UNIVERSIDAD DE CÓRDOBA

Máster en Sistemas Inteligentes

Identificación de arquitecturas software basadas en componentes mediante Programación Evolutiva

---

Aurora Ramírez Quesada

Directores:

Prof. Dr. José Raúl Romero Salguero

Prof. Dr. Sebastián Ventura Soto

Córdoba, 05 de julio de 2013

Índice de contenido

Índice de figuras

Índice de tablas

Resumen

La construcción de sistemas software de calidad constituye uno de los principales retos a los que se enfrentan los ingenieros informáticos en la actualidad, pues deben cumplir las expectativas marcadas por los destinatarios ajustándose al tiempo y coste planificado. La Ingeniería del Software, como método sistemático para el desarrollo del software, facilita esta labor permitiendo reducir fallos y fomentando su reutilización. El análisis arquitéctonico constituye una fase muy importante del diseño del software, pues en él se identifican las funcionalidades del mismo, así como sus relaciones, permitiendo obtener una visión global del sistema en una fase temprana de su desarrollo.

En este contexto, donde la experiencia del arquitecto es determinante, la obtención de métodos y herramientas semiautomáticos que apoyen en la toma de decisiones de diseño abre un nuevo marco para la aplicación de técnicas de Inteligencia Artificial. Este trabajo presenta un modelo de identificación de arquitecturas basadas en componentes mediante un algoritmo de Programación Evolutiva (EP), que simula la abstracción de modelos arquitectónicos a partir de otro tipo de información de análisis, como la presente en los diagramas de clases. Para ello se ha abordado la representación, evaluación y manejo de soluciones para ser procesadas adecuadamente por un algoritmo evolutivo. Los resultados obtenidos reflejan la posibilidad de "evolucionar" arquitecturas software para encontrar aquellas que mejor cumplen los criterios de diseño requeridos por los expertos.

Palabras clave:

Search-Based Software Engineering

Programación Evolutiva

Arquitecturas software basadas en componentes

Categorías ACM:

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search - *Heuristic methods*

D.2.11 [**Software Engineering**]: Software Architectures

Capítulo 1

Introducción

Los sistemas software constituyen una de las principales herramientas con las que abordar diariamente multitud de tareas tanto en el ámbito personal como profesional, de forma que constituyen un activo destacado en la sociedad actual. Por otro lado, a medida que se han logrado avances en la tecnología que los soportan (procesadores más rápidos, mayores capacidades de almacenamiento, distribución a través de Internet, etc.), estos sistemas han crecido tanto en complejidad como en coste.

Ante tales factores, el desarrollo del software debe estar enfocado no solo a la obtención de sistemas de calidad que satisfagan las necesidades para las cuales son ideados, sino a conseguirlo mediante procedimientos que optimicen aspectos como el tiempo de desarrollo y los recursos empleados [1]. En este dominio, los problemas a resolver están fuertemente vinculados a decisiones humanas que los expertos en modelado y construcción de sofware deben acometer, especialmente, en las fases de análisis, diseño y pruebas.

Dentro de estos mecanismos adquieren gran relevancia las primeras decisiones que se toman sobre el software. Estas cuestiones son formuladas habitualmente durante el análisis arquitectónico, mediante el cual se realiza una abstracción de los bloques elementales que conforman el software. Esta tarea se realiza en las primeras fases de análisis, en la que se toman las decisiones que más repercuten en la calidad del producto, responsabilidad que recae casi de forma exclusiva en el equipo de ingenieros software.

En concreto, la identificación de componentes es una de sus principales actividades, pues en ella se aborda el desarrollo de sistemas complejos mediante la determinación de los bloques constituyentes del software y sus funcionalidades, enfoque útil cuando se trabaja

con sistemas complejos [2]. Se trata pues de una tarea de gran importancia para la que se han creado desde recomendaciones generales hasta métodos sistemáticos [3], pero donde la experiencia y la habilidad del arquitecto siguen jugando un papel determinante.

Sin embargo, es habitual que, ante sistemas aparentemente pequeños o cuando se dispone de una planificación ajustada, se aborden directamente modelos de análisis más cercanos a la implementación. Entonces, a medida que las funcionalidades del sistema aumentan o este debe ser modificado, la concepción original de la funcionalidad del software se difumina y los modelos se vuelven demasiado complejos, especialmente para personas que se incorporan al proyecto para abordar tareas de mantenimiento. Realizar un proceso de identificación de componentes en este momento puede ser un enfoque útil para descubrir las distintas funcionalidades que ofrece actualmente el sistema software y cómo se estructuran e interaccionan sus elementos.

Disponer de herramientas de apoyo a la decisión que den soporte al ingeniero en esta actividad es un campo de trabajo interesante en el cual puede resultar beneficiosa la inclusión de técnicas de Inteligencia Artificial (IA) que hagan uso del conocimiento de los expertos para automatizar el proceso. En este sentido, el crecimiento en los últimos años del área denominada *Search Based Software Engineering* (SBSE) [4] muestra el interés de la comunidad investigadora en el tratamiento de las tareas de la Ingeniería del Software como problemas de búsqueda y optimización, susceptibles de ser resueltas por técnicas computacionales. Aspectos como la planificación de proyectos [5] o el desarrollo de casos de prueba [6] han sido ya abordados en el área de SBSE haciendo uso de diferentes técnicas metaheurísticas, las cuales constituyen una alternativa eficiente cuando no es posible aplicar técnicas exactas [7].

La Computación Evolutiva (EC, *Evolutionary Computation*) es una de las metaheurísticas más conocidas y aplicadas, también en el ámbito de SBSE [8]. Inspirada en conceptos de la evolución natural, en ella se manejan simultáneamente varias soluciones al problema que son "evolucionadas" mediante la aplicación de operadores genéticos como el cruce y la mutación, los cuales simulan los cambios en las especies por la combinación y alteración del material genético. Se trata pues de un proceso iterativo en el cual se generan nuevas soluciones, también llamadas individuos, a partir de otras existentes, mediante la modificación de sus características. La Programación Evolutiva (EP, *Evolutionary Programming*) [9, 10], la cual puede considerarse como una rama de la Computación Evolutiva, se caracteriza por

la ausencia de operador de cruce y una representación de las soluciones diseñada específicamente para el problema.

Este trabajo presenta un modelo basado en un algoritmo de Programación Evolutiva para la identificación de componentes en arquitecturas software. A través de un proceso orientado al experto, se ha abordado la optimización de arquitecturas basadas en componentes a partir de modelos de análisis. Para ello se ha diseñado una representación flexible y comprensible, una función de evaluación que incluye criterios de diseño deseables y un conjunto de transformaciones que permiten explorar diferentes tipos de arquitecturas.

Los resultados iniciales obtenidos en el marco de la propuesta son interesantes y prometores, lo cual ha permitido la publicación de tres comunicaciones a congreso, uno internacional con categoría A en el índice CORE (*Genetic and Evolutionary Computation Conference, GECCO'13*) y dos nacionales (*IX Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, MAEB'13*, y *XVIII Jornadas en Ingeniería del Software y Bases de Datos, JISBD'13*), tal y como se detalla el Apéndice A.

La estructura del resto del documento es la siguiente. En el Capítulo 2 se analizan los antecedentes de este trabajo desde la persepctiva de SBSE. A continuación, el modelo propuesto es presentado en el Capítulo 3, incluyendo una descripción detallada del problema abordado. El Capítulo 4 describe los componentes que conforman el algoritmo evolutivo, mientras que la experimentación y los resultados obtenidos son recogidos en el Capítulo 5. Finalmente, las conclusiones y el trabajo futuro son discutidos en el Capítulo 6.

Capítulo 2

Estado del arte

2.1. *Search-Based Software Engineering*

Search-Based Software Engineering (SBSE) es un término propuesto por Harman y Jones en 2001 para denominar al área de investigación destinada a resolver tareas de la Ingeniería del Software mediante técnicas de optimización y búsqueda [11]. Por tanto, se plantea la unión de dos áreas de gran interés como son la Ingeniería del Software, en la cual se plantean actividades destinadas a la creación de software de calidad, y el de las metaheurísticas, que aporta diferentes técnicas para resolver problemas de forma eficiente cuando no es posible aplicar técnicas exactas [7].

En este sentido, las tareas de la Ingeniería del Software deben ser formuladas como problemas susceptibles de ser resueltos mediante técnicas de búsqueda y optmización [12], para lo cual suele ser necesario definir tres aspectos: una representación del problema, una función de evaluación para las soluciones y el conjunto de operadores que permitan al algoritmo moverse por el espacio de búsqueda.

El crecimiento del área en los últimos años [4] ha permitido expandir tanto el ámbito de la problemática a resolver como el de las técnicas aplicadas. Aunque la primera y más fructífera área en SBSE es la de las pruebas del software [13, 14], también se han destinado esfuerzos a problemas de predicción de costes [15] o al diseño [16]. Por otro lado, aunque la Computación Evolutiva continúa siendo una de las metaheurísticas más aplicadas [8], también han surgido trabajos que utilizan otras técnicas como la optimización basada en colonias de hormigas [17], la búsqueda tabú [18] o el enfriamiento simulado [19].

A pesar de los avances mostrados en este campo de investigación desde su concepción, aún existen retos que resolver. La complejidad de algunas de las tareas más importantes de la Ingeniería del Software, como las de análisis y diseño y, sobre todo, aquellas que están centradas en el conocimiento y experiencia de los expertos, hace que surja la necesidad de plantear nuevos modelos y algoritmos. En este sentido, la hibridación de técnicas [20, 21] y la inclusión de interactividad en el proceso de búsqueda [22, 23] se plantean como áreas emergentes de gran interés para la comunidad científica.

2.2. Optimización de arquitecturas software

El análisis arquitectónico del software constituye una de las tareas más importantes para manejar el desarrollo de sistemas software complejos. En ella se dedican esfuerzos para encontrar soluciones de diseño de alto nivel que cumplan ciertos requisitos de calidad como el desempeño, la cohesión o la reutilización, entre otros [24].

En este contexto, existen diferentes tareas que ya han sido abordadas mediante técnicas computacionales como el clustering [25] y, más recientemente, desde la perspectiva de SBSE [26]. Decisiones arquitectónicas como la selección de componentes [27], la replicación y despliegue de componentes basados en atributos como el rendimiento, el coste y la fiabilidad [28, 29] o la refactorización e introducción de patrones arquitectónicos [30] son algunos ejemplos de aplicación.

En la mayoría de propuestas de este ámbito se plantea la optimización arquitectónica como un proceso de toma de decisiones en el cual se exploran, mediante un algoritmo de búsqueda, algunas alternativas sobre los diseños iniciales que satisfagan los requisitos planteados por el experto.

Otra vertiente a considerar es la identificación de diseños arquitectónicos a partir de información a un nivel de abstracción diferente. En este sentido existen algoritmos para la recuperación de modelos arquitectónicos a partir de la información disponible en el código fuente [31] o la síntesis arquitectónica a partir de requisitos [32]. Este tipo de propuestas constituyen otro enfoque al problema, más centrado en las aptitudes "humanas" de los ingenieros, de forma que el mayor reto consiste en automatizar adecuadamente la capacidad de abstracción de los expertos para conseguir buenos diseños.

2.3. Programación Evolutiva

La Computación Evolutiva es una de las metaheurísticas mas conocidas y aplicadas en la resolución de problemas de búsqueda y optimización combinatoria [7]. Inspirada en conceptos de la evolución natural como la supervivencia de los individuos mejor adaptados al entorno o la presencia de mutaciones, es englobada dentro de las metaheurísticas basadas en poblaciones, donde se manejan simultáneamente varias soluciones candidatas del problema que son "evolucionadas" durante varias iteraciones o generaciones. Para ello se define una función de *fitness* que permite evaluar la calidad de las soluciones, también denominadas individuos, y que junto a operadores como el cruce y la mutación, destinados a crear nuevas soluciones, guía al algoritmo hacia la solución óptima [33]. Finalmente, los mecanismos de selección, para elegir los individuos a los que se les aplican los operadores, y de reemplazo, que determina qué individuos sobreviven en la siguiente generación, conforman el resto de componentes de este tipo de algoritmos.

La gran variedad en cuanto a esquemas de evolución, codificaciones y operadores ha permitido aplicar esta técnica a diferentes problemas de SBSE como la modularización del software mediante el algoritmo multiobjetivo NSGA-II [34], o la composición de servicios mediante Programación Genética [35].

La Programación Evolutiva fue propuesta por Fogel en los años 60 como un enfoque evolutivo para la Inteligencia Artificial [9, 10]. Aunque comparte la mayoría de los principios básicos de EC, presenta tres características que la diferencian de ella: la utilización de representaciones cercanas al dominio del problema, la ausencia de operador de cruce y un operador de selección determinista. Por tanto, la "evolución" de las soluciones recae fuertemente en el operador de mutación diseñado, así como en el mecanismo de reemplazo, el cual sí suele ser estocástico.

Capítulo 3

Modelo propuesto

3.1. Visión general del proceso de optimización

El proceso de identificación de componentes propuesto consiste en la abstracción de información de modelos de análisis, representados en este caso como diagramas de clases, para obtener posibles arquitecturas basadas en componentes que representen la funcionalidad de un sistema software. Para ello se propone la utilización de un algoritmo evolutivo mediante el cual se realiza un proceso de búsqueda eficiente, explorando el espacio de posibles soluciones (arquitecturas software) de manera "inteligente", esto es, profundizando en las zonas más prometedoras, evitando así realizar una búsqueda exhaustiva que implicaría un tiempo de cómputo prohibitivo.

La inclusión del algoritmo evolutivo como parte principal del método propuesto puede verse en la Figura 3.1, donde se muestra la secuencia de tareas que componen el proceso de identificación de componentes. Como entrada se recibe un modelo de análisis almacenado en el formato estándar XMI [36], el cual manejan la mayoría de herramientas de modelado que utilizan los ingenieros software y que, por tanto, permite al experto iniciar el proceso cómodamente. A continuación, se realiza la extracción de la información referida a las clases que intervienen en el modelo, así como las relaciones entre ellas. La herramienta de modelado utilizada ha sido Magic Draw [1], mientras que el proceso de extracción ha sido automatizado mediante un intérprete de XMI.

[1] http://www.nomagic.com/products/magicdraw.html

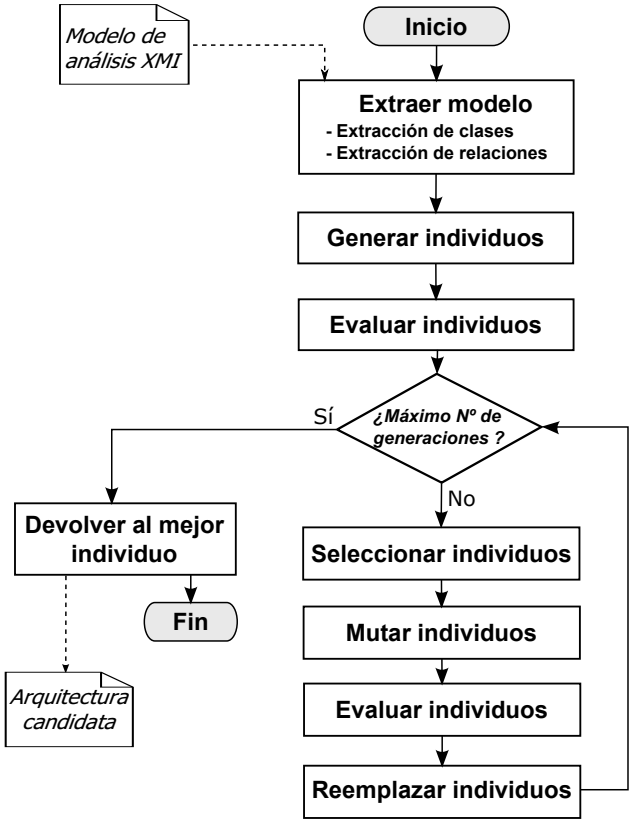


Figura 3.1: Diagrama de flujo del proceso de identificación de componentes

El método de búsqueda comienza con la ejecución del algoritmo evolutivo, iniciado con la creación de una población de n soluciones generadas aleatoriamente a partir de la información del modelo (véase la Sección 4.3). Estas soluciones deben ser evaluadas mediante una función de *fitness* que permitirá determinar su calidad, tal y como se detallará en la Sección 4.4. El resto del algoritmo es un proceso iterativo que finaliza cuando la condición de parada es alcanzada (un número máximo de generaciones en este caso). En cada una de ellas se crea una nueva población de soluciones a partir de los individuos de la generación anterior, que toman el rol de padres. Para ello, los operadores genéticos, detallados en la Sección 4.5, son aplicados sobre dichos individuos, generando otros nuevos. Tras la evaluación de las nuevas soluciones se dispone de n padres y n hijos entre los cuales debe seleccionarse aquellos que continuarán en la siguiente generación. El mecanismo de reemplazo, explicado en la Sección 4.6, establece los criterios de supervivencia entre padres e hijos. Finalmente, la mejor solución encontrada es devuelta, indicando la arquitectura resultante y los valores de las medidas consideradas.

3.2. Descripción del problema

El problema que aquí se plantea puede ser definido como la búsqueda de la configuración óptima de un conjunto de elementos (componentes, interfaces y conectores) que permitan representar la arquitectura de un sistema software. De forma más precisa, se trata de descubrir la existencia de estos elementos a partir de otro tipo de información, los modelos de análisis del software, de forma que de ellos se abstraiga un modelo arquitectónico basado en componentes. Estos modelos pueden ser procesados directamente desde herramientas de modelado con las que trabajan los ingenieros software.

En este punto resulta necesario definir adecuadamente algunos de los conceptos relacionados con el alcance del problema propuesto, el cual ha sido enfocado siguiendo la definición de Szyperski [2]. Un *componente* es «una unidad de composición que especifica contractualmente interfaces y solo hace explícitas dependencias del contexto». En este sentido, el modelo arquitectónico contempla la definición de las *interfaces*, que determinan el conjunto de operaciones que pueden ser invocadas y que permiten separar su especificación funcional de la implementación real. Dicha implementación es realizada en el interior del componente mediante otros elementos. Finalmente, los *conectores* enlazan dos o más puntos de interacción entre interfaces.

Las definiciones anteriores son recogidas en la especificación del estándar UML 2 [37], que ofrece una semántica bien definida para el análisis arquitectónico. Esta notación ha sido considerada en este trabajo a la hora de representar el problema, de forma que se utilizan los modelos y artefactos definidos por UML 2.

Por otro lado, las métricas disponibles para evaluar los diferentes modelos permiten establecer los criterios de calidad exigidos a las soluciones propuestas. Conceptos genéricos como la funcionalidad, reusabilidad, flexibilidad o comprensión del software son traducidos en medidas cuantificables y ampliamente extendidas como la presencia de abstracción, el uso de la herencia y el polimorfismo [24]. Concretamente, los conceptos de cohesión y acoplamiento están aceptados en el ámbito del diseño basado en componentes como criterios básicos y fundamentales de diseño. La cohesión se define como el grado en el cual el componente desarrolla una funcionalidad bien definida. El acoplamiento mide la interdependencia entre componentes a causa de las interacciones entre sus módulos y los flujos de datos.

