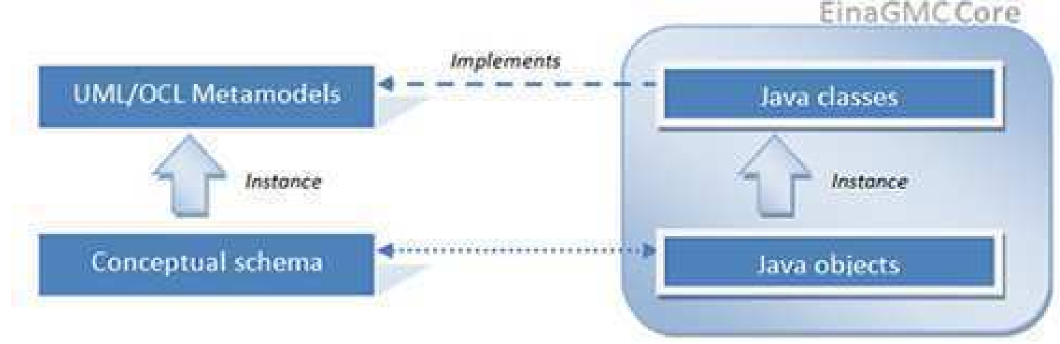


Figura 6 - Funcionalidad de EinaGMC

En términos de implementación, cada esquema conceptual instanciado es del tipo: ***Project***

4.1.3. XMI Parser

Una vez establecida la entrada a nuestra herramienta, que será un esquema conceptual codificado en XMI y utilizando la librería EinaGMC, la forma de guardar esta información en nuestro entorno de programación (Java), solo falta implementar un mecanismo que realice esta tarea de una forma automática.

Es un proceso que a simple vista no alberga dificultad, sin embargo es un proceso complejo, ya que debemos asegurar al diseñador que su modelo fue cargado exactamente como él lo definió, no hay lugar a error. Además, la lectura de un archivo siempre es un proceso más lento, por esa razón es vital maximizar la eficiencia de este proceso. A grandes rasgos, el método se encarga de identificar cada elemento del diagrama de clases y cada operación, y realiza la invocación a los métodos de la EinaGMC creando un nuevo *proyect* e instanciando cada objeto del modelo. El proceso interno se describe en términos de implementación en el capitulo 5.3.2.

4.2.Traducción de operaciones OCL a Lógica

Las pruebas de validación que realiza Aurus, solo tienen en cuenta el esquema estructural y su objetivo es la comprobación de que la creación de instancias cumplan unas determinadas propiedades y restricciones de integridad, con el fin de encontrar un estado de la información base que satisfaga dichas propiedades o restricciones. En este caso, la traducción de las clases, atributos y asociaciones se traducen como predicados base que puedan crear instancias, siempre y cuando cumplan con todas las restricciones definidas.

Sin embargo ya que ahora consideraremos el esquema de comportamiento, las clases y asociaciones están determinadas por los acontecimientos que han ocurrido en un cierto instante de tiempo. En otras palabras, el estado de la información base en un cierto tiempo ***t*** es solo el resultado de todas las operaciones que han sido ejecutadas antes de ***t***.

Por ejemplo, de acuerdo con nuestro esquema de la Figura 5 y las operaciones definidas, *Pedro* solo puede ser una instancia de *Registered* en un tiempo ***t*** si se ha ejecutado la operación *registerUser* en algún momento antes de ***t*** y la operación *unregisterUser* no lo ha eliminado entre el momento de su creación y ***t***.

Para la traducción, las operaciones serán los predicados básicos. Las clases y asociaciones estarán representadas por medio de predicados derivados y sus reglas de derivación se aseguran de que sus casos son precisamente los propuestos por las operaciones realizadas.

La explicación de las reglas formales para la traducción de las operaciones a lógica sigue los métodos descritos en la tesis “Validation of UML conceptual schemas with OCL constraints and operations”, sin embargo, en el punto 5.3.3 veremos que las reglas han sufrido pequeñas modificaciones con el fin de optimizar y adaptarnos a la traducción realizada por Aurus.

4.2.1. Interpretación formal

Las clases y asociaciones se representan por medio de predicados derivados cuyas reglas de derivación se aseguran de que sus casos solo se dan por la ocurrencia de las operaciones, que son los predicados base de la formalización de la lógica.

Una instancia de un predicado *p* (clase o asociación) existe en un tiempo *t* si ha sido añadido por una operación en un tiempo *t2* antes de *t*, y no ha sido borrado por cualquier otra operación entre *t2* y *t*. Formalmente la regla de derivación es:

$p([P,] P_1,\ldots,P_n,T) \leftarrow addP([P,] P_1,\ldots,P_n,T2) \wedge \neg deletedP(P_i,\ldots,P_j,T2,T) \wedge T2 \leq T \wedge time(T) \wedge time(T2)$
$deletedP(P_i,\ldots,P_j,T1,T2) \leftarrow delP(P_i,\ldots,P_j,T) \wedge T>T1 \wedge T \leq T2 \wedge time(T) \wedge time(T1) \wedge time(T2)$
donde:

- P es el OID (Object Identifier), si *p* es una clase.
- $P_i,\ldots,P_j$ son los términos suficientes para identificar una instancia de *p*
- En particular si *p* es una clase o una clase asociativa, $P= P_i = P_j$

El predicado *time* indica que son variables que representan el tiempo en que se producen los eventos, que aparecen en los predicados derivados que estamos definiendo. Estos predicados son base y no se pueden deducir del esquema.

Los predicados *addP* y *delP* también son predicados derivados que contienen alguna operación si ha creado o eliminado una instancia de *p* en el tiempo T.

Para cada operación que especifique la creación de una instancia se define la siguiente regla:
$addP([P,] Par_i,\ldots,Par_k,T) \leftarrow op\text{-}addP_i([P,] Par_1,\ldots,Par_m,T) \wedge pre_i(T_{pre}) \wedge T_{pre}=T-1 \wedge time(T)$

El predicado $op\text{-}addP_i$, es una operación del esquema de comportamiento, con parámetros $Par_1,\ldots,Par_m$ y con precondición $pre_i(T_{pre})$ de manera que su postcondición especifica la creación de una instancia de *p.* Hay que tener en cuenta que la precondición deber tener una aparición antes que la operación, el tiempo para todos estos hechos es T-1.

Del mismo modo para cada operación $op\text{-}delP_i(Par_1,\ldots,Par_n,T)$ con precondición pre_i que elimine una instancia de *p*, se difine la siguiente regla:

$delP(Par_i,\ldots,Par_j,T) \leftarrow op\text{-}delP_i(Par_1,\ldots,Par_n,T) \wedge pre_i(T_{pre}) \wedge T_{pre}=T-1 \wedge time(T)$

donde $Par_i,\ldots,Par_j$ son los parámetros de la operación que identifican la instancia a eliminar. Por lo tanto, si *p* es una clase o clase asociativa, delP además de T, tiene un único termino que corresponde con el OID de la instancia a eliminar.

4.2.2. Identificación de creación y eliminación de instancias en OCL

Para definir todas las reglas de derivación del esquema de comportamiento, tenemos que saber que operaciones OCL son responsables de crear o eliminar instancias. Se supone que las operaciones crean instancias con la información suministrada por los parámetros o eliminan instancias que se dan como parámetros. Una sola operación puede crear y/o eliminar varios casos. Las operaciones de consulta no interesan, ya que no afectan la exactitud del esquema.

Varias expresiones OCL se pueden utilizar para especificar que una instancia existe o no en un momento de la postcondición. Para simplificar la traducción, se considera una sola forma de especificar cada una de estas condiciones, ya que otras expresiones OCL con el mismo significado, pueden ser reescritas fácilmente en términos de los que aquí se describen.

Bajo este supuesto, se definen unas reglas para identificar la creación y eliminación de instancias en las postcondiciones OCL:

R1.
Una instancia c(I, A_1,….., A_n, T) de una clase C es creada por una operación si en su postcondición incluye la expresión OCL:
`C.AllInstances() -> exists(i | i.oclIsNew() and i.attr`$_i$` = a`$_i$`)`
o la expresión:
`i.oclIsTypeOf(C) and i.attr`$_i$` = a`$_i$
donde cada attr$_i$ es un único valor del atributo de C.

