Búsqueda de errores en programas usando Java PathFinder y ACOhg

Francisco Chicano y Enrique Alba

Resumen— **Model checking es una técnica automática bien conocida para comprobar si un determinado programa cumple una serie de propiedades deseadas, como un invariante o la ausencia de interbloqueos. El uso de esta técnica es un deber en sistemas críticos tales como aviones y centrales nucleares, por ejemplo. La mayoría de los model checkers que se pueden encontrar en la literatura hacen uso de algoritmos exhaustivos y deterministas para comprobar estas propiedades. La memoria requerida para la verificación con estos algoritmos crece de forma exponencial con el tamaño del programa a verificar. Sin embargo, cuando la búsqueda de errores con pocos recursos computacionales es una prioridad, se puede hacer uso de algoritmos no exhaustivos para realizar la búsqueda. En este trabajo proponemos el uso de ACOhg, un algoritmo basado en colonias de hormigas para llevar a cabo la búsqueda de errores en programas concurrentes. En el pasado este algoritmo ha dado muy buenos resultados en el model checker HSF-SPIN. En esta ocasión pretendemos estudiar su eficacia en otro model checker: Java PathFinder.**

Palabras clave— **Optimización basada en Colonias de Hormigas, Model Checking**

I. Introducción

Desde el principio de la Informática, los ingenieros están interesados en técnicas que permitan conocer si un fragmento de código cumple una serie de requisitos (la especificación). Estas técnicas son especialmente importantes en el software para sistemas críticos responsables de vidas humanas como, por ejemplo, los controladores para aviones, los sistemas de control de plantas nucleares o el software para herramientas médicas. También tienen especial relevancia en el software para entidades financieras, donde un error software puede implicar la pérdida de grandes cantidades de dinero. Además de esto, el software actual es muy complejo y estas técnicas se han convertido en una necesidad en muchas compañías de software. La *verificación formal* constituye un ejemplo de estos métodos donde las propiedades del software se demuestran como si de un teorema matemático se tratase. Una lógica muy conocida usada en esta verificación es la *lógica de Hoare*. No obstante, la verificación formal usando lógicas no es completamente automática. Aunque los demostradores de teoremas pueden ayudar en el proceso, es necesaria aún la intervención humana.

Otra técnica bien conocida y completamente automática es *model checking* [1], usada especialmente en el caso de programas concurrentes, que permite comprobar si un determinado programa satisface una propiedad dada, como por ejemplo, la ausencia de interbloqueos (*deadlocks*), la posposición indefinida (*starvation*), el cumplimiento de invariantes, etc. En este caso todos los posibles estados del programa se analizan para demostrar (o negar) que éste satisface una determina propiedad. Las propiedades suelen especificarse usando lógicas temporales como la *lógica temporal lineal* (LTL) o la *lógica de árboles de computación* (CTL). El principal problema de esta técnica es que la memoria requerida para realizar tal verificación suele crecer exponencialmente con el tamaño del programa a verificar. A esto se le conoce como *problema de la explosión de estados* y limita el tamaño de los programas que se pueden verificar. En este trabajo nos referiremos al *model checking explícito*, en el cual los distintos estados de un programa son generados y representados explícitamente en memoria. Esto contrasta con el *model checking simbólico* [1], en el cual se usa una estructura de datos muy eficiente, el *diagrama de decisión binaria ordenado*, para almacenar conjuntos de estados.

En lugar de aplicar *model checking* al programa directamente, se aplica normalmente a un modelo suyo, que será más pequeño, reduciendo así el problema de la explosión de estados. Hay varios lenguajes diseñados específicamente para modelar software, como por ejemplo, Promela (*Process Meta-Language*), el lenguaje de SMV o el lenguaje intermedio de SAL. Una interesante excepción es la del *model checker* Java PathFinder [2], que usa el lenguaje Java para modelar y, en sus últimas versiones, trabaja directamente sobre ficheros de *bytecodes* de Java.

Además del uso de modelos existen varias técnicas desarrolladas para aliviar el problema de la explosión de estados. Éstas reducen la memoria necesaria para la búsqueda siguiendo diferentes enfoques [3]. Por un lado hay técnicas que reducen la cantidad de estados a explorar para realizar la verificación como, por ejemplo, la reducción de orden parcial y la reducción de simetría. Por otro lado, encontramos técnicas que reducen el espacio que ocupa un estado en memoria: compresión de estados, representación mínima de autómatas y tablas hash de un bit. No obstante, las técnicas de búsqueda exhaustivas siempre tienen problemas para verificar programas reales porque la mayoría de estos programas son demasiado complejos incluso para las técnicas más avanzadas.

Cuando la búsqueda de errores con una cantidad reducida de recursos computacionales (memoria y tiempo) es una prioridad (por ejemplo, en las

Departamento de lenguajes y Ciencias de la Computación, Universidad de Málaga. E-mail: {chicano, eat}@lcc.uma.es.

primeras etapas de la implementación de un programa), se pueden usar algoritmos no exhaustivos que utilizan información heurística. Los algoritmos no exhaustivos pueden encontrar errores en programas usando menos recursos computacionales que los exhaustivos, pero no pueden usarse para verificar una propiedad: cuando no se encuentra ninguna traza de error usando un algoritmo no exhaustivo no podemos asegurar que no existan dichas trazas.

La búsqueda de errores en programas concurrentes se puede transformar en un problema de optimización y, por tanto, se pueden aplicar algoritmos metaheurísticos. De hecho, se han aplicado en el pasado algoritmos genéticos a este problema. En una primera propuesta, Alba y Troya [4] usaron algoritmos genéticos para detectar interbloqueos, estados inútiles y transiciones inútiles en protocolos de comunicación. Más tarde, Godefroid y Kurshid [5], en un trabajo independiente, aplicaron algoritmos genéticos al mismo problema usando una codificación similar de las trayectorias en el cromosoma. Su algoritmo se integró dentro de VeriSoft [6], un *model checker* que puede verificar programas en C.

