Aprendizaje automático con programación genética gramatical para la detección de patrones de diseño

Rafael Barbudo, José Raúl Romero, Sebastián Ventura
Dpto. de Informática y Análisis Numérico, Universidad de Córdoba
{rbarbudo, jrromero, sventura}@uco.es

***Resumen*—Las técnicas de aprendizaje automático han sido ampliamente utilizadas en diversos dominios. En el caso del desarrollo de software, la exploración de repositorios de código puede ayudar a descubrir buenas prácticas empleadas por otros desarrolladores. En este contexto, la detección automática de patrones de diseño puede generar múltiples beneficios relacionados con la mantenibilidad y la escalabilidad del software. Estos patrones son soluciones generales y reutilizables, aplicables a la resolución de un problema de diseño. No obstante, la falta de documentación suele dificultar su trazabilidad, provocando que sus implementaciones se pierdan entre miles de líneas de código.**

Este trabajo propone un modelo de aprendizaje en dos fases para la detección automática de patrones de diseño. En primer lugar, un algoritmo de programación genética gramatical extrae aquellas propiedades que mejor describen al patrón. El uso de la gramática dota a la propuesta de una gran flexibilidad. A continuación, se construye el clasificador que permita detectar las implementaciones de dichos patrones. La propuesta ha sido empíricamente validada para tres patrones. Además, los resultados demuestran la competitividad del modelo frente a las propuestas actuales.

***Keywords*—detección de patrones de diseño, programación genética gramatical, clasificación asociativa, ingeniería inversa**

I. Introducción

Un patrón de diseño (DP, *Design Pattern*) [1] es una solución efectiva y reutilizable, aplicable a la resolución de un determinado problema de diseño del software. Estos patrones no son soluciones definitivas que puedan ser directamente codificadas, pero proporcionan una plantilla sobre cómo solucionar dicho problema. Su uso da lugar a una serie de beneficios como (1) proporcionar un lenguaje y un marco de desarrollo común; (2) mejorar la legibilidad y la mantenibilidad del código; y (3) aportar soluciones probadas y demostradamente útiles. Esto ha propiciado que su aplicación se haya estandarizado para cualquier tipo de sistema software. A pesar de ello, los DPs no suelen aparecer explícitamente documentados y, por lo tanto, sus implementaciones quedan ocultas entre miles de líneas de código. Desde la perspectiva del mantenimiento del software, la identificación de estos patrones puede aportar varios beneficios. Por ejemplo, puede facilitar la comprensión y reusabilidad del código. En este contexto, la detección de patrones de diseño (DPD) se posiciona como una tarea de gran importancia en el campo de la ingeniería inversa del software. Dado que la inspección manual de código es una labor subjetiva que requiere de un gran esfuerzo, la definición de métodos automáticos ha recibido un gran interés por la comunidad científica.

La mayoría de propuestas actuales tienen un punto en común: en mayor o menor medida, todas requieren de la inyección de conocimiento sobre la estructura y propiedades del DP a detectar. Este conocimiento representa la visión particular de un grupo de expertos y podría no ser suficiente para detectar las implementaciones o versiones realizadas por otros desarrolladores. Para paliar este problema, algunos métodos han introducido cierto grado de flexibilidad a la hora de llevar a cabo la detección. En este contexto, varios autores han propuesto el uso de técnicas de aprendizaje automático (ML, *Machine Learning*) que, a diferencia de otras, permiten analizar las implementaciones ya conocidas para inferir cuáles son las propiedades que mejor las describen. Estos métodos analizan el código en base a, o bien, una serie de métricas software, o de propiedades estructurales y categóricas.

En este trabajo se propone un modelo de dos fases que aplica técnicas de computación evolutiva y ML en el ámbito de la DPD. La primera fase es responsable de aprender cuáles son las propiedades que mejor describen a un patrón. Estas propiedades pueden referirse a (1) métricas software, como el acoplamiento entre clases; (2) propiedades de comportamiento, como la invocación de métodos; y (3) propiedades estructurales, como la relación de herencia entre clases. El encargado de realizar este aprendizaje es un algoritmo de programación genética gramatical [2] (G3P, *Grammar Guided Genetic Programming*). Dicho conocimiento se representa como un conjunto de reglas de asociación (AR, *Association Rule*) cuya estructura es conforme a una gramática libre de contexto (CFG, *Context-Free Grammar*). A continuación, en la segunda fase, se construye el detector a partir de dichas reglas. Para ello, antes es necesario definir un mecanismo de poda que asegure el uso exclusivo de las mejores reglas, así como de una estrategia que defina cómo emplearlas para llevar a cabo la detección.