R2.
Una instancia c(I, P_1, ….., P_n, A_1, ….., A_m, T) de una clase asociativa es creada por una operación si en su postcondición incluye la expresión OCL:
`C.AllInstances() -> exists(i | i.oclIsNew() and i.part`$_1$` = p`$_1$` and ... and i.part`$_n$` = p`$_n$` and i.attr`$_1$` = a`$_1$` and ... and i.attr`$_m$` = a`$_m$`)`
o la expresión:
`i.oclIsTypeOf(C) and i.part`$_1$` = p`$_1$` and ... and i.part`$_n$` = p`$_n$` and i.attr`$_1$` = a`$_1$` and ... and i.attr`$_m$` = a`$_m$
donde cada part$_i$ es un participante que define la asociación, y cada attr$_j$ es un único valor del atributo de C.

R3.

Una instancia r(C_1, C_2, T) de una asociación binaria R entre los objetos C_1 y C_2 , con roles *role-c_1* y *role-c_2* en *r* es creada por una operación si en su postcondición incluye la expresión OCL:

`ci.role-cj = cj,` si la multiplicidad de *role-c_j* es al menos 1

o la expresión:

`ci.role-cj -> includes(cj)`, si la multiplicidad de *role-c_j* es más grande que 1.

Esta regla también se aplica a los valores de los atributos de múltiples valores. La creación o eliminación de instancias de asociaciones n-arias con n>2 no pueden expresarse en OCL a menos que sean clases asociativas, como las consideras en la regla anterior.

R4.

Una instancia c(I, A_1,….., A_n, T) de una clase es eliminada por una operación si en su postcondición incluye la expresión OCL:

`Cgen.allInstances() -> excludes(i)`

o la expresión:

`not i.oclIsTypeOf(Cgen)`

donde C_{gen} es o bien la clase C o una superclase de C.

R5.

Una instancia c(I, P_1, ….., P_n, A_1, ….., A_m, T) de una clase asociativa es eliminada por una operación si en su postcondición incluye la expresión:

`C.allInstances() -> excludes(i)`

o la expresión:

`not i.oclIsTypeOf(C)`

o si alguno de sus participantes (P_1, ….., P_n) es eliminado.

R6.

Una instancia r(C_1, C_2, T) de una asociación binaria R entre los objetos C_1 y C_2, con roles *role-c_1* y *role-c_2* en *r* es eliminada por una operación si en su postcondición incluye la expresión OCL:

`ci.role-cj -> excludes(cj)`

o si alguno de sus participantes (C_1, C_2) es eliminado.

De acuerdo con las reglas de traducción establecidas, el predicado derivado de la clase *Product* del esquema conceptual de la Figura 5, se representa de la siguiente manera:

Product(PR,ID,T) <- addProduct(PR,ID,T2) ^ ¬deletedProduct(PR,T2,T) ^ T2<=T ^ time(T) ^ time(T2)

deletedProduct(PR,T1,T2) <- delProduct(PR,T) ^ T>T1 ^ T<=T2 ^ time(T) ^ time(T2)

donde PR corresponde con el único OID requerido por cada instancia de una clase. ID es el atributo de la clase y T la variable que representa el instante de tiempo en que se crea una instancia.
addProduct y delProduct a su vez también son predicados derivados cuya definición depende de las operaciones del esquema de comportamiento, que crean o eliminan instancias de la clase *Product*.

addProduct(PR,ID,T) <- offerProduct(PR,ID,US,T) ∧ time(T)

La operación *offerProduct* crea una instancia de *Product* de acuerdo a la regla R1, ya que en su postcondición incluye la expresión:

```
Product.allInstances()->exists(p | p.oclIsNew() and p.id=idp ...)
```

Por otra parte, hemos de encontrar que operaciones son responsables de eliminar las instancias de *Product*, para poder especificar la regla de *delProduct.* Como vemos en el esquema de comportamiento Figura 5, la única operación que realiza dicha acción es *removeProduct,* ya que en su postcondición incluye la expresión:

```
Product.allInstances()->excludes(p)
```

por tanto la regla quedaría de la siguiente manera:

delProduct(PR,T) <- removeProduct(PR,T) ∧ Product(PR,ID,T_{pre}) ∧ T_{pre}=T-1 ∧ time(T)

Dado que la operación *removeProduct* recibe por parámetro una instancia de *Product*, hemos de garantizar que esta instancia fue creada en un instante de tiempo anterior a la ejecución de la operación, por lo que es añadida como una precondición a la regla de derivación.

4.2.3. Restricciones

Jerarquías

De acuerdo con las reglas establecidas anteriormente, cuando se crea una instancia de una superclase explícitamente por una postcondición, la traducción garantiza la creación de una única instancia en la superclase. Por el contrario cuando una instancia de una subclase es creada por una postcondición, la traducción no asegura la creación de una instancia en la superclase. Y en ningún caso es posible que una instancia pertenezca a una subclase y no a su superclase.

Para solucionar esto, se define una regla de derivación que indica que una instancia de una subclase implica la existencia de la misma instancia en la superclase.
En nuestro ejemplo:

```
User(US,N,T) <- Registered(US,T)
User(US,N,T) <- Unregistered(US,T)
```

Restricciones temporales

Se debe asegurar que dos acontecimientos no pueden ocurrir simultáneamente, para ello es necesario definir unas restricciones que garanticen que dos operaciones no ocurran en el mismo instante de tiempo. A diferencia de las reglas y restricciones anteriores, estas restricciones se expresan en forma de negación, que representan las condiciones que no pueden suceder en cualquier estado de la información base.

Para cada operación *O* con los parámetros P_1,.....,P_n , se definen las siguientes restricciones para cada parámetro P_j :

$$\leftarrow o(P_1 1,\ldots,P_n 1, T) \wedge o(P_1 2,\ldots P_n 2, T) \wedge P_i 1 <> P_i 2$$

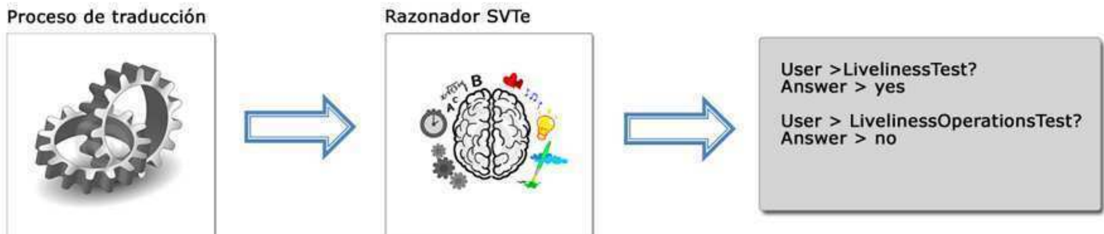
Y para cada par o, o2 de operaciones se definen las siguientes restricciones:

$$\leftarrow o(P_1,\ldots,P_n, T) \wedge o2(Q_1,\ldots,Q_m, T)$$

En nuestro ejemplo, para la operación *offerProduct*:

```
<- offerProduct(PR,ID,T) ^ offerProduct(PR2,ID2,T) ^ PR<>PR2
<- offerProduct(PR,ID,T) ^ offerProduct(PR2,ID2,T) ^ ID <> ID2

<- placeBid(BI,PR,US,T) ^ offerProduct(PR2,I,T)
<- deleteProduct(PR,T) ^ offerProduct(PR2,ID,T)
<- offerProduct(PR,ID,T) ^ registerUser(US2,T)
```

4.3.Validación del esquema



La validación de un esquema conceptual como bien se ha expuesto desde el inicio, ha tener en cuenta tanto la parte estructural (esquema y restricciones de integridad) como la parte del comportamiento (operaciones). Así pues, el primer paso es crear una traducción común entre las dos partes, que será nuestra base para todas las pruebas.

Para esta traducción común, se ha tenido que realizar una pequeña adaptación a la traducción realizada por Aurus del esquema estructural, ya que ahora los predicados de las clases y asociaciones son expresados en términos de una nueva variable que representa el tiempo, *time,* se puede ver un ejemplo de esta adaptación en el punto 5.3.3 y también en los anexos del ejemplo completo.