Los autores del presente artículo han propuesto anteriormente el uso de colonias de hormigas para este problema, ya que, a su juicio, se trata de un algoritmo muy natural para un problema que se formula como una búsqueda en un grafo [7]. Los buenos resultados obtenidos los animaron a avanzar en esa línea de investigación [8] y aplicar otras metaheurísticas [9]. En dichos trabajos el *model checker* en el que se integró la propuesta basada en colonias de hormigas fue HSF-SPIN en todos los casos. Esta herramienta, desarrollada por Edelkamp, Leue y Lluch-Lafuente, tiene el inconveniente de que toma como entrada código Promela, un lenguaje poco conocido. En el presente trabajo pretendemos avanzar en la línea de investigación presentando el algoritmo de búsqueda en el *model checker* Java PathFinder (JPF), que trabaja con código Java, lenguaje mucho más conocido y utilizado.

El artículo se organiza como sigue. En la siguiente sección se presentan los fundamentos de *model checking* y en la Sección III se formaliza el problema. La Sección IV describe el algoritmo ACOhg, utilizado para resolver el problema. Posteriormente, en la Sección V se evalúa la propuesta mediante un estudio experimental. Finalmente, la Sección VI presenta las conclusiones finales y el trabajo futuro.

II. Fundamentos del problema

Sea S el conjunto de *estados* de un programa, S^ω el conjunto de secuencias infinitas de estados del programa, y S^* el conjunto de secuencias finitas de estados. Llamaremos a los elementos de S^ω *ejecuciones* y a los elementos de S^* *ejecuciones parciales*. Una *propiedad* P es un conjunto de ejecuciones, $P \subseteq S^\omega$. Decimos que una ejecución $\sigma \in S^\omega$ *satisface* la propiedad P si $\sigma \in P$, y decimos que σ *viola* la propiedad si $\sigma \notin P$. En el primer caso usamos la notación $\sigma \vdash P$, y en el segundo $\sigma \nvdash P$. Una propiedad P es de *seguridad* si para todas las ejecuciones σ que violan la propiedad existe un prefijo σ_i (ejecución parcial) tal que todas las extensiones de σ_i violan la propiedad. Formalmente,

$$\forall \sigma \in S^\omega : \sigma \nvdash P \Rightarrow (\exists i \geq 0 : \forall \beta \in S^\omega : \sigma_i \beta \nvdash P) \ , \quad (1)$$

donde σ_i es la ejecución parcial compuesta por los primeros i estados de σ. Algunos ejemplos de propiedades de seguridad son la ausencia de interbloqueos y el cumplimiento de invariantes. Por otro lado, una propiedad P es de *viveza* si para todas las ejecuciones parciales α del programa existe al menos una extensión que satisface la propiedad, es decir,

$$\forall \alpha \in S^* : \exists \beta \in S^\omega, \alpha\beta \vdash P \ . \quad (2)$$

Un ejemplo de propiedad de viveza es la posposición indefinida. La única propiedad que es a la vez de seguridad y viveza es la propiedad trivial $P = S^\omega$. Se puede demostrar que una propiedad cualquiera puede expresarse como la intersección de una propiedad de viveza y otra de seguridad [10].

Las propiedades de un programa se especifican normalmente usando una lógica temporal como la LTL o la CTL. En ese caso, se definen proposiciones atómicas basadas en el valor de las variables y los contadores de programa de los procesos. Las propiedades se especifican entonces usando fórmulas lógicas temporales sobre estas proposiciones atómicas. Por ejemplo, una propiedad LTL podría ser $\Box p$ (leído "siempre en el futuro p"), donde $p \equiv x > 3$. Esta propiedad especifica un invariante: para que un programa la cumpla la variable x debe ser siempre mayor que 3. La propiedad en este caso está formada por todas las ejecuciones σ en las que la variable x es mayor que 3 en todos los estados de σ.

A. Verificación en JPF

JPF no permite la especificación de propiedades mediante lógicas temporales o autómatas. En principio, con JPF sólo se puede comprobar la ausencia de interbloqueos y de excepciones no capturadas, propiedades de seguridad para las que no es necesario el uso de autómatas. No obstante, JPF es un *model checker* abierto que permite la incorporación de nuevas propiedades implementadas por el usuario.

Para la búsqueda de errores, JPF toma el programa ya compilado (*bytecodes* de Java) y usa su propia implementación de la máquina virtual Java (JPF-JVM en adelante), en la que puede hacer avanzar el programa objeto de la verificación instrucción a instrucción. Además, puede consultar en cualquier momento el estado de la máquina virtual así como almacenar y restaurar estados previamente guardados. Desde el punto de vista del código a verificar la JPF-JVM no es distinta de cualquier otra máquina virtual Java, la ejecución de las instrucciones tendrá el

mismo efecto[1]. La JPF-JVM estará controlada en todo momento por un objeto que se encarga de realizar la búsqueda. Este objeto debe ser una instancia de una subclase de la clase `Search`. Por tanto, para implementar un nuevo algoritmo de búsqueda dentro de JPF hay que crear una nueva clase e implementar los métodos correspondientes. Al hacer esto, JPF amplía las opciones de extensión de la plataforma, un aspecto que se echa en falta en otros *model checkers* como SPIN. La labor de los algoritmos de búsqueda consistirá en controlar la máquina virtual de acuerdo a su estrategia e identificar la presencia de violaciones de las propiedades en los estados que se alcancen dentro de la misma.