La propuesta ha sido empíricamente validada para 3 tipos de DPs (*Singleton*, *Adapter* y *Factory Method*). Además, los resultados obtenidos se han comparado con los de la herramienta MARPLE [3] para los datos del repositorio DPB [4] (*Design Pattern detection Benchmark Platform*), el cual contiene implementaciones de DPs de 9 proyectos reales. Los resultados experimentales demuestran como, además de la flexibilidad y escalabilidad que otorga la gramática, el modelo propuesto

Trabajo financiado por el Ministerio de Economía y Competitivad, proyecto TIN2017-83445-P

alcanza un gran rendimiento en términos de detección, siendo altamente competitivo frente a las propuestas actuales.

El resto del trabajo se organiza como sigue. La Sección II presenta el problema de la DPD. En la Sección III se presenta una visión general del modelo propuesto. A continuación, en la Sección IV, se describe la primera fase de la propuesta, es decir, el algoritmo de G3P para la extracción de ARs. La segunda fase, responsable de generar el detector, se explica en la Sección V. En la Sección VI la propuesta es validada empíricamente y comparada con otra propuesta actual. Por último, las conclusiones se recogen en la Sección VII.

II. Marco conceptual

Los patrones de diseño se clasifican en función del tipo de problema de diseño que solucionan [1]: los *patrones de comportamiento* se centran en las interacciones entre clases y objetos; los *patrones creacionales* proporcionan una forma de crear objetos mientras se oculta la complejidad del proceso; y los *patrones estructurales* simplifican la composición de clases y objetos. El *Adapter* es un ejemplo del último grupo, cuyo objetivo es el de permitir el uso de una clase ya existente cuya interfaz no coincide con la requerida por el sistema. Cada DP describe un número fijo de roles. Un rol representa una determinada tarea que debe desempeñar un elemento del patrón. En sistemas software orientados a objeto, estos roles son desempeñados por clases o por sus instancias, o incluso interfaces en el caso de lenguajes como Java. Por ejemplo, el *Adapter* define cuatro roles diferentes: *adaptee*, *target*, *client* y *adapter*. El rol *adaptee* es desempañado por la interfaz que necesita ser adaptada, mientras que *target* se refiere a la interfaz requerida por el sistema (*client*). Finalmente, el *adapter* define la correspondiente adaptación de los servicios proveídos por *adaptee* al formato requerido por *target*. Es importante destacar que estos roles podrían ser desempeñados por más de un elemento, dificultando el proceso de detección.

Un DP es una plantilla de una solución y, en consecuencia, pueden existir distintas implementaciones para un mismo patrón dependiendo del programador, requisitos del sistema, etc. Esto dificulta que una descripción estática de las propiedades de un patrón pueda detectar sus posibles variantes. En este contexto, un gran número de propuestas se basan en el uso de técnicas de *similarity scoring* [5], las cuales suelen representar la información estructural del patrón y del proyecto en forma de grafos. Por lo general, estos métodos buscan subestructuras que se correspondan con la del patrón dentro del grafo del proyecto. En contraposición, las técnicas de ML aprenden cuáles son las propiedades que mejor describen a dichos patrones mediante el análisis de repositorios de código. En [6], se construye un clasificador para llevar a cabo la detección mediante el análisis de una serie de implementaciones de patrones. Estos patrones se representan como vectores de características que están formados por *n*k* elementos, donde *n* es el número de métricas software consideradas y *k* es el número de roles. Por otra parte, MARPLE [3] usa propiedades estructurales y de comportamiento como entrada de las técnicas de ML. Como ya se mencionó, no existe ninguna propuesta que considere tanto métricas software, como ambos tipos de propiedades simultáneamente durante el aprendizaje.

