Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

Automatización de la Corrección de Prácticas de Programación a través del Compilador Clang

Pedro Delgado-Pérez
Departamento de Ingeniería Informática
Universidad de Cádiz
11519 Puerto Real, Cádiz
pedro.delgado@uca.es

Inmaculada Medina-Bulo
Departamento de Ingeniería Informática
Universidad de Cádiz
11519 Puerto Real, Cádiz
inmaculada.medina@uca.es

Resumen

El proceso de corrección de ejercicios para la enseñanza de lenguajes de programación se ha realizado tradicionalmente de forma manual. En los últimos años se ha avanzado en este aspecto hacia la automatización de la evaluación de las entregas de los alumnos, lo cual viene a mejorar tanto el aprendizaje del alumno como las tareas del profesor. No obstante, uno de los mayores inconvenientes para lograr este objetivo es la limitación en el análisis sintáctico del código para comprobar el cumplimiento de ciertos requisitos en el mismo. Este trabajo muestra cómo el uso de *Clang*, un compilador maduro de código abierto, puede proporcionar una capacidad de análisis superior mediante el recorrido por el árbol de sintaxis abstracta. Además, se plantea el uso de la prueba de mutaciones como técnica que conciencie a los alumnos de la importancia de diseñar un conjunto completo de casos de prueba para probar sus programas.

Abstract

The process of correction of exercises for the teaching of programming languages has been traditionally accomplished in a manual way. In recent years, several advances have been made regarding this aspect towards the automation of the evaluation of the solutions provided by the students. This support aims to improve both the learning of students and the work of lecturers. However, one of the main drawbacks to achieve this goal is the limitation in the static analysis of the code to check the fulfilment of certain requirements. This paper shows how using *Clang*, a full-fledged open-source compiler, can provide a greater capacity for analysis through the traversal of the abstract syntax tree. Moreover, the technique known as mutation testing is suggested to make students aware of the importance of designing an adequate test suite to test their applications.

Palabras clave

Programación, C++, corrección de prácticas, árbol de sintaxis abstracta, compilador, prueba de mutaciones.

1. Introducción

El aprendizaje de un lenguaje de programación es una tarea difícil que requiere de la realización de una gran diversidad de ejercicios para su dominio. Este proceso se vuelve más exigente cuando hablamos de lenguajes de propósito general, como lo son C o C++. Sin embargo, la corrección de ejercicios no es un proceso que habitualmente se haga de forma efectiva. De parte del profesor, se vuelve tedioso analizar la solución de cada uno de los alumnos, lo cual se suele realizar de manera manual y visual. Desde el punto de vista del alumno, la realización de ejercicios puede no ser una experiencia provechosa si no dispone de una corrección de los mismos, pudiendo perder la motivación pues no siempre se le indican los fallos expresamente.

Los ejercicios de programación pueden ser comprobados a partir de un conjunto de casos de prueba que valide que el programa creado tiene el comportamiento esperado [4]. No obstante, hay una serie de requerimientos que no pueden ser verificados de esta manera y que, principalmente en asignaturas de iniciación a la programación, deberían ser evaluados, como puede ser un buen estilo de programación o que la solución se ajuste a ciertas especificaciones planteadas en el ejercicio a resolver [11]. Por ejemplo, comprobar si los atributos de una clase se inicializan en la lista de inicialización de los constructores de una clase en C++. Estos requerimientos vienen determinados por el profesor en función de lo que espera de sus alumnos en una práctica concreta.

Este trabajo presenta un enfoque para la automatización de la corrección de ejercicios que hayan de cumplir unas características determinadas en lenguajes de la familia C. Este enfoque supone la creación

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

de programas que acompañen a los ejercicios de programación, a los que llamamos *programas-solución*, que verifiquen esos requerimientos. Estos programas-solución pueden ser utilizados tanto por el profesor para la corrección, como pueden ser proporcionados al alumno para que los emplee a fin de saber si su solución es acertada. La creación de estos programas es posible gracias al *árbol de sintaxis abstracta* (AST) generado internamente por el compilador *Clang*, lo cual permite un análisis directo del código escrito por el alumno. El uso de la estructura intermedia empleada por un compilador va un paso más allá en cuanto al trabajo realizado en este campo al conseguir una mayor capacidad de análisis que los analizadores simples creados para este fin. Este análisis además puede ser realizado de una sola vez sobre un conjunto de ficheros, es decir, que es posible estudiar las soluciones de todos los alumnos en una única ejecución.

