Actas de las Jenui, vol. 2. 2017.
Páginas: 297-304

Biblioteca CAC++ para la corrección automática de prácticas de programación en C++

Pedro Delgado-Pérez
Escuela Superior de Ingeniería
Universidad de Cádiz
Av. de la Universidad de Cádiz
10, 11519 - Puerto Real
pedro.delgado@uca.es

Inmaculada Medina-Bulo
Escuela Superior de Ingeniería
Universidad de Cádiz
Av. de la Universidad de Cádiz
10, 11519 - Puerto Real
inmaculada.medina@uca.es

Daniel Pérez-Caro
Escuela Superior de Ingeniería
Universidad de Cádiz
Av. de la Universidad de Cádiz
10, 11519 - Puerto Real
daniel.perezcaro@mail.uca.es

Resumen

La corrección automática de prácticas de programación facilita el aprendizaje y la evaluación en asignaturas en las que se enseñan estas habilidades. El análisis estático es una opción que permite detectar sobre el propio código del alumno si se cumplen ciertas condiciones establecidas en el enunciado de las prácticas. No obstante, la implementación de programas de análisis estático es compleja ya que se requiere de un mecanismo para analizar el código, teniendo que contemplar detalles de bajo nivel. Además, cuando las comprobaciones que se realizan sobre el código de los alumnos se implementan para atajar casos particulares, se hace difícil su reutilización. En este artículo se presenta la biblioteca CAC++, la cual ofrece un método intuitivo para aplicar comprobaciones de manera sencilla, ya que se abstraen los detalles de implementación, pudiendo sin embargo personalizarlas para cada práctica concreta. En este artículo se presenta su estructura y principales características así como un ejemplo de uso que permite observar el alcance de la biblioteca desarrollada.

Abstract

The automatic correction of programming practices facilitates learning and evaluation in subjects where these skills are taught. The static analysis is a plausible option that allows analyzing the source code implemented by students to check whether the code meets the established conditions in the practice. However, implementing static analysis programs is a complex task that requires a mechanism to anlayze the code and to handle low-level details. Moreover, reusing these programs is difficult when the verifications implemented in them are guided by the particularities of each specific practice. This paper presents the CAC++ library, which offers an intuitive method to apply verifications in a simple way, since the library abstracts away implementation details, being still possible to tailor them to each practice. This paper shows the library structure and main features as well as an illustrative example that allows the teacher to observe the extent of the developed library.

Palabras clave

Corrección automática, prácticas de programación, análisis estático, orientación a objetos, C++.

1. Introducción

En la enseñanza de habilidades de programación en informática, la transmisión de conocimientos por parte del profesor y el aprendizaje por parte del alumno constituyen un proceso complejo y que requiere de mucho trabajo por ambas partes para que el mismo tenga éxito. Normalmente, la realización de prácticas de programación en estas asignaturas supone un aspecto clave para que el alumno pueda poner en funcionamiento y ver un sentido a lo aprendido en la teoría. Nuevamente, es fundamental que exista una interacción profesor-alumno para que el aprendizaje sea efectivo. No obstante, esta interacción se ve en ocasiones dificultada por varios factores.

- Número elevado de alumnos en clase.
- El profesor solo puede corregir las prácticas cuando estos las terminan y envían en la fecha fijada.
- El alumno recibe retroalimentación de sus soluciones pasado un tiempo desde su realización, pudiendo olvidar los detalles que le llevaron a cometer tales errores.

Es por ello que en los últimos años varios autores han buscado surtir tanto al profesor como al alumno

de herramientas que les ayuden a superar esas dificultades y mejorar el proceso en general [5, 6, 10]. Una de las soluciones propuestas para paliar el problema es el análisis de estático de código, el cual busca verificar sobre el código del alumno que se cumplen ciertas condiciones establecidas para la práctica (de ahora en adelante, *comprobaciones*) [1, 8, 9]. A diferencia de lo que aporta un conjunto de casos de prueba, el *análisis estático* [7] es útil cuando se quieren comprobar requisitos que no pueden ser detectados a través de la ejecución de pruebas (como que el alumno utilice una característica determinada del lenguaje para implementar la práctica), o cuando el código ni siquiera puede ser ejecutado porque se produce un error al enlazarlo con el fichero que contiene las pruebas.