B. *Model checking heurístico*

En general, la búsqueda en un programa de ejecuciones que violen una propiedad de seguridad puede tratarse como la búsqueda de un nodo (un estado del programa que cumple cierta condición) en un grafo (el formado por todos los estados del programa junto con los arcos que indican cuál es sucesor de cuál). Por ejemplo, en el caso de JPF, si queremos comprobar la ausencia de interbloqueos los nodos a buscar son aquéllos que no tienen sucesores. Esta información se puede obtener consultando la JPF-JVM.

Al transformar el problema a una búsqueda de un nodo en un grafo podemos hacer uso de algoritmos clásicos para la exploración de grafos como la búsqueda primero en profundidad (*Depth First Search*, *DFS*) o la búsqueda primero en anchura (*Breadth First Search*, *BFS*). También es posible aplicar algoritmos de búsqueda que hagan uso de información heurística, como A*, *Weighted* A*, *Iterative Deeping* A*, y *búsqueda primero del mejor* (*Best-First search*). Cuando se usa información heurística es necesario asociar a cada estado un valor heurístico que depende de la propiedad a verificar y que indica la preferencia por explorar ese estado. Esto se hace por medio de una función heurística h que asigna a cada estado un valor numérico. Cuanto menor sea dicho valor mayor es la preferencia por explorar dicho estado, ya que se supone que puede encontrarse más cerca del estado de error.

El uso de heurísticas para guiar la búsqueda de errores en *model checking* se conoce como *model checking heurístico* o *guiado*. Las funciones heurísticas se diseñan para dirigir la exploración en primer lugar a la región del espacio de estados en la que es más probable encontrar un estado de error. De este modo, el tiempo y la memoria requeridos para encontrar un error en un programa se reduce en término medio. No obstante, el uso de las heurísticas no supone ninguna ventaja cuando el objetivo es verificar que un programa dado cumple una determinada propiedad. En este caso, todo el espacio de estados se debe explorar exhaustivamente.

[1]En realidad, en JPF los usuarios pueden proporcionar nuevos comportamientos para las instrucciones, lo cual ha dado lugar a interesantes extensiones de JPF que permiten, entre otras cosas, la ejecución simbólica de *bytecodes*.

III. Formalización del problema

Como se ha mencionado anteriormente, el problema de buscar una violación de una propiedad de seguridad se puede traducir en la búsqueda de un nodo en un grafo comenzando en un nodo inicial. En realidad lo que interesa al programador que quiere descubrir errores en el programa es la secuencia de estados que va desde el estado inicial del programa al estado de error, es decir, la ejecución parcial que acaba con un error. Este problema se puede formalizar del siguiente modo.

Sea $G = (S, T)$ un grafo orientado donde S es el conjunto de nodos y $T \subseteq S \times S$ es el conjunto de arcos. Sea $q \in S$ el *nodo inicial* del grafo y $F \subseteq S$ un conjunto de nodos distinguidos que llamamos *nodos finales*. Denotamos con $T(s)$ al conjunto de sucesores del nodo s. Un camino finito sobre el grafo es una secuencia de nodos $\pi = s_1 s_2 \ldots s_n$ donde $s_i \in S$ para $i = 1, 2, \ldots, n$. Denotamos con π_i al i-ésimo nodo de la secuencia y usamos $|\pi|$ para referirnos a la longitud del camino, es decir, el número de nodos de π. Decimos que un camino π es un *camino inicial* si el primer nodo del camino es el nodo inicial del grafo, es decir, $\pi_1 = q$. Usaremos π_* para referirnos al último nodo de la secuencia π, es decir, $\pi_* = \pi_{|\pi|}$.

Dado un grafo orientado G el problema consiste en encontrar un camino inicial π que termine en un nodo final. Es decir, encontrar π con $\pi_1 = q \wedge \pi_* \in F$.

El grafo G usado en el problema es, en el caso que nos ocupa, el grafo de estados del programa, donde un nodo t es sucesor de otro s si en dicho programa se puede pasar de s a t mediante la ejecución de una instrucción. El nodo inicial q de G es el estado inicial del programa y el conjunto de nodos finales F de G es el conjunto de estados en los que se incumple la propiedad que queremos verificar: bien hay un interbloqueo o bien una excepción no capturada.

IV. ACOhg

Para resolver el problema formulado en la sección anterior hemos hecho uso de una nueva variante de algoritmo basado en colonias de hormigas llamado ACOhg (*Ant Colony Optimization for huge graphs*). Esta variante fue propuesta por los autores del presente trabajo para resolver mediante ACOs problemas de optimización en los que el grafo de construcción tiene un tamaño desconocido o es muy grande como para almacenarlo completamente en memoria.

Los algoritmos basados en colonias de hormigas [11] son algoritmos de optimización global inspirados en el comportamiento de algunas especies de hormigas cuando buscan comida. La idea principal consiste en simular este comportamiento en un grafo, el *grafo de construcción*, para buscar el camino más corto desde un nodo inicial a uno objetivo. La cooperación entre las hormigas simuladas es un fac-

tor clave en la búsqueda que se lleva a cabo de forma indirecta por medio de los *rastros de feromona*, un modelo de las sustancias químicas que las hormigas reales usan para su comunicación. Las principales operaciones de un ACO son la *fase de construcción* y la *actualización de feromona*. En la primera, cada hormiga artificial sigue un camino en el grafo de construcción. En la actualización de feromona, los rastros de feromona asociados a los nodos o arcos del grafo se actualizan teniendo en cuenta los caminos seguidos por las hormigas.