Estos programas-solución también pueden ejecutar un conjunto de casos de prueba sobre el ejercicio del alumno para comprobar su correcto funcionamiento. Cuando estos casos de prueba son diseñados por el alumno, se vuelve complicado evaluar si el conjunto obtenido es apropiado. En este artículo se propone la prueba de mutaciones [1] como técnica que permita determinar la calidad del conjunto de casos de pruebas creado y que además sirva como vehículo para la mejora del mismo. Este hecho supone un aliciente para que el alumno le dé la relevancia adecuada a esta fase del desarrollo de aplicaciones.

El artículo presenta las ventajas de la creación de estos programas, desde la mejora en la corrección y ahorro de tiempo por parte del profesor, hasta la mejora en la implicación del alumno en su aprendizaje al permitirle conocer sus aciertos/fallos de primera mano. En la Sección 2 se explica en profundidad el problema detectado y cómo *Clang* nos permite un análisis del código fuente a través del AST. La Sección 3 presenta el enfoque propuesto y las amplias posibilidades que introduce, como la automatización de la evaluación y seguimiento del alumno, o el análisis conjunto de las soluciones propuestas para detectar las mejores. En la siguiente sección se muestra un ejemplo concreto de un programa-solución. En la Sección 5 se discuten las limitaciones de la propuesta y en la última sección se comentan las conclusiones y trabajo futuro.

2. Antecedentes

2.1. Motivación

Conseguir que el aprendizaje de un nuevo lenguaje de programación sea efectivo, especialmente en asignaturas introductorias, ha sido uno de los grandes afanes de los profesores encargados de esta labor. Por esta razón, han surgido multitud de trabajos que tratan de mejorar este aprendizaje, incluyendo la propuesta de nuevos métodos innovadores para el refuerzo de los conocimientos y la implicación del alumno [7, 9].

La correcta función de los ejercicios que se plantean para su resolución pueden ser comprobados utilizando técnicas de *testing*. Sin embargo, esto raramente se realiza adecuadamente y tampoco se lleva a cabo una evaluación del progreso de los alumnos al no poderse realizar de manera informatizada. Por consiguiente, la corrección de ejercicios que buscan otro tipo de habilidades se hace aún más dificultosa, pues la única manera de abordarlas es de forma visual. Una de las causas principales para que no se siga el proceso oportuno es el tiempo que supone la corrección una a una de las soluciones aportadas por los alumnos, considerando la gran cantidad de ejercicios que normalmente es necesario para un aprendizaje apropiado.

De esta manera, por parte del profesor es complejo llevar un amplio seguimiento de sus alumnos y, por tanto, evaluar si efectivamente se están consiguiendo los objetivos deseados. Esto además puede tener un efecto secundario, que puede ser la desmotivación del alumnado al sentir el proceso de enseñanza del lenguaje como una tarea individual, así como al no ser consciente de sus errores por no disponer de una retroalimentación correcta y a tiempo.

Ante esto, se han comenzado a emplear herramientas que permitan una evaluación automática de las prácticas entregadas por los alumnos, en lenguaje Java [3, 7] y en otros lenguajes como C/C++ y Ada [11]. A diferencia de este último trabajo, la propuesta de este artículo va un paso más allá en cuanto a la revisión del código. En el trabajo citado, que propone la misma visión para la evaluación de prácticas de programación, se utiliza un analizador sintáctico simple que ha sido diseñado para ese propósito y que es capaz de detectar ciertos errores de especificación, como la correcta utilización de declaraciones o paquetes, o la corrección en el estilo y diseño. En nuestro caso, el analizador sintáctico está basado en un compilador maduro, que garantiza poder estudiar todas aquellas características cubiertas por el compilador gracias al uso de sus bibliotecas. Con esto se consigue un análisis en mayor profundidad del contexto completo del código, por ejemplo al estudiar estructuras más complejas como las de la programación orientada a objetos [8]. Dicho análisis está bastante limitado en un analizador que esté basado en la propia sintaxis del lenguaje.

Por otra parte, existen analizadores estáticos para estos y otros lenguajes, como *PMD* o *FindBugs*, que nos proporcionan ciertas métricas sobre el código y permiten encontrar determinados fallos potenciales. Estas herramientas pueden ser muy útiles para mejorar la calidad del código, pero pueden resultar insuficientes a

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

la hora de centrarnos en detalles específicos, ya que no nos permiten crear reglas propias a comprobar en el código.

