UNIVERSIDAD
COMPLUTENSE
MADRID

Proyecto de Innovación

Curso 2016/2017

Proyecto Nº 49

Validación en línea de aspectos formales y corrección asistida de ejercicios en asignaturas con evaluación continua

Responsable: Manuel Montenegro Montes

Facultad de Informática

Departamento de Sistemas Informáticos y Computación

1. Objetivos propuestos en la presentación del proyecto

El objetivo de este proyecto es el desarrollo de una serie de herramientas que permitan a un profesor agilizar la tarea de corrección de ejercicios y prácticas entregadas en formato electrónico. Este proyecto se enmarca en un contexto de evaluación continua para ciertas asignaturas de los grados de la Facultad de Informática. En particular, aquellas asignaturas donde las prácticas y ejercicios alcanzan una complejidad demasiado elevada como para ser procesadas y comprobadas de modo automático con ayuda de jueces.

La idea clave de las herramientas desarrolladas se basa en el hecho de que la corrección de prácticas es más llevadera cuando todas las entregas de una misma práctica presentan un aspecto homogéneo en cuanto a su contenido y estructura. Para ello, el sistema propuesto en el proyecto comprende la especificación, por parte del profesor, de una serie de requisitos formales que ha de cumplir la entrega de un ejercicio. Por tanto, el primer objetivo era:

[H1] Desarrollo de una herramienta de creación de especificaciones, destinada al profesor. Esta herramienta sirve para indicar los requisitos que han de cumplir las entregas de un ejercicio, en cuanto al formato de las mismas. Por ejemplo, el profesor puede exigir una estructura de directorios determinada, o la existencia de ciertas líneas en el contenido de uno de los ficheros entregados.

Este objetivo tenía como requisito previo la concepción de un formalismo para especificar las entregas. Este también se ha llevado a cabo durante el transcurso del proyecto.

Cuanto más detallados son los requisitos impuestos, existirá mayor homogeneidad en el conjunto de entregas realizadas por los estudiantes. Sin embargo, si se incluyen demasiadas restricciones en una entrega, o estas alcanzan una cierta complejidad, resulta difícil para el estudiante tener en cuenta todas estas restricciones, por lo que es probable que algunas de ellas resulten inadvertidas para algunos de los estudiantes. Para ello se establece el segundo objetivo del proyecto:

[H2] Verificador automático de aspectos formales. Los usuarios de esta herramienta serán los estudiantes. Tiene como objetivo comprobar que el ejercicio a entregar por el estudiante respeta las especificaciones de entrega que el profesor ha creado mediante la herramienta [H1].

De este modo, cada alumno ejecuta en su máquina la herramienta [H2] para conocer, antes de entregar un ejercicio, si este se adecua a los requisitos especificados por el profesor. Los requisitos no satisfechos por el ejercicio serán mostrados al estudiante para que este último pueda modificar su entrega conforme a los mismos.

Cuando el profesor recibe todas las entregas, las procesa mediante una tercera herramienta, que ocupa el tercer objetivo del proyecto:

[H3] Asistente de corrección. Los usuarios de esta herramienta serán, de nuevo, los profesores. Una vez que cada estudiante ha verificado la entrega del ejercicio mediante la herramienta [H2], el profesor tiene la certeza de que todas las entregas

tienen un mismo formato. Esto le permitirá procesar las entregas de una manera uniforme, realizando por lotes las operaciones requeridas en la corrección, es decir, el profesor realizará una operación en cada una de las entregas, y esta será replicada automáticamente en el resto. Entre las operaciones contempladas están la creación de ficheros, compilación de prácticas, ejecución de casos de prueba, etc. Otra de las funcionalidades de la herramienta consiste en la extracción y recolección de fragmentos de cada una de las entregas para agilizar su corrección posterior.

Esta última herramienta permite al profesor realizar las tareas rutinarias de corrección de manera automática. Por ejemplo, en el ámbito de ejercicios que consistan en el desarrollo de un determinado programa, estas tareas incluyen la compilación del programa, la ejecución de sus casos de prueba, configuración del entorno previo a la ejecución del programa, puesta en marcha de las bases de datos requeridas por el mismo, etc.