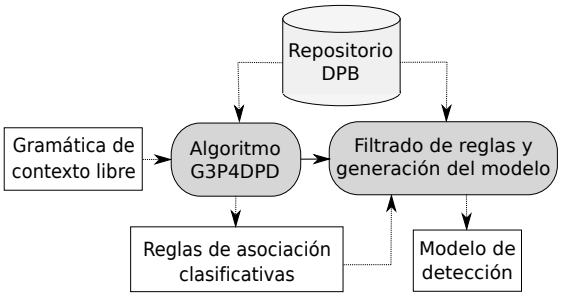


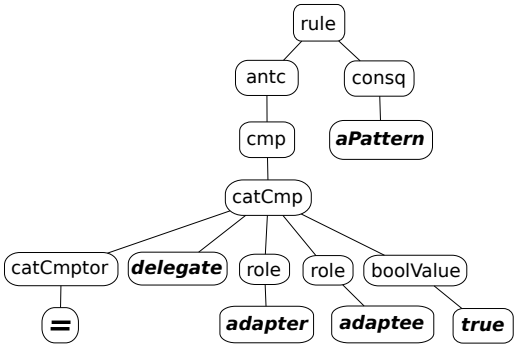
Figura 1: Modelo para la detección de patrones de diseño

III. Modelo para la DPD

En este trabajo se presenta un modelo en dos fases para la DPD que combina el análisis de métricas software y de propiedades estructurales y de comportamiento del código (véase la Figura 1). El objetivo de la primera fase es aprender el conjunto de propiedades software que mejor describen al DP que se está estudiando. Para ello, se analiza el repositorio de código que alberga implementaciones de diversos patrones. Más concretamente, se ha empleado como caso de estudio el repositorio DPB ya que, a diferencia de otros repositorios como P-mart [7], DPB no solo contiene implementaciones correctas de patrones (instancias positivas), sino también implementaciones que no deberían ser consideradas patrones (instancias negativas). Estas últimas son similares a los patrones y, por lo tanto, podrían dar lugar a una detección errónea.

Un algoritmo basado en G3P (G3P4DPD) se encarga de aprender las propiedades que mejor describen a un DP y a sus variantes. Este algoritmo genera un conjunto de reglas conformes a la CFG y las evoluciona aplicando una serie de operadores genéticos. Esta CFG declara la sintaxis del lenguaje que permite expresar las propiedades del software y las restricciones que describen dichas instancias en forma de ARs [8]. Sea I = $\{i_1, ..., i_n\}$ un conjunto de *items*, una regla de asociación es una implicación del tipo $A \rightarrow C$ donde A $\subset$ I, C $\subset$ I, y A $\cap$ C = $\emptyset$. El uso de esta CFG dota a la propuesta de una gran flexibilidad al permitir combinar propiedades software de diferente tipo como métricas software y propiedades estructurales y de comportamiento. Las reglas, que se obtienen en base a esta gramática se denominan reglas de asociación clasificativas [9] (CAR, *Class Association Rule*). Una CAR es un tipo de AR cuyo consecuente se limita a identificar la clase (implementar o no un DP).

Debido a su carácter descriptivo, estas reglas no se pueden utilizar directamente para llevar a cabo la detección. Además, dado que su número puede llegar a ser muy elevado, produciendo reglas redundantes y de baja calidad, hay que garantizar que únicamente se empleen las mejores reglas. Para ello se ha integrado un método de filtrado conocido como *database coverage* [9]. Por otra parte, es necesario definir una estrategia que establezca cómo utilizar dichas reglas para determinar si



(a) Ejemplo de genotipo

(= delegate adapter adaptee true) → *(aPattern)*

(b) Ejemplo de fenotipo

Figura 2: Ejemplo de representación de un individuo

<rule> ::= <antc> <consq>
<antc> ::= <cmp> | *and* <antc> <cmp>
<cmp> ::= <numCmp> | <catCmp>
<numCmp> ::= <numCmptor> <numOp> <role> *const*
<catCmp> ::= <catCmptor> *isFinal* <role> <boolValue>
| <catCmptor> *typeOf* <role> <typeOfValue>
| ...
<numCmptor> ::= ≥ | ≤ | > | <
<catCmptor> ::= = | !=
<numOp> ::= *DIT* | *NOC* | *CBO* | *NOM*
<role> ::= *adapter* | *adaptee* | *target*
<boolValue> ::= *true* | *false*
<typeOfValue> ::= *class* | *absClass* | *intface* | *enum*
...
<consq> ::= *aPattern* | *notAPattern*

Figura 3: Reglas de producción de la gramática

una nueva instancia implementa o no un patrón. Para esta tarea se ha utilizado la estrategia definida por el algoritmo de clasificación asociativa CMAR [10].

IV. Algoritmo de G3P para la generación de AR

En esta sección se describe en detalle el algoritmo para la extracción de CARs. En primer lugar, se explica la CFG y la codificación de los individuos.