Las diferentes pruebas que realizaremos, se definen mediante el control de satisfactibilidad de un predicado derivado. Para lo cual, utilizaremos un razón lógico llamado SVTe implementado por Guillem Rull [3], es un programa externo y la única forma de comunicarnos con él, es mediante archivos de texto, así pues, la lógica generada anteriormente debe quedar plasmada en un fichero.

4.3.1. Satisfactibilidad de un esquema

Un esquema es muy satisfactible, si existe al menos un estado pleno de la información base que satisface todas las restricciones, es decir, si es posible tener una instancia de todas las clases y asociaciones del esquema.

Para realizar esta prueba, la regla formal es:

$$\texttt{Sat <- } \texttt{clase}_i\texttt{(X,…,T)}$$
$$\texttt{Sat <- } \texttt{asociación}_j\texttt{(X,…,T)}$$

En nuestro ejemplo:

```
Sat <- Product(P,I,T)
Sat <- User(U,N,T)
Sat <- Bid(B,P,U,T)
Sat <- Registered(U,T)
Sat <- Unregistered(U,T)
Sat <- OfferedBy(P,U,T)
```

La diferencia con respecto a la formalización de Aurus, es que al tener en cuenta el esquema de comportamiento, estamos limitando los casos en que una instancia es satisfactible, ya que ahora una instancia además de satisfacer las restricciones de integridad se ha de dar por ocurrencia de una operación. Por esta razón, aunque la parte estructural sea satisfactible, el esquema conceptual en conjunto no lo sea cuando se tienen en cuenta las operaciones.

4.3.2. Otros test

Además de las pruebas anteriores para determinar la satisfactibilidad de las clases y asociaciones, hay algunas pruebas que se pueden extraer de forma automática. Por ejemplo, se pueden realizar unas pruebas adicionales para determinar la aplicabilidad y viabilidad de cada operación.

Una operación es **aplicable** si hay un estado donde su precondición se cumple.
La formalización de una operación *O* con precondición pre(t) es:
Aplicable_O <- pre(T)

Una operación es **ejecutable** si puede ser ejecutada al menos una vez, es decir, si hay un estado donde su postcondición se cumple, junto con las restricciones de integridad y de manera que su precondición era también cierta en un estado anterior.
La formalización de una operación:
$ejecutable_O \leftarrow o(P_1, ..., P_n) \wedge pre(T-1)$

5. Herramienta

5.1.Especificación

En este capítulo se presenta la especificación de la herramienta, con el objetivo de dar una visión general, analizar cuáles son sus funcionalidades y la manera de interactuar con ellas.

5.1.1. Visión general

En este punto, tenemos una visión general completa de la herramienta, donde podemos identificar todos los elementos de los cuales hemos hablado en capítulos anteriores y además ver el flujo para llegar a la validación de un esquema conceptual UML con operaciones.

5.1.2. Modelo de casos de uso

Como podemos observar en la Figura 7, el único actor que contemplamos es el propio diseñador del esquema conceptual, ya que es el único tiene conocimiento sobre, qué es lo que se quiere expresar en el modelo y cuáles son los requisitos de los usuarios.

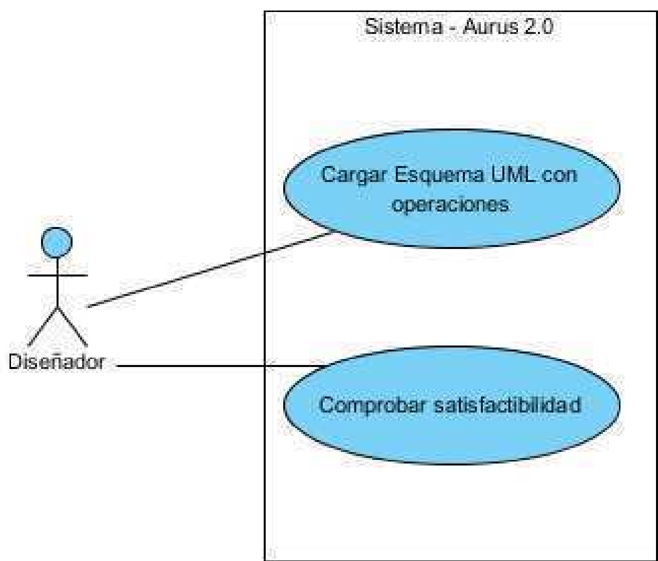


Figura 7 - Diagrama de casos de uso

Especificación del caso de uso: Cargar Esquema UML con operaciones	
Descripción	Carga un esquema conceptual UML con operaciones dentro de la herramienta.
Flujo normal de eventos	
1. El Diseñador introduce un modelo conceptual UML/OCL codificado en XMI 2. El Sistema responde si ha cargado correctamente el modelo	
Flujos alternativos	
1. El sistema informa al Diseñador que el modelo ha generado algún error al intentar cargarlo.	

<table>
<tr><td colspan="2">Especificación del caso de uso: Comprobar satisfactibilidad</td></tr>
<tr><td>Descripción</td><td>Proceso para realizar pruebas, puede darse de dos formas:
Pruebas generadas automáticamente para validar la satisfactibilidad del esquema, como por ejemplo las pruebas definidas en el punto 4.3.1 o pruebas manuales por medio de un predicado derivado.</td></tr>
<tr><td>Precondición</td><td>El caso de uso: Cargar Esquema UML con operaciones</td></tr>
<tr><td colspan="2">Flujo normal de eventos</td></tr>
<tr><td colspan="2">1. El Diseñador lanza una prueba automática o manual.
2. El Sistema responde si la prueba es satisfactible y muestra un ejemplo en caso afirmativo.</td></tr>
</table>

5.2.Diseño

5.2.1. Arquitectura de la herramienta

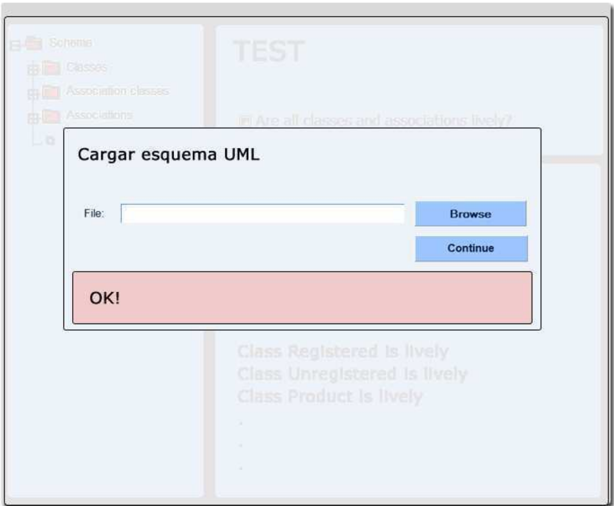
La herramienta se diseño bajo el patrón arquitectónico de tres capas. Teniendo en cuenta que la aplicación tiene una interfaz web es indispensable separar conceptos. Así, este patrón nos proporciona tres subsistemas diferenciados, Capa de presentación, Capa de dominio y Capa de datos.

5.2.2. Capa de presentación

La capa de presentación se encarga de la interacción con el usuario. Esta capa recibe los eventos generados por el usuario y le muestra la información que proviene del dominio.

En nuestro caso, los eventos equivalen al modelo de casos de uso descritos en el capítulo 5.1.2, que describen la forma de interactuar del diseñador con la herramienta. Dado que nuestro sistema tiene pocos casos de uso, el diseño de las pantallas es bastante simple. De hecho solo especificaremos una pantalla por cada caso de uso.

Pantalla: **Cargar Esquema UML con operaciones**

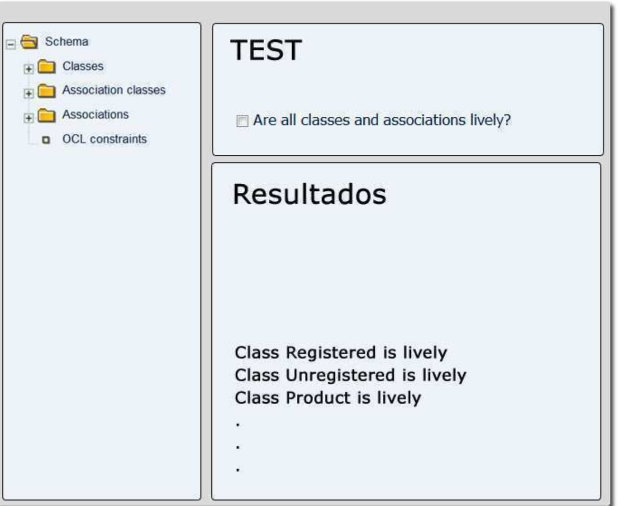


Esta pantalla contiene dos elementos, un formulario mediante el cual se cargará un archivo XMI y una zona de mensajes, que informará al usuario si se ha cargado correctamente el modelo.