2. Objetivos alcanzados

Todos los objetivos expuestos en la sección anterior han sido conseguidos conforme a la planificación contemplada en la solicitud del proyecto. Todas las herramientas han sido desarrolladas y liberadas bajo una licencia GPL. Puede encontrarse su código fuente en `https://github.com/PoVALE`.

[H1] Herramienta de generación de especificaciones.

- Como paso previo se dispone de un lenguaje de especificación de requisitos basado en el formalismo de la lógica de primer orden.
- Se dispone de una representación de este lenguaje de formalización en formato XML, de manera que pueda ser procesado fácilmente por las herramientas restantes (`PoVALE-reader`)
- El lenguaje es extensible mediante añadidos (plugins), de modo que un usuario puede añadir símbolos de función y predicado nuevos.
- Se han desarrollado dos plugins. Uno de ellos (`PoVALE-plugin-files`) incorpora funcionalidad para manejar ficheros: lectura, listado de ficheros dentro de un directorio, etc. El otro plugin (`PoVALE-plugin-lines`) permite acceder al contenido textual de un fichero e imponer restricciones sobre el mismo. En particular, se permite realizar búsquedas mediante expresiones regulares.
- Basándose en esta arquitectura, hemos desarrollado la herramienta [H1] mediante la cual un profesor puede generar de manera interactiva y mediante una interfaz gráfica documentos XML con especificaciones de requisitos (Figura 2-1).

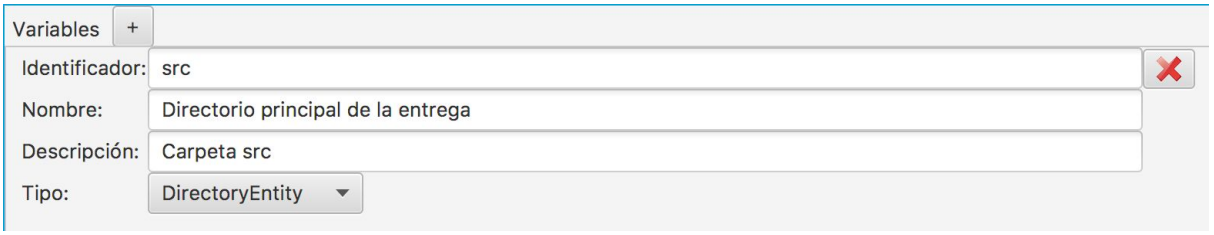

Figura 2-1. Herramienta de generación de especificaciones de requisitos

[H2] Verificador automático de aspectos formales.

- Se ha diseñado una representación (en el lenguaje Java) del formalismo de especificación de requisitos. Esta representación se construye a partir del fichero XML proporcionado por el profesor.
- El fichero de especificación, además de contener los requisitos que ha de cumplir la entrega de un ejercicio, permite incorporar variables cuyos valores serán proporcionados por el estudiante en el momento de verificar su entrega. Por ejemplo, puede solicitarse el nombre de los estudiantes, o el número de grupo al que pertenecen. Los valores de estas variables pueden formar parte de los requisitos, de modo que, por ejemplo, puede exigirse que

el ejercicio entregado contenga una carpeta cuyo nombre coincide con el DNI del estudiante.

- Una vez que el estudiante ha proporcionado la información de estas variables, se inicia el proceso de validación, indicando si los requisitos han sido cumplidos o no. En ambos casos, se proporciona información detallada sobre los requisitos que han sido satisfechos y los que no lo han sido (Figura 2-2).

Figura 2-2. Herramienta de validación de requisitos

[H3] Asistente de corrección

- Se ha diseñado un lenguaje de proceso por lotes que permite utilizar el resultado de una entrega validada mediante el lenguaje de especificación de requisitos.
- Al igual que el lenguaje de especificación, el lenguaje de proceso por lotes es extensible mediante plugins.
- Se ha diseñado una herramienta de proceso por lotes que, a partir de una secuencia de comandos representada en el lenguaje de proceso por lotes, genera un script que puede ejecutarse en cada una de las entregas proporcionadas por los estudiantes.