En una asignatura de programación orientada a objetos en la que se enseña este paradigma a través de C++, se puso en práctica la corrección automática de código de prácticas a través de *Clang*. Por cada uno de los enunciados de prácticas se desarrollaron sendos programa de comprobaciones. Estos programas, en lugar de ser usados para evaluar las prácticas, eran entregados a los alumnos para que pudieran disponer de una herramienta que les informe en todo momento y de primera mano de los errores que están cometiendo durante la realización de las prácticas y no después de las mismas. Esta solución viene a paliar los problemas mencionados al inicio de esta sección en la interacción profesor-alumno.

Sin embargo, los programas que se desarrollaron para esta asignatura como experiencia piloto presentaban ciertos problemas, como la dificultad para reutilizar las comprobaciones y la falta de homogeneidad y de documentación al ser desarrolladas de forma individual para cada práctica. Todo esto motivó la creación de la *biblioteca de Comprobación Automática de Código C++ (biblioteca CAC++)*. La biblioteca recoge las comprobaciones realizadas (principalmente respecto a la programación orientada a objetos) pero de una manera genérica para abarcar la casuística de cada comprobación. Esto permite su reutilización para cualquier práctica y asignatura, pudiendo personalizar las comprobaciones en cada práctica. En la biblioteca se ofrece el listado de comprobaciones documentadas, las cuales siguen un mismo esquema tanto internamente como externamente (para que sean de uso intuitivo para el profesor). Por último, la biblioteca oculta los detalles de implementación. A pesar de ello, su estructura modular (los diferentes elementos que conforman cada comprobación están agrupados en módulos), facilita la actualización y fomenta la mejora de la misma.

En la siguiente sección ahondaremos en los aspectos que motivaron el desarrollo de la biblioteca CAC++. En la Sección 3 se listan las novedades que presenta el recurso frente a otras herramientas similares. Seguidamente, se mostrará un ejemplo en la Sección 4, que será usado en la Sección 5 para describir la estructura de la biblioteca y el flujo que sigue internamente. En la Sección 6 se listarán las características que se contemplaron en su diseño. Finalmente, se comentarán las conclusiones y trabajo futuro en la última sección.

2. Motivación para la creación de la biblioteca CAC++

Para implementar programas de análisis estático se requiere de un mecanismo con el que podamos llevar a la práctica las comprobaciones necesarias y que nos permita analizar con garantías el código. Analizar sintácticamente el código de un lenguaje de propósito general como C++ es una tarea compleja, especialmente si nos enfocamos en las estructuras que involucran a la programación orientada a objetos. En el caso de C++, *Clang* [2] nos ofrece un método robusto para el análisis del código a partir del *árbol de sintaxis abstracta*, que es una representación estructurada del código más sencilla de procesar. No obstante, por la propia magnitud de la gramática de un lenguaje como C++, el empleo de estas bibliotecas no es trivial y se requiere de cierta experiencia en el uso de *Clang*, tal y como se comenta en [3].

Por esa razón, al diseñar las comprobaciones de cada práctica en particular desde cero, observamos los siguientes inconvenientes:

- La creación de programas de comprobaciones independientes para cada práctica dificulta el seguimiento de las diferentes comprobaciones que se realizan, ya que se implementan de manera específica para cada situación. Esto dificulta reutilizar dichas comprobaciones para nuevas prácticas en el futuro.
- El desarrollo de programas *ad-hoc* para las prácticas también resulta en programas con poca modularidad y con estructura diversa. Esto dificulta la corrección y actualización de los mismos.
- Siendo el uso de las bibliotecas de *Clang* complejo, cualquier profesor no experto podría renunciar a cambiar los programas o incluir nuevas comprobaciones. Este hecho se acentúa teniendo en cuenta que normalmente los programas se actualizarán de curso en curso (los conocimientos adquiridos en su manejo pueden no servir de un año a otro).