Capítulo 4

Diseño del algoritmo evolutivo

4.1. Fenotipo y genotipo

La identificación de componentes constituye una tarea muy compleja, de forma que es conveniente realizar una descomposición en problemas de menor envergadura cuya resolución pueda ser abordada mediante técnicas computacionales. Por tanto, la resolución del problema es aquí formulada en términos de las siguientes consideraciones:

- Un componente es definido como un grupo cohesionado de clases, de forma que entre ellas llevan a cabo el comportamiento del componente de manera conjunta.
- Una relación navegable entre clases pertenecientes a diferentes componentes representa una interfaz candidata, pues establece una interacción entre tales componentes. En función de la navegabilidad de la relación, se considerará una interfaz requerida o proveída.
- Los conectores describen la unión de una pareja de interfaces, una requerida y otra proveída, entre dos componentes.

Los elementos anteriores son presentados gráficamente en la Figura 4.1, la cual muestra el fenotipo del problema, esto es, lo que representa la solución, aquí modelado con UML 2 pero que en realidad es tratado de forma genérica en formato XMI.

A continuación puede definirse el genotipo del problema, esto es, la representación interna del fenotipo tal cual es manejada por el algoritmo de búsqueda. Trasladar los conceptos

anteriores a una estructura de datos eficiente desde el punto de vista computacional es uno de los aspectos clave para el éxito de este tipo de algoritmos. A su vez, otro factor importante es que dicha representación sea interpretable por parte de usuarios no expertos en Computación Evolutiva, como puede ser el caso de los ingenieros software.

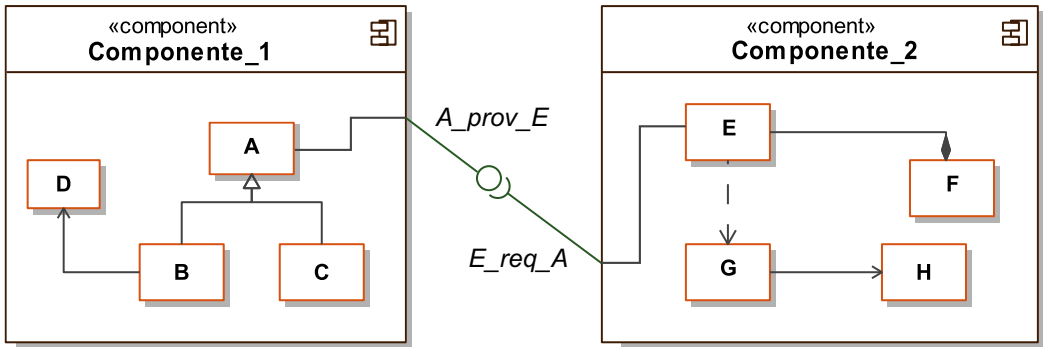


Figura 4.1: Fenotipo del problema

Este último aspecto es una de las principales diferencias de esta propuesta frente a otros trabajos basados en SBSE, donde la representación lineal de las soluciones, esto es, vectores de valores binarios o numéricos, ha venido siendo la codificación más habitual. Sin embargo, a medida que los problemas son más complejos, este tipo de codificación se complica y su interpretación se vuelve tediosa. La utilización de otro tipo de estructuras, especialmente si han sido aplicadas con éxito en otros ámbitos, se convierte en una alternativa interesante.

En este sentido, una de las representaciones habituales de los modelos de análisis es en forma de árbol, donde se establece una relación jerárquica entre los elementos que intervienen en el modelo así como referencias entre los nodos. La representación en árbol ofrece las dos características deseables, interpretabilidad y eficiencia, al tratarse de una estructura de datos comprensible y computacionalmente manejable.

La Figura 4.2 muestra la codificación en forma de árbol del problema representado en la Figura 4.1. El nodo raíz comprende el modelo arquitectónico completo, compuesto por un conjunto de componentes y conectores, los cuales se muestran sombreados para indicar que se trata de elementos prefijados. Estos, a su vez, son nodos no terminales del árbol, pues pueden ser descompuestos en otros de menor abstracción. Los componentes son definidos a partir del conjunto de clases que implementan su funcionalidad y las interfaces que requiere y provee. Por otro lado, los conectores se descomponen en la pareja de interfaces que enlazan. Finalmente, las clases e interfaces conforman el último nivel de descomposición

del árbol, de forma tal que las distintas agrupaciones que se hagan de ellas representarán arquitecturas de un número determinado de componentes y conectores.

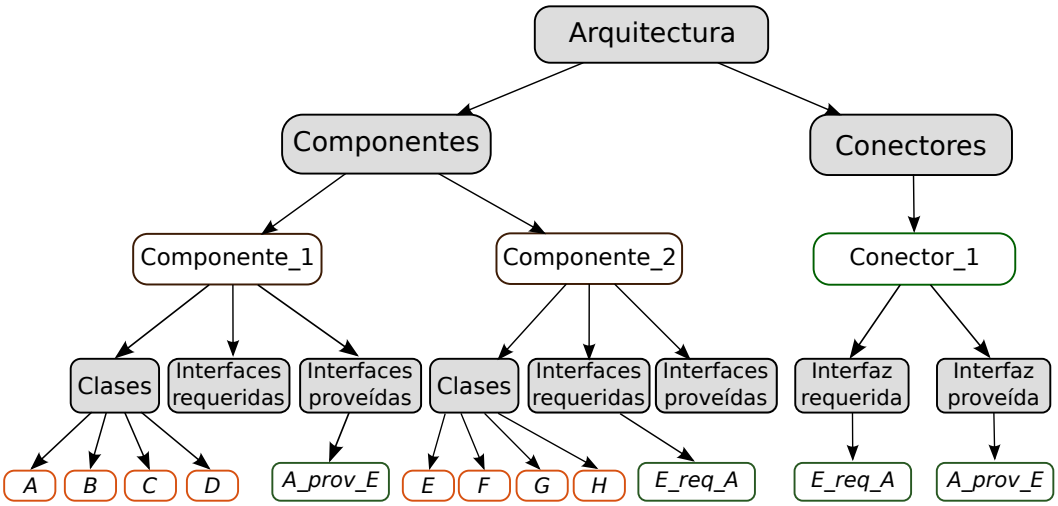


Figura 4.2: Genotipo del problema

4.2. Restricciones

El problema de optimización propuesto presenta varias restricciones que deben ser consideradas durante el proceso evolutivo. En primer lugar, cada una de las clases que componen el modelo inicial deben pertenecer a un único componente, proceso que es controlado en la inicialización del algoritmo.

Otros aspectos relacionados con la composición de los individuos resultantes también deben ser tenidos en cuenta. Por un lado, no pueden existir componentes vacíos, ya que no implementarían ninguna funcionalidad. Por otro lado, todos los componentes deben presentar, al menos, una interfaz que le permita interactuar con el resto de componentes. Finalmente, la presencia de componentes mutuamente dependientes, esto es, donde uno provee y requiere al otro (y viceversa), no son admitidas desde el punto de vista del análisis arquitectónico. Estas características tienen que ser consideradas no solo durante la creación de los individuos pertenecientes a la población inicial, sino que también han de ser cumplidas cuando se apliquen los operadores genéticos, tal y como se detalla en la Sección 4.5.

4.3. Inicialización de la población

Tras abordar la representación del problema, es posible concretar la forma en la que se crean las soluciones que conforman la población inicial del algoritmo. Este proceso consiste en la generación aleatoria de diferentes arquitecturas válidas a partir de la información extraída del diagrama de clases. Concretamente, las clases son distribuidas de manera aleatoria en un número, también aleatorio, de componentes, el cual varía entre un mínimo y un máximo configurable. Tras realizar este proceso, se determinan las interfaces y conectores establecidos entre los componentes y se comprueba el cumplimiento de las restricciones.

4.4. Función de *fitness*

La evaluación de los individuos está basada en criterios de diseño ampliamente extendidos en el desarrollo basado en componentes. La cohesión hace referencia a la obtención de componentes con funcionalidades bien definidas. Por otro lado, el acoplamiento indica la interrelación existente entre componentes. Finalmente, otro aspecto interesante a considerar es el tamaño de los componentes, directamente relacionado con la complejidad de los mismos. Estos tres conceptos han sido traducidos a tres métricas cuantificables sobre el problema propuesto, tal y como se detalla a continuación.

La cohesión global de la arquitectura se obtiene como la media de la cohesión calculada sobre cada componente (coh_i) mediante la Ecuación 4.1. Esta expresión se compone de dos términos ponderados que cuantifican dos características: el grado de unión entre las clases presentes en el interior del componente y el número de agrupaciones que lo conforman. El primero, que varía entre 0 y 1, contabiliza el número de relaciones establecidas entre las clases, clasificadas en función de los distintos tipos de relaciones UML y ponderadas en función de su importancia relativa: asociaciones (n_{as}, w_{as}), dependencias (n_{de}, w_{de}), agregaciones y composiciones (n_{ag}, n_{co}, w_{co}) y generalizaciones (n_{ge}, w_{ge}). El segundo término, definido en el rango [-1,0], representa una penalización por el número de agrupaciones (n_{gr}) en el interior del componente, fundamentado en el concepto de *componentes conexas* en la Teoría de Grafos y bajo la idea de que varias agrupaciones representan dispersión en cuanto a la funcionalidad del componente. En dicha ecuación, n_{cl} representa el número de clases en el interior del componente, mientras que los pesos presentes en ella han sido

fijados experimentalmente.

$$\begin{aligned} coh_i &= \frac{w_{c1}}{(w_{as}+w_{de}+w_{co}+w_{ge})\cdot(n_{cl}-1)} \cdot \Big[w_{as}\cdot n_{as} + w_{de}\cdot n_{de} + \\ &+ w_{co}\cdot(n_{ag}+n_{co}) + w_{ge}\cdot n_{ge}\Big] + w_{c2}\cdot\frac{1-n_{gr}}{n_{cl}-1} \end{aligned} \tag{4.1}$$

El siguiente término a considerar es el exceso de interrelaciones entre los componentes, lo cual se refleja en la medida de acoplamiento (*acopl*) propuesta en la Ecuación 4.2. En ella se contabilizan las relaciones existentes entre clases pertenecientes a distintos componentes que no son especificadas a través de las interfaces. Por ejemplo, esta situación puede ocurrir durante el proceso evolutivo ante una relación de generalización en la que la clase padre y las subclases no están ubicadas en el mismo componente. C indica el número de componentes, mientras que R, definido en la Ecuación 4.3, establece una penalización por la presencia de este tipo de relaciones. Concretamente, R acumula, para cada par de componentes i y j, el peso de la relación más fuerte entre las k relaciones ($k = 0...m$), $r^k_{i,j}$, establecidas entre ellos. La normalización de la métrica se consigue mediante el término R_{max}, pues representa la peor configuración posible (todos los componentes están mutuamente conectados por medio de la relación más fuerte). Por tanto, la medida toma valores entre 0 (no existen relaciones entre los componentes salvo las definidas a través de las interfaces) y 1 (todos los componentes están fuertemente acoplados).

$$acopl = \frac{1}{R_{max}} \cdot \frac{2\cdot R\cdot(C-2)!}{C!} \tag{4.2}$$

$$R = \sum_{i=1}^{C}\sum_{j=i+1}^{C} max\ r^k_{i,j} \tag{4.3}$$

El coeficiente de variación, CV, mide la dispersión en el tamaño de los componentes, definido como el número de clases que los conforman. Su cálculo se obtiene por medio de la Ecuación 4.4, donde μ y σ representan la media y la desviación estándar del tamaño de los componentes, respectivamente. Esta medida puede tomar valores entre 1 y 8.

$$CV = \frac{\sigma}{\mu} \tag{4.4}$$

Finalmente, la función de *fitness* es construída a partir de la suma ponderada de los tres términos anteriores. Puesto que el acoplamiento y el coeficiente de variación deben ser minimizados y esta última, a su vez, normalizada, se han aplicado las correspondientes transformaciones aritméticas con los límites superiores de cada medida, tal y como se observa en la Ecuación 4.5.

$$fitness = w_{coh} \cdot \frac{\sum_{i=1}^{n} coh_i}{n} + w_{acopl} \cdot (1 - acopl) + w_{cv} \cdot \left(\frac{8 - CV}{8}\right) \tag{4.5}$$

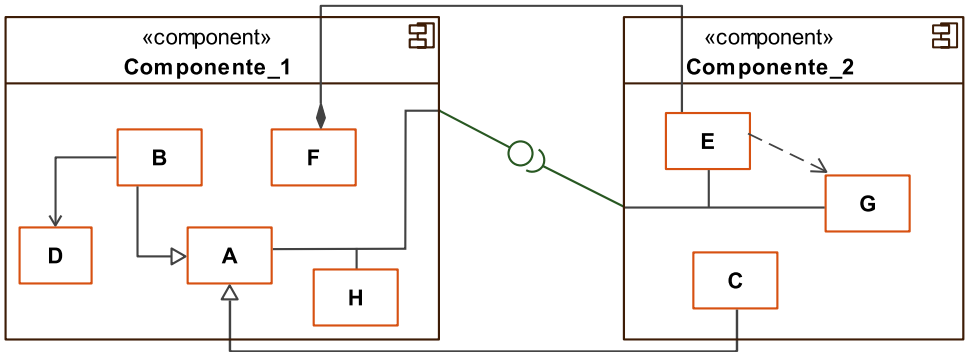


Figura 4.3: Individuo de ejemplo a evaluar por el algoritmo evolutivo

A modo de ejemplo, la Figura 4.3 muestra una solución no óptima de la arquitectura presentada en la Sección 4.1. El cálculo del valor de *fitness* es recogido en la Tabla 4.1, donde se ha considerado $w_{as} = 1$, $w_{de} = 1$, $w_{co} = 3$, $w_{ge} = 5$, $w_{c1} = 0,5$ y $w_{c2} = 0,5$. El primer componente presenta cinco clases, de las cuales dos, H y F, no están relacionadas con ninguna otra clase en el interior del componente. Entre el resto se establece una generalización y una asociación. El segundo presenta tres clases, dos agrupaciones y una relación de dependencia. En ambos casos la penalización por el número de grupos es alta mientras que el número de relaciones internas es baja, de forma que se alcanzan valores negativos para la cohesión. En cuanto al acoplamiento, puede verse cómo existen dos relaciones entre clases de ambos componentes, una generalización entre A y C y una composición entre E

y F. Como $w_{ge} = 5$ y $w_{co} = 3$, R toma el valor 5. A su vez, R_{max} también es igual a 5 pues la peor situación posible para dos componentes es que entre sus clases se establezca una relación de generalización (la de mayor peso). La media y la desviación estándar del número de clases por componente son fácilmente obtenidas. Finalmente, las tres medidas son combinadas según la configuración de los pesos $w_{coh} = 0,3$, $w_{aco} = 0,4$ y $w_{cv} = 0,3$.

Tabla 4.1: Cálculo del *fitness* sobre el individuo de ejemplo

Cohesión	**Acoplamiento**	**CV**
$n_{gr_1} = 3, n_{cl_1} = 5, n_{as_1} = 1, n_{ge_1} = 1$ $coh_{comp\_1} = -0,1750$ $n_{gr_2} = 2, n_{cl_2} = 3, n_{de_2} = 1$ $coh_{comp\_2} = -0,2250$ $coh = \frac{-0,1750+(-0,2250)}{2} = -0,2000$	$r^1_{1,2} = w_{co} = 3$ $r^2_{1,2} = w_{ge} = 5$ $R = max(3,5) = 5$ $R_{max} = w_g = 5$ $acopl = \frac{1}{5} \cdot \frac{2 \cdot 5 \cdot 1}{2} = 1$	$\mu = \frac{5+3}{2} = 4$ $\sigma = \sqrt{(5-4)^2 + (3-4)^2} = 1,4142$ $CV = \frac{1,4142}{4} = 0,3536$
$fitness = 0,3 \cdot (-0,2) + 0,4 \cdot (1-1) + 0,3 \cdot \left(\frac{8-0,3536}{8}\right) = 0,2267$		

4.5. Operadores genéticos

Los operadores genéticos están destinados a realizar alteraciones sobre unos individuos, denominados padres, para producir otros nuevos, sus descendientes. En Programación Evolutiva, donde no existe operador de cruce, las modificaciones genéticas se realizan por medio del operador de mutación de forma que, habitualmente, cada padre genera un único descendiente. En este sentido, la ausencia de operador de cruce es provocada por la presencia de varias restricciones en el problema, lo cual dificulta la obtención de soluciones válidas tras recombinar a los individuos que ejercen como padres.

En este trabajo se han desarrollado varios operadores de mutación con el fin de obtener un conjunto variado de transformaciones arquitectónicas que permitan modificar tanto el número de componentes como sus elementos internos. De esta forma, algunos de ellos implican alteraciones más profundas que otros, permitiendo que sea el experto el que configure la importancia de cada uno de ellos. Esta configuración se traduce en una probabilidad de selección del operador, de forma que, para cada individuo, se genera una ruleta basada en

probabilidades entre los distintos operadores.

Por otro lado, en función de las características de cada individuo es posible que algunas de las transformaciones propuestas no puedan realizarse, pues se obtendrían individuos no válidos. En este sentido, para cada individuo se crea un ruleta específica con aquellos operadores que se le pueden aplicar. Finalmenre, si tras la transformación se genera un individuo no válido, se repite el proceso de selección del mutador hasta que se obtenga un individuo válido o se alcance un número máximo de intentos, en cuyo caso se retorna el individuo original.

Para facilitar la comprensión del funcionamiento de los operadores propuestos se ha elaborado un ejemplo gráfico que acompañará a la explicación de cada operador. El individuo mostrado en la Figura 4.4 se corresponde con el individuo padre al que se le aplicarán los distintos operadores.

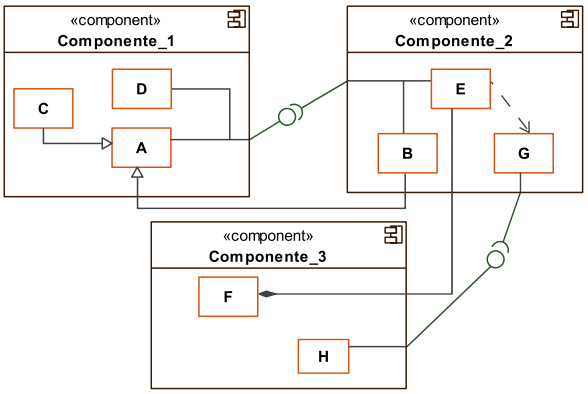


Figura 4.4: Individuo de ejemplo para actuar como padre

4.5.1. Operador de mutación *Añadir Componente*

El objetivo de este operador es incluir un nuevo componente en la arquitectura original. Para ello es necesario seleccionar algunas de las clases pertenecientes a otros componentes, que pasarán a formar parte del nuevo componente. Dado que esta operación implica el aumento en el número de componentes, solo puede ser aplicado cuando el número actual de componentes no sea el máximo configurado.