3. Metodología empleada en el proyecto

El desarrollo de un proyecto ha requerido un proceso de selección previa de tecnologías y herramientas a utilizar, así como un proceso de familiarización con estas. El plan de trabajo ha sido el siguiente:

Tarea 1. Diseño de lenguaje de especificación de requisitos formales de entrega (herramientas [H1], [H2], [H3])

Para ello se han recopilado enunciados de ejercicios y prácticas de años anteriores para poder analizar las restricciones más comunes que se plantean en la entrega de un ejercicio. A partir de ahí se ha diseñado un lenguaje de especificación de restricciones.

Tarea 2. Diseño de la arquitectura del sistema (herramientas [H1], [H2], [H3])

Se ha diseñado una arquitectura genérica en Java que permite cargar añadidos dinámicamente. Para ello ha sido necesario identificar las partes extensibles del lenguaje de requisitos: tipos de entidad, símbolos de función, predicado y acciones.

Tarea 3. Verificador de requisitos de entrega (herramienta [H2])

Esta tarea se ha llevado a cabo en dos subsistemas. En el proyecto `PoVALE-core` se ha implementado el subsistema de validación. Basándose en este, hemos diseñado la interfaz gráfica del mismo (`PoVALE-view`) en Java, haciendo hincapié en la generación de mensajes informativos para aquellas restricciones de la especificación no satisfechas.

Tarea 4. Generador interactivo de especificaciones de requisitos (herramienta [H1])

Una vez definido el lenguaje de especificación y su representación en XML, se ha diseñado una aplicación en Java que implementa la herramienta [H1]. El código fuente de esta herramienta se encuentra en el proyecto `PoVALE-specification`.

Tarea 5. Descripción del lenguaje de proceso por lotes (herramienta [H3])

Para el desarrollo de esta tarea se han analizado las acciones más frecuentes que realizan los profesores a la hora de procesar las entregas por los alumnos: compilación, ejecución de tests y extracción de fragmentos de código. A partir de ahí se ha definido la sintaxis del lenguaje de proceso por lotes.

Tarea 6. Desarrollo de aplicación para manipular por lotes y extraer información de entregas (herramienta [H3])

Se ha extendido el sistema de añadidos para incorporar acciones dentro de las entidades, y se ha desarrollado una aplicación que permite generar un script sobre un conjunto de entregas de ejercicios.

4. Recursos humanos

En esta sección se describe la contribución de cada uno de los integrantes del proyecto al mismo.

- **Laura Hernando**: Ha implementado las clases básicas que implementan los asertos y términos del lenguaje de especificación. Utilizando dichas clases ha implementado tanto la interfaz como la lógica interna del validador de ejercicios (herramienta [H2]). También ha diseñado junto con Daniel Rossetto la interfaz y la lógica de la herramienta generadora de especificaciones [H1].

- **Daniel Rossetto**: Ha desarrollado la representación XML del lenguaje de especificación de requisitos e implementado la lectura del mismo dentro del proyecto PoVALE-reader. Con respecto a la herramienta comprobadora [H2], ha implementado la parte referente al manejo de variables externas del lenguaje de especificación. También ha colaborado en la implementación del generador de especificaciones.

- **Santiago Saavedra:** Ha participado en el desarrollo de la arquitectura global del sistema, y gestionado los distintos proyectos dentro de GitHub. También ha participado en la implementación del proceso de carga de los distintos plugins, y del uso de las funciones y predicados importados en los mismos dentro de las herramientas [H1], [H2] y [H3]. Ha codirigido junto con Manuel Montenegro el Trabajo de fin de Grado *Asistente de Corrección y Validación de ejercicios*.

- **Manuel Montenegro**: Además de realizar las labores de coordinación necesarias en el proyecto, ha codirigido junto con Santiago Saavedra el Trabajo de fin de Grado *Asistente de Corrección y Validación de ejercicios* de los estudiantes Laura Hernando y Daniel Rossetto. Ha participado la arquitectura general de las tres herramientas objeto de este proyecto, y ha desarrollado los *plugins* relativos al manejo de ficheros y al manejo de líneas dentro de un fichero para su uso dentro de las herramientas [H1], [H2] y [H3]. También ha implementado una versión preeliminar de la herramienta [H3].

- **Clara Segura**: Ha diseñado, junto con Fernando Rosa, el lenguaje de especificación de requisitos, inspirándose en los requisitos extraídos de las prácticas de las asignaturas *Estructuras de Datos y Algoritmos* y *Fundamentos de Programación*. Además, ha colaborado en el desarrollo del lenguaje de proceso por lotes.