Las razones anteriores son por tanto la motivación del desarrollo de la biblioteca CAC++. La creación de una biblioteca como esta ya se había planteado en un trabajo anterior [4].

3. Novedad del recurso

El análisis estático se había empleado a nivel docente con anterioridad a la creación de la biblioteca CAC++. Existen herramientas tanto para C++ como para otros lenguajes. Para C++ podemos destacar la herramienta de análisis estático Style++ [1], que está basada en el front-end de *Edison Design Group*[1]. A nivel nacional, podemos mencionar la herramienta CUESTOR para C y Java [9] o el sistema de evaluación de prácticas con analizadores sintácticos para C, C++ y Ada [8]. Por otra parte, y tal y como se comenta en [3], existen otros analizadores estáticos como *PMD* y *FindBugs*, los cuales señalan determinados fallos potenciales y nos proporcionan información sobre la calidad de nuestro código respecto a varias métricas. Aunque estas herramientas pueden ser aplicadas a cualquier contexto, también pueden ser útiles a la hora de evaluar la calidad del código desarrollado en las prácticas.

El recurso que se presenta en este trabajo presenta las siguientes novedades con respecto al trabajo previo:

- En este trabajo no se presenta una herramienta cerrada para su simple utilización, sino una biblioteca personalizable y abierta también a la mejora. La diferencia estriba en que la biblioteca no ciñe a unas comprobaciones determinadas, sino que su estructura interna propicia la actualización y extensión con nuevas comprobaciones. De esa manera, se pueden atajar casos aún no contemplados cuando estos se presenten.
- Las herramientas comentadas suelen permitir una configuración que se limita a habilitar/deshabilitar comprobaciones. Las comprobaciones que están habilitadas, después se aplican al código al completo, informando sobre aquellos puntos donde se detectan problemas. Al usar CAC++, el profesor no solo selecciona que comprobaciones quiere aplicar, sino que puede indicar en qué casos concretos se aplican. Por ejemplo, el profesor puede querer realizar ciertas comprobaciones en una clase pero no en otra. De esta forma, el programa que se deriva del uso de la biblioteca se alinea de forma perfecta con las condiciones que se impongan en la práctica (se evita que las comprobaciones se apliquen a otras partes del código que no se desean analizar).

Por otra parte, las mejoras que presenta este enfoque sobre la realización de programas individuales para cada práctica son:

- Se consigue un repositorio que recoge todas las comprobaciones realizadas hasta el momento de una manera genérica, de forma que se documentan adecuadamente y se fomenta su reutilización.

[1] `https://www.edg.com/c`

```
class Ejemplo{

    public:
        Ejemplo(): a(0) { }
        void metodo_que_reutiliza(int p){
            a = metodo_a_reutilizar() + p;
        }
        int metodo_a_reutilizar(){
            return a * 2;
        }
    private:
        int a;
};
```

Figura 1: Código correcto para la práctica

- La biblioteca puede ser utilizada fácilmente ya que las comprobaciones se presentan de manera que se abstraen los detalles de su implementación.

4. Ejemplo

En esta sección mostramos un ejemplo donde el uso de la biblioteca puede ser útil. Supongamos la existencia de una práctica para nuestro ejemplo, de la cual el profesor extrae las siguientes comprobaciones que desea validar en el código de sus alumnos:

- En primer lugar, comprobar que se ha incluido una clase `Ejemplo`.
- En ese caso, validar que se ha definido un constructor por defecto para esta clase.
- También validar en esa clase que se está llamando al método con nombre `metodo_a_reutilizar` (o `metodo_a_reusar`) dentro del método `metodo_que_reutiliza(int)`. En este ejemplo, el hecho de que el método llamado tenga dos posibles nombres sirve para ilustrar más adelante que la biblioteca contempla no limitar a un único nombre específico para los elementos sobre los que se realiza la comprobación.