Puesto que se trata de una operación que implica un cambio importante en la estructura de la solución, se ha considerado una heurística a la hora de seleccionar las clases que conformarán el componente añadido. Esta heurística se basa en la idea de que es impor-

tante mantener unidas aquellas clases que parecen estar funcionalmente relacionadas, esto es, forman un grupo dentro de algún componente. Por tanto, no interesa disociar estos grupos pero sí separar dos grupos independientes que actualmente estén asignados al mismo componente, pues influye negativamente en su cohesión. A su vez, existe una probabilidad asociada a la participación de cada componente como suministrador de clases para el nuevo componente. Esta probabilidad viene determinada por la Ecuación 4.7, según la cuál existe una mayor probabilidad de que el componente proporcione clases cuantas más agrupaciones presente en su interior. En dicha ecuación, $grupos(Componente_i)$ devuelve el número de grupos del componente i, mientras que max_{grupos} indica el número mayor de grupos presentes en un componente, tal y como refleja el segundo término de la expresión, donde C indica el número de componentes que existen en una determinada solución.

$$Probabilidad(Componente_i \rightarrow Nuevo) = \frac{grupos(Componente_i)}{max_{grupos}} \tag{4.6}$$

$$max_{grupos} = max(grupos(j)) \quad j \in [1, C] \tag{4.7}$$

A continuación, si el componente es seleccionado, se contabiliza el número de clases que conforman cada uno de sus grupos, de forma que se elige uno entre los que contienen el número mínimo de clases. Finalmente, si no ha sido posible encontrar ningún grupo aislado de clases se realiza un proceso alternativo, completamente aleatorio, que selecciona algunas de las clases presentes en los distintos componentes. Para este segundo mecanismo debe considerarse la restricción de que de en cada componente no pueden seleccionarse todas sus clases, pues el componente quedaría vacío. El Algoritmo 4.1 recoge el pseudocódigo de este proceso, donde se utilizadan son las siguientes funciones:

- $grupos(i)$ calcula el número de grupos de clases en el interior del componente i.
- $aleatorio(min, max)$ devuelve un número aleatorio entre min y max.
- $contarClases(i, j)$ obtiene el número de clases que conforman el grupo j para el componente i.
- $elegirGrupo(indice)$ retorna el conjunto de clases dado el índice del grupo.
- $elegirClase(j)$ devuelve la j-ésima clase del modelo.

Algoritmo 4.1 Heurística del operador de mutación *Añadir Componente*

```
max = 0
for i = 1...C do
  if grupos(i) > max then
    max = grupos(i)
  end if
end for
for i = 1...C do
  nGrupos = grupos(i)
  prob = aleatorio(0, 1)
  if prob > (nGrupos/max) then
    if nGrupos > 1 then
      minClases = ∞
      for j = 1...nGrupos do
        nClases[j] = contarClases(i, j)
        if nClases[j] < minClases then
          minClases = nClases[j]
        end if
      end for
      candidatos = 0
      for j = 1...nGrupos do
        if nClases[j] == minClases then
          candidatos + +
        end if
      end for
      if candidatos > 0 and nGrupos > 1 then
        indice = aleatorio(0, candidatos)
        clases ← elegirGrupo(indice)
        elegidos + +
      end if
    end if
  end if
end for
if elegidos == 0 then
  while elegidos == 0 do
    for i = 1...C do
      for j = 1...nClases do
        prob = aleatorio(0, 1)
        if prob > 0,5 then
          clases ← elegirClase(j)
          elegidos + +
        end if
      end for
    end for
  end while
end if
return clases
```

La Figura 4.5 muestra un ejemplo gráfico del funcionamiento de este operador, donde las clases B y F, las cuales estaban aisladas del resto de clases en el interior de los componentes *2* y *3* repectivamente, son seleccionadas para construir el componente *4*. Este movimiento implica la identificación de una nueva interfaz candidata entre el componente creado y el componente *1*, establecida por la relación entre las clase D y B.

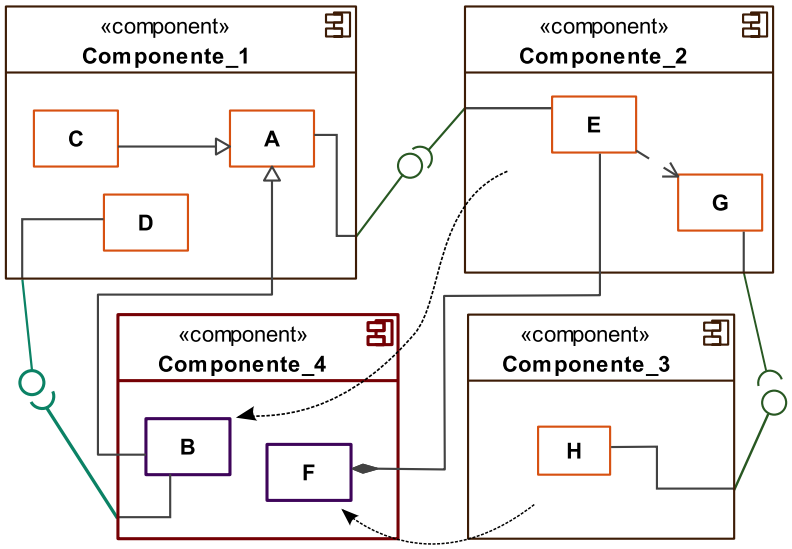


Figura 4.5: Individuo resultante tras aplicar el operador *Añadir Componente*

4.5.2. Operador de mutación *Eliminar Componente*

El objetivo de este operador es eliminar un componente de la arquitectura, de forma que se selecciona uno entre aquellos con mayor acoplamiento, intentando así favorecer la reducción de esta medida. El proceso consiste pues en determinar el componente con mayor acoplamiento, o en seleccionar uno al azar si todos ellos tienen el mismo grado de acoplamiento, tal y como se recoge en el Algoritmo 4.2. En dicho código, la función $acoplamiento(i)$ devuelve el valor de acoplamiento para el componente i, mientras que la función $aleatorio(min, max)$ es equivalente al procedimiento explicado en la Sección 4.5.1 y C representa el número de componentes.

Es necesario indicar que este operador podrá ser aplicado sobre aquellas soluciones cuyo número de componentes actuales no sea el mínimo fijado. Finalmente, las clases pertenecientes al componente seleccionado son distribuidas de manera aleatoria entre el resto de componentes.

Algoritmo 4.2 Heurística del operador de mutación *Eliminar Componente*

```
max = 0
componente = −1
for i = 0...C do
  if acoplamiento(i) > max then
    max = acoplamiento(i)
    componente = i
  end if
end for
if componente == −1 then
  componente = aleatorio(1, C)
end if
return componente
```

En la Figura 4.6 se puede apreciar un ejemplo de aplicación de este mutador, donde se ha seleccionado al componente *2* como el componente a eliminar, pues es el que tiene mayor acoplamiento. Sus clases se han distribuido entre los dos componentes restantes. Como resultado se obtienen dos componentes para cuya interacción se define una interfaz identificada únicamente a partir de la relación entre A y E, pues las clases D y B pertenecen ahora al mismo componente.

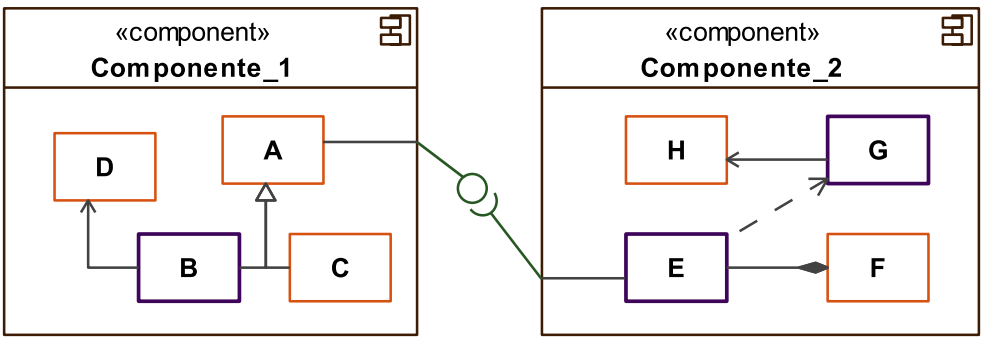


Figura 4.6: Individuo resultante tras aplicar el operador *Eliminar Componente*

4.5.3. Operador de mutación *Unir Componentes*

La unión de componentes consiste en la selección de dos componentes del modelo arquitectónico para conformar un nuevo componente que una las funcionalidades de ambos. Puesto que esta transformación implica la reducción del número de componentes, solo podrá ser aplicada si el número actual de componentes es mayor que el mínimo configurado.

Algoritmo 4.3 Heurística del operador de mutación *Unir Componentes*

```
max = 0
componentes = null
candidatos = null
componente1 = −1
componente2 = −1
for i = 1...C do
  if acoplamiento(i) > max then
    max = acoplamiento(i)
    candidatos ← i
  end if
end for
if tam(candidatos) > 1 then
  componente1 = aleatorio(1, tam(candidatos))
  while componente1 == componente2 do
    componente1 = aleatorio(1, tam(candidatos))
  end while
else
  componente1 = candidatos(0)
  while componente1 == componente2 or acoplamiento(componente2) == 0 do
    componente2 = aleatorio(1, C)
  end while
end if
componentes ← componente1
componentes ← componente2
return componentes
```

La heurística utilizada para la selección de los dos componentes a unir también está basada en la intención de reducir el acoplamiento entre componentes. Para ello se identifican aquellos con el máximo acoplamiento. Si existe más de un componente con el valor máximo de acoplamiento, se eligen dos de ellos para ser unidos. En caso contrario, se selecciona al de mayor acoplamiento y a otro elegido de manera aleatoria, tal y como refleja el Algoritmo 4.3. En este código *candidatos* y *componentes* son dos listas donde se almacenan los índices de los componentes candidatos a ser unidos y los finalmente seleccionados, respectivamente. La función $tam(l)$ devuelve el tamaño de la lista l, mientras que las funciones

$acoplamiento(i)$ y $aleatorio(min, max)$ se definen en los mismos términos que para los operadores anteriores.

Gráficamente, la Figura 4.7 recoge un ejemplo del funcionamiento de este operador. En él se ha seleccionado al componente *2* por ser el que presenta mayor acoplamiento y al componente *3*, elegido al azar, para ser unidos.

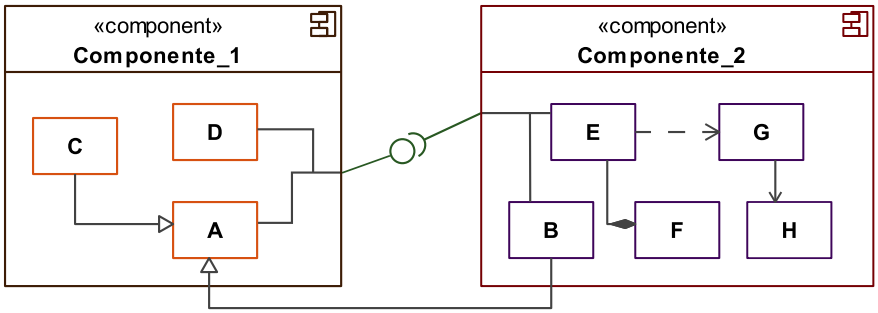


Figura 4.7: Individuo resultante tras aplicar el operador *Unir Componente*

4.5.4. Operador de mutación *Dividir Componente*

Este operador de mutación realiza el proceso inverso al del operador anterior, pues consiste en la partición de un componente para generar dos nuevos. En este caso, al igual que en el caso del operador *Añadir Componente*, existe la precondición de que el número actual de componentes sea inferior al máximo configurado.

La idea subyacente al proceso de selección del componente a transformar consiste en la búsqueda de los componentes que presentan más de un grupo en su interior, de forma que, como estos grupos podrían estar representando funcionalidades diferentes del componente, resultaría interesante separar dichas funcionalidades en dos nuevos componentes.

El Algoritmo 4.4 recoge, de forma detallada, el procedimiento utilizado. En él se realiza la búsqueda de los componentes con más de un grupo diferenciado de clases en su interior (almacenados en la lista *candidatos*) para, a continuación, distribuir estos grupos entre aquellos que permanecen en el componente actual y los que serán enviados al nuevo componente.

En el caso de que no existan componentes con estas características (al menos dos para que no siempre sea elegido el mismo en sucesivas aplicaciones del operador), se procede a realizar una selección aleatoria de uno de ellos. El procedimiento aleatorio debe, por otro

lado, garantizar que no se seleccionan todas las clases del componente, para impedir que quede vacío. La variable *clases* es una lista en la cual se almacenan las clases pertenecientes a los grupos seleccionados para formar el nuevo componente, mientras que las clases de los grupos restantes se mantendrán en el componente actual, representado por la variable *componente*.

Algoritmo 4.4 Heurística del operador de mutación *Dividir Componente*

```
max = 0
clases = null
componente = −1
grupo = −1
for i = 1...C do
  if grupos(i) > 1 then
    candidatos ← i
  end if
end for
if tam(candidatos) > 1 then
  componente = aleatorio(1, tam(candidatos))
  grupo = aleatorio(1, grupos(componente))
  clases ← elegirClases(componente, grupo)
else
  componente = aleatorio(1, C)
  nClases = clases(componente)
  elegidas = 0
  for j = 1...nClases do
    prob = aleatorio(0, 1)
    if prob > 0,5 and elegidas < nClases then
      clases ← j
      elegidas + +
    end if
  end for
end if
return clases
```

La Figura 4.8 muestra el resultado de aplicar este operador sobre el individuo original de la Figura 4.4. Como puede observarse, el componente *2*, que presentaba dos grupos de clases (uno con B y otro formado por E y G), es dividido en dos componentes, el propio componente *2* y el *4*, de forma que cada uno de ellos mantiene uno de los grupos originales. Esta transformación implica no solo el movimiento de las clases pertenecientes a cada grupo, sino también de las interfaces vinculadas a estas clases, como es el caso de la clase B, la cual permite la identificación de una interfaz entre el componente *4* y el *1*.

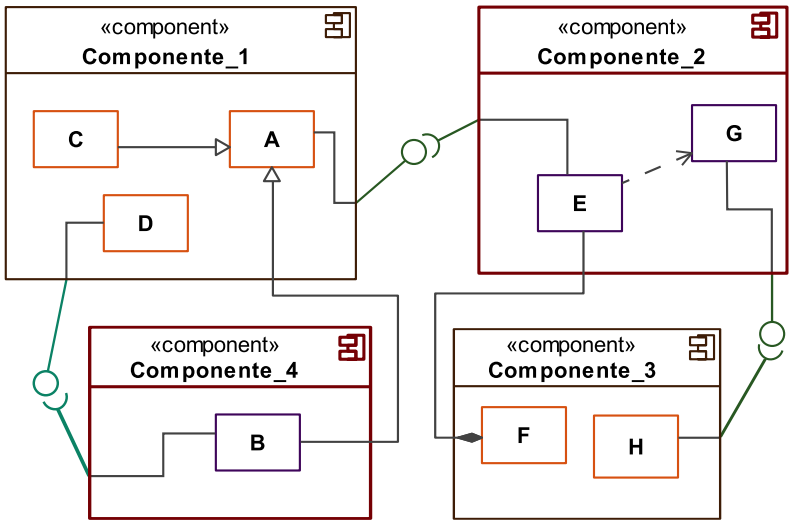


Figura 4.8: Individuo resultante tras aplicar el operador *Dividir Componente*

4.5.5. Operador de mutación *Mover Clase*

Este operador realiza el movimiento de una clase de un componente origen hacia uno destino. Dado que se trata de una transformación que no implica grandes alteraciones en las soluciones generadas, la selección tanto de la clase a reubicar como de los componentes implicados se realiza de manera aleatoria.

Respecto a las restricciones a considerar, no será posible seleccionar como componente origen a ninguno en cuyo interior solo se aloje una clase, pues resultaría en un componente vacío. Una vez excluídos del proceso dichos componentes, la selección del componente origen también es realizada de manera aleatoria sobre el resto de componentes. Por tanto, únicamente no podría aplicarse este operador si todos los componentes del modelo arquitectónico estuviesen conformados por una única clase en su interior.

El proceso de selección de los componentes afectados y de la clase a mover se recoge en el Algoritmo 4.9. Una vez seleccionado el componente origen y la clase a mover, si $C = 2$, entonces el componente destino será el otro componente, mientras que si $C > 2$, el componente destino se elige de forma aleatoria. Finalmente, la variable *movimiento*, de tipo lista, almacena el movimiento que se va a realizar, indicando el componente origen, el destino y la clase elegida. Un ejemplo de la aplicación de este proceso se recoge en la Figura 4.9, donde la clase *B* es desplazada desde el componente *2* hasta el *1*.

Algoritmo 4.5 Heurística del operador de mutación *Mover Clase*

$movimiento = null$
$origen = aleatorio(1, C)$
$nClases = clases(origen)$
$clase = aleatorio(1, nClases)$
if C==2 **then**
  if $origen == 1$ **then**
    $destino = 2$
  else
    $destino = 1$
  end if
else
  while $destino == origen$ **do**
    $destino = aleatorio(1, C)$
  end while
end if
$movimiento \leftarrow origen, destino, clase$
return *movimiento*

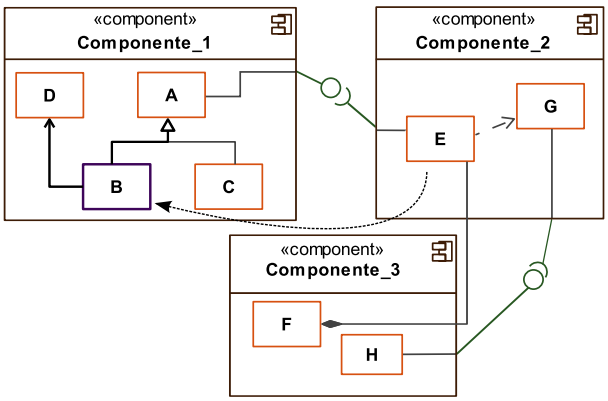


Figura 4.9: Individuo resultante tras aplicar el operador *Mover Clase*

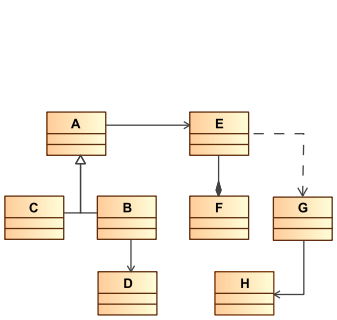
4.6. Selección y reemplazo

Las técnicas de selección y reemplazo indican el mecanismo mediante el cual se eligen los individuos que actúan como padres y la estrategia utilizada para actualizar la población de una generación a otra, respectivamente. En el primer caso, el proceso de selección es determinista, como es habitual en Programación Evolutiva, de forma que cada individuo de la población toma el rol de padre. El reemplazo se realiza mediante una competición entre cada padre y su descendiente, de forma que aquel con mejor *fitness* es incluido en la población de la siguiente generación.