- **Fernando Rosa**: Ha diseñado el lenguaje de proceso por lotes, basándose en el proceso de corrección de las prácticas de la asignatura *Fundamentos de Programación* de las titulaciones impartidas en la Facultad de Informática. También ha participado, junto con Clara Segura, en la creación del lenguaje de especificación de requisitos.

5. Desarrollo de las actividades

El grueso de las actividades de implementación dentro de este trabajo se ha realizado en el contexto del Trabajo de Fin de Grado titulado *Asistente de Corrección y Validación de Ejercicios*, realizado por Laura Hernando Serrano y Daniel Rossetto Bermejo, siendo Manuel Montenegro y Santiago Saavedra los codirectores de dicho trabajo. Este trabajo se ha presentado durante el mes de junio de 2017, obteniendo una calificación de 10 (Sobresaliente).

Antes del comienzo de este Trabajo de Fin de Grado, se esbozó durante el mes de septiembre de 2016 el lenguaje de especificación de requisitos. Para ello nos basamos en los requisitos de prácticas de las asignaturas de *Fundamentos de Programación*, *Estructuras de Datos y Algoritmos* y *Tecnología de Programación* correspondientes a cursos pasados. Se constató que gran parte de las restricciones que podían imponerse a las prácticas de dichas asignaturas son expresables mediante el formalismo de la Lógica de Primer Orden. Por ello se incorporaron al lenguaje de especificación los distintos elementos de este formalismo (términos, asertos y entidades) con el objetivo de definir la sintaxis y semántica del lenguaje de manera más precisa. En el Anexo 6.1 se encuentra una descripción del lenguaje de especificación de requisitos.

Con el lenguaje de especificación ya creado, dio comienzo el TFG mencionado anteriormente. Como primera tarea se abordó la representación de este formalismo de especificación en una jerarquía de clases en el lenguaje de programación Java para su uso desde los programas implementados en dicho lenguaje. A partir de ahí, se implementó la lógica de validación que permitía, dado un objeto entregable, y una lista de restricciones expresadas mediante la jerarquía de clases mencionada previamente, comprobar si el objeto satisfacía dichas restricciones. Tanto la lógica de validación como la jerarquía de clases se encuentran implementadas en el siguiente repositorio:

`https://github.com/PoVALE/PoVALE-core`

Paralelamente a esto, se desarrolló el sistema de añadidos (*plugins*) y se integró con la lógica de validación. A modo de ejemplo, se desarrollaron dos plugins:

- `PoVALE-plugin-files`: Permite especificar restricciones sobre los ficheros y directorios contenidos en la entrega de un ejercicio, sin entrar en su contenido. Por ejemplo, mediante este plugin puede imponerse que el fichero entregado tenga extensión `.xml`, o que el fichero entregado sea un directorio que no contenga ficheros cuyo nombre comience por `backup`. La implementación de este plugin se encuentra en `https://github.com/PoVALE/PoVALE-plugin-files`.
- `PoVALE-plugin-lines`: Permite especificar restricciones sobre el contenido de un fichero, suponiendo que este contenido sea de tipo textual. En particular, permite especificar la existencia de lineas dentro de un fichero de texto que contengan una determinada cadena. Esto resulta útil para exigir, por ejemplo, que el fichero entregado contenga el nombre de los estudiantes que la realizan. Tanto la documentación como la implementación de este plugin se encuentran en el repositorio `https://github.com/PoVALE/PoVALE-plugin-files`.

Una vez desarrollada la arquitectura básica basada en plugins y la lógica de comprobación, se ejemplificaron diversos casos de uso correspondientes a asignaturas impartidas en la Facultad de Informática. Concretamente, se especificaron las restricciones exigidas a las entregas de algunos ejercicios de asignaturas impartidas anteriormente y se ejecutaron pruebas sobre entregas para comprobar si el validador las aceptaba o rechazaba.