En cuanto a la validación del comportamiento de un programa, uno de los métodos más habituales es a través de la composición de un conjunto de casos de prueba. Entre los enfoques más empleados está el proporcionar al alumno herramientas para que diseñe las pruebas de forma que pueda autoevaluar su práctica antes de la entrega de la misma para depurar posibles errores. Por ejemplo, en [8] se pide a los alumnos crear sus casos de prueba con *JUnit* y utilizar la herramienta *Cobertura* para asegurar una cobertura mínima, aunque el cumplimiento de ciertos criterios de cobertura tampoco implica una evaluación rigurosa de los casos de prueba[1]. Estos casos de prueba también han de ser enviados junto con la práctica; no obstante, tal y como se explica en [11], la propia evaluación de las pruebas de los alumnos es una tarea difícil ya que sería necesario comprobar que, efectivamente, los casos de prueba cumplen su misión de detectar los posibles errores existentes en el código.

2.2. Árbol de sintaxis abstracta en Clang

LLVM es un proyecto de código abierto que busca la creación de compiladores para cualquier lenguaje de programación, proporcionando la infraestructura necesaria para su desarrollo. *Clang* es uno de los subproyectos originados del proyecto *LLVM* y que se dedica a la familia de lenguajes de C: C, C++, Objective-C y Objective-C++. Se trata de uno de los proyectos más consolidados, ya que incluso se distribuye con las versiones de *LLVM*, y, como proyecto de código abierto, es posible reutilizar las bibliotecas que emplea.

La llegada del proyecto *LLVM* y del compilador *Clang* ha supuesto un progreso importante respecto a los compiladores precedentes para estos lenguajes, al permitirnos llevar a cabo análisis de código de la misma manera que lo hace un compilador internamente. Esto evita tener que crear herramientas que nos sirvan como solución específica del problema a tratar, que además es difícil que consigan llegar al nivel de profundidad de conocimiento sobre el código que se puede obtener con *Clang*.

Este análisis se consigue a través del recorrido del árbol de sintaxis abstracta o AST. Este árbol es generado a partir de la salida obtenida por el analizador sintáctico, conteniendo toda la información del código de una forma estructurada. En él se representan las distintas expresiones del código mediante sus ramas y, a diferencia del árbol de sintaxis concreta, no se muestran todos los tokens del código. En el AST generado por *Clang*, cada elemento del lenguaje se representa con una clase de nodo. Esto facilita la búsqueda uniforme de aquello que se desea localizar dentro del árbol.

Clang y sus bibliotecas pueden emplearse para utilidades diversas, como el análisis estático, refactorización o generación de código. Así se reduce la complejidad de detectar ciertos elementos en el código, ya que la búsqueda se realiza de forma mecánica en base a unas reglas predefinidas [6]. El uso que se puede dar a las herramientas que se desarrollen es muy diverso [10]. Como ejemplo, podemos citar *Clang Static Analyzer* para encontrar defectos en los lenguajes de la familia C, herramienta integrada dentro del proyecto *Clang*. En el proyecto *LLVM* están involucradas empresas tan importantes como *Apple*, *Google* o *Intel*.

Debemos comentar que *Clang* se sigue adaptando de forma completa a los nuevos estándares que están siendo publicados, como el estándar C++11 [5]. Esto permite que este enfoque pueda seguir siendo implementado para el aprendizaje de las características propuestas en los nuevos estándares.

2.3. Prueba de mutaciones

El trabajo previo que da raíz a esta propuesta de innovación, y que utiliza el AST de *Clang*, es la creación de un marco de prueba de mutaciones para el lenguaje C++ [2]. En la prueba de mutaciones, una técnica de prueba de software, se crean versiones modificadas del programa que está siendo analizado en base a unos operadores de mutación, que surgen del estudio de los errores más comunes en el desarrollo de aplicaciones.

Gracias al análisis de código a través del AST, es posible determinar las localizaciones donde introducir esas mutaciones: eliminar o insertar elementos en el código, así como realizar reemplazos con otros candidatos que pueden provenir del propio análisis del código. El empleo de *Clang* posibilitó la introducción de las modificaciones oportunas de una forma uniforme y robusta, pudiendo observarse el empleo fructífero de este método de implementación y de la técnica [1].