De esta manera, un código que sería correcto para nuestra práctica sería el que se muestra en la Figura 1. En la siguiente sección recurriremos a este ejemplo para explicar la estructura de la biblioteca CAC++. Así las cosas, se mostrará desde la generación del fichero de comprobaciones hasta la ejecución del programa mostrando los resultados.

5. Biblioteca CAC++

5.1. Estructura

La estructura de la biblioteca ha sido diseñada para separar claramente cada parte del código según su

```
#include "caclibrary.h"
#include <iostream>

int main(int argc, const char **argv){

    checkCode c(argc, argv, "codigo_alumno.cpp", "Orden:␣./ejemplo␣codigo␣alumno.cpp");

    c.setCorrectMessage("Clase␣Ejemplo␣correcta");
    c.setIncorrectMessage("Revisa␣los␣errores␣de␣la␣clase␣Ejemplo");

    std::cout << "Programa␣de␣comprobaciones␣para␣la␣clase␣Ejemplo" << std::endl;
    std::cout << "**********************************************" << std::endl;

    //Validar la existencia de la clase Ejemplo
    if(c.findClass("Ejemplo")){

        //Validar que se define constructor por defecto
        c.defaultConstructor("Ejemplo", "Revisa␣el␣enunciado␣respecto␣a␣los␣constructores");

        //Validar que se reutiliza
        c.methodWithReferencedMethod({"metodo_que_reutiliza"}, {{"int"}}, "Ejemplo", {"?"},
                                     {"metodo_a_reu.*ar"}, {{}}, "Ejemplo", {"?"},
                                     "No␣se␣reutiliza␣tal␣como␣indica␣el␣enunciado");

        //Ejecutar comprobaciones
        c.check();
    }
    else{
        std::cout << "No␣se␣ha␣encontrado␣la␣clase␣Ejemplo" << std::endl;
    }

    return 0;
}
```

Figura 2: Fichero de comprobaciones para la clase Ejemplo.

finalidad así como evitar al usuario tener que manejar elementos relacionados con la implementación interna. Se persigue que la biblioteca permita una fácil mantenibilidad y que las comprobaciones sigan un patrón de funcionamiento similar para facilitar la labor del usuario, entre otras características que serán vistas en la Sección 6. De forma general, podemos destacar tres partes diferenciadas en la arquitectura:

Fichero de comprobaciones

Es el fichero principal y desde donde se ordena al sistema qué acciones realizar haciendo uso de la biblioteca CAC++.

A través de este fichero el profesor emplea la biblioteca, implementando *un fichero de comprobaciones para cada práctica*. En ellos, el profesor se encarga de adecuar las diferentes comprobaciones que se desean realizar sobre el código de los alumnos en base a los requisitos de las prácticas. En la biblioteca existe un método asociado a cada una de las comprobaciones que se han contemplado en la misma. Las comprobaciones se invocan acorde a las indicaciones establecidas en la biblioteca, usando los parámetros del método para determinar las condiciones concretas de la comprobación.

Ejemplo Retomando el ejemplo iniciado en la Sección 4, la Figura 2 muestra el código del fichero de comprobaciones que tendría que desarrollar el profesor para hacer las validaciones pertinentes en el código implementado por un alumno. Como puede observarse, en este programa se suprime toda referencia a cómo se llevan a cabo las comprobaciones internamente. De esta manera, el profesor solo tiene que centrarse en indicar las comprobaciones:

- **findClass** (línea 15): permite comprobar que la clase `Ejemplo` se ha definido en el código. Al estar como condición en una sentencia condicional, el resto de comprobaciones no se ejecutarán si la clase no se detecta.
- **defaultConstructor** (línea 18): busca la existencia de un constructor por defecto para esta clase.
- **methodWithReferencedMethod** (línea 21): detecta que un método concreto invoca a otro en el cuerpo de su definición. Recibe por parámetros:
    1. Nombre del método (método 1) que va a referenciar a otro método (método 2).
    2. Tipos de los parámetros del método 1 (re-