Pantalla: **Comprobar satisfactibilidad**

Esta pantalla tiene tres elementos: Un árbol del esquema conceptual, es un árbol grafico donde se especifica toda la información del modelo conceptual, por ejemplo, las clases, asociaciones, operaciones del sistema, etc.

Zona de test, formulario donde se especifican los test automáticos que el diseñador puede realizar para evaluar su modelo conceptual. Y por último la zona de resultados, donde la herramienta informa cual fue el resultado del último test realizado.



5.2.3. Capa de dominio

La capa de dominio se encarga de gestionar el estado y las acciones del sistema. Recibe los eventos de la capa de presentación, y ejecuta las acciones pertinentes, procesando la información y si es necesario comunicándose con la capa de datos. Por último retorna los datos necesarios a la capa de presentación.

Puesto que la capa de dominio, es el núcleo de este proyecto, en el capítulo 5.3 se describirá a fondo la implementación del funcionamiento de la herramienta.

5.2.4. Capa de datos

La capa de datos, en nuestra implementación no es más que un gestor de ficheros. Ya que la herramienta no contempla almacenar ningún tipo de información después de terminar de validar un esquema conceptual. Así pues, el único momento donde se almacenará información, es en el proceso de razonamiento con SVTe, ya que como hemos comentado antes, la entrada de SVTe es a través de un fichero de texto que contiene la representación lógica del modelo. Por lo tanto, solo cuando se lance un evento de validación, nos comunicaremos con la capa de datos para crear un fichero *input.txt* el cual solo estará presente en el sistema mientras se realizan los procesos de validación. Acto seguido será eliminado.

5.3.Implementación

En este apartado se comenta como se estructura el sistema, cómo interactúan todos los mecanismos y cuáles fueron las soluciones aportadas para cumplir con los objetivos. Además, se hará una comparación entre la formalización de la traducción de las operaciones y cuál fue el resultado en la implementación.

5.3.1. Visión general

En la siguiente figura se pueden ver los componentes principales del sistema, que concuerdan con las fases propuestas en el capítulo 4. En los siguientes puntos se comentará de una forma más detallada cada clase y su respectiva función.

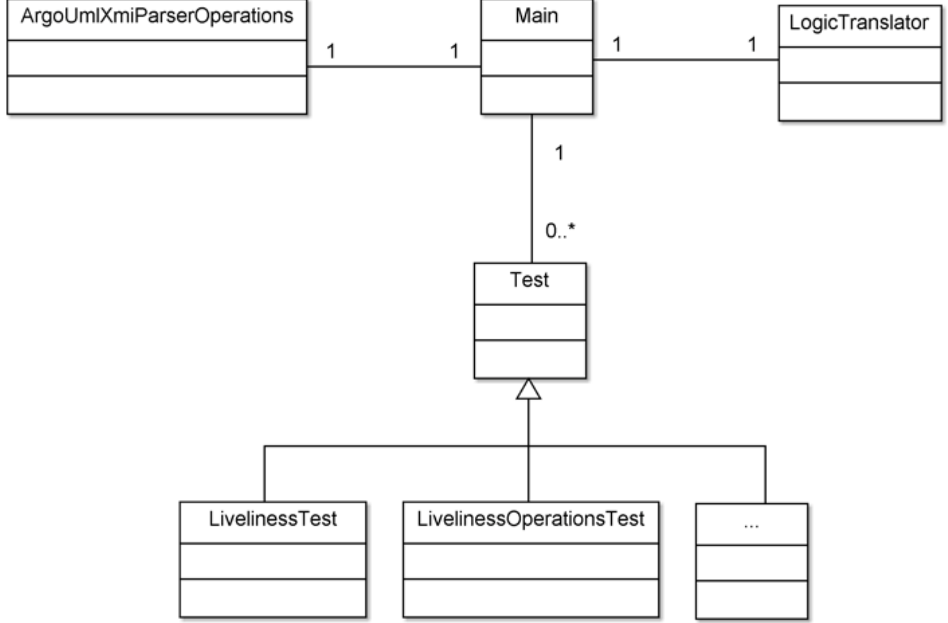


Figura 8 - Estructura básica del sistema

Por motivos de claridad, en los diagramas de clases que se presentarán en las demás páginas, hemos obviado algunos atributos y métodos, ya que solo son cuestiones técnicas concretas que no son relevantes para la explicación.

5.3.2. Carga de un esquema conceptual

Como hemos establecido en el punto 4.1.1, la entrada a nuestra herramienta será un fichero de texto, que contendrá un esquema conceptual UML con operaciones codificado en XMI, y el resultado de la ejecución de esta parte del código, será la instanciación del esquema con operaciones en los metamodelos de la librería EinaGMC.

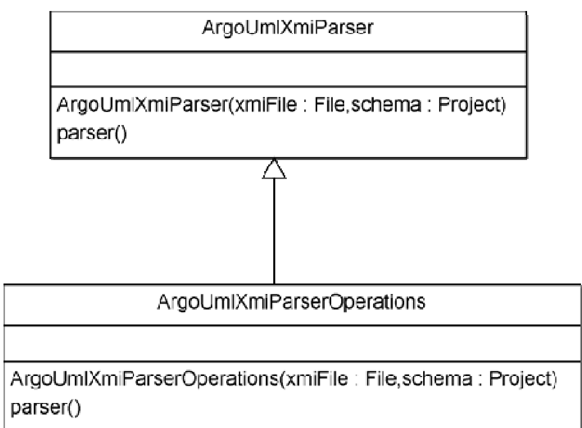


Figura 9 - Diagrama de clase ArgoUmlXmiParserOperations

Este es el fragmento del diagrama de clases que corresponde con la tarea de cargar un esquema dentro de la herramienta. Como se puede ver, para tener en cuenta las operaciones definidas en el esquema conceptual y no interferir con la implementación de la clase *ArgoUmlXmiParser* se ha creado una especialización

ArgoUmlXmiParserOperations recibe dos parámetros: un ***xmiFile*** de tipo *File* que especifica el fichero XMI y un ***schema*** de tipo *Project* que es el formato que utiliza EinaGMC para sus modelos, contiene las interfaces para acceder a los metamodelos de UML y OCL.

El funcionamiento de este proceso, comienza con la lectura del fichero *xmiFile* y la identificación de cada clase, asociación, herencia, operación, etc. En lugar de leer e instanciar directamente en el *schema*, se utilizan unas clases intermedias que representan la estructura de XMI, de esta forma se traslada la estructura XMI del fichero a las clases intermedias. Esto nos ayuda agilizar el proceso, por ejemplo, en XMI una asociación guarda sus participantes como referencias, esto nos obligaría a volver a buscar dentro del fichero de texto a que participante hace referencia, con las clases intermedias esta acción es inmediata, todo esto aporta eficiencia y modularidad.

Una vez trasladado el esquema XMI, solo hay que invocar los métodos de la librería de la EinaGMC [5] e instanciar el modelo en el parámetro *schema*.

Todos los elementos de un diagrama de clases, clases, asociaciones, herencias etc, son instanciados en el metamodelo de UML y las restricciones de integridad escritas en OCL son instanciadas en el metamodelo de OCL (tipo *constraint* de la librería).