Las dos diferencias principales entre ACOhg y los ACOs tradicionales son las siguientes. En primer lugar, la longitud de los caminos (definido como el número de arcos del camino) que describen las hormigas durante la fase de construcción está limitado. Es decir, cuando el camino de una hormiga alcanza una determinada longitud λ_{ant} la hormiga se detiene. En segundo lugar, las hormigas comienzan su camino en distintos nodos del grafo durante la búsqueda. Al principio, las hormigas se colocan en el nodo inicial del grafo, y el algoritmo se ejecuta durante un número de determinado de pasos σ_s (a este periodo se le llama *etapa*). Si no se encuentra ningún nodo objetivo, los últimos nodos de los mejores caminos encontrados se usan como nodos de inicio para las hormigas de la siguiente etapa. De esta forma, durante la siguiente etapa las hormigas avanzarán su exploración en el grafo. (véase [7] para más detalles). En el Algoritmo 1 presentamos el pseudocódigo de ACOhg.

En lo que sigue describiremos el algoritmo, pero antes debemos aclarar algunas cuestiones relacionadas con la notación empleada en el pseudocódigo. Para empezar, el camino descrito por la k-ésima hormiga se denota con a^k. Por esta razón usamos la misma notación que en la Sección III para referirnos a la longitud del camino ($|a^k|$), el j-ésimo nodo del camino (a^k_j), y el último nodo del camino (a^k_*). Usamos el operador $+$ para referirnos a la concatenación de dos caminos. El conjunto *init* contiene caminos iniciales y *next_init* es el conjunto de los mejores caminos encontrados en una etapa. El valor de la variable *stage* no afecta al comportamiento del algoritmo, se incluye únicamente para indicar cuándo hay un cambio de etapa. Como en todas las metaheurísticas, para guiar la búsqueda es necesario definir una función objetivo que asigne un valor real a cada camino indicando su coste y que el algoritmo deberá minimizar. Esta función, que denotaremos con f, se define como sigue

$$f(a^k) = \begin{cases} |\pi + a^k| \text{ si } a^k_* \in F \\ |\pi + a^k| + p \text{ si } a^k_* \notin F \end{cases}, \quad (3)$$

donde π es el camino inicial de *init* cuyo último nodo es el primero de a^k y p es una penalización añadida cuando la hormiga no termina en un nodo final.

El algoritmo funciona como sigue. Al principio, se inicializan las variables (líneas 1-5). Los rastros de feromona son todos inicializados con el mismo valor: un número aleatorio entre τ_0^{min} y τ_0^{max}. En el conjunto *init* se inserta un camino inicial formado únicamente por el estado inicial (línea 1). De esta forma, todas las hormigas de la primera etapa comienzan la construcción de sus caminos en el nodo inicial.

Algoritmo 1 Algoritmo ACOhg

```
1: init ← {q};
2: next_init ← ∅;
3: τ ← initialize_pheromone();
4: step ← 1;
5: stage ← 1;
6: while step ≤ m ∧ ∄i ∈ [1..csize] • a^i_* ∈ F do
7: for k = 1 to csize do {Ant operations}
8: a^k ← ∅;
9: a^k_1 ← select_init_node_randomly(init);
10: while |a^k| ≤ λ_ant ∧ a^k_* ∉ F do
11: node ← select_successor(a^k_*, T(a^k_*), τ, η);
12: a^k ← a^k + node;
13: end while
14: next_init ← select_best_paths(init, next_init, a^k);
15: if f(a^k) < f(a^best) then
16: a^best ← a^k;
17: end if
18: end for
19: τ ← pheromone_evaporation(τ, ρ);
20: τ ← pheromone_update(τ, a^best);
21: if step ≡ 0 mod σ_s then
22: init ← next_init;
23: next_init ← ∅;
24: stage ← stage + 1;
25: τ ← pheromone_reset();
26: end if
27: step ← step + 1;
28: end while
```

Tras la inicialización, el algoritmo entra en un bucle que se ejecuta hasta que se alcanza un determinado número de pasos o alguna hormiga alcanza un nodo final (línea 6). En el cuerpo del bucle, cada hormiga describe una camino comenzando en el nodo final de un camino anterior (línea 9). Este camino se selecciona de forma aleatoria del conjunto *init* utilizando una distribución de probabilidad que asigna a cada camino una probabilidad proporcional a su valor de fitness. Para la construcción del camino, las hormigas entran en un bucle (líneas 10-13) en el que cada hormiga k selecciona aleatoriamente el siguiente nodo de acuerdo al rastro de feromona (τ_j) y el valor heurístico (η_j) asociado con cada nodo j (línea 11). La probabilidad de que la k-ésima hormiga ubicada en el nodo i escoja el nodo $j \in T(i)$ es

$$p^k_{ij} = \frac{[\tau_j]^\alpha [\eta_j]^\beta}{\sum_{s \in T(i)} [\tau_s]^\alpha [\eta_s]^\beta}, \text{ para } j \in T(i), \quad (4)$$

donde α y β son dos parámetros del algoritmo que determinan la influencia relativa de los rastros de feromona y el valor heurístico en la construcción

del camino, respectivamente (véase la Figura 1). De acuerdo con la expresión anterior, las hormigas artificiales prefieren aquellos caminos con mayor concentración de feromona, como las hormigas reales en el mundo real. Cuando una hormiga tiene que elegir un nodo, el último nodo del camino actual se expande. La hormiga selecciona un nodo sucesor y los demás son descartados. Toda la fase de construcción se repite hasta que la hormiga alcanza la longitud máxima permitida λ_{ant} o encuentra un nodo final.

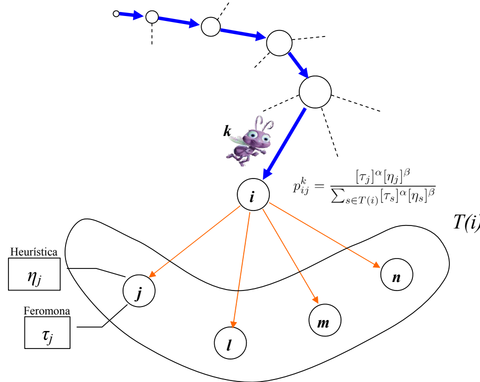

Fig. 1. Una hormiga durante la fase de construcción.