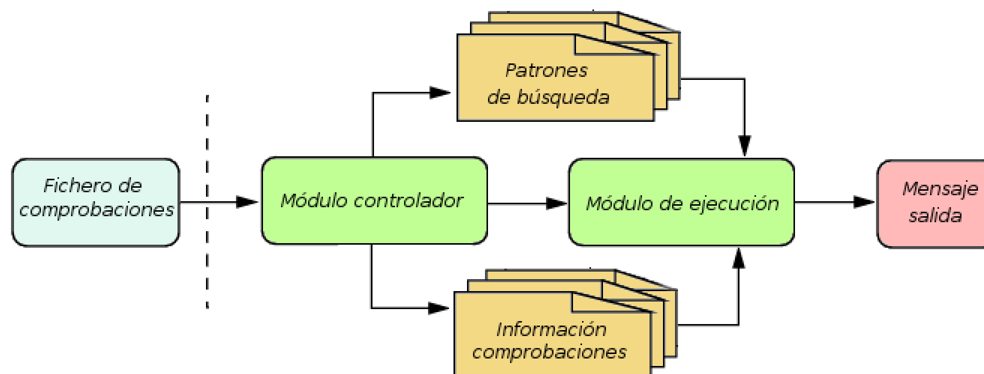

Figura 3: Diagrama simplificado de funcionamiento de la biblioteca CAC++. El profesor desarrolla el fichero de comprobaciones en base a la interfaz de la biblioteca (marcada con una línea discontinua). Tras la ejecución de las comprobaciones, se muestra un mensaje como salida.

cuérdese que en C++ un método puede ser sobrecargado, teniendo una clase varios métodos con el mismo nombre pero distintos parámetros).

3. Clase a la que pertenece el método 1.
4. Constante/No constante (método 1).

Estos mismos cuatro parámetros también se indican seguidamente para identificar al método 2 (línea 22). En la Sección 6.2 se dan más detalles sobre los parámetros de esta comprobación.

Estas dos últimas comprobaciones reciben como último parámetro el mensaje que se ha de mostrar en caso de que no se cumpla la comprobación. Si no se provee ningún mensaje, se imprimirá un mensaje por defecto.

Módulo controlador

Es el módulo encargado de procesar las instrucciones dadas por el usuario en el fichero principal para establecer las búsquedas concretas que se han de llevar a cabo en el código.

En este módulo no se ejecutan las comprobaciones aún, sino que se prepara la información necesaria para la posterior ejecución de las comprobaciones solicitadas. En concreto, en cada uno de los métodos asociados a las diferentes comprobaciones, se pueden observar las siguientes tareas:

- Según los parámetros indicados por el usuario, se almacena información relativa a cada comprobación, que será necesaria para determinar si esta se cumple o no durante la fase de ejecución. Para facilitar la mantenibilidad del código, esa información se mantiene en una clase independiente.
- A partir de la información de la comprobación introducida, se establece un nuevo *patrón de búsqueda* para analizar el código del alumno. Este patrón se incluye a la lista del resto de patrones de búsqueda que se van a tener en cuenta a la hora de recorrer el árbol de sintaxis abstracta a fin de poder realizar las comprobaciones oportunas. Todos los patrones de búsqueda implementados en la biblioteca están recogidos en un mismo fichero para su fácil localización.

En la Figura 3 se puede observar un diagrama del flujo que siguen las comprobaciones en la biblioteca desde que se añaden al fichero de comprobaciones hasta que se ejecutan para mostrar el resultado.

Módulo de ejecución

Es el módulo que procede a ejecutar todas las comprobaciones indicadas por el usuario.

En este módulo se toma el código del alumno y se deriva el árbol de sintaxis abstracta del mismo. Este árbol es entonces recorrido para validar si se cumplen las condiciones que se especifican en las comprobaciones introducidas. Es decir, se trata de buscar elementos en el código coincidentes con los patrones de búsqueda añadidos en el módulo controlador. El recorrido se realiza una única vez con todos los patrones al mismo tiempo, lo cual evita realizar un recorrido completo del árbol de sintaxis abstracta por cada una de las comprobaciones indicadas.