Las operaciones forman parte de ambos metamodelos, es decir, la cabecera que contiene nombre de la operación y cada parámetro con su tipo, es instanciado en el metamodelo de UML y las pre - postcondiciones deberían formar parte del metamodelo de OCL. Sin embargo, este fue uno de los primero obstáculos que nos encontramos, el metamodelo de OCL que implementa la librería EinaGMC no soporta la operación *oclIsNew()* en las postcondiciones, y está claro que es una de las formas más habituales de crear una instancia en OCl, además es una de las reglas de identificación descritas en el método de la tesis. Después de hablarlo detenidamente, la conclusión fue guardar las postcondiciones como *String,* ya que otra opción conllevaría a un retraso en la implementación de la herramienta. Tratar las postcondiciones como *String* no es ningún problema, basta que coincida con un patrón de identificación de las reglas.

Así pues, tanto la pre como la postcondición de una operación son guardadas como restricciones (tipo *constraint* de la librería), pero la pre es instanciada en el metamodelo de OCL y la post simplemente guardada como *String*.

5.3.3. Traducción de operaciones OCL a lógica

La implementación de las reglas de traducción para las operaciones, descritas en el capítulo 4.2 se pueden encontrar en la clase *LogicOperationsTranslator,* sin embargo, como también necesitamos la traducción del esquema estructural, veremos todos los elementos necesarios para realizar la traducción completa.

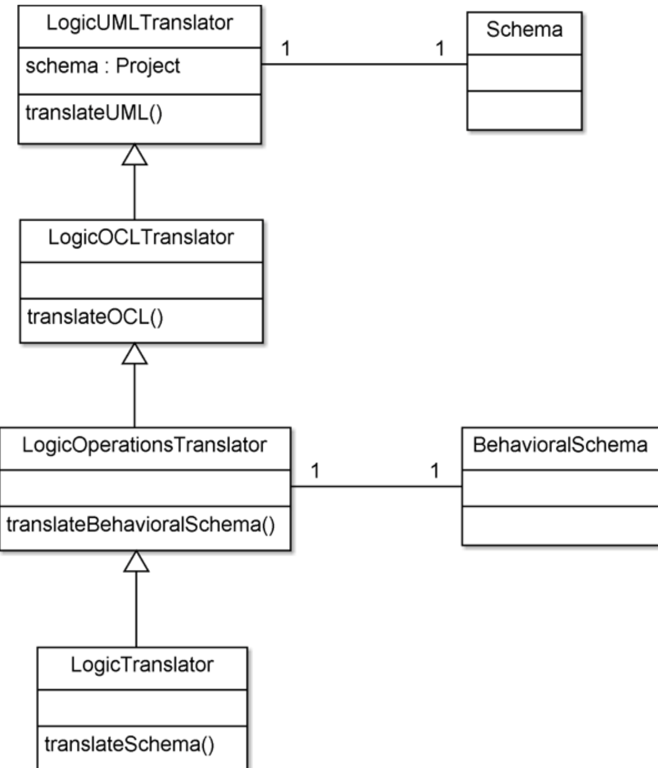


Figura 10 - Diagrama de clases para la traducción

Los traductores necesitan acceder a los elementos del metamodelo UML/OCL en el cual esta instanciado el esquema conceptual, para efectuar las reglas de traducción pertinentes. Como podemos observar en la superclase *LogicUMLTranslator,* tiene un parámetro ***schema*** que es el esquema instanciado resultado del apartado anterior.

La clase ***Schema*** representa la lógica generada de la traducción del esquema estructural, y para mantener una independencia en la implementación, la lógica generada a partir del esquema de comportamiento se instancia en un modelo diferente, ***BehavioralSchema***.

La especificación de los métodos de consulta para acceder a los elementos de los metamodelos UML/OCL no se especificarán en esta memoria, pero se pueden encontrar en [5]. Sin embargo quiero puntualizar en unos de los métodos de consulta proporcionados por la librería.

`Public List<Constraint> getAllConstraints()` : Retorna todas las restricciones contenidas en el esquema conceptual importado en el proyecto.

Este método es utilizado por Aurus para obtener las restricciones de integridad (invariantes), dado que es lo único que nos proporciona la librería EinaGMC. Sin embargo este método como bien lo específica retorna todas las restricciones, no concreta que sean restricciones de integridad. Pues bien, nuestras pre y postcondiciones de las operaciones al ser también restricciones (*constraint*) generaba un conflicto, cuando Aurus quería tratas los invariantes también obtenía las restricciones de las operaciones y por tanto la ejecución generaba un error.

La solución pasaba por crear un nuevo método que retornara solo los invariantes, ya que la operación anterior cumplía con su especificación. Así bien, después de una reunión con el director de este pfc Ernest Teniente y con uno de los implementadores de la librería Albert Tort Pugibet, se decidió que sería enriquecedor para mi implementar este nuevo método, aunque no fuera algo que albergara gran dificultad implicaría conocer mejor la librería EinaGMC, tanto en los metamodelos como en la implementación.

`Public List<Constraint> getAllInvariants()` : Retorna todos los invariantes contenidos en el esquema conceptual importado en el proyecto.

Debido a esta solución, hubo que modificar la implementación de Aurus, solo tuve que substituir todos los *`getAllConstraints()`* por *`getAllInvariants()`*.

5.3.3.1. Metamodelo lógico

Retomando la Figura 10, la representación de la lógica generada del esquema de comportamiento se instancia en ***BehavioralSchema***, que es una implementación del mismo metamodelo lógico que utiliza Aurus para la representación del esquema estructural.

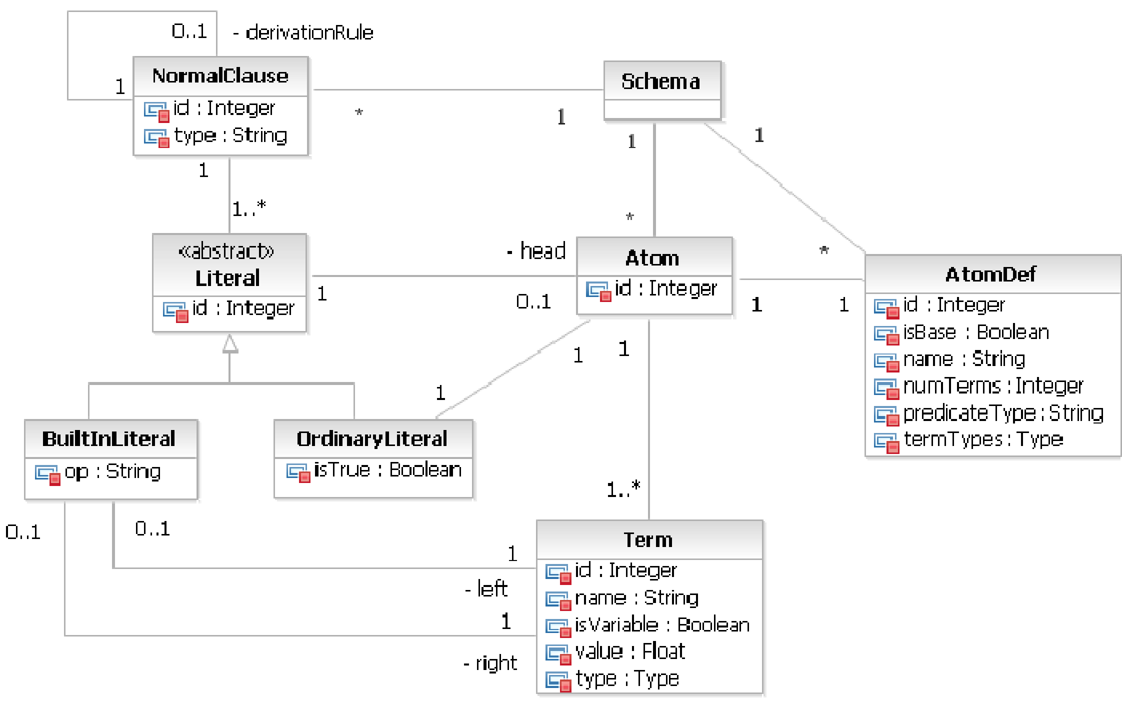


Figura 11 - Diagrama de clases del metamodelo lógico

Este es el fragmento de diagrama de clases que corresponde a la explicación de los conceptos del capítulo 2.3.

La clase AtomDef implementa el concepto de predicado, podemos destacar el campo *predicateType* el cual nos permite especificar si estamos representando una clase, asociación, operación, etc.