Durante esta fase de pruebas surgió la necesidad, no prevista inicialmente, de extender el lenguaje de especificación con variables externas, cuyos valores serían proporcionados por los estudiantes durante su entrega y serían tenidos en cuenta durante el proceso de validación. Por ejemplo, supongamos que en una asignatura tenemos distintos grupos de prácticas numerados, y queremos que, a la hora de entregar un determinado ejercicio, cada grupo de prácticas realice su entrega mediante un fichero cuyo nombre contenga el número del grupo. En este caso, el número de grupo es una variable externa cuyo valor deberá ser introducido por el estudiante que realice la entrega. La herramienta validadora se encargará de comprobar que el nombre del fichero entregado comprende el número introducido por el estudiante. Se llevó a cabo la incorporación de este tipo de variables a la lógica del validador previamente implementada.

Una vez implementada la lógica interna del sistema de validación, el siguiente paso consistió en la interacción con el estudiante. En primer lugar, el estudiante debe recibir un fichero que contenga la descripción de los requisitos para un determinado ejercicio. Este fichero es no es más que una representación en XML del formalismo de especificación de requisitos. En el Anexo 6.2 se describe dicha representación. El proyecto `PoVALE-reader` es una librería que permite al programador cargar un fichero XML y convertir su contenido a objetos expresados en la jerarquía de clases implementada previamente en `PoVALE-core`, para su posterior validación. El código fuente de este proyecto puede obtenerse en la siguiente dirección:

`https://github.com/PoVALE/PoVALE-reader`

La implementación realizada en el proyecto `PoVALE-reader` sirvió como base para realizar la interfaz gráfica de la herramienta validadora [H2]. En esta fase surgió la necesidad de abordar el problema de generar mensajes de error legibles para el usuario. Resulta deseable que cuando la entrega del estudiante no satisface los requisitos impuestos por el profesor, se muestre al estudiante un mensaje informativo indicando exactamente dónde está el problema. Se ha puesto especial cuidado en esta parte, diseñando métodos para generar mensajes de error automáticamente a partir de restricciones. No obstante, y con el fin de mantener la flexibilidad, el profesor puede especificar sus propios mensajes de error en el fichero de especificación, que pueden dar información más precisa de la que se generaría automáticamente.

Una vez terminada la herramienta [H2] se hicieron pruebas con algunas especificaciones de ejemplo, obteniendo resultados satisfactorios. La implementación de dicha herramienta se encuentra bajo el siguiente proyecto:

`https://github.com/PoVALE/PoVALE-view`

Las acciones necesarias para el desarrollo de la herramienta [H2] provocaron la consolidación de la arquitectura básica del sistema y del lenguaje de especificación de requisitos. Esto propició que el desarrollo de las siguientes fases se hiciese a un ritmo más elevado. El siguiente paso fue el desarrollo de la herramienta [H1] de generación de especificación de requisitos. Este paso, pese a ser de realización laboriosa, no introdujo ninguna complicación conceptual ni técnica posterior, ya que se trataba de una aplicación independiente destinada a funcionar en una arquitectura ya asentada. Puede encontrarse en el siguiente repositorio:

`https://github.com/PoVALE/PoVALE-specification`

El último paso consistió en el desarrollo de la herramienta [H3]. En este caso se trata de poder leer los ficheros de una entrega y realizar diversas acciones sobre ellos: compilación, ejecución de casos de prueba, obtención de los resultados de dichos test, etc. Para ello se ha desarrollado un sencillo lenguaje de secuencias de comandos (*scripts*) mediante extensión del lenguaje de especificaciones definido previamente añadiendo la posibilidad de añadir acciones. Por ejemplo, dado un ejercicio cuyas entregas contienen varios programas escritos en Java, un posible script consistiría en recorrer todos los ficheros de la entrega con extensión `.java` y ejecutar el compilador de Java sobre cada uno de ellos.

Partiendo de este lenguaje de scripts pasó a implementarse la herramienta [H3], que se encarga de ejecutar un script dado en todas las entregas de un mismo ejercicio. En una versión inicial, esta herramienta genera un fichero de proceso por lotes en cada una de las entregas generadas para su ejecución mediante el sistema *Bash*. Una vez descrito previamente el lenguaje de guiones, la traducción de dicho lenguaje a ficheros de proceso por lotes no introdujo ninguna dificultad reseñable.

6. Anexos

6.1 Lenguaje de especificación de requisitos de entrega

El objetivo de este anexo es describir el lenguaje que utilizaremos para expresar las restricciones de entrega que una tarea debe cumplir. El principal objetivo es disponer de un lenguaje muy genérico, pero que sea extensible mediante plugins. En esta sección se presenta la estructura del lenguaje, sin tener en cuenta aspectos de representación del mismo mediante XML.