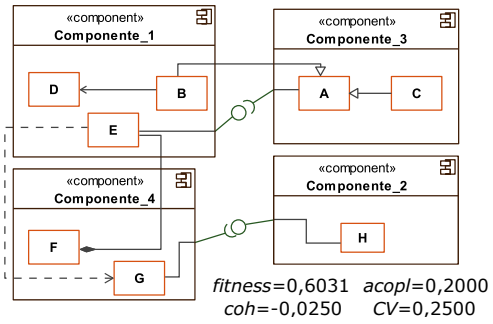
4.7. Desarrollo de un ejemplo

A continuación se desarrolla un pequeño ejemplo sobre la ejecución del algoritmo evolutivo, de forma que se podrá comprobar cómo las soluciones iniciales generadas son optimizadas progresivamente.

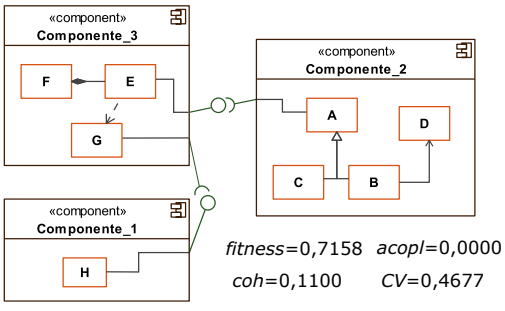
En la Figura 4.10 se recogen cuatro momentos del proceso de identificación de componentes. En primer lugar se dispone de un modelo de análisis (un diagrama de clases) del cual se han extraído las clases participantes y sus relaciones (a). Esta información es utilizada para la creación y evaluación de los individuos de la población inicial (b). Se puede observar cómo el mejor individuo de la población inicial presenta varias relaciones externas entre las clases. Tras 5 generaciones (c), el mejor individuo de la población ha mejorado en cuanto a la distribución de las clases, aunque mantiene el número de componentes. Alcanzadas las 10 generaciones (d), el mejor individuo presenta una arquitectura simplificada en la cual se han distribuido de forma óptima las clases del modelo.



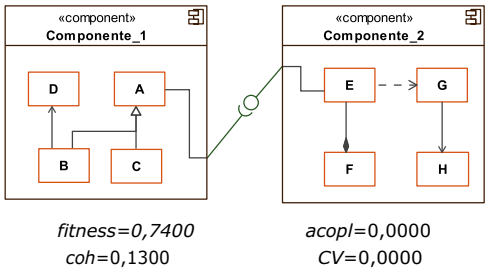
(a) Diagrama de clases



(b) Individuo de la población inicial



(c) Mejor individuo (generación 5)



(d) Mejor individuo (generación 10)

Figura 4.10: Ejemplo de funcionamiento del algoritmo evolutivo

Capítulo 5

Experimentación

5.1. Software utilizado e instancias del problema

El algoritmo de Programación Evolutiva propuesto ha sido desarrollado en Java con el *framework* JCLEC [38]. Se han utilizado dos librerías Java públicas para el preprocesado de la información procedente de los modelos de análisis: SDMetrics Open Core [1] y Datapro4j [2].

Para realizar la experimentación del algoritmo se han considerado los modelos de análisis de tres sistemas software, cuyas características se recogen en la Tabla 5.1. En ella se muestran el número de clases (#Clases) y el número de apariciones de cada tipo de relación UML: asociaciones (#Asoc), dependencias (#Depen), agregaciones (#Agreg), composiciones (#Compos) y generalizaciones (#Gener).

Tabla 5.1: Instancias del problema y sus características

	#Clases	#Asoc	#Depen	#Agreg	#Compos	#Gener
NekoHTML	24	16	5	0	0	10
Aqualush	58	69	6	0	0	20
Datapro4j	59	4	18	1	4	50

[1] http://www.sdmetrics.com/OpenCore.html
[2] http://www.uco.es/grupos/kdis/datapro4j

5.2. Configuración de parámetros

En la Tabla 5.2 se detallan los parámetros del algoritmo y los valores asociados a cada uno de ellos, obtenidos tras haber efectuado una serie de experimentos previos. Se han realizado 30 ejecuciones con diferentes semillas aleatorias sobre los tres problemas considerados con el objetivo de obtener resultados no sesgados.

Puesto que no existen otras propuestas ante las cuales comparar, se ha realizado también una búsqueda aleatoria (RS, *Random Search*). Dada la configuración anterior, el algoritmo evolutivo genera un máximo de 100 individuos válidos en cada una de las 100 generaciones, por lo que la búsqueda aleatoria debe generar 10.000 soluciones válidas para realizar una comparación justa.

Tabla 5.2: Configuración de parámetros

Parámetro	**Valor**	**Parámetro**	**Valor**
Tamaño de la población	100	Prob. Mut. *Añadir componente*	0,250
Número de generaciones	100	Prob. Mut. *Eliminar componente*	0,125
Nº máx. mutaciones/ind.	10	Prob. Mut. *Dividir componente*	0,125
W_{coh}	0,300	Prob. Mut. *Unir componentes*	0,250
W_{aco}	0,400	Prob. Mut. *Mover clase*	0,250
W_{cv}	0,300	Número de componentes	Entre 2 y 8

5.3. Resultados y discusión

La Tabla 5.3 recoge los resultados obtenidos tras la ejecución de ambos algoritmos. En ella se muestran los valores medios de *fitness*, así como de las tres métricas consideradas para su cálculo: cohesión (*coh*), acoplamiento (*acopl*) y CV. Cabe recordar que tanto el *fitness* como la cohesión deben ser maximizados, mientras que el acoplamiento y el CV son minimizadas.

Como puede observarse, el algoritmo de EP es capaz de alcanzar mejores valores de *fitness* que los aportados por la búsqueda aleatoria, de forma que el proceso evolutivo propuesto permite alcanzar arquitecturas más adecuadas según los criterios de diseño considerados.

Tabla 5.3: Resultados del algoritmo de EP y la búsqueda aleatoria

		NekoHTML	AquaLush	Datapro4j
EP	*Fitness*	$0,7301 \pm 0,0014$	$0,6793 \pm 0,0254$	$0,4780 \pm 0,0206$
	Cohesión	$0,1977 \pm 0,0105$	$0,1262 \pm 0,0446$	$0,0767 \pm 0,0991$
	Acopl.	$0,0000 \pm 0,0000$	$0,0600 \pm 0,0899$	$0,9868 \pm 0,2571$
	CV	$0,7806 \pm 0,1208$	$0,9222 \pm 0,0754$	$0,9868 \pm 0,2571$
RS	*Fitness*	$0,6198 \pm 0,0402$	$0,4640 \pm 0,0112$	$0,3316 \pm 0,0102$
	Cohesión	$-0,0390 \pm 0,0828$	$-0,2510 \pm 0,0401$	$0,0541 \pm 0,0980$
	Acopl.	$0,1420 \pm 0,0776$	$0,3694 \pm 0,0346$	$0,9522 \pm 0,1001$
	CV	$0,3129 \pm 0,2576$	$0,3460 \pm 0,1014$	$0,1008 \pm 0,0963$

Analizando los resultados en cuanto a la métrica de cohesión, cabe destacar la habilidad del algoritmo para encontrar agrupaciones de clases funcionalmente relacionadas. Este hecho queda reflejado al observar que la búsqueda aleatoria obtiene en algunos casos valores negativos, lo cual indica que las relaciones entre las clases ubicadas en los componentes no son lo suficientemente fuertes frente al número de agrupaciones generadas, que constituyen una penalización en esta métrica.

Respecto al acoplamiento se puede apreciar cómo el algoritmo encuentra soluciones con el mínimo valor de esta medida para el primer problema considerado, mientras que encuentra más dificultades para *Datapro4j*. La diferencia de complejidad entre los sistemas software considerados permite explicar esta circunstancia, pues *Datapro4j* presenta un número muy superior de relaciones de generalización, las cuales implican la máxima penalización en el cómputo del acoplamiento.

Aunque la búsqueda aleatoria obtiene arquitecturas más balanceadas en cuanto al tamaño de los componentes, obtiene valores muy inferiores para el resto de métricas. En este sentido, el algoritmo de EP es capaz de mantener un mejor compromiso entre todas las medidas consideradas.

Otro aspecto interesante a comparar es el tiempo de ejecución necesario para cada algoritmo, tal y como se recoge en la Tabla 5.4. En ella puede apreciarse como el algoritmo evolutivo es mucho más efectivo que la búsqueda aleatoria, sobre todo en el segundo problema considerado, *AquaLush*. La presencia de restricciones juega aquí un papel importante debido a la dificultad de crear de manera totalmente aleatoria un número de soluciones válidas equivalente al total de soluciones que explora el algoritmo evolutivo. Por el contrario,

una vez creada una población de soluciones de menor tamaño es más fácil generar otras nuevas a partir de ellas, que también cumplan las restricciones del problema.

Tabla 5.4: Tiempo medio de ejecución (s) del algoritmo de EP y la búsqueda aleatoria

	NekoHTML	AquaLush	Datapro4j
EP	$14,4062 \pm 0,7761$	$888,2714 \pm 69,2335$	$32,6848 \pm 1,5754$
RS	$759,6504 \pm 31,5627$	$83{,}410,2443 \pm 929,6426$	$1{,}125,8845 \pm 13,9348$

También es posible destacar otros aspectos relacionados con la tipología de las soluciones generadas. En primer lugar, esta propuesta es capaz de manejar arquitecturas software de diferentes características, esto es, número de componentes y conectores, permitiendo al experto decidir entre soluciones de diversa naturaleza. A su vez, los componentes de las arquitecturas resultantes se asemejan a las soluciones reales. En concreto, para el sistema *Aqualush*, basado en capas, el algoritmo es capaz de obtener las agrupaciones de clases que constituyen algunas de estas capas.

Finalmente, con el objetivo de comparar estadísticamente los resultados obtenidos por ambos procedimientos [39, 40], se ha realizado el test de Wilcoxon bajo la hipótesis nula, H_0, de que ambos algoritmos tienen un rendimiento similar. Con un 90 % de confianza, H_0 puede ser rechazada, por lo que existen diferencias significativas entre la propuesta de EP y la búsqueda aleatoria.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

En este trabajo se ha presentado un modelo de Programación Evolutiva para la identificación de arquitecturas basadas en componentes a partir de modelos de análisis. Esta tarea es una de las principales a realizar en el análisis arquitectónico cuando se modelan sistemas software de gran envergadura, de forma que abordar su automatización constituye un reto de gran interés.

La problemática ha sido formulada como un problema de búsqueda y optimización, de forma que se han definido los componentes básicos de este tipo de propuestas: una representación adecuada, que además de ser eficiente es interpretable por los expertos en análisis arquitectónico; una función de fitness, basada en criterios de diseño como la cohesión y el acoplamiento; y un conjunto de operadores genéticos que representan diferentes transformaciones arquitectónicas.

Tras analizar los resultados obtenidos, es posible destacar la adecuación de la propuesta al problema planteado. La obtención de los modelos arquitectónicos anteriores permiten al analista abordar la identificación de las unidades funcionales que constituyen el software, proceso difícil de lograr cuando se trabaja con sistemas software complejos. Mediante las métricas aquí consideradas se consigue un compromiso entre distintos aspectos deseables en el diseño basado en componentes, es más, el propio experto puede determinar la necesidad de optimizar uno u otro en mayor o menor medida según el problema concreto. Además, el tiempo de cómputo necesario no es excesivo, lo cual ofrece al ingenierio la posibilidad de

realizar varias ejecuciones para obtener diferentes soluciones arquitectónicas. En definitiva, el método propuesto consigue apoyar la labor del ingeniero software, de forma que le aporta una nueva herramienta, flexible y potente, con la que abordar su trabajo.

Finalmente, la novedad de la propuesta y del problema abordado han sido resaltados tanto por la comunidad investigadora enfocada en el ámbito de las metaheurísticas como en el área de la Ingeniería del Sofware y SBSE, lo cual desmuestra que se trata de un campo de interés en la actualidad.

6.2. Trabajo futuro

El algoritmo propuesto supone una primera aproximación a la optimización de arquitecturas software mediante técnicas metaheurísticas. Como trabajo futuro se plantea, en primer lugar, la extensión del modelo para incluir otro tipo de información de análisis que pueda ser de interés, como la descomposión de clases en atributos y métodos o la presencia de subcomponentes. La inclusión de otras métricas para la evaluación de las soluciones bajo más criterios de diseño o para la comparación con las soluciones diseñadas por los propios expertos constituye otra línea de trabajo interesante. Además, existen otras tareas relacionadas con el análisis y diseño del software susceptibles de ser tratadas desde la perspectiva de SBSE.

Por otro lado, el campo de las metaheurísticas aporta muchas técnicas que pueden ser combinadas con el algoritmo evolutivo diseñado para mejorar su rendimiento. En especial, la hibridación con métodos de búsqueda local podría ser interesante para refinar las soluciones generadas por el algoritmo a lo largo de la evolución. A su vez, la inclusión de mecanismos de interacción que permitan al ingeniero software participar activamente en el proceso de optimización puede ser otra línea de trabajo a considerar. Finalmente, la variedad dentro del conjunto de metaheurísticas, en constante evolución, es otro aliciente para la creación de nuevos algoritmos para la optimización de arquitecturas software.

Referencias

[1] N. Ashrafi, "The impact of software process improvement on quality: in theory and practice," *Information and Management*, vol. 40, no. 7, pp. 677–690, 2003.

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2002.

[3] D. Birkmeier and S. Overhage, "On Component Identification Approaches — Classification, State of the Art, and Comparison," in *Proceedings of the 12th International Symposium on Component-Based Software Engineering (CBSE'09)*, (Berlin, Heidelberg), pp. 1–18, Springer-Verlag, 2009.

[4] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, pp. 11:1–61, 2012.

[5] L. L. Minku, D. Sudholt, and X. Yao, "Evolutionary algorithms for the project scheduling problem: runtime analysis and improved design," in *Proceedings of the 14th International Conference on Genetic and Evolutionary Computation Conference (GECCO '12)*, (Philadelphia, USA), pp. 1221–1228, ACM, 2012.

[6] J. Ferrer, P. M. Kruse, F. Chicano, and E. Alba, "Evolutionary Algorithm for Prioritized Pairwise Test Data Generation," in *Proceedings of the 14th International Conference on Genetic and Evolutionary Computation Conference (GECCO '12)*, (Philadelphia, USA), pp. 1213–1220, ACM, 2012.

[7] I. Boussaïd, J. Lepagnot, and P. Siarry, "A survey on optimization metaheuristics," *Information Sciences*, vol. 237, no. 0, pp. 82 – 117, 2013.

[8] M. Harman, "Software Engineering Meets Evolutionary Computation," *Computer*, vol. 44, no. 10, pp. 31–39, 2011.

[9] L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial Intelligence through Simulated Evolution*. New York, USA: John Wiley, 1966.

[10] D. B. Fogel and L. J. Fogel, "An introduction to evolutionary programming," in *Artificial Evolution*, vol. 1063 of *Lecture Notes in Computer Science*, pp. 21–33, Springer Berlin Heidelberg, 1996.

[11] M. Harman, P. McMinn, J. T. Souza, and S. Yoo, "Search Based Software Engineering: Techniques, Taxonomy, Tutorial," in *Empirical Software Engineering and Verification*, vol. 7007 of *Lecture Notes in Computer Science*, pp. 1–59, Springer Berlin Heidelberg, 2012.

[12] J. A. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. F. Jones, M. Lumkin, B. S. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. J. Shepperd, "Reformulating Software Engineering as A Search Problem," *IEEE Proceedings - Software*, vol. 150, no. 3, pp. 161–175, 2003.

[13] P. McMinn, "Search-based software test data generation: a survey: Research Articles," *Software Testing, Verification & Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[14] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[15] J. Dolado, "On the problem of the software cost function," *Information and Software Technology*, vol. 43, no. 1, pp. 61 – 72, 2001.

[16] O. Räihä, "A survey on search-based software design," *Computer Science Review*, vol. 4, no. 4, pp. 203 – 249, 2010.

[17] W.-N. Chen and J. Zhang, "Ant Colony Optimization for Software Project Scheduling and Staffing with an Event-Based Scheduler," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 1–17, 2013.

[18] E. Díaz, J. Tuya, R. Blanco, and J. J. Dolado, "A Tabu Search Algorithm for Structural Software Testing," *Computers & Operations Research*, vol. 35, no. 10, pp. 3052–3072, 2008.

[19] S. Bouktif, H. Sahraoui, and G. Antoniol, "Simulated Annealing for Improving Software Quality Prediction," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, (Seattle, Washington, USA), pp. 1893–1900, ACM, 2006.

[20] S. Yoo and M. Harman, "Using Hybrid Algorithm For Pareto Effcient Multi-Objective Test Suite Minimisation," *Journal of Systems and Software*, vol. 83, no. 4, pp. 689–701, 2010.

[21] R. Britto, P. S. Neto, R. Rabelo, W. Ayala, and T. Soares, "A Hybrid Approach to Solve the Agile Team Allocation Problem," in *Proceedings of IEEE Congress on Evolutionary Computation (CEC '12)*, (Brisbane, Australia), pp. 1–8, IEEE, 2012.

[22] G. Bavota, F. Carnevale, A. De Lucia, M. Di Penta, and R. Oliveto, "Putting the Developer in-the-Loop: An Interactive GA for Software Re-modularization," in *Proceedings of the 4th International Symposium on Search Based Software Engineering (SSBSE'12)*, (Berlin, Heidelberg), pp. 75–89, Springer-Verlag, 2012.

[23] C. Simons and J. Smith, "A comparison of meta-heuristic search for interactive software design," *Soft Computing*, pp. 1–16, 2013.

[24] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationship between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245–273, 2000.

[25] O. Maqbool and H. Babri, "Hierarchical Clustering for Software Architecture Recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759 –780, 2007.

[26] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software Architecture Optimization Methods: A Systematic Literature Review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013.

[27] V. Cortellessa, F. Marinelli, and P. Potena, "Automated Selection of Software Components Based on Cost/Reliability Tradeoff," in *Software Architecture*, vol. 4344 of *Lecture Notes in Computer Science*, pp. 66–81, Springer Berlin Heidelberg, 2006.

[28] S. Malek, N. Medvidovic, and M. Mikic-Rakic, "An Extensible Framework for Improving a Distributed Software System's Deployment Architecture," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 73–100, 2012.