Para cada una de las comprobaciones se almacena un resultado, el cual se emplea finalmente para cotejar si las comprobaciones terminaron con éxito o se detectaron algunos errores. Por cada problema detectado en el código *se mostrará al alumno un mensaje que proporciona información orientativa* sobre el mismo.

Ejemplo En nuestro ejemplo, el alumno primero desarrolla un código que no es correcto según las espe-

cificaciones del enunciado (véase Figura 4). En concreto, no invoca el método `metodo_a_reutilizar` (compárese con Figura 1).

```
class Ejemplo{

    public:
        Ejemplo(): a(0) { }
        void metodo_que_reutiliza(int p){
            a = a * 2 + p;
        }
        int metodo_a_reutilizar(){
            return a * 2;
        }
    private:
        int a;
};
```

Figura 4: Código incorrecto para la práctica (no se cumple la reutilización).

Al ejecutar el programa de comprobaciones sobre este código incorrecto, se mostrará el mensaje relacionado con la comprobación que incluyó el profesor en el desarrollo del programa (ver Figura 5). Justo tras mostrar los errores, también se imprimirá el mensaje que se fijó con el método `setIncorrectMessage` (línea 9 en Figura 2).

Basándose en este mensaje, el alumno tratará de corregir su código, hasta obtener una versión correcta como la de la Figura 1. Cuando todas las comprobaciones pasen con éxito, se mostrará el mensaje que puede observarse en la Figura 6, el cual se fijó con el método `setCorrectMessage` (línea 8 en Figura 2).

```
./ejemplo codigo_alumno.cpp --

Programa de comprobaciones para la clase Ejemplo
************************************************
No se reutiliza tal como indica el enunciado
--------------------
Revisa los errores de la clase Ejemplo
```

Figura 5: Ejecución con errores

```
./ejemplo codigo_alumno.cpp --

Programa de comprobaciones para la clase Ejemplo
************************************************
Clase Ejemplo correcta
```

Figura 6: Ejecución sin errores

5.2. Detalles técnicos

CAC++ ofrece actualmente un rango de comprobaciones, que pueden agruparse en varias categorías generales:

- *Clase*, como comprobar si una clase existe.
- *Atributos*, como saber sobre su nivel de acceso.
- *Métodos*, como saber si están marcados como *inline* o *noexcept*.
- *Construcción/destrucción*, como conocer sobre la existencia de ciertos constructores.
- *Funciones*, como comprobar si una función está marcada como amiga de una clase.
- *Miscelánea*, como validar que ciertas funciones invocadas están siendo tomadas de la cabecera correcta.

La biblioteca ha sido desarrollada usando la versión 3.9 de *Clang* en un sistema con Ubuntu 16.04. Se encuentra disponible en la dirección:

`https://ucase.uca.es/cac`

Conviene remarcar que la biblioteca también permite establecer la orden que el alumno ha de proveer para que el programa de comprobaciones se ejecute como es debido (línea 6 en Figura 2). Esta orden (o cualquier otra información que se desee transmitir al alumno) se imprimirá al ejecutar el programa con la opción `--help`.

6. Diseño

6.1. Alcance y limitaciones

El desarrollo de programas de análisis estático, como los que aquí se abordan, viene de la mano con una limitación inherente: existen infinidad de comprobaciones que se podrían llevar a cabo en el código. Como tal, es imposible en la práctica que la biblioteca que en este trabajo se presenta contenga todas las comprobaciones que un profesor pueda idear o necesitar. Además, como ya hemos comentado anteriormente, actualmente la biblioteca se centra en la programación orientada a objetos, pues se originó a partir de una asignatura enfocada en este paradigma de programación.

Por otra parte, se pretendía dotar a la biblioteca CAC++ de flexibilidad para abarcar en la medida de lo posible diferentes variantes que pudieran derivarse de cada comprobación. Esta medida, sin embargo, podía comprometer la simplicidad de empleo de la biblioteca, lo cual es un factor primordial para su uso efectivo por parte de cualquier profesor.