IV-A. Codificación de los individuos

Cada individuo del evolutivo se compone de dos elementos: (1) un genotipo, expresado en forma de árbol sintáctico, y (2) un fenotipo que representa la CAR. La Figura 2a muestra un genotipo de ejemplo, mientras que la Figura 2b muestra su correspondiente fenotipo, esto es, la regla.

La Figura 3 muestra las reglas de producción de la CFG, donde los símbolos no terminales se representan entre <..> y los terminales en cursiva. El símbolo raíz, a partir del cual se inicia el proceso de derivación, es <*rule*> y se deriva en <*ant*> y <*consq*>, los cuales representan al antecedente y consecuente de la regla, respectivamente. El antecedente está formado por una serie de comparaciones (<*cmp*>) que pueden ser numéricas (<*numCmp*>) o categóricas (<*catCmp*>). Independientemente del tipo, cada comparación está formada por un comparador, un operador, un conjunto de argumentos y un valor. Más concretamente, las comparaciones numéricas están formadas por un comparador numérico (>, <, ≥ o ≤), un operador numérico (p.ej. *NOC*), un rol como argumento del operador, y un valor constante (*const*). Nótese como estos operadores se corresponden con métricas software. Un ejemplo de comparación numérica podría ser "`NOC(target)<1`", la cual indica que la clase que desempeña el rol *target* no tiene hijos (subclases). Por otra parte, las comparaciones categóricas se componen de un comparador categórico (= o !=), un operador categórico (p.ej. *typeOf*), un número variable de roles como argumentos, y un posible valor asociado a dicho operador. En este caso, un ejemplo podría ser "`typeOf(target)=intface`", la cual indica que el elemento que desempeña el rol *target* es una interfaz. El conjunto completo de operadores se recoge en la Tabla I. En relación al consecuente (<*consq*>), este solo puede ser derivado en *aPattern* o *notAPattern* para indicar si la regla hace referencia o no a un DP, respectivamente.

Es importante destacar como la mayoría de operadores categóricos se basan en los patrones de diseño elementales (en inglés, *elemental design patterns*) [11], micro patrones (en inglés, *micro patterns*) [12] y pruebas de patrones de diseño (en inglés, *design pattern clues*) [3], los cuales pueden ser vistos como operadores categóricos que comprueban si un fragmento de código satisface o no una determinada propiedad en base a valores de *true* o *false*. Por ejemplo, *Abstract Interface* es un patrón de diseño elemental que comprueba si un artefacto del código es abstracto o no. En algunos lenguajes como Java, un artefacto no solo puede ser una clase abstracta o concreta sino también una interfaz. Esto se podría solucionar añadiendo diferentes operadores como *isAbstract*, *isConcrete* o *isInterface*. No obstante, los predicados resultantes estarían fuertemente correlados y podrían introducir ruido al proceso de aprendizaje. En este contexto, la CFG permite el uso de operadores categóricos que pueden devolver múltiples valores. Por ejemplo, *typeOf* analiza un artefacto del código y devuelve un valor diferente según se trate de una clase concreta o abstracta, una interfaz o una enumeración. Este operador englobaría a los anteriores y, además de solucionar los problemas ya citados, su uso da lugar a reglas más compactas e interpretables.

IV-B. Descripción del algoritmo

El Algoritmo 1 muestra el esquema general del evolutivo. Como se puede apreciar, recibe cinco entradas: el número de iteraciones (*maxGen*), el tamaño de la población (*popSize*), el número de reglas a devolver (*extPopSize*), la gramática (*grammar*) y el repositorio de patrones (*repo*). Este repositorio está compuesto por un conjunto de instancias de DPs (positivas y negativas) y su código fuente. Cada una de estas instancias define los elementos que componen el patrón, así como los roles que desempeñan. El algoritmo devuelve una población externa (*extPop*) compuesta por las mejores reglas.