[29] A. Koziolek, D. Ardagna, and R. Mirandola, "Hybrid multi-attribute qos optimization in component based software systems," *Journal of Systems and Software*, (en prensa).

[30] A. C. Jensen and B. H. Cheng, "On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns," in *Proceedings of the 12th International Conference on Genetic and Evolutionary Computation (GECCO '10)*, (Portland, Oregon, USA), pp. 1341–1348, ACM, 2010.

[31] S. Kebir, A.-D. Seriai, A. Chaoui, and S. Chardigny, "Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code," in *Proceedings of the 5th International C* Conference on Computer Science and Software Engineering (C3S2E'12)*, (New York, NY, USA), pp. 1–8, ACM, 2012.

[32] S. Vathsavayi, O. Räihä, and K. Koskimies, "Using Quality Farms in Multi-objective Genetic Software Architecture Synthesis," in *Proceedings of IEEE Congress on Evolutionary Computation (CEC '12)*, (Brisbane, Australia), pp. 1–8, IEEE, 2012.

[33] T. Back, U. Hammel, and H.-P. Schwefel, "Evolutionary computation: comments on the history and current state," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 3–17, 1997.

[34] K. Praditwong, M. Harman, and X. Yao, "Software Module Clustering as a Multi-Objective Search Problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2011.

[35] P. Rodríguez-Mier, M. Mucientes, M. Lama, and M. Couto, "Composition of web services through genetic programming," *Evolutionary Intelligence*, vol. 3, pp. 171–186, 2010.

[36] OMG, *MOF 2 XMI Mapping Specification*. OMG, aug 2011. formal/2011-08-09, http://www.omg.org/spec/XMI/2.4.1/.

[37] OMG, *Unified Modeling Language 2.4 Superstructure Specification*. OMG, nov 2010. formal/2010-11-14, http://www.omg.org/spec/UML/2.4/.

[38] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "JCLEC: a java framework for evolutionary computation," *Soft Computing*, vol. 12, no. 4, pp. 381–392, 2007.

[39] J. Derrac, S. García, D. Molina, and F. Herrera, "A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms," *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 3 – 18, 2011.

[40] A. Arcuri and L. Briand, "A Practical Guide for using Statistical Tests to Assess Randomized Algorithms in Software Engineering," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, (Honolulu, Hawái, USA), pp. 1–10, IEEE, 2011.

Apéndices

Apéndice A

Publicaciones asociadas

Publicaciones asociadas a estre trabajo:

- Ramírez, A; Romero, J.R; Ventura, S. A Novel Component Identification Approach Using Evolutionary Programming. 15^{th} *International Conference on Genetic and Evolutionary Computation (GECCO'13)*. Amsterdam (Holanda). ACM. 6-10 julio 2013. *Aceptado*.

- Ramírez, A; Romero, J.R; Ventura, S. Algoritmo de programación evolutiva para identificación de arquitecturas software. *IX Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB'13)*. Madrid (España). SCIE. 17-20 septiembre 2013. *Aceptado*.

- Ramírez, A; Romero, J.R; Ventura, S. Identificación de Componentes en Arquitecturas Software Mediante Programación Evolutiva. *XVIII Jornadas en Ingeniería del Software y Bases de Datos (JISBD'13)*. Madrid (España). SISTEDES. 18-20 septiembre 2013. *Aceptado*.

A Novel Component Identification Approach Using Evolutionary Programming

Aurora Ramírez, José Raúl Romero and Sebastián Ventura
Dept. of Computer Science and Numerical Analysis, University of Córdoba
Rabanales Campus, 14071 Córdoba, Spain
{i72raqua, jrromero, sventura}@uco.es

ABSTRACT

Component identification is a critical phase in software architecture analysis to prevent later errors and control the project time and budget. Obtaining the most appropriate architecture according to predetermined design criteria can be treated as an optimization problem, especially since the appearance of the Search Based Software Engineering, and its combination with bio-inspired metaheuristics. In this work, an evolutionary programming (EP) algorithm is used to identify components, based on a novel and comprehensible representation of software architectures.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search -*Heuristic methods*

General Terms

Algorithms, Software design

Keywords

Component-based architecture, Search Based Software Engineering, Evolutionary Programming

1. INTRODUCTION

Architectural analysis places an important role in current software developments. It is mainly considered a human-centered decision process, where the abilities and prior experiences of software engineers have a marked influence on the end product quality and reusability. Therefore, during the high level analysis, components identification is a critical approach for dealing with complex systems [1], allowing to identify the different system elements, as well as their functionalities and interactions.

Recently, the appearance of SBSE (*Search Based Software Engineering*) [2] brings a new perspective for solving specific problems in Software Engineering through search and optimization approaches. Focusing on architecture optimization [3], aspects like the architecture definition, the quality attributes to be measured and the global objective (refactoring, deployment, etc.) reflects a variety of applications.

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.
ACM 978-1-4503-1964-5/13/07.

In this paper we propose a novel evolutionary programming algorithm for software architecture optimization from analysis models. In a more precise way, a novel encoding of component-based software architectures closest to the expert domain comprehension, a fitness based on design concepts and specific genetic operators are presented.

2. ALGORITHM DESIGN

A component-based architecture can be described as a set of three elements: *components*, defined as a cohesive group of classes working together to satisfy its expected behaviour; *interfaces*, identified from relationships between classes belonging to different components that exchange services (required by one and provided by the other); and *connectors*, the linkage between a pair of required-provided interfaces.

A representation close to the expert domain has been selected, resulting in a trade-off between performance and comprehensibility. The hierarchical composition of these artefacts allows the translation into a tree structure (see Figure 1).

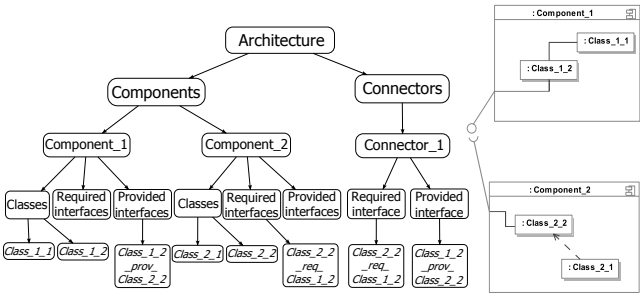


Figure 1: Genotype and phenotype

The initialization process begins with a random distribution of classes into a random number of components. Then, candidate interfaces and connectors are identified.

After their creation, individuals must be evaluated. Cohesion, the degree to which a component performs a well defined functionality, and coupling, related to the interdependence between components, are well-known design criteria. The distribution of classes among components is also important, as architectural solutions tend to look for balanced components in terms of size and inner complexity. Thus, these concepts are translated into quantifiable measures used to define the fitness function, f (see Equation 1).

$$coh_i = \frac{w_{c1}}{(w_a + w_d + w_{ac} + w_g) \cdot (n_{cl} - 1)} \cdot \Big[w_a \cdot n_a + w_d \cdot n_d + \\ + w_{ac} \cdot n_{ac} + w_g \cdot n_g \Big] + w_{c2} \cdot \frac{1 - n_{gr}}{n_{cl} - 1}$$

$$cop = \frac{1}{R_{max}} \cdot \frac{2 \cdot R \cdot (C-2)!}{C!}; R = \sum_{i=1}^{C} \sum_{j=i+1}^{C} \max r_{i,j}^{k}$$

$$cv = \frac{\sigma}{\mu}$$

$$f = w_{coh} \cdot \frac{\sum_{i=1}^{n} coh_i}{n} + w_{cop} \cdot (1 - cop) + w_{cv} \cdot \left(\frac{8 - cv}{8} \right) \quad (1)$$

The global cohesion is calculated as the average cohesion of each component (coh_i), reflecting the strength of their internal structures. It considers a weighted sum of relationships among classes, based on the number of different types of UML relationships: associations (w_a, n_a), dependencies (w_d, n_d), aggregations and compositions (w_{ac}, n_{ac}), and generalizations (w_g, n_g). The presence of unconnected clusters of classes (n_{gr}) is penalized, since it might imply internal dispersion. n_{cl} is the number of internal classes.

Coupling (*cop*) concerns the relationships among classes belonging to different components, based on the number of combinations of C components. It varies between 0 (components only related through interfaces) and 1 (all components are mutually connected). R accumulates a penalty rate for each pair of linked components based on the strongest relationship between them ($\max r_{i,j}^{k}$). R_{max} represents the worst situation, i.e. all components are related with the rest by means of the strongest relationship.

The coefficient of variation, *cv*, provides a normalized measure of the component size dispersion. μ and σ represent the mean and standard deviation of the component internal size, respectively. In opposition to *coh*, *cv* and *cop* must be minimized, so arithmetical transformations using its maximum values are realised in order to maximize f.

Finally, each individual in the population is selected to act as parent, generating a new solution. A probabilistic roulette with five mutators is applied in order to obtain offsprings. Each one represents an architectural transformation: *add* and *remove* components, *split* and *merge* them or move classes from one component to another one. The replacement strategy establishes a competition between each parent and its offspring, so only the best individual survives.

3. EXPERIMENTATION

The complete approach has been written in Java using the Datapro4j library [1] and JCLEC framework [4]. To analyse the performance and accuracy of this proposal, 30 executions were performed over three diverse problems, i.e. architectural specifications. In absence of other proposals to compare with, a random search (RS) has been also performed. Table 1 shows the average of fitness values of the best solutions found. The parameter configuration used was: 100 individuals as population, 100 generations, 2-8 components, $w_{coh} = 0.3$, $w_{coupl} = 0.4$ and $w_{cv} = 0.3$.

In general terms, the EP algorithm obtains better solutions in all problem instances. Special attention must be placed in the first instance, where all solutions obtained achieve the minimum value for the coupling measure. Even when RS is able to find a set of components with similar sizes, the rest of measures are significantly worse. As for the EP approach, a more appropriate trade-off between size dispersion and the rest of measures has been achieved, which mainly benefits the cohesion. Finally, the EP algorithm is able to evolve and keep architectures with different number of components and connectors during the search.

[1] http://www.uco.es/grupos/kdis/datapro4j

Table 1: Results for EP and RS algorithms

		NekoHTML	AquaLush	Datapro4j
EP	Fitness	0.7216	0.6483	0.4690
	Cohesion	0.1762	0.1430	0.0586
	Coupling	0.0000	0.1411	0.5162
	CV	0.8333	1.0175	1.1218
RS	Fitness	0.6198	0.4640	0.3316
	Cohesion	-0.0390	-0.2510	0.0541
	Coupling	0.1420	0.3694	0.9522
	CV	0.3129	0.3460	0.1008

The Wilcoxon signed-rank statistical test has demonstrated that the EP algorithm performs significantly better than RS with 90% confidence.

4. CONCLUSIONS

This paper presents a novel approach for the identification of component-based architectures from analysis models. The proposed encoding uses trees structures, similar to the one used by the underlying specification models, which brings the approach closer to the software architect. Specific genetic operators simulating architectural transformations and a fitness function inspired in cohesion and coupling concepts conform the core of the evolutionary search.

Experimentation has shown very promising results in terms of cohesion and coupling. Furthermore, our approach can generate and evolve individuals with different number and configurations of components and connectors, showing a flexible handling of software architectures.

Acknowledgments

Work supported by the Ministry of Science and Technology, project TIN2011-22408, and FEDER funds.

5. REFERENCES

[1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Boston, USA: Addison-Wesley Longman Publ. Co., 2nd ed., 2002.

[2] M. Harman, S. A. Mansouri, and Y. Zhang, "Search Based Software Engineering: Trends, Techniques and Applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–61, 2012.

[3] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Trans. on Software Engineering*, no. 99, pp. 1–26, 2012.

[4] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "JCLEC: a java framework for evolutionary computation," *Soft Comput.*, vol. 12, no. 4, pp. 381–392, 2007.

Algoritmo de programación evolutiva para identificación de arquitecturas software

Aurora Ramírez, José Raúl Romero y Sebastián Ventura

Dpto. de Informática y Análisis Numérico
Universidad de Córdoba, Campus de Rabanales, 14071 Córdoba
{aramirez, jrromero, sventura}@uco.es

Resumen La aplicación de técnicas metaheurísticas para la resolución de tareas de la Ingeniería del Software se engloba dentro del área denominada SBSE (*Search Based Software Engineering*). Por otra parte, el análisis y diseño del software y, en concreto, el análisis arquitectónico son fases importantes en el desarrollo de proyectos, ya que permiten prevenir errores en el futuro. No obstante, este tipo de tareas no ha sido muy tratada en SBSE a pesar de que la obtención de la arquitectura del software puede ser abordada como un problema de búsqueda y optimización. En este trabajo se presenta un algoritmo de programación evolutiva para la optimización de arquitecturas basadas en componentes. Para ello se ha desarrollado una nueva representación del problema, una función de *fitness* basada en métricas de diseño y un conjunto de operadores genéticos. La propuesta ha sido validada con arquitecturas reales, mostrando resultados interesantes y prometedores.

Keywords: Programación evolutiva, arquitecturas software, *Search Based Software Engineering*

1. Introducción

La utilización de técnicas de optimización y búsqueda en el dominio de la Ingeniería del Software, denominada *Search Based Software Engineering* (SBSE), ha experimentado un gran crecimiento en los últimos años [1], donde las técnicas bio-inspiradas han sido las metaheurísticas más empleadas [2].

En este dominio, los problemas a resolver están fuertemente vinculados a decisiones humanas, especialmente en las fases de análisis, diseño y pruebas. El análisis arquitectónico constituye un ejemplo de un proceso centrado en el experto que, a su vez, tiene una gran repercusión en la calidad y reusabilidad del software. En concreto, la identificación de componentes es el proceso mediante el cual se detectan los bloques constituyentes del software y sus funcionalidades, enfoque útil cuando se trabaja con sistemas complejos [3].

Este proceso puede ser abordado a partir de la información disponible en los modelos de análisis, pues en ellos se detallan las clases y las relaciones entre ellas, representando conjuntamente la estructura global del sistema. Habitualmente, estos modelos son los más utilizados pero, a medida que el sistema crece,

se vuelven demasiado complejos y es necesario realizar un análisis a nivel arquitectónico. De este modo, los componentes son identificados a partir de conjuntos de clases que definen funcionalidades específicas. En este punto, el problema puede ser enfocado como un proceso de optimización de estos componentes a partir de la búsqueda de agrupaciones de clases funcionalmente relacionadas.

En este trabajo se propone el desarrollo de un algoritmo de programación evolutiva para la identificación de arquitecturas software basadas en componentes, el cual es orientado como un método de apoyo a la decisión del ingeniero. El problema es enfocado desde un nivel de abstracción elevado, a diferencia de otras propuestas existentes, más cercanas a la refactorización de código y el mantenimiento del software. Se ha definido una representación cercana al experto, donde se consideran los elementos (componentes, clases, etc.) con los que este trabaja. La función de *fitness* propuesta se basa en criterios de diseño, mientras que los operadores genéticos representan transformaciones arquitectónicas.

El resto del artículo está estructurado de la siguiente manera. En la Sección 2 se describen propuestas relacionadas con este trabajo, centrándose en la aplicación de técnicas bio-inspiradas. El diseño del algoritmo evolutivo se detalla en la Sección 3. A continuación, la Sección 4 incluye la experimentación realizada y los resultados obtenidos. Finalmente, las conclusiones y el trabajo futuro se discuten en la Sección 5.

2. Trabajo relacionado

En el ámbito del análisis y diseño del software existen multitud de aspectos que pueden ser optimizados mediante la utilización de metaheurísticas [4]. En estos casos, el principal objetivo es apoyar al ingeniero en la toma de decisiones, de forma que se ofrecen diferentes alternativas entre las cuales elegir.

En [5] se desarrolla un algoritmo genético interactivo donde las funcionalidades identificadas en los casos de uso son traducidas en un conjunto de clases, atributos y métodos.

La introducción de patrones de diseño en el contexto de la refactorización del software ha sido estudiado en [6], donde se hace uso de la programación genética para manejar grafos y árboles que codifican la arquitectura resultante y el conjunto de transformaciones realizadas, respectivamente. En [7], un algoritmo genético es combinado con técnicas de clustering para extraer arquitecturas basadas en componentes a partir de código fuente.

La presencia de enfoques multi-objetivo también es frecuente en SBSE cuando se deben optimizar conjuntamente varios factores de diseño. La búsqueda de arquitecturas alternativas basándose en aspectos como la disponibilidad y el coste [8] o la modificabilidad y la eficiencia [9] son ejemplos de ello.

Aunque las propuestas anteriores trabajan a nivel de arquitecturas software, abarcan problemas diferentes al aquí considerado. La refactorización y búsqueda de alternativas de diseño requieren la existencia de una arquitectura inicial, aspecto que no es necesario en esta propuesta. Por otro lado, la extracción de componentes a partir de código fuente utiliza otro tipo de información al que

aquí se propone, pues los modelos de análisis están centrados en aspectos de alto nivel, permitiendo abordar el problema en una fase temprana del desarrollo.

3. Diseño del algoritmo

3.1. Genotipo y fenotipo

El problema que aquí se plantea puede ser definido como la búsqueda de la configuración óptima de un conjunto de elementos (componentes, interfaces y conectores) que permitan representar la arquitectura de un sistema software. De forma más precisa, se trata de descubrir la existencia de estos elementos a partir de otro tipo de información, los modelos de análisis del software, de forma que de ellos se abstraiga un modelo arquitectónico basado en componentes. Estos modelos pueden ser procesados directamente desde herramientas de modelado con las que trabajan los ingenieros del software y que, habitualmente, utilizan la notación UML.

En este punto resulta necesario definir adecuadamente algunos de los conceptos relacionados con el alcance del problema propuesto [3]. Los *componentes* son unidades de composición que presentan una funcionalidad bien definida y que es puesta a disposición de otros mediante sus *interfaces*. La implementación del componente es realizada en su interior por un conjunto de clases, que permanece oculto e inaccesible desde el exterior del componente. Las interfaces, por tanto, son el medio de interacción entre los componentes, pues en ellas se especifica el conjunto de operaciones que un componente provee al resto y que pueden identificarse a partir de la interacción entre clases pertenecientes a distintos componentes. Finalmente, los *conectores* representan la unión de varios puntos de interacción entre componentes, habitualmente pares de interfaces proveídas y requeridas. La Figura 1 representa un ejemplo de arquitectura modelada con UML, que constituye el fenotipo del problema.