El lenguaje se asemeja bastante a la lógica de primer orden, pero esto es previsible, dado el tipo de restricciones que queremos capturar. Incluimos varios tipos de elementos: entidades, colecciones, funciones, términos, relaciones y asertos.

Entidades

Una entidad es algo sobre lo que se pueden poner restricciones. Por ejemplo:

- Un fichero es una entidad, porque podemos exigir que tenga un nombre determinado, una extensión, o un tipo de contenido.
- Una clase de Java también es una entidad. Podemos exigir que la clase se llame de una determinada manera, o que tenga un atributo dado.
- Los atributos y métodos de Java también son entidades. Incluso, si queremos, las expresiones y sentencias de Java también pueden ser consideradas entidades. De este modo, podríamos exigir que un determinado método no tenga dos bucles while anidados, por ejemplo.
- Una línea de un fichero de texto también es una entidad. Podemos pedir, por ejemplo, que entre las diez primeras líneas de un fichero, una de ellas contenga el nombre del estudiante.
- Un elemento XML también es una entidad. Se puede exigir, que sea de una determinada etiqueta, que contenga o no contenga un determinado atributo, etc.

Por tanto, tenemos distintos tipos de entidad (ficheros, carpetas, clases, métodos, atributos, etc.). La idea es que sólo haya unos cuantos tipos de entidad que vengan incluidos *de serie*, como por ejemplo los ficheros y carpetas, y que mediante plugins puedan extenderse el conjunto de tipos de entidad soportados. Por ejemplo, el plugin de Java puede proporcionar las entidades clase, atributo y método.

Además de los ficheros y carpetas, supondremos que algunos tipos de entidad vienen *de serie*; estos son los enteros y cadenas de texto. Esto resultará útil para realizar comparaciones. También suponemos que existe una operación de igualdad entre dos entidades del mismo tipo.

Colecciones

Una colección es un conjunto de entidades, del mismo o de distinto tipo. Por ejemplo, si queremos imponer que una determinada carpeta contenga ficheros con la extensión `.cpp`, tendremos que hacer referencia a la colección de ficheros que tiene esta carpeta, para poder decir que cada uno de ellos tiene extensión `.cpp`.

Funciones

Una función recibe varias entidades o colecciones como parámetros y produce una entidad o colección como resultado. Por ejemplo, podemos definir una función `name` que reciba una entidad de tipo fichero como parámetro y devuelva una entidad de tipo cadena con el nombre de ese fichero. Otra posibilidad es tener una función llamada `files` que reciba una entidad de tipo carpeta y devuelva una colección de entidades de tipo fichero. Al igual que los tipos de entidad, es recomendable tener un conjunto de funciones básicas incluidas de serie y poder ampliar el conjunto de funciones mediante plugins. Por ejemplo, el plugin de Java puede introducir las funciones `attributes` y `methods`, que devuelvan, respectivamente, la lista de atributos y métodos de la clase pasada como parámetro.

Términos

Los términos sirven para denotar una entidad. Se corresponden con la categoría de términos en lógica de primer orden. Es decir, incluyen variables, aplicaciones de símbolos de función, etc. La sintaxis de los términos viene dada por la siguiente gramática:

$$t ::= ROOT \mid c \mid X \mid f(t_1, ..., t_n) \mid [t_1, ..., t_n] \mid [t \mid q_1, ..., q_n]$$

$$q ::= X \in t \mid \varphi$$

donde:

- $ROOT$ es una variable especial que denota la entidad que el estudiante entrega. Por ejemplo, si en un ejercicio el usuario ha de entregar un fichero de C++, la variable ROOT contendrá ese fichero. Si ha de entregar una carpeta con varias clases, la variable ROOT hará referencia a la carpeta.
- c hace referencia a literales, que pueden ser enteros o cadenas.
- X es una variable. En un determinado momento, esta variable hará referencia a una entidad. Las variables se introducen mediante cuantificadores o expresiones let (ver la sección de asertos).
- $f(t_1, ..., t_n)$ es la aplicación de la función f a los términos $t_1, ..., t_n$.
- $[t_1, ..., t_n]$ es la colección formada por los términos $t_1, ..., t_n$.
- $[t \mid q_1, ..., q_n]$ denota una lista intensional, al estilo de las de Haskell. Cada q_i es un generador ($X \in t$) o un filtro φ. Los filtros están definidos en las categorías de asertos.