Tabla I: Operadores numéricos y categóricos

Operadores numéricos	
Signatura	**Descripción**
NOM(R_1)	Número de métodos de R_1
NOC(R_1)	Número de subclases directas de R_1
DIT(R_1)	Profundidad en el árbol de herencia de R_1
RFC(R_1)	Número de métodos que pueden ser invocados cuando una instancia de R_1 recibe un mensaje
Operadores categóricos	
Signatura	**Descripción**
isSubclass(R_1)	*true* si R_1 es una subclase, *false* en caso contrario
isFinal(R_1)	*true* si R_1 no puede ser heredado
controlledInit(R_1)	*true* si R_1 se instancia así mismo dentro de un bloque *if* o *while*
staticField(R_1)	*true* si R_1 tiene un campo estático
staticFlag(R_1)	*true* si R_1 tiene un campo estático y booleano
conglomeration(R_1)	*true* si R_1 declara algún método que llame al menos a otros 2 métodos de R_1
returned(R_1,R_2)	*true* si algún método declarado en R_1 devuelve un elemento del tipo R_2
received(R_1,R_2)	*true* si algún método declarado en R_1 recibe un elemento del tipo R_2 como argumento
createObj(R_1,R_2)	*true* si R_1 instancia a R_2
delegate(R_1,R_2)	*true* si R_1 invoca algún método de R_2
sameElem(R_1,R_2)	*true* si R_1 y R_2 son el mismo artefacto
typeOf(R_1)	Devuelve el tipo del artefacto que implementa a R_1 (*class*, *absClass*, *enum* o *intface*)
linkMethod(R_1,R_2)	Devuelve *directOver*, *indirOver*, *directImpl* o *indirImpl* si R_1 directa o indirectamente sobrescribe o implementa un método de R_2, *noLink* en cualquier otro caso
linkArtefact(R_1,R_2)	Devuelve *directInherit*, *indirInherit*, *directImpl*, *indirImpl* si R_1 directa o indirectamente extiende o implementa a R_2, *noLink* en cualquier otro caso
ctorVisibility(R_1)	Devuelve la visibilidad del constructor menos restrictivo de R_1, i.e. *private, protected, package o public*
aggregation(R_1,R_2)	Devuelve información de un atributo del tipo de R_2 declarado en R_1 en términos de su visibilidad y de su instanciabilidad.
adapter(R_1,R_2,R_3)	Devuelve si un método declarado (*decl*) o heredado (*inhr*) de R_1, implementado de R_3, delega en un método de R_2, *noLink* en cualquier otro caso

Se comienza generando, de manera aleatoria, *popSize* individuos en base a las restricciones definidas por *grammar*. La población externa (*extPop*) es inicializada al conjunto vacío. A continuación, se evalúan los individuos de *pop* en base al soporte (Ecuación 1). Esta métrica mide la frecuencia de aparición de una determinada regla en el conjunto de datos, siendo *t* el conjunto de transacciones en el conjunto de datos *T* que contiene al *itemset X*.

$$supp(X) = \frac{|\{t \in T; X \subseteq t\}|}{|T|} \quad (1)$$

A partir de este punto, el algoritmo, a lo largo de *maxGen* iteraciones, evoluciona a los individuos mediante la aplicación de operadores genéticos. En primer lugar, el operador de selección escoge *popSize* individuos de la unión entre *pop* y *extPop*. A continuación, el operador de cruce es aplicado, con una probabilidad prefijada, sobre pares de individuos para combinar sus genotipos. Por último, se aplica uno de dos posibles operadores de mutación. La selección se realiza de manera aleatoria, siendo ambos operadores equiprobables.

Algoritmo 1: Extracción de reglas de asociación

```
Entrada: maxGen, popSize, extPopSize, grammar, repo
Salida  : extPop
pop ← generateRules(popSize, grammar)
extPop ← ∅
evaluate(pop, repo)
while generation < maxGen do
    pop ← select(pop ∪ extPop, popSize)
    pop ← crossover(pop)
    if random() < 0,5 then
        pop ← diversityMutator(pop);
    else
        pop ← dpdMutator(pop);
    end
    evaluate(pop, repo)
    extPop ← update(pop ∪ extPop, extPopSize)
    generation++
end
```

El primero (*diversityMutator*) es un operador genérico que promueve la diversidad entre los individuos, mientras que el segundo (*dpdMutator*) es específico del dominio de la DPD y busca generar reglas que cubran todas las instancias. Tras su aplicación, *extPop* es actualizada con las mejores reglas.

La población externa se actualiza uniendo y filtrando a los individuos de la población actual (*pop*) y de la población externa de la generación anterior. En primer lugar, se ordenan las reglas en base a su confianza. La confianza (Ecuación 2) mide con qué frecuencia se cumple el consecuente (*C*) en aquellas reglas en las que además se cumple el antecedente (*A*). A continuación, estas reglas son filtradas en base al concepto de precedencia [9], esto es, dadas dos reglas R_1 y R_2, se dice que R_1 precede a R_2 si y sólo si: (1) conf(R_1) > conf(R_2); (2) conf(R_1) = conf(R_2) y supp(R_1) > supp(R_2); o (3) conf(R_1) = conf(R_2), supp(R_1) = supp(R_2) y el número de atributos de R_1 es menor que el de R_2.