La jerarquía Literal, implementa los dos tipos de literales que podemos tener. Podemos diferenciar si un Literal Ordinal es negativo gracias al campo *isTrue,* y saber el operador de un Literal Built-In gracias al campo *op.*

5.3.3.2. *Lógica generada*

Por lo que respecta a la traducción y resultado del esquema de comportamiento a lógica, es importante destacar los cambios que sufrió la traducción final con respecto a las reglas formales.

Predicado instancia

Uno de los cambios significativos es, cambiar la aparición del predicado de una instancia en una regla de derivación, por su significado en términos de *addInstance, deletedInstance.*

Traducción formal:

```
delProduct(PR,T) <- removeProduct(PR,T) ∧ Product(PR,ID,Tpre) ∧ Tpre=T-1
∧ time(T)
```

Implementación:

```
delProduct(PR,T) :- removeProduct(PR,T), addProduct(PR,ID, Tpre),
not(deletedProduct(PR, Tpre,T)), Tpre=T-1, time(T)
```

Tiempo acotado inferiormente

Las variables de tiempo tienen que ser limitadas a valores positivos, porque si existe una precondición, T tiende a disminuir, y si no existe ninguna cota inferior el proceso puede no terminar. Por tanto, todas las apariciones de variables que representen el tiempo, tienen que tener una limitación del tipo $T_i > 0$.

Traducción formal:

```
Product(PR,ID,T) <- addProduct(PR,ID,T2) ^ ¬deletedProduct(PR,T2,T) ^
T2<=T ^ time(T) ^ time(T2)
```

Implementación:

```
Product(PR,ID,T) :- addProduct(PR,ID,T2), time(T), time(T2), T>0,
T2>0, T2<=T, not(deletedProduct(PR,T2,T))
```

Precondición Tpre

El razonador lógico SVTe, no permite resolver Tpre = T-1, por esa razón hemos de reemplazar esa expresión por otra con significado igual. Me tomo el atrevimiento de cambiar Tpre por T2, es una cuestión de implementación y no afecta el significado.
Ahora tenemos T2 = T-1, lo cambiamos por T2<T, dado que T-1 significa en un instante de tiempo anterior, el nuevo literal sigue manteniendo ese significado y por tanto es correcto.

Traducción formal:

```
delProduct(PR,T) <- removeProduct(PR,T) ∧ Product(PR,ID,Tpre) ∧ Tpre=T-1
∧ time(T)
```

Implementación:

```
delProduct(PR,T) :- removeProduct(PR,T), addProduct(PR,ID,T2),
time(T), T>0, time(T2), T2>0, T2<T, not(deletedProduct(PR,T2,T))
```

Predicados para los atributos

En la tesis, se hace una simplificación de las reglas de derivación para cada predicado de cada atributo de una clase. Ejemplo, una clase con sus atributos, se representaba de la siguiente manera:

```
employee(E)
employeeName(E,Name)
employeeSalary(E,Salary)
```

con la simplificación:

```
employee(E,Name,Salary,T)
```

Puesto que hemos de tener en cuenta todos estos aspectos, en la implementación no se utiliza ninguna simplificación. Sin embargo, para la validación de la parte comportamiento de un esquema conceptual, los atributos no influyen, si tenemos en cuenta que lo único que puede interferir en la integridad de un estado del sistema, son las instancias de los objetos en sí.

Si tenemos en cuenta todas las modificaciones antes mencionadas, la clase *Product* de nuestro ejemplo, queda representada por las siguientes reglas de derivación:

```
Product(PR,T) :- addProduct(PR,ID,T2), time(T), time(T2), T>0, T2>0,
T2<=T, not(deletedProduct(PR,T2,T))

ProductId(PR,ID,T) :- addProduct(PR,ID,T2), time(T), time(T2), T>0,
T2>0, T2<=T, not(deletedProduct(PR,T2,T))

addProduct(PR,ID,T) :- offerProduct(PR,ID,US,T), time(T), T>0

delProduct(PR,T) :- removeProduct(PR,T), addProduct(PR,ID,T1),
time(T), T>0, time(T1), T1>0, T1<T, not(deletedProduct(PR,T1,T))

deletedProduct(PR,T1,T2) :- delProduct(PR,T), time(T), time(T1),
time(T2), T>0, T1>0, T2>0, T>T1, T<=T2
```

5.3.4. Validación del esquema

Esta es la última etapa que nos permitirá razonar con la lógica generada a partir del esquema estructural y de comportamiento. Podemos entender este proceso como una comunicación, es decir, nosotros como usuarios le haremos preguntas al razonador SVTe (en función de predicados derivados) y este nos dirá si es posible construir un estado que satisfaga todas las restricciones.

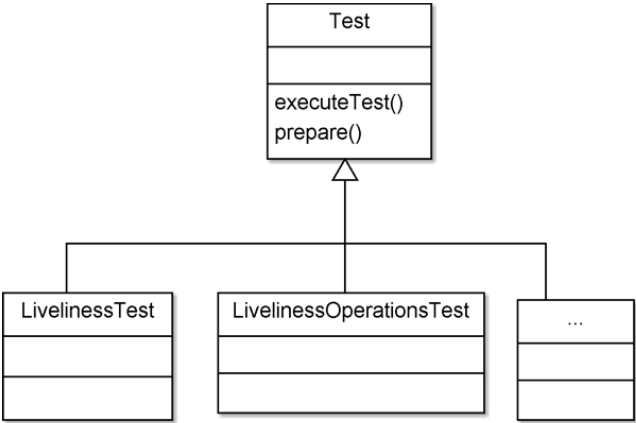


Figura 12 - Diagrama de clases correspondiente a los Test

Este es el diagrama de clases que corresponde con los tests automáticos que se generan para la validación de un esquema conceptual UML con operaciones. Podemos apreciar dos subclases *LivelinessTest* y *LivelinessOperationsTest* , que corresponden con las validaciones del esquema estructural y el esquema de comportamiento respectivamente.

LivelinessOperationsTest es el test de validación explicado en el capitulo 4.3.1 y además existen una serie de subclases que corresponden a otros tests, unos definidos para el esquema estructural y otros que podrían ser los explicados en el capitulo 4.3.2, pero que no tratamos en este pfc.

El proceso de ejecución de los test se divide en dos etapas, que corresponden con los métodos de la clase *Test.(prepare, executeTest)*

5.3.4.1. *prepare*

Este método se encarga básicamente de dos cosas. Crear el predicado derivado (Goal) del que se quiere evaluar su satisfactibilidad. Y además, generar un archivo conjunto con toda la lógica, tanto del esquema estructural como del esquema de comportamiento.

La lógica generada por Aurus del esquema estructural, sufre un ligero cambio debido a la presencia de las operaciones, ahora tiene que ser expresada en términos de una nueva variable, que corresponde al tiempo en que se crea una instancia. Por ejemplo, una de las restricciones del esquema estructural es:

```
:- OfferedBy(P,U), not(Product(P))
```

"No puede ser que exista la asociación OfferedBy entre un Product P y un User U, y no exista la instancia de Product P"

La misma restricción tiene que ser expresa en términos la variable *T.*

```
:- OfferedBy(P,U,T), not(Product(P,T))
```

De esta forma, nos aseguramos que las instancias de clases, asociaciones, etc. son creadas por la intervención de una operación en un cierto instante de tiempo T.
Así pues, el método *prepare* genera un archivo de texto con toda la lógica y con el predicado derivado que se quiere evaluar, dicho archivo será la entrada al razonador lógico SVTe.

5.3.4.2. *executeTest*

El razonador SVTe es un software externo [3] que tiene por objetivo la construcción de un estado que cumpla una meta (goal) y satisfaga todas las restricciones establecidas. SVTe requiere como parámetro de entrada un archivo de texto con todas las restricciones lógicas, reglas de derivación y un predicado goal a satisfacer. El resultado de la ejecución, es un archivo de texto donde indica o bien un estado que satisface todas las restricciones o bien que restricciones se han violado y no se pueden reparar.

Dicho esto, la ejecución del método *executeTest* pone en marcha el razonador indicando los parámetros de entrada antes mencionados y recupera el resultado. De esta forma, una vez completados los tests que se quieran evaluar, finalmente cumplimos con el objetivo final de evaluar la (in)corrección de un esquema conceptual UML con operaciones.