3. Programas-solución

3.1. Proceso de creación y uso

El recorrido por los nodos del AST de *Clang* puede realizarse de dos formas:

1. *Patrón visitante:* Se visita todos los nodos de un mismo tipo, realizando el procesamiento deseado en todos ellos.
2. *Uso de patrones de búsqueda:* Se desarrolla un patrón que enlace aquellos nodos que cumplan un criterio determinado. Los nodos que cumplan las condiciones impuestas pueden ser recuperados en

[1] `http://www.mutiny.eu/essays/code_coverage_versus_mutation_testing`

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

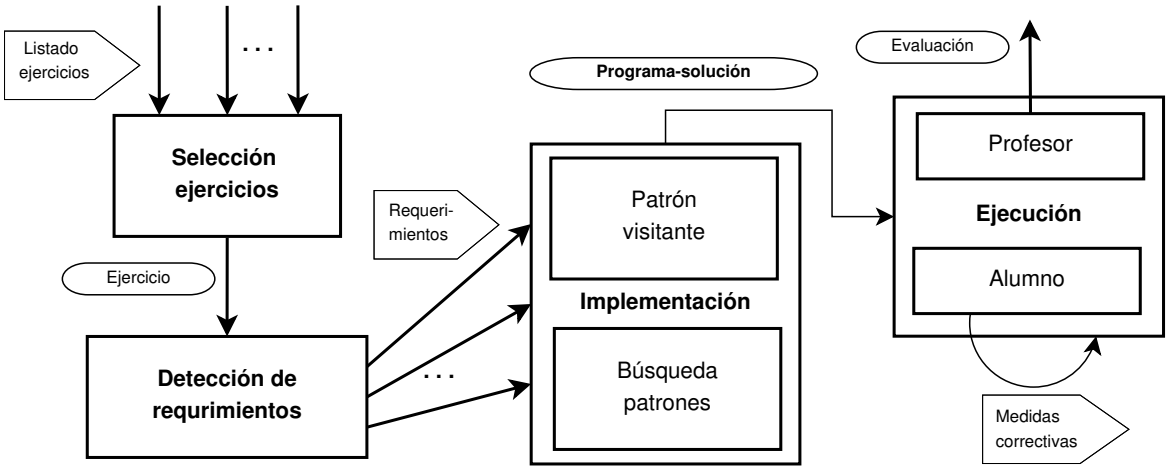

Figura 1: Flujo de creación y uso de programas-solución

el código para su posterior procesamiento. Esta opción es posible gracias a un lenguaje de diseño específico de *Clang*.

Ambos métodos pueden ser empleados para los mismos fines, pudiendo el usuario seleccionar el que más se ajuste a las necesidades de su problema o facilite la implementación del mismo. Tanto en uno como en otro, es posible realizar la búsqueda de varios tipos de nodos a fin de que se cumplan diversos requerimientos en un mismo recorrido del árbol, es decir, que no es necesaria la creación de varios programas-solución por cada una de las especificaciones que han de cumplirse en un mismo ejercicio. Además, la búsqueda de todos los requerimientos indicados se realizan en una única pasada sobre el árbol, lo cual reduce considerablemente el tiempo de ejecución.

El proceso que sigue la corrección automática de prácticas es el siguiente (ver Figura 1):

1. El profesor selecciona y valora qué problemas pueden realizar un uso efectivo de la creación de un programa-solución.
2. En cada ejercicio, se determinan aquellos requerimientos que se querrán comprobar.
3. Esos requerimientos son los que han de ser implementados en el programa-solución. Es aquí donde el desarrollador debe seleccionar cual de los dos métodos de búsqueda va a emplear.
4. El profesor debe también decidir a qué destina esta implementación: si el programa-solución *se distribuye al alumno* junto con el ejercicio que ha de resolver para que se autoevalúe, o bien *lo emplea el propio profesor* para evaluar al alumno.
5. En este último caso, la corrección puede ser de manera directa (se ejecuta la solución contra el ejercicio entregado) o puede integrarse junto con el sistema de administración de cursos que emplee la Universidad [11], de forma que la práctica sea corregida al subirla a la plataforma.
6. El alumno es informado de su resultado, con el cual puede emprender las acciones correctivas pertinentes. La integración con la plataforma virtual puede permitir la retroalimentación rápida hacia el alumno, así como guardar los resultados de los alumnos de cara a su evaluación.

Cuando una herramienta es utilizada por el alumno para su autoevaluación, es importante que la misma sea de fácil instalación y uso como se comenta en [7]. En este sentido, *Clang* se instala en sistemas GNU/Linux de forma sencilla como un paquete más o mediante la apropiada descarga e instalación manual.

3.2. Empleo del programa-solución

La corrección automática de prácticas permite al profesor llevar a cabo un amplio abanico de acciones para apoyar el aprendizaje de sus alumnos, que se presta a la imaginación y que depende de la profundidad que quiera alcanzar en sus correcciones, solo limitado por el tiempo y pericia desarrollando las soluciones.