En este momento debemos aclarar algo acerca de las dos funciones heurísticas que se han presentado anteriormente: η y h. La función heurística η depende de cada nodo del grafo de construcción y se define en el contexto de los algoritmos basados en colonias de hormigas. Cuanto mayor es dicho valor η_j, mayor será la probabilidad de que el nodo j sea elegido durante la fase de construcción. La segunda función heurística, h, se define en el contexto del problema (véase la discusión anterior sobre *model checking* heurístico), depende de cada estado del programa y está pensada para ser minimizada. En nuestro caso debemos definir η en función de h. La expresión exacta que usaremos es $\eta_j = 1/(1+h(j))$. De esta forma, η_j aumenta cuando $h(j)$ decrece (alta preferencia para explorar el nodo j).

Tras la fase de construcción, la hormiga se usa para actualizar el conjunto *next_init* (línea 14), que será el conjunto *init* en la siguiente etapa. En *next_init*, sólo puede haber caminos iniciales y, además, todos los caminos deben tener nodos finales distintos (esta condición la asegura la función *select_best_paths*). Un camino a^k se inserta en este conjunto si su último nodo no es el último de un camino inicial π ya incluido en el conjunto. Si esto no se cumple, el nuevo camino a^k reemplaza al camino inicial π de *next_init* sólo si $f(a^k) < f(\pi)$. Antes de la inclusión, el camino debe concatenarse con el camino inicial correspondiente de *init*, es decir, el camino inicial π con $\pi_* = a_1^k$ (este camino existe y es único). De este modo, sólo se incluyen en el conjunto *next_init* caminos iniciales. La cardinalidad de *next_init* está acotada por un parámetro ι. Cuando este límite se alcanza y se debe incluir un nuevo camino, el camino inicial cuya imagen por f sea mayor se elimina del conjunto.

Cuando todas las hormigas han construido sus caminos se actualiza el valor de los rastros de feromona. Primero, todos los rastros son reducidos (evaporación) de acuerdo con la expresión $\tau_j \leftarrow (1-\rho)\tau_j$ (línea 19), donde ρ es la *tasa de evaporación de feromona* y cumple $0 < \rho \leq 1$. Después, los rastros de feromona asociados con los nodos visitados por la mejor hormiga encontrada (a^{best}) aumentan (línea 20) usando la expresión

$$\tau_j \leftarrow \tau_j + \frac{1}{f(a^{best})}, \ \forall j \in a^{best} \ . \tag{5}$$

De este modo, el mejor camino encontrado se premia con una cantidad extra de feromona y las hormigas seguirán dicho camino con mayor probabilidad en el próximo paso.

Finalmente, con una frecuencia de σ_s pasos, una nueva etapa comienza. El conjunto *init* se reemplaza con *next_init* y todos los rastros de feromona son eliminados de memoria (líneas 21-26). Los nodos asociados a dichos rastros también son eliminados de memoria (a no ser que formen parte de un camino de *next_init*). Este paso de eliminación permite al algoritmo reducir la cantidad de memoria requerida para la búsqueda.

V. Experimentos

En esta sección realizamos un estudio experimental en el que usamos una implementación de ACOhg integrada en JPF para buscar violaciones de propiedades de seguridad en un conjunto de programas concurrentes implementados en Java. Además aplicamos algunos de los algoritmos tradicionalmente usados para este problema, implementados ya en JPF y los comparamos con ACOhg. En la siguiente sección describimos los programas usados en los experimentos. Tras esto, presentamos la configuración utilizada para los algoritmos en la Sección V-B. Posteriormente, se muestran y discuten los resultados obtenidos en dichos experimentos.

A. Conjunto de programas concurrentes

Hemos seleccionado tres programas escalables escritos en Java que implementan algoritmos de especial importancia en la programación concurrente. Los dos primeros programas son dos versiones del problema de los filósofos de Edsger Dijkstra mientras que el tercero es una solución concurrente al problema de las parejas estables.

Una breve descripción del problema de los filósofos es la siguiente. Un cierto número n de filósofos se sientan en una mesa circular para comer arroz, tras lo cual se retiran a pensar. En la mesa hay n palillos chinos distribuidos de forma circular. Cada filósofo necesita dos palillos para comer, el de su izquierda y el de su derecha, los cuales toma en ese orden. Este

escenario se puede modelar en un computador usando n procesos que toman los "palillos" para "comer" y los liberan tras saciarse, permitiendo comer a otros "filósofos". Puede producirse un interbloqueo cuando todos los filósofos cogen el palillo de su izquierda y esperan a que su compañero suelte el palillo de su derecha. En la primera versión que hemos empleado, a la que llamaremos `din`, los filósofos sólo intentan comer una vez. En la segunda versión, denominada `phi`, los filósofos ejecutan indefinidamente un bucle en el que comen y piensan alternativamente, aumentando así las posibilidades de crear un interbloqueo.

En el problema de las parejas estables existen dos conjuntos de n elementos: los hombres y las mujeres. Cada elemento de cada conjunto ordena a los del otro según su preferencia. El problema consiste en emparejar cada hombre con una mujer de forma que no exista un hombre y una mujer que no se han asociado como pareja pero que se prefieren respecto a sus actuales parejas. Llamaremos `mar` al programa concurrente que resuelve este problema.

Los tres problemas poseen un parámetro n que permite escalarlos, es decir, podemos obtener instancias del tamaño deseado aumentando el valor de n. El tamaño del espacio de estados de los programas aumenta de forma exponencial con respecto a este parámetro. Para la realización de los experimentos hemos usado valores de n entre 2 y 16 en el caso de los programas `din` y `phi` y entre 2 y 6 para el programa `mar`. En todos los casos la propiedad que violan los programas es la ausencia de interbloqueo.