En el capítulo 6, se presentará la ejecución completa de la herramienta, donde podremos razonar sobre los resultados de los tests, y los cambios que necesitaría el esquema de comportamiento para ser satisfactible.

5.3.5. Problemas de validación

Me gustaría destacar que en la fase de validación hemos tenido varios inconvenientes, y las posibles alternativas que surgieron para dar solución a dichos problemas.

SVTe instanciaba las variables con números reales, por tanto uno de los primero problemas surgieron debido a que expresamos las precondiciones como un valor anterior a un cierto tiempo T, el razonador podía intentar construir un estado de forma indefinida.

$T_{pre} < T$, por tanto la solución fue, a todos los tiempos añadir una restricción ($T_{pre} > 0$, $T > 0$) y modificar SVTe para trabajar con numero enteros.

Otro inconveniente se daba con la formalización siguiente:

```
delProduct(PR,T) :- removeProduct(PR,T), addProduct(PR,ID,T1),
time(T), T>0, time(T1), T1>0, T1<T, not(deletedProduct(PR,T1,T))

deletedProduct(PR,T1,T2) :- delProduct(PR,T), time(T), time(T1),
time(T2), T>0, T1>0, T2>0, T>T1, T<=T2
```

Podemos ver que existe una recursividad en las reglas de derivación, dado que ***deletedProduct*** invoca a ***delProduct*** y esta a su vez, aun que sea a través de un literal negativo vuelve a verificar ***deletedProduct*** .

En primera instancia, el SVTe estaba implementado con unas operaciones recursivas que no admitían este tipo de formalización, ya que las llamadas recursivas eran tantas que sobrepasaban el límite de la pila de procesos, y ocasionaba la terminación inesperada de la ejecución. El problema fue solucionado cambiando la implementación de las operaciones recursivas, pasándolas a iterativas.

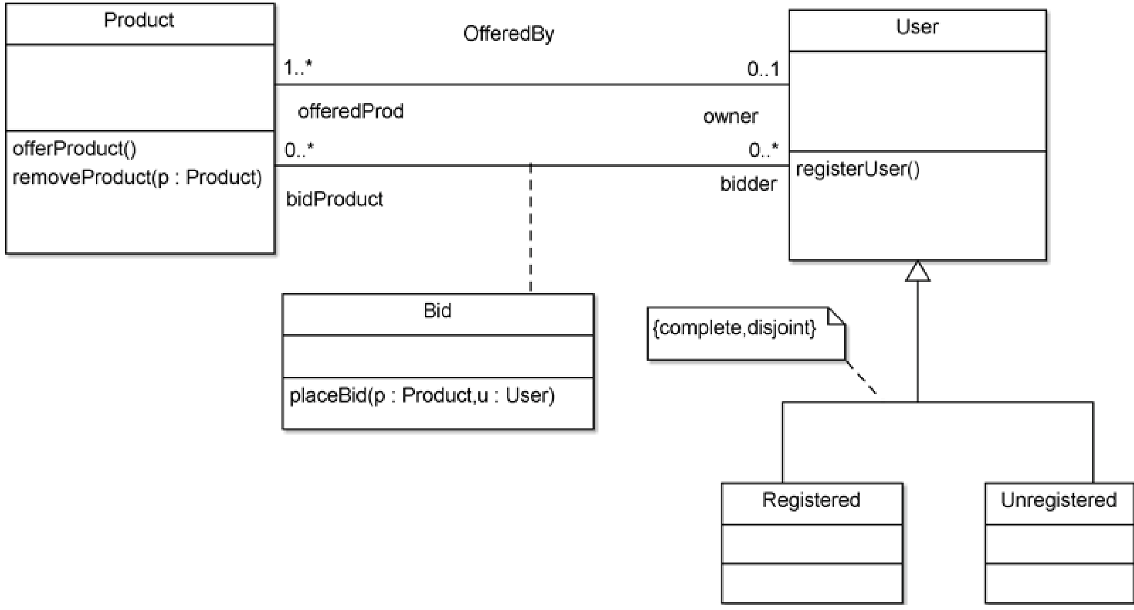
Por último, en el siguiente caso:

```
@5 :- OfferedBy(PR,US,T), not(Product(PR,T))

OfferedBy(PR,US,T) :- addOfferedBy(PR,US,T2), time(T), time(T2), T>0,
T2>0, T2<=T, not(deletedOfferedBy(PR,US,T2,T))
```

Cuando se violaba la restricción 5, el SVTe intentaba reparar la restricción y entraba en una recursividad que no terminaba. Al ver que no se encontraba el origen del problema, se planteo cambiar la formalización a B-Schema, que básicamente se trata en convertir todas las reglas de derivación a restricciones, ya que esta solución fue probada manualmente y funcionaba. Sin embargo el problema fue solucionado modificando el SVTe ya que no estaba realizando bien la tarea de reparación de las restricciones violadas.

6. Ejemplo

A continuación, veremos la ejecución completa de la herramienta para el mismo esquema que presentamos en el capítulo 3. Se mostrara la traducción tanto del esquema estructural como de comportamiento, además, el resultado del test (ver capitulo 4.3.1) que comprueba si es posible tener una instancia de todas las clases y asociaciones del esquema.




```
Op: registerUser()
Pre:
Post: Registered.allInstances()-> exists(u | u.oclIsNew())


Op: placeBid(p: Product, u: User)
Pre: u.oclIsTypeOf(Registered)
Post: Bid.allInstances()-> exists(b | b.oclIsNew() and b.bidder = u and
b.bidProduct = p)


Op: offerProduct()
Pre:
Post: Product.allInstances()->exists(p| p.oclIsNew())


Op: removeProduct(p: Product)
Pre:
Post: Product.allInstances()->excludes(p)
```

A modo de aclaración para las siguientes traducciones, las reglas sin *head* (restricciones) van precedidas por un identificador del estilo *@1* , esto lo utiliza el razonador para identificar las reglas y si el resultado de una ejecución es un identificador, quiere decir que la restricción identificada por ese número es violada.

Para las clases o asociaciones que no son creadas por el acontecimiento de una operación, se crea un predicado derivado que siempre sea falso, de esta forma nos aseguramos que esta instancia nunca puede ser creada.

6.1.Traducción del esquema estructural

```
%+ Classes and associations
%  Registered(R)
%  Unregistered(U)
%  Product(P)
%  User(U)
%  OfferedBy(P1,U2)
%  Bid(B1,P2,U3)


%+ Constraints for OIDs

@0 :- Bid(B0,P0,US0,T), Bid(B0,P1,US1,T), P0<>P1
@1 :- Bid(B0,P0,US0,T), Bid(B0,P1,US1,T), US0<>US1
@2 :- Product(X,T), User(X,T)
@3 :- Product(X,T), Bid(X,P0,US0,T)
@4 :- User(X,T), Bid(X,P0,US0,T)


%+ Constraints for hierarchies

@12 :- Registered(X,T), not(User(X,T))
@13 :- Unregistered(X,T), not(User(X,T))

IsKindOfUser(US0,T) :- Unregistered(US0,T)
IsKindOfUser(US0,T) :- Registered(US0,T)
@14 :- User(US0,T), not(IsKindOfUser(US0,T))
@15 :- User(US0,T), not(IsKindOfUser(US0,T))
%+ Constraints for associations

@5 :- OfferedBy(P0,US0,T), not(Product(P0,T))
@6 :- OfferedBy(P0,US0,T), not(User(US0,T))
@9 :- Bid(OID,P0,US0,T), not(Product(P0,T))
@10 :- Bid(OID,P0,US0,T), not(User(US0,T))
@11 :- Bid(OID1,P0,US0,T), Bid(OID2,P0,US0,T), OID1<>OID2


%+ Constraints for association cardinalities

MinOfferedProd(US0,T) :- OfferedBy(P0,US0,T)
@7 :- User(US0,T), not(MinOfferedProd(US0,T))
@8 :- OfferedBy(P0,US0,T), OfferedBy(P0,US1,T), US1<>US0
```