Desde el punto de vista del alumno, puede ser muy útil no solo conocer si su solución es correcta o no, sino que el programa-solución le informe de los problemas concretos que presenta su práctica. Es más, *Clang* nos permite de una manera sencilla realizar cambios en el código gracias al mapeo directo del árbol con el código a alto nivel, por lo que podría ser más útil aún *introducir comentarios o pistas en el código del alumno* en las localizaciones específicas que requieren de corrección (un ejemplo de esto puede ser visto en la Sección 4.1).

Por parte del profesor, el uso de este enfoque es una fuente de información sobre el desarrollo de las clases. En primer lugar, podrá conocer en qué ejercicios están los alumnos encontrando mayor dificultad, por lo que

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

```
visita_clases : clase → resultado
        //Comprobamos que estamos en una clase definida
        si es_definicion(clase) entonces
            resultado ← falso
            //Recorremos los atributos.
            para cada atributo y perteneciente a clase hacer
                  //Si hay un atributo privado, se cumple el requisito
                  si es_privado(y) entonces
                      resultado ← verdadero
                  fin si
            fin para cada
            //Informamos y finalizamos si no se encuentra el atributo
            si resultado es falso entonces
                escribir("Existen clases que no cumplen los requisitos.")
                devolver resultado
            fin si
        fin si
        devolver resultado
```

Figura 2: Método que visita las clases definidas para determinar si existe alguna sin un atributo privado.

podrá incidir en la instrucción de ese aspecto, o qué ejercicios pueden necesitar un replanteamiento para la aclaración de su enunciado. En segundo lugar, podrá estar informado de la evolución de los alumnos así como de aquellos que destacan en destreza.

Además, puede ser empleado como una herramienta de motivación de sus alumnos. Así, se pueden plantear *ejercicios que supongan un reto que deban tratar de resolver todos sus alumnos*. Por ejemplo, resolver un problema haciendo referencia el menor número posible de veces a variables. Al poder ejecutar la herramienta de una sola vez sobre todas las entregas de los alumnos, podrá encontrar de forma sencilla qué alumno ha aportado la mejor solución. Este trabajo sería bastante laborioso en caso de hacerse de forma manual, ya que el profesor tendría que analizar visualmente cada solución (lo cual es propenso a errores), e ir apuntando la solución de cada alumno para finalmente ver cuál es la mejor. Del mismo modo, se puede premiar a aquellos que sigan un buen estilo de programación.

3.3. Mejora de casos de prueba

Los programas-solución también se puede emplear para automatizar la ejecución de un conjunto de casos de prueba. De esta manera, se podría comprobar si el funcionamiento del programa del alumno es el esperado [4]. La ejecución del programa contra los casos de prueba puede ser tratada desde dos puntos de vista. En el primero, es el profesor el que diseña los casos de prueba y se ejecutan por todos los alumnos al usar el programa-solución por igual, mostrando si la práctica del alumno pasa el conjunto de pruebas o no.

En segundo lugar, y quizás el más interesante, es que se fomente en la asignatura la creación por parte del alumno de un conjunto de casos de prueba que vengan a comprobar la calidad de la solución que ha propuesto. El programa-solución, en lugar de ejecutar las pruebas del profesor, lanzaría los casos de prueba del alumno, lo que le concienciaría de la importancia de esta fase del ciclo de desarrollo de software.

Sin embargo, como se comentó en la Sección 2.1, evaluar si los casos de prueba que plantea el alumno son adecuados se convierte nuevamente en un proceso complejo. En este trabajo se propone el empleo de la prueba de mutaciones (ver Sección 2.3) para realizar la evaluación de las pruebas diseñadas por el alumno. Esta técnica se estima muy oportuna para esta labor ya que creará versiones modificadas del código aportado por el alumno (llamadas *mutantes*) en base a los errores más comunes que los programadores suelen cometer en el desarrollo de aplicaciones. Si el conjunto de pruebas creado es oportuno, debería poder detectar todos los cambios modelados por los mutantes, es decir, se debería obtener resultados distintos en la ejecución del programa original y los mutantes que representan un cambio en el programa.

Este hecho mostrará al alumno si sus casos de prueba son mejorables e incentivará la creación de nuevos que consigan efectivamente detectar a todos los mutantes. Para la inserción de errores en el código, se han de implementar operadores de mutación que reflejen fallos típicos de programación en el lenguaje empleado, en este caso C++ [1]. Para ello, se propone la utilización del método de creación de mutantes aportado en [2], que es análogo al planteamiento de análisis/mo-

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

```
búsqueda_patron : nodos_encontrados → resultado
        si existe_nodo(nodos_encontrados, patron_atributo_privado) entonces
            escribir("Existen clases que no cumplen los requisitos.")
            devolver falso
        fin si
        devolver verdadero
```

Figura 3: Método que recupera y trata los nodos del árbol encontrados por el patrón de búsqueda creado.

dificación de código en este artículo. El uso de esta técnica puede complementarse, en cualquier caso, con la evaluación de criterios de cobertura.