B. Parámetros de ACOhg

En los experimentos usamos el algoritmo ACOhg con la configuración mostrada en la Tabla I. Estos parámetros se han elegido teniendo en cuenta estudios previos de parámetros existentes para ACOhg.

TABLA I
PARÁMETROS DE ACOHG.

Parámetro	Valor	Parámetro	Valor
m	100	τ_0^{min}	0.0
$csize$	10	τ_0^{max}	1.0
λ_{ant}	40	α	1.0
σ_s	40	β	2.0
ι	10	p	100
ρ	0.2		

Con respecto a la información heurística, h, usamos la heurística `MostBlocked` implementada en JPF y cuya expresión es $h(j) = 10000-(activas(j)-ejecutables(j))$, donde $activas(j)$ y $ejecutables(j)$ es el número de hebras activas y ejecutables, respectivamente, en el estado j. El criterio de parada de nuestros algoritmos consiste en encontrar una traza de error o alcanzar el número máximo de iteraciones permitidas m. Este no es el único criterio de parada interesante; el algoritmo podría seguir la búsqueda después de encontrar una traza de error para optimizar su longitud. No obstante, estamos interesados aquí en observar el esfuerzo requerido por el algoritmo para obtener una traza de error, que es un objetivo prioritario en las primeras fases de diseño de software crítico.

Ya que ACOhg es un algoritmo estocástico, necesitamos realizar varias ejecuciones independientes para tener una idea de su comportamiento. Por esto, realizamos 100 ejecuciones independientes para conseguir muy alta confianza estadística. En las tablas de resultados mostramos la media y la desviación estándar de esas 100 ejecuciones. La máquina usada en los experimentos es un Pentium 4 a 2.8 GHz con 1 GB de RAM y sistema operativo Linux. La cantidad máxima de memoria asignada a los algoritmos es 1 GB (como la cantidad de memoria física de la máquina): cuando un proceso excede esta memoria se detiene automáticamente. Hacemos esto para evitar un gran trasiego de datos desde/hasta la memoria secundaria, que podría afectar significativamente al tiempo requerido en la búsqueda.

C. Resultados Experimentales

En esta sección mostramos los resultados obtenidos al aplicar ACOhg, BFS y DFS a los programas Java descritos anteriormente. En las Tablas II, III y IV presentamos los resultados para los programas `din`, `phi` y `mar`, respectivamente. Mostramos la tasa de éxito (número de ejecuciones independientes en que se encuentra una traza de error), la longitud de la traza de error (número de estados) y el tiempo requerido por el algoritmo para encontrar dicha traza.

No mostramos aquí la memoria requerida en cada caso por el algoritmo porque no se puede medir con precisión. La razón de esto es que cuando una máquina virtual Java dispone de una cantidad de memoria M para ejecutar un programa, ésta sólo lanza el recolector de basura cuando la memoria ocupada por los objetos creados se aproxima a la cantidad máxima permitida M. De esta forma evita reiteradas llamadas innecesarias al recolector y ahorra tiempo de cómputo. Sin embargo, por otro lado, esto implica que durante la ejecución de un programa en el que se crean muchos objetos (como los algoritmos que nos ocupan) la cantidad de memoria máxima que la máquina virtual utilizará estará próxima al valor M, sea cual sea éste. Así pues al consultar la máxima cantidad de memoria utilizada por la JVM no obtendremos con precisión la cantidad máxima de memoria *requerida* por el algoritmo, que es lo que nos interesaría conocer aquí. A continuación comentaremos dichos resultados programa por programa.

En el programa `din` podemos comprobar cómo la tasa de éxito de ACOhg decrece conforme el tamaño de la instancia aumenta, tal y como podíamos esperar. La razón de esto es que el tamaño del espacio de búsqueda es mayor (recordemos que este tamaño aumenta exponencialmente con el parámetro n del

TABLA II
RESULTADOS DE ACOHG, BFS Y DFS PARA `din`

Programa	Medida	ACOhg	BFS	DFS
din2	Tasa éxito	**100/100**	**1/1**	**1/1**
	Longitud	**6.00** $_{0,00}$	**6**	**6**
	Tiempo (s)	0.41 $_{0,49}$	**0**	**0**
din4	Tasa éxito	**100/100**	**1/1**	**1/1**
	Longitud	**13.00** $_{0,00}$	**13**	**13**
	Tiempo (s)	1.43 $_{0,53}$	3	**1**
din6	Tasa éxito	97/100	**1/1**	**1/1**
	Longitud	**19.00** $_{0,00}$	**19**	**19**
	Tiempo (s)	6.21 $_{4,19}$	39	**3**
din8	Tasa éxito	37/100	0/1	**1/1**
	Longitud	**25.00** $_{0,00}$	-	**25**
	Tiempo (s)	**12.78** $_{6,70}$	-	58
din10	Tasa éxito	4/100	0/1	**1/1**
	Longitud	**31.00** $_{0,00}$	-	**31**
	Tiempo (s)	**25.00** $_{8,49}$	-	1536
din12	Tasa éxito	0/100	0/1	**1/1**
	Longitud	-	-	**37**
	Tiempo (s)	-	-	**37573**
din14	Tasa éxito	0/100	0/1	0/1
	Longitud	-	-	-
	Tiempo (s)	-	-	-
din16	Tasa éxito	0/100	0/1	0/1
	Longitud	-	-	-
	Tiempo (s)	-	-	-