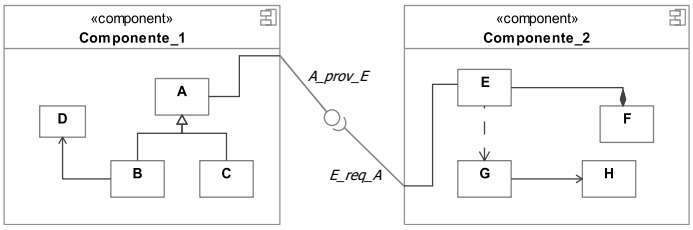


Figura 1. Fenotipo del problema

La selección de una codificación apropiada es un aspecto clave en el desarrollo de cualquier algoritmo de búsqueda, ya que influye en su eficiencia y, a su vez, debe ser interpretable, especialmente cuando el proceso está destinado a usuarios no expertos en computación evolutiva. A diferencia de la mayoría de propuestas

en el área de SBSE, que utilizan codificaciones lineales, se ha optado por una representación en árbol, donde los elementos definidos anteriormente constituyen los nodos del mismo en función de su nivel de composición.

Como se puede apreciar en la Figura 2, que representa el genotipo correspondiente al fenotipo anterior, la raíz del árbol representa la arquitectura completa, compuesta por un conjunto de componentes y conectores. Cada componente, a su vez, es desglosado en los tres elementos prefijados que lo describen (sombreados en la figura): el conjunto de clases que implementa su funcionalidad, las interfaces proveídas y las interfaces requeridas. Finalmente, los conectores se componen de un par de interfaces, una proveída y otra requerida.

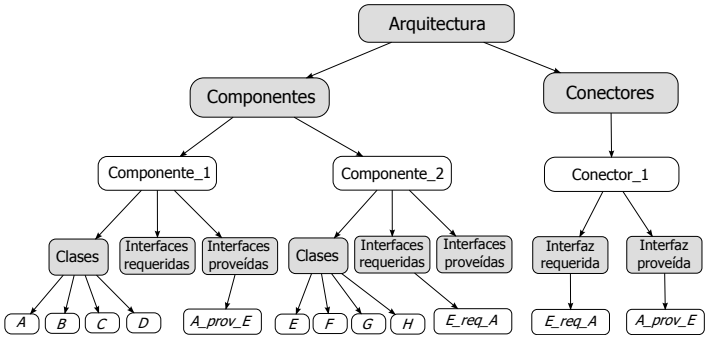


Figura 2. Genotipo del problema

El problema de optimización propuesto presenta varias restricciones que deben ser consideradas durante el proceso evolutivo. En primer lugar, cada una de las clases que componen el modelo inicial deben pertenecer a un único componente, proceso que es controlado en la inicialización del algoritmo.

Otros aspectos relacionados con la composición de los individuos resultantes también deben ser tenidos en cuenta. Por un lado, no pueden existir componentes vacíos, ya que no implementarían ninguna funcionalidad. Por otro lado, todos los componentes deben presentar, al menos, una interfaz que le permita interactuar con el resto de componentes. Finalmente, la presencia de componentes mutuamente dependientes, esto es, donde uno provee y requiere al otro (y viceversa), no son admitidas desde el punto de vista del análisis arquitectónico.

3.2. Inicialización de la población

La población inicial se crea mediante la generación aleatoria de un conjunto de individuos válidos. Cada individuo de la población es construído realizando una distribución aleatoria de las clases del modelo inicial de análisis en un número también aleatorio de componentes, obteniendo así una población diversa respecto al número de componentes. Una vez realizada esta distribución, es posible determinar las interfaces y los conectores establecidos entre estos componentes.

3.3. Función de *fitness*

La evaluación de los individuos está basada en criterios de diseño ampliamente extendidos en el desarrollo basado en componentes. La cohesión hace referencia a la obtención de componentes con funcionalidades bien definidas. Por otro lado, el acoplamiento indica la interrelación existente entre componentes. Finalmente, otro aspecto interesante a considerar es el tamaño de los componentes, directamente relacionado con la complejidad de los mismos. Estos tres conceptos han sido traducidos a tres métricas cuantificables sobre el problema propuesto, tal y como se detalla a continuación.

La cohesión global de la arquitectura se obtiene como la media de la cohesión calculada sobre cada componente (coh_i) mediante la Ecuación 1. Esta expresión se compone de dos términos ponderados que cuantifican dos características: el grado de unión entre las clases presentes en el interior del componente y el número de agrupaciones que lo conforman. El primero, que varía entre 0 y 1, contabiliza el número de relaciones establecidas entre las clases, clasificadas en función de los distintos tipos de relaciones UML y ponderadas en función de su importancia relativa: asociaciones (n_{as}, w_{as}), dependencias (n_{de}, w_{de}), agregaciones y composiciones (n_{ag}, n_{co}, w_{ac}) y generalizaciones (n_{ge}, w_{ge}). El segundo término, definido en el rango [-1,0], representa una penalización por el número de agrupaciones (n_{gr}) en el interior del componente, fundamentado en el concepto de *componentes conexas* en la Teoría de Grafos y bajo la idea de que varias agrupaciones representan dispersión en cuanto a la funcionalidad del componente. n_{cl} representa el número de clases en el interior del componente. Todos los pesos presentes en la Ecuación 1 han sido prefijados experimentalmente.

$$coh_i = \frac{w_{c1}}{(w_{as} + w_{de} + w_{ac} + w_{ge}) \cdot (n_{cl} - 1)} \cdot \Big[w_{as} \cdot n_{as} + w_{de} \cdot n_{de} + \\ + w_{ac} \cdot (n_{ag} + n_{co}) + w_{ge} \cdot n_{ge} \Big] + w_{c2} \cdot \frac{1 - n_{gr}}{n_{cl} - 1} \quad (1)$$

El siguiente término a considerar es el exceso de interrelaciones entre los componentes, lo cual se refleja en la medida de acoplamiento ($acopl$) propuesta en la Ecuación 2. En ella se contabilizan las relaciones existentes entre clases pertenecientes a distintos componentes que no son especificadas a través de las interfaces. Por ejemplo, esta situación puede ocurrir durante el proceso evolutivo ante una relación de generalización en la que la clase padre y las subclases no están ubicadas en el mismo componente. C indica el número de componentes, mientras que R, definido en la Ecuación 3, establece una penalización por la presencia de este tipo de relaciones. Concretamente, R acumula, para cada par de componentes i y j, el peso de la relación más fuerte entre las k relaciones ($k = 0...m$), $r^k_{i,j}$, establecidas entre ellos. La normalización de la métrica se consigue mediante el término R_{max}, pues representa la peor configuración posible (todos los componentes están mutuamente conectados por medio de la relación más fuerte). Por tanto, la medida toma valores entre 0 (no existen relaciones entre los componentes salvo las definidas a través de las interfaces) y 1 (todos los componentes están fuertemente acoplados).

$$acopl = \frac{1}{R_{max}} \cdot \frac{2 \cdot R \cdot (C-2)!}{C!} \tag{2}$$

$$R = \sum_{i=1}^{C} \sum_{j=i+1}^{C} max\ r_{i,j}^{k} \tag{3}$$

El coeficiente de variación, CV, mide la dispersión en el tamaño de los componentes, definido como el número de clases que los conforman. Su cálculo se obtiene por medio de la Ecuación 4, donde μ y σ representan la media y la desviación estándar del tamaño de los componentes, respectivamente. Esta medida puede tomar valores entre 1 y 8.

$$CV = \frac{\sigma}{\mu} \tag{4}$$

Finalmente, la función de *fitness* es construída a partir de la suma ponderada de los tres términos anteriores. Puesto que el acoplamiento y el coeficiente de variación deben ser minimizados y esta última, a su vez, normalizada, se han aplicado las correspondientes transformaciones aritméticas con los límites superiores de cada medida, tal y como se observa en la Ecuación 5.

$$fitness = w_{coh} \cdot \frac{\sum_{i=1}^{n} coh_i}{n} + w_{acopl} \cdot (1 - acopl) + w_{cv} \cdot \left(\frac{8 - CV}{8}\right) \tag{5}$$

3.4. Operadores genéticos

Dada la presencia de varias restricciones en el problema, la recombinación de individuos no ha sido considerada ante la dificultad de generar soluciones válidas. El único operador considerado es la mutación, para el cual se han diseñado diferentes variantes que representan las siguientes transformaciones arquitectónicas:

- *Añadir un componente*: Se crea un nuevo componente, mediante la selección aleatoria de clases e interfaces pertenecientes a otros componentes.
- *Eliminar un componente*: Se selecciona el componente con mayor acoplamiento para ser eliminado de la arquitectura. Una vez identificado, se realojan sus clases e interfaces en el resto de componentes.
- *Unir dos componentes*: Se seleccionan dos componentes, aquel con mayor acoplamiento y otro aleatorio, para ser fusionados en uno solo.
- *Dividir un componente*: Se selecciona aleatoriamente un componente entre aquellos que presentan varios grupos diferenciados de clases, los cuales son distribuidos en dos nuevos componentes.
- *Mover una clase*: Se selecciona de forma aleatoria una clase del modelo para ser transferida desde su componente actual hacia otro elegido aleatoriamente.

La selección de un operador u otro está basada en una ruleta probabilística, la cual es creada para cada padre en función de los operadores que se le pueden aplicar y la probabilidad asociada a cada mutador, configurable por el usuario.

Finalmente, la presencia de restricciones (ver Sección 3.1) debe ser considerada también a la hora de generar nuevos individuos. Los mutadores que extraen clases de unos componentes para ubicarlas en otros comprueban previamente que el movimiento a realizar no implique la ausencia de clases en el componente origen. La creación de individuos con componentes aislados o mutuamente dependientes es también identificada tras la evaluación del individuo, de forma que si el descendiente creado no es válido, se realiza un número máximo de intentos que, si es alcanzado, retorna al individuo inicial.

3.5. Selección y reemplazo

En este trabajo, el método de selección utilizado consiste en la actuación de cada individuo de la población como padre, produciendo un único descendiente mediante la aplicación de un operador de mutación. Por otro lado, la estrategia de competición entre padres e hijos para formar parte de la nueva población consiste en la selección del mejor individuo entre cada padre y su descendiente. La elección de ambos procedimientos se ha llevado a cabo tras la realización de una experimentación previa, de forma que se controla y mantiene la diversidad introducida por los mutadores. A su vez, una selección determinista suele ser el método de selección más habitual en EP.

4. Experimentación

El algoritmo de programación evolutiva propuesto ha sido desarrollado en Java con el *framework* JCLEC [10]. Se han utilizado dos librerías Java públicas para el preprocesado de la información procedente de los modelos de análisis: SDMetrics Open Core [1] y Datapro4j [2].

Tabla 1. Instancias del problema y sus características

	#Clases	#Asoc	#Depen	#Agreg	#Compos	#Gener
NekoHTML	24	16	5	0	0	10
Aqualush	58	69	6	0	0	20
Datapro4j	59	4	18	1	4	50

Para realizar la experimentación del algoritmo se han considerado los modelos de análisis de tres sistemas software, cuyas características se recogen en

[1] http://www.sdmetrics.com/OpenCore.html
[2] http://www.uco.es/grupos/kdis/datapro4j

la Tabla 1. En ella se muestran el número de clases (#Clases) y el número de apariciones de cada tipo de relación UML: asociaciones (#Asoc), dependencias (#Depen), agregaciones (#Agreg), composiciones (#Compos) y generalizaciones (#Gener).

4.1. Configuración del algoritmo

En la Tabla 2 se detallan los parámetros del algoritmo y los valores asociados a cada uno de ellos, obtenidos tras haber efectuado una serie de experimentos previos. Se han realizado 30 ejecuciones con diferentes semillas aleatorias sobre los tres problemas considerados con el objetivo de obtener resultados no sesgados.

Puesto que no existen otras propuestas ante las cuales comparar, se ha realizado también una búsqueda aleatoria (RS, *Random Search*). Dada la configuración anterior, el algoritmo evolutivo genera un máximo de 100 individuos válidos en cada una de las 100 generaciones, por lo que la búsqueda aleatoria debe generar 10.000 soluciones válidas para realizar una comparación justa.

Tabla 2. Configuración de parámetros

Parámetro	**Valor**	**Parámetro**	**Valor**
Tamaño de la población	100	Prob. Mut. *Añadir componente*	0,250
Número de generaciones	100	Prob. Mut. *Eliminar componente*	0,125
Nº máx. de intentos en la mutación	10	Prob. Mut. *Dividir componente*	0,125
W_{coh}	0,300	Prob. Mut. *Unir componentes*	0,250
W_{aco}	0,400	Prob. Mut. *Mover clase*	0,250
W_{cv}	0,300	Número de componentes	Entre 2 y 8

4.2. Resultados experimentales

En la Tabla 3 se recogen los resultados medios de *fitness*, cohesión, acoplamiento y CV obtenidos por los dos algoritmos. Se puede observar cómo el algoritmo de EP consigue mejores soluciones en términos de medias para todas las instancias consideradas. Además, para el primer problema seleccionado se ha alcanzado el menor valor de acoplamiento en todas las ejecuciones.

Aunque la búsqueda aleatoria es capaz de obtener individuos que representan arquitecturas balanceadas en cuanto al tamaño de los componentes, los valores obtenidos para el resto de métricas son muy inferiores a los proporcionados por el algoritmo de EP. Por tanto, es posible alcanzar un compromiso entre cohesión y acoplamiento mediante la evolución de soluciones generadas aleatoriamente.

Más aún, el algoritmo de EP siempre encuentra soluciones con valores de cohesión positivos. Esto resalta la dificultad de generar soluciones donde el número y tipología de las relaciones entre las clases pueda contrarrestar la penalización por la presencia de varias agrupaciones.

Estos resultados también ofrecen información interesante en función de los problemas considerados. Como puede observarse, el valor de *fitness* para la arquitectura de Datapro4j es algo inferior al de los otros problemas debido a la diferencia entre los valores de acoplamiento alcanzados. Este hecho es significativo, pues este sistema presenta un número elevado de relaciones de generalización, las cuales implican la máxima penalización en el cálculo del acoplamiento.

Tabla 3. Resultados del algoritmo de EP y la búsqueda aleatoria

		NekoHTML	AquaLush	Datapro4j
EP	*Fitness*	$0,7301 \pm 0,0014$	$0,6793 \pm 0,0254$	$0,4780 \pm 0,0206$
	Cohesión	$0,1977 \pm 0,0105$	$0,1262 \pm 0,0446$	$0,0767 \pm 0,0991$
	Acopl.	$0,0000 \pm 0,0000$	$0,0600 \pm 0,0899$	$0,9868 \pm 0,2571$
	CV	$0,7806 \pm 0,1208$	$0,9222 \pm 0,0754$	$0,9868 \pm 0,2571$
RS	*Fitness*	$0,6198 \pm 0,0402$	$0,4640 \pm 0,0112$	$0,3316 \pm 0,0102$
	Cohesión	$-0,0390 \pm 0,0828$	$-0,2510 \pm 0,0401$	$0,0541 \pm 0,0980$
	Acopl.	$0,1420 \pm 0,0776$	$0,3694 \pm 0,0346$	$0,9522 \pm 0,1001$
	CV	$0,3129 \pm 0,2576$	$0,3460 \pm 0,1014$	$0,1008 \pm 0,0963$

Considerando la estructura de las soluciones proporcionadas por el algoritmo se pueden apreciar otros aspectos interesantes. En el caso del sistema Aqualush, cuya solución arquitectónica original está basada en capas, se han identificado las agrupaciones de clases que conforman algunas de estas capas. A su vez, cabe destacar la habilidad del algoritmo para generar y mantener diferentes tipos de arquitecturas en cuanto al número de componentes y conectores a lo largo de toda la evolución. Este hecho es importante, pues permite presentar al experto diferentes alternativas, entre las cuales tomará su decisión de diseño.

Finalmente, con el objetivo de comparar estadísticamente los resultados obtenidos por ambos procedimientos, se ha realizado el test de Wilcoxon bajo la hipótesis nula, H_0, de que ambos algoritmos tienen un rendimiento similar. Con un 90 % de confianza, H_0 puede ser rechazada, por lo que existen diferencias significativas entre la propuesta de EP y la búsqueda aleatoria.

5. Conclusiones y trabajo futuro

En este artículo se presenta un algoritmo de programación evolutiva aplicado al ámbito de la identificación de componentes software a partir de modelos de análisis. Se ha definido una codificación en árbol que permite manejar eficientemente las soluciones generadas por el algoritmo, a la vez que ofrece una representación comprensible para usuarios no expertos en técnicas metaheurísticas. La definición de una función de *fitness* basada en criterios de diseño y la creación de operadores genéticos específicos constituyen el resto del modelo.

La aplicación de algoritmos bio-inspirados en el dominio del análisis arquitectónico constituye a su vez una novedad, pues la mayoría de propuestas para

la optimización de arquitecturas en SBSE están orientadas a la refactorización de código o el despliegue.

En este sentido, extender la definición del problema con otros aspectos del análisis y diseño del software constituye una línea de trabajo futura. Finalmente, el algoritmo propuesto puede ser mejorado mediante la inclusión de nuevos operadores que exploren más variantes arquitectónicas. La aplicación de otras metaheurísticas, así como la hibridación de alguna de ellas con EP, constituyen otros retos a considerar en el futuro.

Agradecimientos Este trabajo ha sido apoyado por la Junta de Andalucía y el Ministerio de Ciencia y Tecnología mediante los proyectos P08-TIC-3720 y TIN2011-22408, y fondos FEDER.

Referencias

1. M. Harman, S. A. Mansouri, and Y. Zhang, "Search Based Software Engineering: Trends, Techniques and Applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, 2012.
2. M. Harman, "Software Engineering Meets Evolutionary Computation," *Computer*, vol. 44, no. 10, pp. 31–39, 2011.
3. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2002.
4. O. Räihä, "A survey on search-based software design," *Computer Science Review*, vol. 4, no. 4, pp. 203 – 249, 2010.
5. C. L. Simons, I. C. Parmee, and R. Gwynllyw, "Interactive, Evolutionary Search in Upstream Object-Oriented Class Design," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 798–816, 2010.
6. A. C. Jensen and B. H. Cheng, "On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO '10)*, (Portland, Oregon, USA), pp. 1341–1348, ACM, 7-11 July 2010.
7. S. Kebir, A.-D. Seriai, A. Chaoui, and S. Chardigny, "Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code," in *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, (New York, NY, USA), pp. 1–8, ACM, 2012.
8. A. Martens, D. Ardagna, H. Koziolek, R. Mirandola, and R. Reussner, "A Hybrid Approach for Multi-attribute QoS Optimisation in Component Based Software Systems," in *Proceedings of the 6th International Conference on the Quality of Software Architectures (QoSA '10)*, (Prague, Czech Republic), pp. 84–101, Springer, 23-25 June 2010.
9. S. Vathsavayi, O. Räihä, and K. Koskimies, "Using Quality Farms in Multi-objective Genetic Software Architecture Synthesis," in *Proceedings of IEEE Congress on Evolutionary Computation (CEC '12)*, (Brisbane, Australia), pp. 1–8, IEEE, 10-15 June 2012.
10. S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "JCLEC: a java framework for evolutionary computation," *Soft Comput.*, vol. 12, no. 4, pp. 381–392, 2007.