4. Ejemplo de uso

4.1. Programa-solución

Para ejemplificar lo expuesto, vamos a suponer que se busca que el alumno haya incluido un atributo privado en todas las clases que se han definido en la práctica. Como se ha comentado en la Sección 3, es posible realizar el recorrido del AST de dos formas: con el patrón visitante y a través de patrones de búsqueda.

- Patrón visitante: Ya que necesitamos confirmar que todas las clases cumplan una determinada condición, se debe visitar todas las clases definidas a través del elemento que representa al elemento clase. En la Figura 2 se puede observar una implementación en pseudocódigo de la solución acompañada de comentarios. El método *visita_clases* se ejecutará por cada clase que esté presente en el código analizado, siendo asignada al parámetro de entrada *clase*.
- Patrones de búsqueda: En este caso, para lograr el objetivo podemos diseñar un patrón para buscar una clase que carezca de un atributo privado. El patrón realiza esencialmente la misma tarea que se muestra en la Figura 2, pero en lugar de en lenguaje C++ se utiliza un lenguaje específico de dominio creado por la propia comunidad de *Clang* para facilitar el uso de las bibliotecas.
  Al patrón se le asigna una cadena identificativa que nos permite recuperar los nodos encontrados por este patrón en la búsqueda dentro del árbol. En la Figura 3 se muestra cómo, mediante la cadena que identifica al patrón (en el ejemplo la cadena es "patron_atributo_privado"), se rescatan los nodos específicos de este patrón entre todos los localizados por los diversos patrones (*nodos_encontrados*) ya que, como se comentó en la Sección 3.1, se pueden verificar varios requerimientos en un mismo recorrido. Cuando el patrón encuentre un nodo (*existe_nodo* devuelve verdadero), se podrá acabar la ejecución indicando que al menos una clase no cumple el requisito.

Podemos ir un paso más allá en la corrección de este ejercicio, siendo más específicos con el problema hallado. Como se comentó anteriormente, es posible realizar cambios de forma directa en el código, lo cual se consigue de una manera sencilla gracias a la información que se provee en el AST sobre la ubicación exacta de cada elemento en el código fuente. De esta manera podemos cambiar la implementación de la solución para que incluya un comentario en aquellas clases para las que el usuario no haya incluido el atributo privado. Así se logra que el alumno pueda observar con mayor facilidad dónde se encuentra el problema. A nivel de pseudocódigo, la implementación de la Figura 2 cambiaría la función *escribir* por otra función que insertase los comentarios correspondientes (ver Figura 4).

```
localizacion ← localiza_elemento(clase)
comentario ← "Esta clase no cumple los requisitos"
inserta_texto(localizacion, comentario)
```

Figura 4: Cambio en el código de la Figura 2 para introducir comentarios en el código.

El resultado de cómo quedaría una clase con el comentario introducido, al no cumplir el requerimiento planteado, puede verse en la Figura 5.

```
class A{
  public:
      A() {g = 0;}
      A(int v_a1): a1(v_a1) {}
      int a1;
      int recuperar_a1 () {return a1;}
/*Esta clase no cumple los requisitos*/
};
```

Figura 5: Resultado de aplicar el código de la Figura 4 en una clase que no cumple el requisito.

Como comentarios finales, decir que al término de la ejecución se puede avisar al alumno de la existencia de errores. Si no se quiere sobrescribir el fichero del alumno con los comentarios insertados, se puede mos-

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

trar el resultado por pantalla o simplemente guardar el resultado en un nuevo fichero de código.

4.2. Operador de mutación

Supongamos un ejercicio simple de una calculadora que aplique los operadores aritméticos $+$, $-$, $*$, $/$ sobre dos operandos. Para probar su programa, un alumno podría emplear una única prueba en la que diese los valores 8 y 4 a los operandos. Con esta prueba, si existiese un fallo, sería fácil detectarlo porque las cuatro operaciones aritméticas dan un valor distinto con estos operandos. Sin embargo, si el alumno en su lugar proporciona los valores 2 y 1, teniendo en cuenta que las operaciones $2*1$ y $2/1$ dan el mismo resultado, sería imposible determinar un posible intercambio de ambas operaciones en el código solo con este caso de prueba.