Predicados (o relaciones)

Se corresponde con la misma noción en lógica de primer orden. Una relación puede satisfacerse o no según el valor de los argumentos que reciba. Por ejemplo, podemos disponer de una relación `is-interface?` que reciba una entidad de tipo clase y determine si es una interfaz de Java o no. Al igual que en las funciones, supondremos que el conjunto de predicados y relaciones es extensible mediante plugins.

Asertos

Un aserto representa una restricción sobre una o varias entidades. Son el pilar principal sobre el que construir los requisitos de entrega. El conjunto de asertos viene definido por la siguiente gramática:

$$\varphi ::= \quad true \mid false \mid P(t_1, ..., t_n) \mid t_1 = t_2 \mid \varphi_1 \wedge ... \wedge \varphi_n \mid \varphi_1 \vee ... \vee \varphi_n \mid \neg\varphi$$

$$\mid \quad \varphi_1 \rightarrow \varphi_2 \mid \forall X \in t.\varphi \mid \exists X \in t.\varphi \mid \exists !X \in t.\varphi \mid let\ X = t\ in\ \varphi$$

donde:

- $P(t_1, ..., t_n)$ es la aplicación de un símbolo de predicado.
- Las fórmulas cuantificadas introducen variables en ámbito, pero estas variables toman valores dentro de una colección dada por el término t. El cuantificador $\exists !$ denota existencia y unicidad.
- $let\ X = t\ in\ \varphi$ sirve para introducir definiciones en ámbito.

Ejemplos

Supongamos una práctica de FP en la que los estudiantes han de entregar un único fichero .cpp. En este caso la variable ROOT hace referencia al fichero que entrega el estudiante. Lo siguiente son algunos ejemplos de restricciones:

- El fichero entregado tiene extensión `.cpp` y función llamada buscaagenda:

$$extension(ROOT) = "cpp" \ \wedge \ \exists !X \in functions(ROOT).\ name(X) = "buscaagenda"$$

- No existen definiciones de variables globales en el fichero entregado

$$\neg\exists X \in variables(ROOT).true$$

- El fichero tiene un comentario de línea con el texto “Nombre: “ seguido del nombre del alumno.

$$\exists L \in lines(ROOT).matches(L, "//.*Nombre:.*")$$

En este caso suponemos que la función `extension` devuelve la extensión de un fichero. Un hipotético plugin para C++ introducirá las funciones `functions` y `variables` que devuelven las declaraciones de función y variable, respectivamente, de un fichero `cpp`. Otro añadido para manejo de ficheros de texto plano puede introducir la función `lines` que devuelve la colección de líneas de un fichero, y la relación `matches` que se satisface si una determinada cadena ajusta con una expresión regular.

En la asignatura de TP los estudiantes hacen entregas más complejas, consistentes en un fichero `.zip` con un proyecto de *Eclipse*. Restricciones de ejemplo:

- El fichero contiene una carpeta `src`.
  $\exists !S \in files(ROOT)\ .\ isdirectory?(S) \wedge name(S) = "src"$

- Dentro de la carpeta `src` solamente puede haber ficheros con extensión `java`.
  $\exists !S \in files(ROOT)\ .\ isdirectory?(S) \wedge name(S) = "src" \wedge$
  $\forall F \in filesrec(S)\ .\ extension(F) = "java"$

- Dentro de la carpeta `src` existe, al menos, un fichero que implementa la interfaz `Observable`.
  $\exists !S \in files(ROOT)\ .\ isdirectory?(S) \wedge name(S) = "src" \wedge$
  $\exists F \in filesrec(S)\ .\ extension(F) = "java" \wedge$
  $\exists C \in classes(F).\ \exists D \in implements(C)\ .name(D) = "Observable"$

- Existe un fichero llamado `Vista.java` que no contiene atributos cuyo tipo es alguna de las clases del proyecto que implementa `Observable`.
  $\exists !S \in files(ROOT)\ .\ isdirectory?(S) \wedge name(S) = "src"$
  $\wedge\ \exists !F \in filesrec(S)\ .\ name(F) = "Vista.java"$
  $\wedge\ let\ M = [name(C) \mid F \in filesrec(S),\ C \in classes(F),$
  $\exists D \in implements(C).name(D) = "Observable"]$
  $in\ \exists C \in classes(F).\ \forall A \in fields(C)\ .\forall T \in M.\neg(type(A) = T)$

En estos ejemplos supondremos que un plugin de `Java` introduciría las funciones `implements`, `classes`, `fields`, etc.