Conocedores de estas limitaciones, la fase de análisis y diseño de CAC++ se presentaba como un aspecto clave para abordar estas restricciones de forma que la biblioteca no resultase una herramienta estática y que también existiese un compromiso entre flexibilidad y facilidad de uso. La extensión de la biblioteca con nuevas comprobaciones sigue encontrando una limitación en la dificultad de usar bibliotecas como las de *Clang*.

No obstante, nos gustaría hacer notar que la tarea de añadir nuevas comprobaciones por parte de cualquier profesor es facilitada por la estructura modular y clara de CAC++, el disponer de ejemplos de cómo otras comprobaciones funcionan y la propia documentación de las mismas.

6.2. Características de diseño

Como resultado del estudio de la sección anterior, se contemplaron las siguientes pautas para la implementación:

- **Generalización**: Las comprobaciones llevadas a cabo en cada práctica deben ser analizadas en profundidad y dejar paso a comprobaciones genéricas que abarquen cada uno de los casos específicos. De esta manera, se logra un *número manejable de comprobaciones* pero que a la vez conserva la potencia de derivar en comprobaciones más específicas. Por ejemplo, con la siguiente comprobación se puede verificar de manera general que un constructor de una clase tiene una lista de inicialización:

  – *listInitializerConstructor*(className, parameters, initializers, message);

  El profesor es quien debe obtener una comprobación específica indicando qué lista de inicialización debe tener un constructor particular de una clase determinada.
- **Variabilidad**: La generalización de las comprobaciones lleva a la necesidad de ofrecer variabilidad para que una misma comprobación pueda ser usada con diferentes propósitos (poder llevar a cabo comprobaciones específicas a la práctica). Esto se logra a través de *los parámetros de las comprobaciones*. Para que el profesor ajuste una comprobación a sus necesidades, se contemplan los siguientes parámetros:

  – *Detección de caso positivo o negativo*: El profesor puede querer comprobar que el alumno ha añadido un elemento o, por el contrario, asegurar que no lo ha añadido.
  – *Mensaje para mostrar*: Se puede adaptar al nivel de los alumnos para dar más o menos información. También se puede ofrecer el mensaje en diferentes idiomas.
  – *Elementos buscados*: Se contempla el uso de *expresiones regulares*. De esta manera, se puede indicar un nombre concreto para un elemento o bien una expresión regular de forma que se analicen varios elementos (aquellos que cuadren con la expresión regular). También se pueden emplear *comodines* para indicar que un elemento no se tenga en cuenta. Tanto el uso de expresiones regulares como de comodines es útil cuando no se conoce el nombre exacto o ciertas características de las declaraciones empleadas.

  Respecto a este último punto, si volvemos al fichero de comprobaciones generado en la Figura 2, podemos notar lo siguiente:

  – Según se contempló en el ejemplo de la Sección 4, podía existir dos posibles nombres para el método (`metodo_a_reutilizar` y `metodo_a_reusar`). No se provee el nombre exacto para el método 2, sino que se usa una expresión regular (metodo_a_reu.*ar) para abarcar las dos posibilidades .
  – Como no se especifica nada sobre si los métodos implicados deben ser constantes, se indica con el comodín “?” que no se tenga en cuenta a la hora de realizar la búsqueda.

- **Usabilidad**: En busca de una biblioteca intuitiva para el profesor, esta se desarrolla con el objetivo de facilitar el aprendizaje de la misma mediante la implementación de comprobaciones que guardan una correlación en su estructura, desde el orden de los parámetros hasta su funcionamiento interno, para que todas consideren las mismas situaciones. En resumen, existe una coherencia entre todas las comprobaciones para que el usuario las aplique por analogía con las demás, como por ejemplo en las siguientes comprobaciones:

  – *listInitializerConstructor*(className, parameters, initializers, message);
  – *explicitSpecifiedConstructor*(className, parameters, message);