Un operador de mutación adecuado para este ejercicio sería aquel que reemplaza cada operador aritmético en el código por el resto de operadores aritméticos. Este operador podría ayudar al alumno a entender que su conjunto de casos de prueba es insuficiente cuando justamente la mutación cambie los operadores $*$ y $/$.

5. Limitaciones

5.1. Ejercicios

El enfoque presentado para la corrección de ejercicios de programación solo puede ser útil para un subconjunto de ejercicios como los mencionados a lo largo del documento. En primer lugar, no todos los requerimientos son tan específicos como para poder crear un programa-solución que los verifique de una forma general (la solución se ejecuta en todos los ficheros) y robusta (la corrección se aplica de forma apropiada). Además, hay que tener en cuenta que un análisis estático de código no nos permite conocer si el programa se comportará como debe hacerlo. Es por ello que la solución propuesta puede ser complementada con la ejecución, también automatizada, de un conjunto de casos de prueba tal y como se explica en la Sección 3.3.

En segundo lugar, no siempre será sencillo realizar las comprobaciones oportunas. En este último sentido, el profesor debe ser quien valore si su programa-solución podrá cumplir su función sin problemas y si compensa el tiempo de desarrollar la solución respecto al beneficio que se obtendrá con la misma.

5.2. Profesor y alumno

El profesor pasará a dedicar más tiempo en el desarrollo de las soluciones para sus ejercicios que en la corrección de los mismos, pero ese tiempo dependerá también de la habilidad del mismo en su realización. La experiencia de los autores nos dice que la curva de aprendizaje no es tan empinada para comprobaciones sencillas una vez se tiene una estructura estable para la búsqueda, ya que las bibliotecas son fáciles de usar y existe documentación sobre las mismas[2]. Sin embargo, debido a la magnitud de bibliotecas que se manejan en *Clang* (lo cual es lógico teniendo en cuenta la flexibilidad y las características avanzadas que presenta un lenguaje de propósito general), se requiere de mayor soltura para llevar a cabo acciones más específicas.

Por otra parte, es posible que para la aplicación de este enfoque sea necesario que el alumno se ciña a ciertas reglas para su correcta aplicación. Por ejemplo, en ocasiones puede ser necesario que el alumno emplee un determinado nombre para una declaración de una variable. Del mismo modo, en los casos en los que el profesor desea evaluar los ejercicios de varios alumnos en la misma ejecución, se hace necesario que el alumno asigne un nombre específico al fichero de su ejercicio para su correcta identificación. En caso contrario, será el profesor quien tenga que renombrar los ficheros para la ejecución, lo cual puede ser engorroso en el caso de tratar con un número considerable de ficheros.

5.3. Prueba de mutaciones

En cuanto a la prueba de mutaciones, dos son los inconvenientes más habituales de la técnica. El primero es la existencia de *mutantes equivalentes* que no representan un cambio en el comportamiento del programa que se está probando. Esto significa que no es posible crear un caso de prueba para detectar el fallo simulado en el mutante. Determinar si un mutante es equivalente o no es una tarea que se realiza de forma manual.

El segundo problema radica en la cantidad de mutantes que pueden ser generados y que puede conllevar un alto coste computacional. Es por ello que el profesor deberá seleccionar operadores de mutación ajustados a cada ejercicio, que reduzcan el número de mutantes que se pueden crear y vengan a mejorar las pruebas para las características reseñables de cada ejercicio.

6. Conclusiones y trabajo futuro

Este artículo presenta un enfoque cuyo objetivo es el de mejorar la enseñanza de habilidades de programación mediante la automatización de la solución para los ejercicios propuestos. Los programas-solución buscarán el cumplimiento de ciertos requerimientos que puedan ser analizados de manera estática sobre el código. La irrupción de *Clang* permite mejorar el desarrollo existente en este campo hasta el momento al permitir un análisis más potente del código, pudiendo abarcar una mayor diversidad de requisitos en la corrección

[2] http://clang.llvm.org/doxygen/index.html

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

de las prácticas. Al tratarse de un proyecto de código abierto, el empleo de este recurso está disponible para toda la comunidad universitaria. También se propone el uso de la prueba de mutaciones para evaluar los casos de prueba creados por el alumno para probar sus programas mediante un marco de pruebas generado también con este compilador.

Con el planteamiento expuesto se espera conseguir una mejora en la participación del alumno en su propio proceso de instrucción al disponer de la asistencia directa de un programa sobre el ejercicio planteado, y que este sea consciente de la importancia de la fase de pruebas. Por parte del profesor, la ejecución del programa-solución sobre todas los envíos de los alumnos, incluso de una sola vez, le permitirá poder analizar un mayor número de ejercicios en el mismo espacio de tiempo que si hiciese esta labor de manera manual. Además, los resultados pueden ser más fiables y se pueden plantear ejercicios colectivos con mayor facilidad que hasta el momento.