En base a esto, si una regla A → C precede a otra A' → C' y A ⊂ A', entonces A' → C' es descartada. A continuación, se eliminan aquellas reglas con bajo soporte y confianza. Además, a partir del valor del estadístico χ^2 (Ecuación 3), se realiza un análisis estadístico para comprobar si el antecedente y consecuente de la regla están positivamente correlados. Así se garantiza que las reglas presenten implicaciones fuertes.

$$conf(A \rightarrow C) = \frac{supp(A \cup C)}{supp(A)} \quad (2)$$

$$\chi^2 = \frac{n * (lift(R) - 1)^2 * supp(R) * conf(R)}{(conf(R) - supp(R)) * (lift(R) - conf(R))} \quad (3)$$

$$donde \quad lift(A \rightarrow C) = \frac{supp(A \cup C)}{supp(A) * supp(C)}$$

IV-C. Operadores genéticos

El selector escoge a los individuos que se utilizarán para generar otros nuevos. Más concretamente, se aplica un torneo binario, que toma aleatoriamente dos individuos y mantiene aquel con un mayor valor de *fitness*. Este proceso se repite hasta que se seleccionen *popSize* individuos.

A continuación, el operador de cruce selecciona, aleatoriamente, una comparación de los genotipos de dos individuos y las intercambia. Para localizar una comparación, el genotipo es recorrido en pre-orden hasta que se encuentra el símbolo <*cmp*>. El final de dicha comparación se alcanza cuando se encuentra uno de los siguientes símbolos: *and*, <*cmp*> o <*consq*>. Por último, se devuelven los dos mejores individuos entre el conjunto de padres e hijos.

Finalmente, se aplica uno de los operadores de mutación. El primero (*diversityMutator*) selecciona un número de comparaciones y las reconstruye aleatoriamente desde <*cmp*>. Este número se elige mediante una ruleta aleatoria que promueve las pequeñas modificaciones. Por otra parte, el segundo operador (*dpdMutator*) recorre todas las comparaciones y las niega con una determinada probabilidad, la inversa del número de comparaciones. Para negar una determinada comparación, el comparador se remplaza por su opuesto (p.ej. $>$ cambiaría a $\leq$). Además, el terminal encargado de describir la clase también puede ser intercambiado con una probabilidad de 0,5. Este operador permite obtener reglas que describan tanto a la clase positiva como a la negativa. Así, por ejemplo, si la comparación "`typeOf(target)=intface`" describe una propiedad que ocurre frecuentemente en las instancias positivas, es probable que "`typeOf(target)!=intface`" describa a las negativas. Independientemente del operador aplicado, al final se devuelve el mejor individuo (padre o hijo).

V. Construcción del modelo de detección

Como se mencionó en la Sección III, en primer lugar se filtran las reglas de acuerdo al método de *database coverage*. Este filtro fue introducido inicialmente en el algoritmo CBA [9], y posteriormente utilizado en otros. Este método parte del conjunto de CARs, en orden descendente de precedencia, y del repositorio utilizado para generarlas. Para cada regla se realiza un pase sobre las instancias para encontrar a todas aquellas para las que se cumple el antecedente. Si la regla es capaz de clasificar correctamente al menos a una, se añade y se incrementa el contador de todas las instancias que cubre. Además, cuando una instancia es cubierta por un número de reglas, ésta se descarta. En el caso de que la regla no clasifique correctamente a ninguna instancia, ésta se elimina. Este proceso se repite hasta que todas las instancias hayan sido cubiertas o hasta que no queden más reglas.

Tras filtrar las reglas, es necesario definir la estrategia que empleará el modelo de detección para, a partir de las CARs restantes, identificar si un conjunto de clases está implementando o no un DP. Para este trabajo se ha empleado la estrategia definida por CMAR, de modo que cuando se recibe una nueva instancia se buscan todas aquellas reglas cuyo antecedente se satisface. Si todas tienen el mismo consecuente, se le asigna la etiqueta a la instancia. En caso contrario, las reglas se separan de acuerdo a sus consecuentes, y se calcula la χ^2 ponderada para determinar cuál es el grupo más representativo.