TABLA III
RESULTADOS DE ACOHG, BFS Y DFS PARA `phi`

Programa	Medida	ACOhg	BFS	DFS
phi2	Tasa éxito	**100/100**	**1/1**	**1/1**
	Longitud	14.76 $_{6,59}$	**8**	17
	Tiempo (s)	**1.00** $_{0,00}$	**1**	**1**
phi4	Tasa éxito	**100/100**	**1/1**	**1/1**
	Longitud	28.51 $_{6,48}$	**17**	113
	Tiempo (s)	1.04 $_{0,20}$	59	**1**
phi6	Tasa éxito	**100/100**	0/1	**1/1**
	Longitud	**35.04** $_{4,03}$	-	2469
	Tiempo (s)	2.04 $_{1,10}$	-	**2**
phi8	Tasa éxito	**100/100**	0/1	**1/1**
	Longitud	**41.75** $_{10,44}$	-	23771
	Tiempo (s)	21.75 $_{16,55}$	-	**14**
phi10	Tasa éxito	**100/100**	0/1	**1/1**
	Longitud	**69.06** $_{7,29}$	-	87834
	Tiempo (s)	54.76 $_{1,64}$	-	**43**
phi12	Tasa éxito	**100/100**	0/1	0/1
	Longitud	**75.09** $_{7,66}$	-	-
	Tiempo (s)	**69.10** $_{16,25}$	-	-
phi14	Tasa éxito	**100/100**	0/1	0/1
	Longitud	**98.57** $_{15,83}$	-	-
	Tiempo (s)	**121.41** $_{26,91}$	-	-
phi16	Tasa éxito	**65/100**	0/1	0/1
	Longitud	**110.89** $_{5,65}$	-	-
	Tiempo (s)	**148.78** $_{11,54}$	-	-

programa escalable) y el problema se vuelve más duro para ACOhg, reduciéndose así la probabilidad de encontrar una solución. Este es el efecto que produce la explosión de estados y que afecta a todos los algoritmos. Véase que mientras que ACOhg aún encuentra trazas de error en `din10`, BFS tan sólo puede encontrarlas hasta `din8` y DFS es capaz de llegar hasta `din12`. Lo que impide a DFS y BFS seguir la búsqueda es la memoria: han llegado a consumir toda la memoria disponible. Por el contrario, ACOhg no es capaz de completar la búsqueda en los programas más grandes porque no ha tenido tiempo suficiente de hacerlo. De hecho, si aumentamos el número de hormigas o el número de pasos del algoritmo las tasas de éxito aumentarán y ACOhg será capaz de encontrar trazas de error en los programas más grandes.

En cuanto a la longitud de la traza de error, en este caso sólo existe una traza de error: aquélla en la que todos los filósofos toman un palillo antes de que cualquiera de ellos intente tomar el segundo. Por este motivo la longitud de las trazas de error es siempre la misma para cada instancia concreta. Podemos observar cómo crece la longitud del camino conforme aumenta el número de filósofos.

En último lugar observamos que el tiempo requerido para encontrar una traza en los algoritmos exhaustivos es pequeño en las instancias de menor tamaño pero crece muy rápidamente a medida que lo hace la instancia. ACOhg no se caracteriza por ser un algoritmo especialmente rápido, como otros estudios muestran [7], pero aquí observamos que es capaz de encontrar trazas de error más rápidamente que DFS en las instancias más grandes que ambos resuelven. Es posible que este mayor tiempo requerido por DFS sea debido a la gran cantidad de memoria necesaria para la búsqueda, que obliga al recolector de basura a ejecutarse con mayor frecuencia.

Analizamos ahora los resultados obtenidos en el programa `phi`. En esta ocasión la tasa de éxito de ACOhg es mayor que en el caso anterior. La razón de esto es que, al repetir todos los procesos el mismo bucle indefinidamente, existen múltiples trazas de error, lo cual aumenta las probabilidades de encontrar una. Mientras que los algoritmos DFS y BFS siguen teniendo problemas para encontrar soluciones en los programas de mayor tamaño, vemos que ACOhg es capaz de encontrar soluciones incluso en el mayor de ellos: `phi16`.

No todas las trazas de error tienen la misma longitud en este programa y no siempre ACOhg encontrará la más corta, por lo que el promedio estará por encima de la longitud óptima. Esto puede observarse comparando los resultados de ACOhg con los de BFS, que obtiene siempre una traza óptima. Podemos destacar aquí que las trazas de error obtenidas por ACOhg son, con diferencia, mucho más cortas que las obtenidas por DFS (véase la Figura 2). Esto es debido a la forma en que DFS realiza la búsqueda (profundidad). En resumen, podemos decir que ACOhg es capaz de encontrar trazas de error cortas en programas en los que DFS y BFS no pueden.

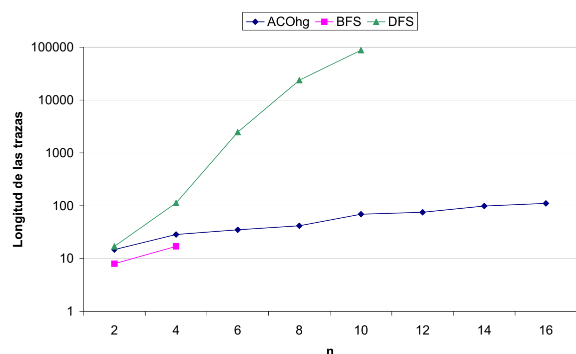

Fig. 2. Longitud de las trazas en `phi`

Vemos que el tiempo de ejecución de ACOhg se mantiene en esta ocasión por encima del de DFS incluso en las instancias más grande que DFS puede resolver. Sin embargo, este tiempo es bajo para una aplicación práctica de la técnica. No existen muchas oportunidades para analizar BFS, pero en las pocas ocasiones que nos brinda puede observarse que su tiempo de ejecución tiende a ser mucho mayor que el de los otros dos algoritmos.