6.2 Representación XML del lenguaje de requisitos

En esta sección describimos cómo se expresan los elementos que componen nuestro lenguaje de especificación, detallado en la sección anterior, mediante XML. La especificación de un ejercicio se encuentra entre las etiquetas `<spec></spec>`, que se compone de plugins, variables, asertos y términos. En la memoria del Trabajo de Fin de Grado *Asistente de corrección y Validación de Ejercicios* puede encontrarse una descripción más detallada de esta representación.

Plugins
Cada *plugin* se encuentra entre las etiquetas `<import></import>`, y en el interior de esas etiquetas debe encontrarse el nombre (incluyendo paquete) de la clase principal del plugin. Por ejemplo: `<import>es.ucm.povaleFiles.FilesPlugin</import>`

Variables
El valor de estas variables está indicado por alumno. Las variables se encuentran entre las etiquetas `<var></var>`:

```
<var>
     // Descripción corta de variable
      <label>Identificador</label>

      // Nombre de variable
      <name>nombre_grupo</name>

      // Descripción larga de variable
      <desc>Formato: LXXGXX </desc>

      // Tipo de entidad contenida en
      // la variable
      <type>StringEntity</type>
</var>
```

Términos

Aparecen siempre en el interior de algunos asertos. Según el tipo de término, su representación en XML es:

- **Cadenas:** `<literalString>cadena de texto</literalString>`
- **Enteros:** `<literalInteger>45</literalInteger>`
- **Variables:** `<variable>x</variable>`
- **Lista de términos:**

  ```
  <listTerm>
        <literalInteger>13</literalInteger>
        <literalInteger>42</literalInteger>
        <literalInteger>24</literalInteger>
  </listTerm>
  ```

- **Aplicaciones de símbolos de función**:
  Se representan mediante la etiqueta `<functionApplication>`:
  ```
  <functionApplication>
        <function>extension</function>
        <variable>x</variable>
  </functionApplication>
  ```

Asertos

Todos los asertos se encuentran bajo una única etiqueta: `<assertion>`, que contiene una lista de asertos, donde cada uno de ellos se encuentra bajo la etiqueta: `<assert>`.

Cada aserto tiene un mensaje por defecto que se muestra durante la validación, aunque si se desea utilizar un mensaje distinto es posible indicarlo mediante el atributo `msg =”descripcion del requisito”`.

Por ejemplo si queremos modificar el mensaje correspondiente una verdad lógica:
`<assertTrue msg="Siempre se cumple este aserto"></assertTrue>`

A continuación mostraremos cómo se representa cada aserto:

- **Verdad y falsedad lógica**:
  `<assertTrue/>` y `<assertFalse/>`

- **Negación:**
```
<not>
      ...
</not>
```

- **Igualdad:**
```
<equals>
       <lhs>
           <variable>x</variable>
       </lhs>
       <rhs>
           <literalInteger>1</literalInteger>
       </rhs>
</equals>
```

- **Conjunción (AND) y disyunción (OR):**
```
<and>
      aserto1
      aserto2
      ...
</and>

<or>
      aserto1
      aserto2
      ...
</or>
```

- **Implicación:**
```
<entail>
```

```
      <lhs>
           aserto1
       </lhs>
       <rhs>
           aserto2
       </rhs>
</entail>
```

- **Cuantificadores universal, existencial y existencial único:**

```
<exist>
      <variable>X</variable>
      termino
      aserto
</exist>
```

Similarmente se utilizaría `<forall>` y `<existsOne>`

- **Aplicaciones de símbolo de predicado:**

```
<predicateApplication>
      <predicate>is-directory?</predicate>
       <variable>x</variable>
</predicateApplication>
```