Tabla II: Configuración experimental

Parámetro	Valor
Número de generaciones	100
Tamaño de la población	100
Probabilidad de cruce	0,8
Número máximo de derivaciones	8
Umbral de soporte	0,01
Umbral de confianza	0,5
Umbral de χ^2	3,841
Umbral de cobertura	4

VI. Validación empírica

El algoritmo G3P4DPD se ha desarrollado en Java tomando como base el *framework* JCLEC [13]. Para la obtención de métricas software se ha empleado la librería ckjm[1] (*Chidamber and Kemerer Java Metrics*). Por otra parte, las propiedades estructurales y de comportamiento se han extraído usando las librerías *Java Parser*[2] y Javassist[3], las cuales extraen dichas propiedades del código fuente y del *bytecode*, respectivamente.

Las instancias usadas para validar la propuesta provienen del repositorio DPB, el cual ha sido creado por los autores de MARPLE y contiene instancias de 9 proyectos reales escritos en Java (p.ej. *Netbeans*). La Tabla II muestra la configuración empleada, donde los valores de los umbrales se han tomado de [10]. Es interesante destacar que el número máximo de derivaciones determina el tamaño del genotipo, es decir, de la regla. Este valor se ha fijado empíricamente en 8, ya que un valor superior difícilmente generaría reglas con un soporte admisible.

Nótese que la CFG ha sido parcialmente modificada para el estudio de cada DP. Cabe destacar que el uso exclusivo de los operadores más representativos por patrón reduce el espacio de búsqueda, mejorando la convergencia y minimizando el tiempo requerido para encontrar las mejores reglas. No obstante, experimentos previos han demostrado que el uso completo de la gramática no afecta al rendimiento del modelo de detección siempre que se incremente el número de generaciones. La Tabla III muestra qué operadores se han empleado para la detección de cada DP. Por una parte, los operadores categóricos se han seleccionado en base a su estructura y colaboraciones según lo definido por Gamma *et al.* [1]. Además, las micro-estructuras (en inglés, *micro-structures*), empleados por Zanoni *et al.* [3], también han sido usadas como referencia. Por otra parte, la selección de operadores numéricos para el *Factory Method* se ha realizado en base al estudio realizado por Issaoui *et al.* [14]. Para los demás casos, la selección de operadores se ha realizado en base a experimentos previos.

Antes de poder establecer un marco experimental, es necesario considerar que no existe ningún método que considere

[1]ckjm: *https://www.spinellis.gr/sw/ckjm*
[2]Java Parser: *https://javaparser.org*
[3]Javassist: *http://jboss-javassist.github.io/javassist/*

Tabla III: Operadores usados en la experimentación

	NOM	NOC	DIT	RFC	isFinal	isSubclass	controlledInit	staticField	staticFlag	conglomeration	returned	received	createObj	delegate	sameElem	typeOf	linkMethod	linkArtefact	ctorVisibility	aggregation	adapter
Singleton		✓	✓		✓		✓	✓	✓		✓		✓			✓			✓	✓	
Adapter		✓							✓		✓	✓	✓	✓		✓	✓	✓		✓	✓
F.Method	✓	✓	✓	✓		✓				✓	✓		✓		✓	✓	✓	✓		✓	

Tabla IV: Resultados experimentales

	Singleton	Adapter	Factory Method
Accuracy	$0,9389 \pm 0,0442$	$0,8554 \pm 0,0199$	$0,8337 \pm 0,0224$
Precision	$0,9283 \pm 0,0833$	$0,8401 \pm 0,0219$	$0,8157 \pm 0,0273$
Recall	$0,9136 \pm 0,0627$	$0,8797 \pm 0,0252$	$0,8954 \pm 0,0483$
Specificity	$0,9541 \pm 0,0571$	$0,8308 \pm 0,0261$	$0,7617 \pm 0,0483$
F_1	$0,9179 \pm 0,0559$	$0,8592 \pm 0,0194$	$0,8529 \pm 0,0226$

Tabla V: Rendimiento de MARPLE

	Singleton	Adapter	Factory Method
Accuracy	0,9281	0,8588	0,8189
F_1	0,9026	0,8478	0,8340

todos los tipos de propiedades durante el aprendizaje. Como se mencionó anteriormente, MARPLE es una herramienta con una gran relevancia en el área, si bien no considera métricas software. A fin de poder medir el rendimiento de la propuesta, se ha realizado una validación cruzada estratificada de 5 particiones. Además, dado que el algoritmo G3P4DPD es estocástico, se han realizado 30 ejecuciones con diferentes semillas aleatorias. Igualmente, se ha de tener presente que MARPLE particiona internamente los datos, impidiendo la reproducibilidad de la experimentación sobre las mismas particiones.