  En ambas comprobaciones sobre constructores de una clase, el constructor se identifica con la misma tupla “(className, parameters)” en sus dos primeros parámetros.
- **Simplicidad**: Ofrecer variabilidad al profesor no debe tampoco entrar en conflicto con la simplicidad de la biblioteca. En ocasiones no tiene sentido ofrecer flexibilidad dentro de una comprobación cuando existen variantes que no tienen aplicabilidad. En definitiva, se opta por comprobaciones tan simples como sea posible acorde a la funcionalidad que se espera de las mismas.
- **Mantenibilidad**: La claridad en la estructura interna de la biblioteca era clave para su mantenibilidad y extensión. Por ello, se ha hecho hincapié en la obtención de una estructura modular, en la que se pueda observar las diferentes partes que conforman una comprobación. Esto facilitará la

actualización de las comprobaciones y la inserción de nuevas comprobaciones por comparación con otras existentes.

- **Documentación**: Por todas las características anteriores, la documentación juega un papel fundamental. Por esa razón, se ofrece una especificación de cada una de las comprobaciones, detallando su objetivo, parámetros y ejemplos de uso.

7. Conclusiones y trabajo futuro

En este artículo se presenta la biblioteca CAC++, un recurso docente que permite validar ciertas condiciones sobre el código elaborado por los alumnos en las prácticas de programación. Este recurso docente es actualmente empleado en una asignatura de programación orientada a objetos impartida en C++, pero se pone a disposición de cualquier profesor que imparta asignaturas similares, tanto para la evaluación de las prácticas como para el uso del alumno. Esta biblioteca, que incluye un surtido de comprobaciones, ha sido desarrollada con especial énfasis en su simplicidad de uso y en su mantenibilidad (para que pueda ser actualizada y extendida continuamente) gracias a su estructura. En especial, la creación de la biblioteca mejora el sistema anterior de programas de comprobaciones específicas para cada práctica al permitir la reutilización de comprobaciones para nuevas prácticas debido al proceso de generalización de las mismas.

Respecto al trabajo futuro, en la parte técnica se requiere de constante actualización de la biblioteca para refinar las comprobaciones actuales, añadir nuevas comprobaciones y adaptarla tanto a los nuevos estándares de C++ como a las nuevas versiones de *Clang*. En la parte de investigación en la docencia, se pretende hacer un seguimiento del uso y el beneficio que reporta esta biblioteca a los alumnos en asignaturas de programación orientada a objetos u otras asignaturas para las que se implementen nuevas comprobaciones.

Referencias

[1] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jarvinen. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, 3(1):245–262, 2004.

[2] Pedro Delgado-Pérez. Clang: un compilador de código abierto. Last access: 2016.12.02.

[3] Pedro Delgado-Pérez and Inmaculada Medina-Bulo. Automatización de la corrección de prácticas de programación a través del compilador Clang. In *Actas de las XXI Jornadas de Enseñanza Universitaria de Informàtica, Jenui 2015*, pages 311–318, Andorra la Vella, Julio 2015.

[4] Pedro Delgado-Pérez and Inmaculada Medina-Bulo. Diseño de un lenguaje específico de dominio para la corrección automatizada de prácticas de programación. In *XVII Congreso Internacional de Investigación Educativa: Investigar con y para la sociedad, AIDIPE 2015*, pages 1217–1224, Cádiz, Spain, 2015.

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

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

[7] Elias Penttilä. Improving C++ software quality with static code analysis. Master's thesis, Aalto University, School of Science, May 2014.

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

[9] Francisco P. Romero, Jesus Serrano-Guerrero, and Hernan Pérez de Inestrosa. Cuestor: Una nueva aproximación integral a la evaluación automática de prácticas de programación. In *Actas de las XVI Jornadas de Enseñanza Universitaria de Informática, Jenui 2010*, pages 493–500, Santiago de Compostela, 2010.

[10] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. Fully automatic assessment of programming exercises. *SIGCSE Bull.*, 33(3):133–136, June 2001.