Identificación de Componentes en Arquitecturas Software Mediante Programación Evolutiva

Aurora Ramírez, José Raúl Romero y Sebastián Ventura

Dpto. de Informática y Análisis Numérico
Universidad de Córdoba, Campus de Rabanales, 14071 Córdoba
{aramirez, jrromero, sventura}@uco.es

Resumen El análisis arquitectónico constituye una tarea indispensable en los grandes sistemas software, pues dedica su esfuerzo a encontrar soluciones de diseño de alto nivel que cumplan requisitos de desempeño, acoplamiento, cohesión o reutilización, entre otros. Sin embargo, aspectos como la experiencia del arquitecto software o la complejidad del sistema repercuten en el resultado. La Inteligencia Artificial (IA) aporta un marco novedoso para el desarrollo de herramientas semi-automáticas en este dominio. En general, el área de SBSE (*Search-Based Software Engineering*) plantea considerar los retos de la Ingeniería del Software como problemas de optimización y búsqueda. Este trabajo presenta una nueva propuesta para la identificación de arquitecturas basadas en componentes a partir de representaciones cercanas al experto y haciendo uso de la metaheurística de Programación Evolutiva (EP). Además, discute los principales retos a los que se enfrenta este tipo de soluciones y desarrolla un estudio experimental que aporta resultados prometedores.

Keywords: Arquitecturas basadas en componentes, SBSE, Programación Evolutiva

1. Introducción

La identificación de componentes, una de las principales actividades del análisis arquitectónico, permite abordar el desarrollo de sistemas complejos mediante la determinación de las distintas unidades funcionales del software y sus interacciones [1]. Realizar el análisis del software a un nivel de abstracción elevado durante las primeras fases del proyecto aporta beneficios en su posterior desarrollo, tanto en esfuerzo como en coste. Por tanto, se trata de una tarea de gran importancia donde la experiencia y habilidad del arquitecto son determinantes.

Disponer de herramientas que apoyen al ingenierio en esta actividad es un campo de trabajo interesante en el cual puede resultar beneficiosa la inclusión de técnicas de IA que hagan uso del conocimiento de los expertos para automatizar el proceso. En este sentido, el crecimiento en los últimos años del área denominada SBSE [2] muestra el interés de la comunidad investigadora en el tratamiento de las tareas de la Ingeniería del Software como problemas de búsqueda y optimización, susceptibles de ser resueltas por técnicas computacionales. Aspectos

como la planificación de proyectos [3] o el desarrollo de casos de prueba [4] han sido ya abordados en el área de SBSE haciendo uso de diferentes metaheurísticas, las cuales constituyen una alternativa eficiente cuando no es posible aplicar técnicas exactas. En este sentido, la Computación Evolutiva (EC, *Evolutionary Computation*) y más concretamente, la Programación Evolutiva (EP, *Evolutionary Programming*), son metaheurísticas muy conocidas que aplican conceptos inspirados en la evolución natural para explorar el espacio de soluciones.

En este trabajo se presenta un estudio experimental para la identificación de componentes en arquitecturas software mediante un modelo de Programación Evolutiva. A través de un proceso orientado al experto, se ha abordado la optimización de arquitecturas basadas en componentes a partir de modelos de análisis. Para ello se ha diseñado una representación flexible y comprensible, una función de evaluación que incluye criterios de diseño deseables y un conjunto de transformaciones que permiten explorar diferentes tipos de arquitecturas.

El resto del artículo está estructurado como sigue. La Sección 2 recoge el estado actual de la problemática, especialmente en lo referido a la presencia de propuestas en el ámbito de SBSE. A continuación, en la Sección 3 se describe de forma detallada el ámbito del problema, centrándose en la codificación de arquitecturas utilizada. El diseño del marco de trabajo y, en concreto, del algoritmo evolutivo, se detallan en la Sección 4. En la Sección 5 se presenta un ejemplo del funcionamiento del algoritmo mientras que la Sección 6 aborda la experimentación y discusión del proceso. Para finalizar, la Sección 7 recoge las conclusiones obtenidas y el trabajo futuro.

2. Problemática y estado actual

La identificación sistemática de componentes software es un área de investigación en la que se dan cabida desde procedimientos generales y recomendaciones, hasta métodos estructurados y algoritmos [5]. La utilización de técnicas basadas en clustering para la extracción de la arquitectura han sido las primeras propuestas destacadas en el área. En estos casos, el problema de recuperar o identificar la arquitectura del software es enfocado como un problema de partición de grafos en el cual se representan sus dependencias funcionales [6].

Recientemente, la optimización de arquitecturas también ha sido tratada desde la perspectiva de SBSE bajo diferentes enfoques [7]. La mayoría de estas propuestas han estado centradas en aspectos cercanos al código fuente, como la refactorización e inclusión de patrones de diseño [8]. En [9], la extracción de arquitecturas basadas en componentes también es realizada a partir de código fuente, donde se utiliza un algoritmo genético para optimizar una arquitectura inicial obtenida tras el análisis de un grafo de dependencias. Sin embargo, el análisis arquitectónico a un nivel de abstracción más elevado, donde se utilice información de los propios modelos de análisis, no ha sido abordado hasta la fecha. Este tipo de propuestas y, en general, todas las de SBSE, están orientadas a dar soporte al experto, de forma que le ayuden en la toma de decisiones, responsabilidad que finalmente recae en el ingeniero software.

Por otro lado, también ha existido variedad en cuanto a las técnicas metaheurísticas aplicadas, destacando la Computación Evolutiva [10]. En ella se manejan simultáneamente varias soluciones al problema, que son «evolucionadas» mediante la aplicación de operadores genéticos como el cruce y la mutación. Habitualmente, cada solución, también denominada individuo, es representada mediante un vector de valores numéricos, cuyas posiciones son denominadas genes, que codifican una posible solución al problema. La Programación Evolutiva [11] se caracteriza por la ausencia de operador de cruce y una representación adaptada al dominio del problema.

3. Definición del problema

La solución propuesta al problema de identificación de componentes ha sido enfocada siguiendo la definición de Szyperski [1], según la cual «un componente es una unidad de composición que especifica contractualmente interfaces y solo hace explícitas dependencias del contexto». En este sentido, el modelo arquitectónico contempla la definición de las interfaces, que determinan el conjunto de operaciones que pueden ser invocadas y que permiten separar su especificación funcional de la implementación real. Dicha implementación es realizada en el interior del componente mediante otros elementos. Finalmente, los conectores enlazan dos o más puntos de interacción entre interfaces.

Las definiciones anteriores son recogidas en la especificación del estándar UML 2 [12], que ofrece una semántica bien definida para el análisis arquitectónico. Esta notación ha sido considerada en este trabajo a la hora de representar el problema, de forma que se utilizan los modelos y artefactos definidos por UML 2.

Puesto que la identificación de componentes constituye una tarea muy compleja, es necesario realizar una descomposición en problemas de menor envergadura cuya resolución pueda ser abordada mediante técnicas computacionales. Por tanto, la resolución del problema es aquí formulada en términos de las siguientes restricciones:

- Un componente es definido como un grupo cohesionado de clases, de forma que entre ellas llevan a cabo el comportamiento del componente.
- Una relación navegable entre clases pertenecientes a diferentes componentes representa una interfaz candidata, pues establece una interacción entre tales componentes. En función de la navegabilidad de la relación, se considerará una interfaz requerida o proveída.
- Los conectores describen la unión de una pareja de interfaces, una requerida y otra proveída, entre dos componentes.

Por otro lado, las métricas disponibles para evaluar los diferentes modelos permiten establecer los criterios de calidad exigidos a las soluciones propuestas. Conceptos genéricos como la funcionalidad, reusabilidad, flexibilidad o comprensión del software son traducidos en medidas cuantificables y ampliamente extendidas como la presencia de abstracción, el uso de la herencia y el polimorfismo [13]. Concretamente, los conceptos de cohesión y acoplamiento están

ampliamente aceptados en el ámbito del diseño basado en componentes como criterios básicos y fundamentales de diseño. La cohesión se define como el grado en el cual el componente desarrolla una funcionalidad bien definida. El acoplamiento mide la interdependencia entre componentes a causa de las interacciones entre sus módulos y los flujos de datos.

4. Diseño del modelo de identificación de componentes

En esta sección se describe el procedimiento propuesto para realizar la identificación de componentes a partir de modelos de análisis. Tras una visión general del mismo, se detallarán los aspectos de mayor interés en la resolución de cualquier problema de SBSE: la representación de las soluciones, las métricas de evaluación de la calidad y las transformaciones de las soluciones propuestas.

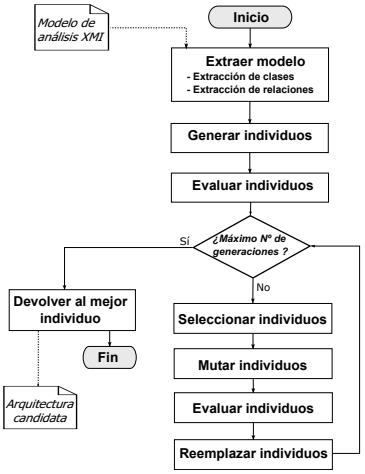


Figura 1: Diagrama de flujo del modelo propuesto

La Figura 1 muestra la secuencia de tareas que componen el proceso. Como entrada al sistema, el experto dispone de un modelo de análisis en el formato estándar XMI [14]. El uso de recomendaciones estándares facilita que no se dependa de representaciones propias, así como la posible integración con herramientas externas. A continuación, se realiza la extracción de la información referida a las clases que intervienen en el modelo, así como las relaciones entre ellas. La herramienta de modelado utilizada ha sido Magic Draw [1], mientras que el proceso de extracción se ha automatizado mediante un intérprete de XMI.

El método de búsqueda comienza con la ejecución del algoritmo evolutivo, el cual es iniciado con la creación de una población de n soluciones generadas

[1] http://www.nomagic.com/products/magicdraw.html

aleatoriamente a partir de la información del modelo, cuya representación se aborda en la Sección 4.1. El proceso de inicialización consiste en la creación de un número aleatorio de componentes en los cuales se ubican, también de forma aleatoria, las clases e interfaces candidatas extraídas del modelo de análisis. A continuación, estas soluciones deben ser evaluadas para determinar su calidad, tal y como se detalla en la Sección 4.2.

El resto del algoritmo es un proceso iterativo que finaliza cuando la condición de parada es alcanzada (un número máximo de iteraciones). En cada una de ellas, denominada generación, se crea una nueva población de soluciones a partir de los individuos de la generación anterior, que toman el rol de padres. Para ello, los operadores genéticos, detallados en la Sección 4.3, son aplicados sobre dichos individuos, generando otros nuevos. Tras la evaluación de las nuevas soluciones se dispone de n padres y n hijos entre los cuales debe seleccionarse aquellos que continuarán en la siguiente generación. En este caso, se mantiene al mejor individuo entre cada padre y su hijo. Finalmente, la mejor solución encontrada es devuelta, indicando tanto la arquitectura resultante como los valores de las diferentes medidas consideradas.

4.1. Representación del problema

Tras definir el marco de la propuesta es posible abordar la codificación de las arquitecturas para ser manejadas por el algoritmo de búsqueda. Los conceptos anteriores deben ser trasladados a una estructura eficiente desde el punto de vista computacional, puesto que es uno de los aspectos clave para el éxito de este tipo de propuestas. A su vez, mantener una representación cercana al experto es otro factor importante, pues influye en la interpretabilidad del proceso.

Este último aspecto es una de las principales diferencias de esta propuesta frente a otros trabajos basados en SBSE, donde la representación lineal de las soluciones, esto es, vectores de valores binarios o numéricos, ha venido siendo la codificación más habitual. Sin embargo, a medida que los problemas son más complejos, este tipo de codificación se complica y su interpretación se vuelve tediosa. La utilización de otro tipo de estructuras, especialmente si han sido aplicadas con éxito en otros ámbitos, se convierte en una alternativa interesante.

Una de las representaciones habituales de los modelos de análisis es en forma de árbol, donde se establece una relación jerárquica entre los elementos que intervienen en el modelo así como referencias entre los nodos. La representación en árbol ofrece las dos características deseables, interpretabilidad y eficiencia, al tratarse de una estructura de datos comprensible y computacionalmente manejable. Gráficamente, la Figura 2 muestra la equivalencia entre el modelo arquitectónico o fenotipo, esto es, lo que representa la solución (aquí modelado con UML 2 pero que en realidad es tratado de forma genérica en formato XMI), y la codificación en forma de árbol o genotipo, su representación interna. El nodo raíz comprende el diagrama completo, compuesto por un conjunto de componentes y conectores. Estos, a su vez, son nodos no terminales del árbol, pues son definidos en función de otros de menor abstracción. Finalmente, las clases e interfaces conforman el último nivel de descomposición del árbol.

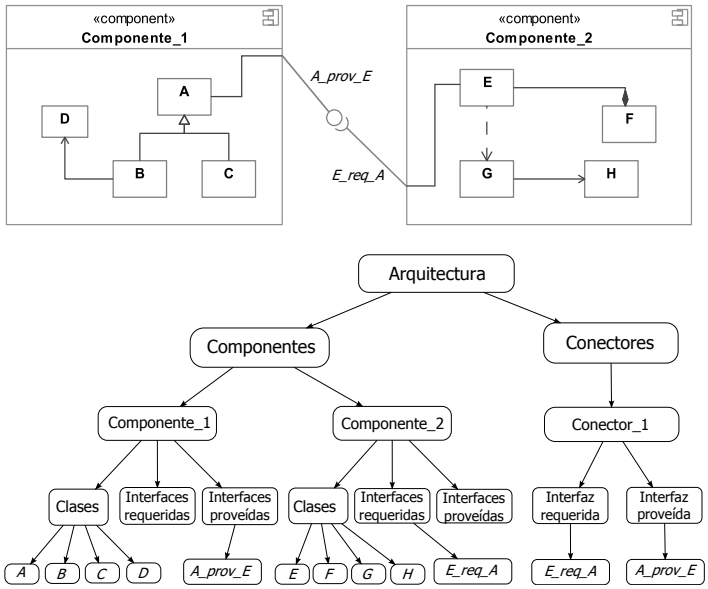


Figura 2: Codificación propuesta del problema (fenotipo y genotipo)

4.2. Función de evaluación

Como se mencionó en la Sección 3, la determinación de las métricas de interés para evaluar la calidad de los modelos arquitectónicos es un aspecto clave para el ingeniero del software. Por otro lado, las metaheurísticas requieren de un mecanismo para decidir cuándo una solución es mejor que otra, de forma que utilizan esta información para dirigir el proceso de búsqueda. La función de evaluación aquí definida (denominada función de *fitness*) está basada en dos características arquitectónicas deseables por el experto, cohesión y acoplamiento, a la vez que sirve como base para establecer una comparación numérica objetiva entre las diferentes soluciones creadas a lo largo del proceso evolutivo.

En primer lugar, la cohesión global de la arquitectura es expresada como la media de la cohesión de cada componente (coh_i), calculada mediante la Ecuación 1. En ella se evalúan dos aspectos: las relaciones entre las clases contenidas en el componente y el número de agrupaciones que lo conforman. El primer término mide el grado de relación entre las clases que implementan la funcionalidad del componente, considerando los distintos tipos de relaciones definidas en UML. El segundo término está fundamentado en el concepto de componentes conectadas en la Teoría de Grafos, bajo la idea de que la presencia de distintos grupos o núcleos de clases puede ser indicativo de la existencia de varias funcionalidades poco relacionadas entre sí. En la Ecuación 1, n_{cl} representa el número de clases dentro del componente y n_{gr} el número de grupos de clases no conectadas entre sí. Los términos n_x indican el número de relaciones internas, clasificadas en asociaciones (as), dependencias (de), agregaciones (ag), compo-

siciones (*co*) y generalizaciones (*ge*). Los dos términos considerados, que varían entre 0 y 1 y entre -1 y 0, respectivamente, son ponderados mediante los pesos w_{c1} y w_{c2}. Tanto estos pesos como los correspondientes a los distintos tipos de relaciones han sido prefijados experimentalmente.

$$coh_i = \frac{w_{c1}}{(w_a + w_d + w_c + w_g) \cdot (n_{cl} - 1)} \cdot \Big[w_a \cdot n_{as} + w_d \cdot n_{de} + w_c \cdot (n_{ag} + n_{co}) + w_g \cdot n_{ge} \Big] + w_{c2} \cdot \frac{1 - n_{gr}}{n_{cl} - 1} \quad (1)$$

A continuación, el cálculo del acoplamiento está enfocado a la cuantificación de las relaciones entre componentes que no son reflejadas por la conexión de sus interfaces. Este tipo de relaciones ocurren a lo largo del proceso evolutivo cuando el agrupamiento de clases no es óptimo y, por tanto, pueden existir relaciones entre clases de diferentes componentes que, en realidad, deberían formar parte del mismo (piénsese en una generalización donde la clase padre y sus respectivas subclases han sido ubicadas en diferentes componentes). La Ecuación 2 define el valor de acoplamiento (*acopl*), que refleja el número de relaciones existentes entre C componentes, variando entre 0 (no existen relaciones fuera de las definidas por las interfaces) y 1 (todos los componentes están conectados con el resto).