Los lenguajes C y C++ son lenguajes de propósito general que continúan siendo lenguajes de referencia para la enseñanza de la programación, por lo que la propuesta mostrada puede ser empleada de forma extensiva. Aunque el análisis estático depende en gran medida del lenguaje de programación empleado, gracias a que *Clang* forma parte del proyecto *LLVM*, cuya finalidad es el desarrollo de compiladores para cualquier lenguaje, este método se podría aplicar a otros lenguajes siempre que existan compiladores que ofrezcan las mismas posibilidades que aquí se presentan.

En el futuro, nos gustaría desarrollar una aplicación que pueda ser empleada por el profesor para la generación de las soluciones, de manera que su estructura permitiera que el profesor se centrase únicamente en la particularidad de la implementación de la solución y pudiese obviar todos aquellos elementos que son comunes. Con esta aplicación también sería posible el análisis de las soluciones de los alumnos, así como llevar un histórico de las evaluaciones de cada alumno, de manera que se pueda observar su progreso en la asignatura. Para finalizar, esta innovación se desea implantar en asignaturas de iniciación a la programación en la Universidad de Cádiz para tratar de recopilar datos de su validez y también del grado de satisfacción obtenida tanto por el profesor como por el alumnado.

Referencias

[1] P. Delgado-Pérez, I. Medina-Bulo, J.J. Domínguez-Jiménez, Antonio García-Domínguez y Francisco Palomo-Lozano. Class mutation operators for C++ object-oriented systems. *Annals of Telecommunications*, 2014.

[2] Pedro Delgado-Pérez, Inmaculada Medina-Bulo y Juan José Domínguez-Jiménez. Generación de mutantes válidos en el lenguaje de programación C++. En *XIX Jornadas de Ingeniería del Software y Base de Datos, JISBD 2014*, Cádiz, España, 2014.

[3] Francisco Durán, Francisco Gutiérrez y Ernesto Pimentel. El uso de herramientas de apoyo para la valoración de actividades prácticas de programación. En *Innovación educativa en las titulaciones de Informática en la Universidad española*, páginas 13–28. Servicio de Publicaciones, 2008.

[4] Marco Antonio Gómez Martín, Guillermo Jiménez Díaz y Pedro Pablo Gómez Martín. Test de unidad para la corrección de prácticas de programación, ¿una estrategia win-win? En *Actas de las XVI Jornadas de Enseñanza Universitaria de Informática, Jenui 2010*, páginas 51–58, Santiago de Compostela, 2010.

[5] ISO. *ISO/IEC 14882:2011 Information technology – Programming languages – C++*. International Organization for Standardization, Ginebra, Suiza, Febrero 2012.

[6] Olaf Krzikalla. Performing source-to-source transformations with Clang. En *2013 European LLVM Conference*, Paris, Francia, Abril 2013.

[7] Germán Moltó y Oscar Sapena. Entorno virtualizado de aprendizaje para facilitar el desarrollo de destrezas de programación. En *Actas de las XIX Jornadas de Enseñanza Universitaria de Informática, Jenui 2013*, páginas 327–334, Castellón, 2013.

[8] E. Mosqueira-Rey. La evaluación continua y la autoevaluación en el marco de la enseñanza de la programación orientada a objetos. En *Actas de las XVI Jornadas de Enseñanza Universitaria de Informática, Jenui 2010*, páginas 223–230, Santiago de Compostela, 2010.

[9] José Otero, Rosario Suárez, Luciano Sánchez y Inés Couso. Tarjetas didácticas digitales en cursos introductorios de programación: experiencia piloto y aplicación cliente servidor para seguimiento del aprendizaje. En *Actas de las XX Jornadas de Enseñanza Universitaria de Informática, Jenui 2014*, páginas 431–434, Oviedo, 2014.

[10] Elias Penttilä. Improving C++ software quality with static code analysis. Tesis de Master, Aalto University, School of Science, Mayo 2014.

[11] Juan Carlos Rodríguez del Pino, Margarita Díaz Roca, Zenón Hernández Figueroa y José Daniel González Domínguez. Hacia la evaluación continua automática de prácticas de programación. En *Actas de las XIII Jornadas de Enseñanza Universitaria de Informática, Jenui 2007*, páginas 179–186, Teruel, 2007.