6.2.Traducción del esquema de comportamiento

```
%+ Operaciones

% placeBid(BI,PR,US,T)
% removeProduct(PR,T)
% offerProduct(PR,T)
% registerUser(US,T)

%+ Clases y asociaciones

%+ OfferedBy

OfferedBy(PR,US,T) :- false


%+ Bid

Bid(BI,PR,US,T) :- addBid(BI,PR,US,T2), time(T), time(T2), T>0, T2>0,
      T2<=T, not(deletedBid(BI,PR,US,T2,T))

addBid(BI,PR,US,T) :- placeBid(BI,PR,US,T), addProduct(PR,T1),
      addRegistered(US,T2), time(T), T>0, time(T1), T1>0, T1<T,
time(T2), T2>0, T2<T, T2<>T1, not(deletedProduct(PR,T1,T))

deletedBid(BI,PR,US,T1,T2) :- delBid(BI,PR,US,T), time(T), time(T1),
      time(T2), T>0, T1>0, T2>0, T>T1, T<=T2

delBid(BI,PR,US,T) :- delProduct(PR,T), addBid(BI,PR,US,T2), time(T),
      time(T2), T>0, T2>0, T2<T, not(deletedBid(BI,PR,US,T2,T))


%+ Product

Product(PR,T) :- addProduct(PR,T2), time(T), time(T2), T>0, T2>0,
      T2<=T, not(deletedProduct(PR,T2,T))

addProduct(PR,T) :- offerProduct(PR,T), time(T), T>0

delProduct(PR,T) :- removeProduct(PR,T), addProduct(PR,T1), time(T),
      T>0, time(T1), T1>0, T1<T, not(deletedProduct(PR,T1,T))

deletedProduct(PR,T1,T2) :- delProduct(PR,T), time(T), time(T1),
      time(T2), T>0, T1>0, T2>0, T>T1, T<=T2


%+ User
%+ ---
```

```
%+ Registered

Registered(US,T) :- addRegistered(US,T2), time(T), time(T2), T>0,
      T2>0, T2<=T

addRegistered(US,T) :- registerUser(US,T), time(T), T>0


%+ Unregistered

Unregistered(US,T) :- false


%+ Restricciones para herencias

  User(US,T) :- Registered(US,T)
  User(US,T) :- Unregistered(US,T)


%+ Restricciones generales

  @18 :- placeBid(BI,PR,US,T), placeBid(BI2,PR2,US2,T), BI<>BI2
  @19 :- placeBid(BI,PR,US,T), placeBid(BI2,PR2,US2,T), PR<>PR2
  @20 :- placeBid(BI,PR,US,T), placeBid(BI2,PR2,US2,T), US<>US2
  @21 :- removeProduct(PR,T), removeProduct(PR2,T), PR<>PR2
  @22 :- offerProduct(PR,T), offerProduct(PR2,T), PR<>PR2
  @23 :- registerUser(US,T), registerUser(US2,T), US<>US2
  @24 :- placeBid(BI,PR,US,T), removeProduct(PR2,T)
  @25 :- placeBid(BI,PR,US,T), offerProduct(PR2,T)
  @26 :- placeBid(BI,PR,US,T), registerUser(US2,T)
  @27 :- removeProduct(PR,T), offerProduct(PR2,T)
  @28 :- removeProduct(PR,T), registerUser(US2,T)
  @29 :- offerProduct(PR,T), registerUser(US2,T)
```

6.3.Test y conclusiones

Las pruebas que realizaremos, son para determinar si es posible tener una instancia de todas las clases y asociaciones del esquema. A continuación, se mostrará cada predicado que se quiere satisfacer y el resultado del razonador.

Goal	SVTe
`Sat <- Product(P,T)`	`time(1)` `offerProduct(0,1)`
`Sat <- User(U,T)`	**@7**
`Sat <- Registered(U,T)`	**@7**
`Sat <- Unregistered(U,T)`	`No satisfactible`
`Sat <- Bid(B,P,U,T)`	**@7**
`Sat <- OfferedBy(P,U,T)`	`No satisfactible`

En primer lugar, podemos observar de entrada que, crear una instancia de *Unregistered* y *OfferedBy* no es satisfactible, ya que no existe ninguna operación que pueda crear dichas instancias. Efectivamente, si comprobamos los contratos de las operaciones, podemos darnos cuenta del error que se cometió en la especificación.

Otro error que llama la atención, son las instancias de *User*, *Registered* y *Bid* , todas violan la restricción identificada con el número 7.

```
@7 :- User(US0,T), not(MinOfferedProd(US0,T))
MinOfferedProd(US0,T) :- OfferedBy(P0,US0,T)
```

"No puede ser que exista una instancia de *User* US0 y no exista la asociación *OfferedBy* entre US0 y un *Product* P0"

La restricción @7 hace referencia a la multiplicidad (1..*) de la asociación *OfferedBy*, si se da de alta un *User* obliga a crear o asociar dicho usuario con un *Product.*

En vista de lo anterior, una posible solución es modificar la operación *registerUser* de la siguiente manera:

```
Op: registerUser(p:Product)
Pre:
Post: Registered.allInstances()-> exists(u | u.oclIsNew() and
u.offeredProduct=p)
```

Y para solucionar la creación de una instancia de *Unregistered*, crear una nueva operación del estilo:

```
Op: unregisterUser(u:Usuario)
Pre: u.oclIsTypeOf(Registered)
Post: u.oclIsTypeOf(Unregistered) and not u.oclIsTypeOf(Registered)
```

7. Conclusiones

7.1.Sobre la herramienta y trabajos futuros

A pesar de los contratiempos, el resultado en general a cumplido con los objetivos propuestos. Ahora la herramienta permite importar de forma automática un esquema conceptual UML con operaciones y realizar su posterior validación. Sin embargo, creo que es importante comentar, que debido a los problemas con el razonador lógico SVTe, no se pudo realizar un conjunto de pruebas más amplio que hubiera ayudado a la detección de posibles anomalías.

Por otro lado, dada la complejidad con la que se pueden especificar los modelos UML, y teniendo en cuenta que, actualmente el test de validación solo se centra en la satisfactibilidad de las instancias de cada clase y asociación, una de las posibles ampliaciones a la herramienta, es la creación de nuevos test automáticos basados en el esquema en concreto, que ayudaran al diseñador a detectar situaciones potencialmente indeseables.

7.2.Valoración del proyecto

Después de unos meses de trabajo, puedo decir que la experiencia ha sido gratificante. Es un proyecto que desde el primer día me despertó fascinación, ya que es una herramienta realmente útil para cualquier diseñador. Gracias a todo eso, no solo he puesto en práctica los conocimientos adquiridos durante la carrera, sino que además me ha servido para comprender la complejidad que conlleva un proyecto de este tipo, y como saber adaptarme a las situaciones no previstas. Sin lugar a dudas, ver en la práctica como se utiliza la lógica para resolver problemas, me ha parecido muy interesante y del que seguro podre sacar provecho de una manera u otra en el futuro.

8. Referencias y bibliografía

[1] Validation of UML conceptual schemas with OCL constraints and operations
Anna Queralt, director Ernest Teniente
http://upcommons.upc.edu/handle/123456789/195216

[2] Aurus
Guillermo Lubary Fleta, directora: Anna Queralt
Proyecto Final de Carrera, FIB - UPC
http://upcommons.upc.edu/handle/123456789/143394

[3] Razonador lógico SVTe
SVTe - A Tool to Validate Database Schemas giving Explanations - Guillem Rull, Carles Farré, Ernest Teniente, Toni Urpí.

[4] EinaGMC Project
Grupo de trabajo GMC (Modelització Conceptual) de la UPC y la UOC
http://guifre.lsi.upc.edu/eina_GMC

[5] EinaGMC API specification
Grupo de trabajo GMC (Modelització Conceptual) de la UPC y la UOC
http://guifre.lsi.upc.edu/eina_GMC/resources_eina/core/doc/index.html

[6] UML 2.0 specification
OMG (Object Management Group)
http://www.omg.org/spec/UML/2.0/

[7] OCL 2.0 specification
OMG (Object Management Group)
http://www.omg.org/spec/OCL/2.0/

[8] Validación de esquemas de bases de datos SQL Server
Carlos Beltrán; director: Carles Farré
Proyecto Final de Carrera, FIB-UPC

[9] Reasoning on UML Conceptual Schemas with Operations
Anna Queralt, Ernest Teniente