La Tabla IV recoge los resultados obtenidos para los tres DPs. Debido al espacio limitado, para cada medida, únicamente se muestra su media y su desviación estándar. Por otra parte, la Tabla V recoge los resultados de MARPLE [3]. Además de las ventajas de flexibilidad y escalabilidad ya mencionadas, la propuesta alcanza valores competitivos en términos de las medidas de evaluación analizadas. Los valores de desviación estándar sugieren que el evolutivo es robusto y sus resultados están poco condicionados por la semilla aleatoria. También es interesante destacar como se alcanzan valores muy equilibrados entre las diferentes medidas.

VII. Conclusiones y trabajo futuro

Este trabajo propone un modelo de dos fases para la detección automática de patrones de diseño que combina técnicas de computación evolutiva y aprendizaje automático. En primer lugar, el algoritmo G3P4DPD extrae las propiedades que mejor describen a un patrón mediante el estudio del código. Este conocimiento se representa como un conjunto de reglas de asociación conformes a una gramática. A partir de estas reglas, en una segunda fase, se construye el modelo de detección. El uso de la gramática dota a la propuesta de una gran flexibilidad ya que permite el estudio simultáneo de métricas software y otras propiedades estructurales y de comportamiento.

La propuesta ha sido empíricamente validada para tres patrones del repositorio DPB. Los resultados obtenidos han sido comparados con los de la herramienta MARPLE mostrando que, además de los beneficios cualitativos, la propuesta es competitiva y robusta en base a medidas objetivas de evaluación como *accuracy* y F_1. Así pues, el estudio conjunto de métricas software, así como de propiedades estructurales y de comportamiento resulta de gran interés al describir las implementaciones de los patrones.

Como trabajo futuro se planea incorporar nuevos operadores que den soporte a un mayor número de patrones de diseño y que mejoren los resultados de los ya soportados. Además, podría ser interesante integrar este modelo con IDEs existentes como Eclipse, y agregar capacidades para recomendar el desarrollo de un determinado patrón en un punto concreto del código de acuerdo con el conocimiento aprendido.

Referencias

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] R. I. Mckay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'Neill, "Grammar-based genetic programming: A survey," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 365–396, Sep. 2010.

[3] M. Zanoni, F. Arcelli Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *J. Syst. Softw.*, vol. 103, no. C, pp. 102–117, May 2015.

[4] F. A. Fontana, A. Caracciolo, and M. Zanoni, "DPB: A benchmark for design pattern detection tools," in *2012 16th European Conference on Software Maintenance and Reengineering*, March 2012, pp. 235–244.

[5] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, Nov. 2006.

[6] A. Chihada, S. Jalili, S. M. H. Hasheminejad, and M. H. Zangooei, "Source code and design conformance, design pattern detection from source code by classification approach," *Appl. Soft Comput.*, vol. 26, no. C, pp. 357–367, Jan. 2015.

[7] Y.-G. Guéhéneuc, "P-mart: Pattern-like micro architecture repository," *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*, 2007.

[8] J. M. Luna, J. R. Romero, and S. Ventura, "Design and behavior study of a grammar-guided genetic programming algorithm for mining association rules," *Knowl. Inf. Syst.*, vol. 32, no. 1, pp. 53–76, Jul. 2012.

[9] B. Liu, W. Hsu, and Y. Ma, "Integrating classification and association rule mining," in *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, ser. KDD'98. AAAI Press, 1998, pp. 80–86.

[10] W. Li, J. Han, and J. Pei, "CMAR: Accurate and efficient classification based on multiple class-association rules," in *Proceedings of the 2001 IEEE International Conference on Data Mining*, ser. ICDM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 369–376.

[11] J. M. Smith, *Elemental Design Patterns*, 1st ed. Addison-Wesley Professional, 2012.

[12] J. Y. Gil and I. Maman, "Micro patterns in Java code," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 97–116.

[13] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "JCLEC: a Java framework for evolutionary computation," *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, vol. 12, no. 4, pp. 381–392, 2008.

[14] I. Issaoui, N. Bouassida, and H. Ben-Abdallah, "Using metric-based filtering to improve design pattern detection approaches," *Innov. Syst. Softw. Eng.*, vol. 11, no. 1, pp. 39–53, Mar. 2015.