De forma equivalente a la expresión de cohesión, el tipo de relación UML es considerada a la hora de penalizar la presencia de conexiones entre componentes. Mediante el término R, definido en la Ecuación 3, se evalúan las relaciones establecidas entre clases pertenecientes a cada par de componentes, contabilizando la más fuerte de todas ellas. Dados dos componentes, i y j, conectados por m relaciones entre las clases alojadas en cada uno, $r^k_{i,j}$, $k = 0...m$, representa el peso del tipo de la relación *k-ésima* entre ellos, establecidos en las mismas condiciones que para la cohesión. El término R_{max} permite normalizar la métrica, de forma que representa la peor situación posible (todos los componentes están acoplados por medio de la relación más fuerte).

$$acopl = \frac{1}{R_{max}} \cdot \frac{2 \cdot R \cdot (C-2)!}{C!} \quad (2)$$

$$R = \sum_{i=1}^{C} \sum_{j=i+1}^{C} max r^k_{i,j} \quad (3)$$

Finalmente, se ha considerado una tercera característica relacionada con el tamaño de los componentes resultantes, con la cual se intentan obtener arquitecturas balanceadas en cuanto al tamaño y complejidad de los componentes. El coeficiente de variación, CV, calculado mediante la Ecuación 4, ha sido introducido como medio para controlar la dispersión en el tamaño de los mismos. En dicha ecuación, μ y σ representan la media y la desviación estándar del tamaño interno del componente (medido como el número de clases), respectivamente.

$$CV = \frac{\sigma}{\mu} \quad (4)$$

Finalmente, es posible construir una función de *fitness* mediante la suma ponderada de las tres componentes anteriores (véase la Ecuación 5). Mientras que la cohesión es una medida a maximizar, el acoplamiento y el coeficiente de variación deben ser minimizados. Para que todas ellas se maximicen, es necesario aplicar las modificaciones aritméticas simples que se observan, donde se utilizan los límites superiores de cada término.

$$fitness = w_{coh} \cdot \frac{\sum_{i=1}^{n} coh_i}{n} + w_{aco} \cdot (1 - acopl) + w_{cv} \cdot \left(\frac{8 - CV}{8}\right) \quad (5)$$

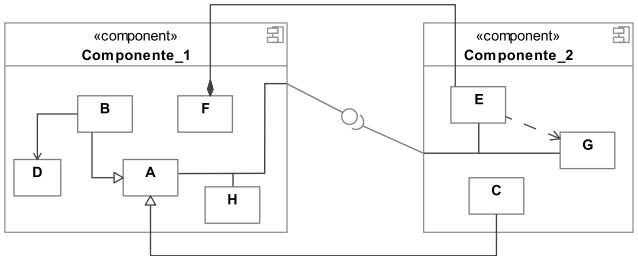


Figura 3: Individuo de ejemplo a evaluar por el algoritmo evolutivo

Tabla 1: Cálculo del *fitness* sobre el individuo de ejemplo

Cohesión	**Acoplamiento**	**CV**
$n_{gr_1} = 3, n_{cl_1} = 5, n_{as_1} = 1, n_{ge_1} = 1$ $coh_{comp\_1} = -0,1750$ $n_{gr_2} = 2, n_{cl_2} = 3, n_{de_2} = 1$ $coh_{comp\_2} = -0,2250$ $coh = \frac{-0,1750+(-0,2250)}{2} = -0,2000$	$r^1_{1,2} = w_{co} = 3$ $r^2_{1,2} = w_{ge} = 5$ $R = max(3,5) = 5$ $R_{max} = w_g = 5$ $acopl = \frac{1}{5} \cdot \frac{2 \cdot 5 \cdot 1}{2} = 1$	$\mu = \frac{5+3}{2} = 4$ $\sigma = \sqrt{(5-4)^2 + (3-4)^2} = 1,4142$ $CV = \frac{1,4142}{4} = 0,3536$
$fitness = 0,3 \cdot (-0,2) + 0,4 \cdot (1-1) + 0,3 \cdot \left(\frac{8-0,3536}{8}\right) = 0,2267$		

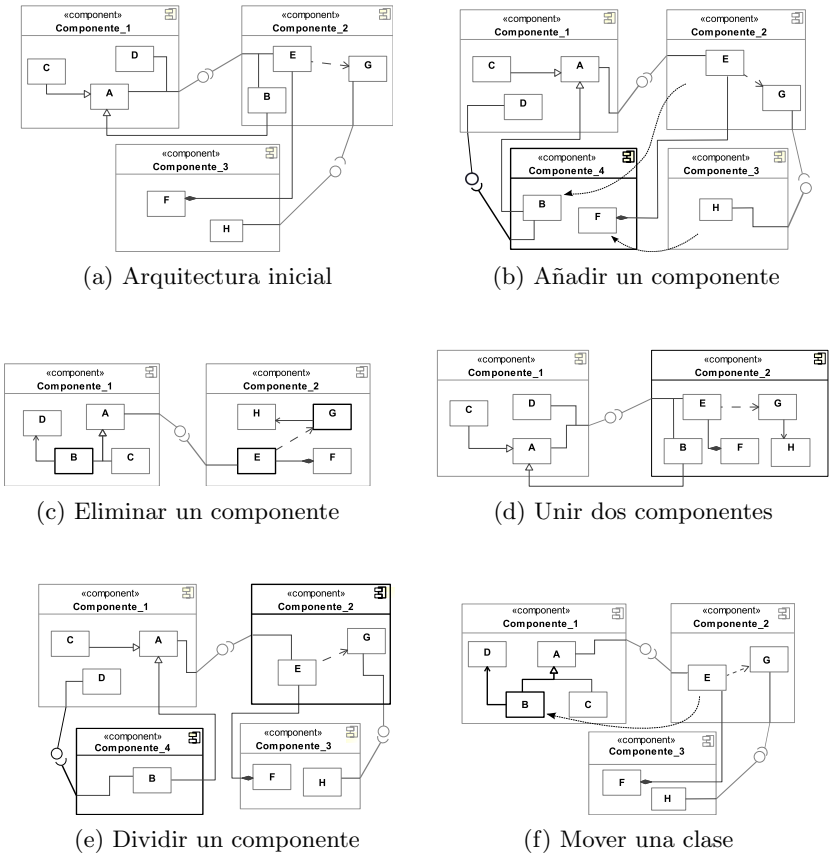
A modo de ejemplo, la Figura 3 muestra una solución no óptima de la arquitectura presentada en la Sección 4.1. El cálculo del valor de *fitness* es recogido en la Tabla 1, donde se ha considerado $w_a = 1$, $w_d = 1$, $w_c = 3$, $w_g = 5$, $w_{c1} = 0,5$ y $w_{c2} = 0,5$. El primer componente presenta cinco clases, de las cuales dos, H y F, no están relacionadas con ninguna otra clase en el interior del componente. Entre el resto se establece una generalización y una asociación. El segundo

presenta tres clases, dos agrupaciones y una relación de dependencia. En ambos casos la penalización por el número de grupos es alta mientras que el número de relaciones internas es baja, de forma que se alcanzan valores negativos para la cohesión. En cuanto al acoplamiento, puede verse cómo existen dos relaciones entre clases de ambos componentes, una generalización entre A y C y una composición entre E y F. Como $w_g = 5$ y $w_c = 3$, R toma el valor 5. A su vez, R_{max} también es igual a 5 pues la peor situación posible para dos componentes es que entre sus clases se establezca una relación de generalización (la de mayor peso). La media y la desviación estándar del número de clases por componente son fácilmente obtenidas. Finalmente, las tres medidas son combinadas según la configuración de los pesos $w_{coh} = 0,3$, $w_{aco} = 0,4$ y $w_{cv} = 0,3$.

4.3. Operadores genéticos

Los operadores de mutación son desarrollados para realizar alteraciones sobre unos individuos (padres) para producir otros nuevos (descendientes). En este problema, los operadores representan transformaciones arquitectónicas que modifican el número de componentes y su distribución interna. Obsérvese que es posible que estas operaciones produzcan arquitecturas inválidas, por lo que debe fijarse un máximo de intentos que, si es alcanzado, finaliza el proceso devolviendo al individuo inicial. A su vez, la utilización de un operador u otro estará condicionada a su factibilidad y a una probabilidad asociada a cada uno de ellos. Cabe destacar que el movimiento de componentes y clases puede implicar la reasignación de otros elementos asociados a ellos, como clases, interfaces y conectores, en función de la nueva estructura definida. En la Figura 4 se muestra el comportamiento de los mutadores propuestos, donde la primera arquitectura corresponde al individuo inicial o padre. De forma más detallada, el funcionamiento de estos operadores es el siguiente:

- *Añadir un componente*: Se crea un nuevo componente extrayendo agrupaciones de clases de otros componentes, seleccionadas aleatoriamente. En el caso (b) se muestra un posible resultado de su aplicación.
- *Eliminar un componente*: Se selecciona el componente con mayor acoplamiento para ser eliminado de la arquitectura. Sus elementos internos (clases, relaciones e interfaces) son realojados en el resto de componentes de forma aleatoria, tal y como se representa en el caso (c).
- *Unir dos componentes*: Se seleccionan dos componentes, el de mayor acoplamiento y uno aleatorio, para formar un nuevo componente. En el ejemplo del caso (d) se han unido los componentes 1 y 3 de la aquitectura inicial.
- *Dividir un componente*: Se selecciona aleatoriamente un componente entre aquellos que presentan más de un grupo diferenciado de clases en su interior, resultando en dos nuevos componentes donde cada uno de ellos mantiene alguno de estos grupos, tal y como se aprecia en el caso (e).
- *Mover una clase*: Tras seleccionar aleatoriamente dos componentes, se produce el movimiento de una clase (también aleatoria) de un componente a otro. En el ejemplo del caso (f), la clase B ha sido desplazada desde el componente 2 hasta el 1.



(a) Arquitectura inicial (b) Añadir un componente

(c) Eliminar un componente (d) Unir dos componentes

(e) Dividir un componente (f) Mover una clase

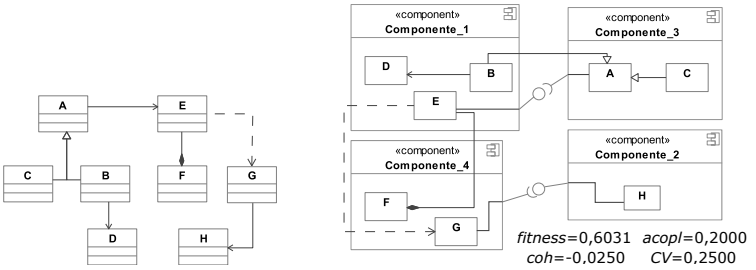
Figura 4: Operadores de transformación definidos

5. Desarrollo de un ejemplo

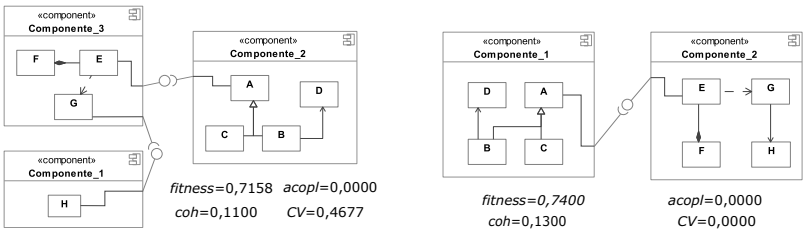
A continuación se desarrollará un pequeño ejemplo sobre la ejecución del algoritmo evolutivo, de forma que se podrá comprobar cómo las soluciones iniciales generadas son optimizadas progresivamente.

En la Figura 5 se recogen cuatro momentos del proceso de identificación de componentes. En primer lugar se dispone de un modelo de análisis (un diagrama de clases) del cual se han extraído las clases participantes y sus relaciones (a). Esta información es utilizada para la creación y evaluación de los individuos de la población inicial (b). Se puede observar cómo el mejor individuo de la población inicial presenta varias relaciones externas entre las clases. Tras 5 generaciones (c), el mejor individuo de la población ha mejorado en cuanto a la distribución de las clases, aunque mantiene el número de componentes. Alcanzadas las 10

generaciones (d), el mejor individuo presenta una arquitectura simplificada en la cual se han distribuido de forma óptima las clases del modelo.



(a) Diagrama de clases (b) Individuo de la población inicial



(c) Mejor individuo (generación 5) (d) Mejor individuo (generación 10)

Figura 5: Ejemplo de funcionamiento del modelo propuesto

6. Experimentación y discusión

La implementación de todos los aspectos del proceso ha sido realizada en Java. Para ello se han utilizado diversas librerías públicas: SDMetrics Open Core [2] para el manejo de archivos XMI, Datapro4j [3] para el preprocesado y manejo de las estructuras de datos intermedias y el *framework* JCLEC [15] para la implementación del algoritmo evolutivo.

Para analizar el rendimiento del algoritmo, se han considerado algunos modelos de análisis correspondientes a sistemas software de diferentes características y complejidad (véase la Tabla 2). NekoHTML [4] y Datapro4j son sistemas reales, mientras que AquaLush [5] es un sistema creado con propósitos educativos.

[2] http://www.sdmetrics.com/OpenCore.html
[3] http://www.uco.es/grupos/kdis/datapro4j
[4] http://nekohtml.sourceforge.net
[5] http://www.ifi.uzh.ch/rerg/research/aqualush.html

Tabla 2: Instancias del problema y sus características

	#Clases	#Asoc	#Depen	#Agreg	#Compos	#Gener
NekoHTML	24	16	5	0	0	10
AquaLush	58	69	6	0	0	20
Datapro4j	59	4	18	1	4	50

6.1. Configuración del algoritmo

La Tabla 3 detalla los parámetros del algoritmo evolutivo y los valores que han sido considerados tras realizar experimentos preliminares. Estos parámetros son comunes para las tres instancias del problema. A su vez, dada la aleatoriedad del algoritmo, se han realizado 30 ejecuciones con diferentes semillas aleatorias. Puesto que no se dispone de procedimientos similares con los cuales comparar esta propuesta, se ha realizado también una búsqueda aleatoria (RS, *Random Search*), proceso habitual en el ámbito de las metaheurísticas.

Tabla 3: Configuración de parámetros

Parámetro	**Valor**	**Parámetro**	**Valor**
Tamaño de la población	100	Prob. Mut. *Añadir componente*	0,250
Número de generaciones	100	Prob. Mut. *Eliminar componente*	0,125
Nº de intentos en la mutación	10	Prob. Mut. *Dividir componente*	0,125
W_{coh}	0,300	Prob. Mut. *Unir componentes*	0,250
W_{aco}	0,400	Prob. Mut. *Mover clase*	0,250
W_{cv}	0,300	Número de componentes	Entre 2 y 8

6.2. Ejecución y discusión de resultados

La Tabla 4 recoge los valores medios de *fitness*, cohesión, acoplamiento y CV obtenidos sobre las tres arquitecturas propuestas. Cabe recordar que, mientras que el *fitness* y la cohesión deben ser maximizadas, el acoplamiento y el CV son minimizadas. Como puede apreciarse, el algoritmo es capaz de obtener mejores resultados que la búsqueda aleatoria, especialmente en lo que a cohesión y acoplamiento se refiere.

Un resultado reseñable es la diferencia en la medida de acoplamiento entre las dos primeras arquitecturas propuestas y Datapro4j. La arquitectura de este último presenta un número de generalizaciones muy superior al resto, lo cual tiene un impacto notable en el acoplamiento, pues este tipo de relaciones acumulan la máxima penalización en esta medida.

Tabla 4: Resultados del algoritmo evolutivo (EP) y la búsqueda aleatoria (RS)

	EP				**RS**			
Problema	*Fitness*	Cohesión	Acopl.	CV	*Fitness*	Cohesión	Acopl.	CV
NekoHTML	0,7301	0,1977	0,0000	0,7806	0,6198	-0,0390	0,1420	0,3129
AquaLush	0,6793	0,1262	0,0600	0,9222	0,4640	-0,2510	0,3694	0,3460
Datapro4j	0,4780	0,0767	0,5200	0,9868	0,3316	0,0541	0,9522	0,1008

En general, las arquitecturas obtenidas por el algoritmo presentan agrupaciones interesantes de clases. A su vez, el algoritmo es capaz de encontrar componentes presentes en la arquitectura real, así como mantener soluciones con diferente número de componentes y conectores a lo largo de toda la evolución. Estos resultados permiten mostrar al ingeniero software diferentes alternativas entre las cuales tomar una decisión.

Además, para validar la propuesta se ha realizado un test estadístico para analizar el rendimiento de ambos algoritmos. Se ha utilizado el test de Wilcoxon, el cual permite comparar dos algoritmos bajo la hipótesis nula (H_0) de que ambos obtienen un rendimiento similar. Con un 90 % de confianza, H_0 puede ser rechazada, de forma que existen diferencias significativas entre los dos algoritmos.

7. Conclusiones y trabajo futuro

En este artículo se ha presentado un modelo de identificación y optimización de arquitecturas basadas en componentes mediante un algoritmo evolutivo. La problemática planteada ha sido reformulada como un problema de búsqueda, de forma que se han trasladado los conceptos y modelos que maneja habitualmente el arquitecto del software. En este sentido, este trabajo representa una novedad en el nivel de abstracción considerado, pues no existen propuestas de SBSE que utilicen modelos de análisis para la extracción de componentes.

Mediante la definición de una representación apropiada, un conjunto de métricas que permiten guiar la búsqueda y un conjunto de operadores de mutación que codifican transformaciones comprensibles, se ha obtenido un proceso de análisis arquitectónico que combina la comprensión del experto y la potencia de la Programación Evolutiva.

Como trabajo futuro, pretendemos extender la información utilizada en el proceso, por ejemplo, con la definición de atributos y métodos o de subcomponentes. Por otro lado, las metaheurísticas ofrecen una gran variedad de técnicas y procedimientos que pueden ser explorados para mejorar el funcionamiento del algoritmo propuesto.

Agradecimientos Trabajo apoyado por el Ministerio de Ciencia y Tecnología, proyecto TIN2011-22408, y fondos FEDER.

Referencias

1. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2002.
2. M. Harman, S. A. Mansouri, and Y. Zhang, "Search Based Software Engineering: Trends, Techniques and Applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, 2012.
3. L. L. Minku, D. Sudholt, and X. Yao, "Evolutionary algorithms for the project scheduling problem: runtime analysis and improved design," in *Proceedings of the 14th International Conference on Genetic and Evolutionary Computation Conference (GECCO '12)*, (Philadelphia, USA), pp. 1221–1228, ACM, 2012.
4. J. Ferrer, P. M. Kruse, F. Chicano, and E. Alba, "Evolutionary Algorithm for Prioritized Pairwise Test Data Generation," in *Proceedings of the 14th International Conference on Genetic and Evolutionary Computation Conference (GECCO '12)*, (Philadelphia, USA), pp. 1213–1220, ACM, 2012.
5. D. Birkmeier and S. Overhage, "On Component Identification Approaches — Classification, State of the Art, and Comparison," in *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, CBSE '09, (Berlin, Heidelberg), pp. 1–18, Springer-Verlag, 2009.
6. O. Maqbool and H. Babri, "Hierarchical Clustering for Software Architecture Recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759 –780, 2007.
7. A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software Architecture Optimization Methods: A Systematic Literature Review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013.
8. A. C. Jensen and B. H. Cheng, "On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns," in *Proceedings of the 12th International Conference on Genetic and Evolutionary Computation (GECCO '10)*, (Portland, Oregon, USA), pp. 1341–1348, ACM, 7-11 July 2010.
9. S. Kebir, A.-D. Seriai, A. Chaoui, and S. Chardigny, "Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code," in *Proceedings of the 5th International C* Conference on Computer Science and Software Engineering*, C3S2E '12, (New York, NY, USA), pp. 1–8, ACM, 2012.
10. M. Harman, "Software Engineering Meets Evolutionary Computation," *Computer*, vol. 44, no. 10, pp. 31–39, 2011.
11. D. B. Fogel and L. J. Fogel, "An introduction to evolutionary programming," in *Artificial Evolution*, vol. 1063 of *Lecture Notes in Computer Science*, pp. 21–33, Springer Berlin Heidelberg, 1996.
12. OMG, *Unified Modeling Language 2.4 Superstructure Specification*. OMG, nov 2010. formal/2010-11-14, http://www.omg.org/spec/UML/2.4/.
13. L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationship between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, 2000.
14. OMG, *MOF 2 XMI Mapping Specification*. OMG, aug 2011. formal/2011-08-09, http://www.omg.org/spec/XMI/2.4.1/.
15. S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "JCLEC: a java framework for evolutionary computation," *Soft Comput.*, vol. 12, no. 4, pp. 381–392, 2007.