TABLA IV
RESULTADOS DE ACOHG, BFS Y DFS PARA `mar`

Programa	Medida	ACOhg	BFS	DFS
mar2	Tasa éxito	**100/100**	**1/1**	**1/1**
	Longitud	12.72 $_{0,81}$	**12**	14
	Tiempo (s)	1.21 $_{0,55}$	**1**	**1**
mar3	Tasa éxito	**100/100**	**1/1**	**1/1**
	Longitud	25.08 $_{3,78}$	**19**	35
	Tiempo (s)	1.10 $_{0,30}$	3	**1**
mar4	Tasa éxito	**100/100**	**1/1**	**1/1**
	Longitud	38.24 $_{2,36}$	**27**	64
	Tiempo (s)	1.68 $_{0,82}$	510	**1**
mar5	Tasa éxito	**100/100**	0/1	**1/1**
	Longitud	**62.24** $_{10,04}$	-	99
	Tiempo (s)	31.24 $_{7,12}$	-	**3**
mar6	Tasa éxito	**100/100**	0/1	**1/1**
	Longitud	**74.56** $_{4,48}$	-	141
	Tiempo (s)	40.43 $_{11,66}$	-	**5**

Analizamos por último los resultados obtenidos en el programa `mar`. En este programa ACOhg consigue una tasa de éxito del 100 % en todos los casos al igual que DFS. Por el contrario, BFS es capaz sólo de encontrar trazas de error hasta `mar4`. En cuanto a la longitud de las trazas, podemos observar que en las instancias pequeñas ACOhg consigue trazas muy cercanas a las óptimas. A medida que el tamaño del espacio de búsqueda crece estas longitudes se van alejando de la óptima. De la misma forma, la longitud de las trazas que obtiene DFS se alejan cada vez más de las obtenidas por ACOhg conforme aumenta el tamaño del programa. El tiempo de ejecución sigue la misma tendencia que en los programas anteriores: ACOhg es más rápido que BFS pero más lento que DFS.

Como resumen final de este estudio podemos decir que ACOhg es un algoritmo que combina las buenas características de BFS y DFS. Por un lado, obtiene trazas de error cortas, como BFS, y por otro se ejecuta en un corto intervalo de tiempo, como DFS. Todo esto acompañado además de que es capaz de encontrar errores usando una cantidad reducida de memoria, a diferencia de BFS y, en menor medida, DFS.

VI. CONCLUSIONES Y TRABAJO FUTURO

En este artículo abordamos el problema de la búsqueda de violaciones de propiedades de seguridad en programas concurrentes usando un enfoque basado en *model checking*. Para ello hacemos uso de ACOhg, un algoritmo basado en colonias de hormigas, integrado en el *model checker* Java PathFinder. Presentamos un estudio experimental en el que se ha usado ACOhg para descubrir interbloqueos en tres programas escalables. Además, comparamos los resultados con los obtenidos por dos algoritmos tradicionales para el mismo problema: BFS y DFS.

Los resultados muestran que ACOhg es capaz de encontrar errores en ocasiones en las que DFS y BFS fallan debido a la falta de memoria. Asimismo, hemos comprobado que la longitud de las trazas de error obtenidas son cortas, lo que facilita la labor de comprensión del error por parte del programador. Por último, hemos podido comprobar que, aunque no es el algoritmo más rápido, los tiempos de ejecución de ACOhg son suficientemente reducidos como para usar la propuesta en la práctica.

Como trabajo futuro inmediato se planea la aplicación de ACOhg para buscar propiedades expresadas mediante fórmulas LTL y propiedades de viveza dentro de JPF. Esto requiere estudiar con detalle JPF para crear extensiones que permitan verificar tales propiedades, ya que actualmente sólo es posible comprobar ciertas propiedades de seguridad. Además de esto, pretendemos estudiar la compatibilidad y la influencia en los resultados de ACOhg de técnicas para reducir el espacio de búsqueda a explorar como la ejecución delta, la compresión de estados o la reducción por simetría. Otra línea posible de investigación es la ejecución en paralelo de ACOhg para que pueda disponer de este modo de más memoria para la búsqueda.

AGRADECIMIENTOS

Este trabajo está parcialmente financiado por la Consejería de Innovación, Ciencia y Empresa de la Junta de Andalucía con número de proyecto P07-TIC-03044 (proyecto DIRICOM).

REFERENCIAS

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, The MIT Press, January 2000.
[2] A. Groce and W. Visser, "Heuristics for model checking Java programs," *Intl. Jnl. on Software Tools for Technology Transfer*, vol. 6, no. 4, pp. 260–276, 2004.
[3] G. J. Holzmann, *The SPIN Model Checker*, Addison-Wesley, 2004.
[4] E. Alba and J. M. Troya, "Genetic Algorithms for Protocol Validation," in *Proceedings of the PPSN IV International Conference*, Berlin, 1996, pp. 870–879, Springer.
[5] P. Godefroid and S. Khurshid, "Exploring Very Large State Spaces Using Genetic Algorithms," in *LNCS, 2280*. 2002, pp. 266–280, Springer.
[6] P. Godefroid, "VeriSoft: A tool for the automatic analysis of concurrent reactive software," in *Proc. of the 9th Conference on Computer Aided Verification, LNCS 1254*, 1997, pp. 476–479.
[7] E. Alba and F. Chicano, "Finding safety errors with ACO," in *Proc. of GECCO*, 2007, pp. 1066–1073.
[8] F. Chicano and E. Alba, "Searching for liveness property violations in concurrent systems with ACO," in *Proceedings of Genetic and Evolutionary Computation Conference*, 2008.
[9] E. Alba, F. Chicano, M. Ferreira, and J. Gomez-Pulido, "Finding deadlocks in large concurrent Java programs using genetic algorithms," in *Proceedings of Genetic and Evolutionary Computation Conference*. July 2008, pp. 1735–1742, ACM.
[10] B. Alpern and F. B. Schneider, "Defining liveness," *Inform. Proc. Letters*, vol. 21, pp. 181–185, 1985.
[11] M. Dorigo and T. Stützle, *Ant Colony Optimization*, The MIT Press, 2004.