Coevolución para el Diseño Automático de Máquinas de Estados Finitos*

J. Manuel Colmenar[1], Alfredo Cuesta-Infante[1], José L. Risco-Martín[2], J. Ignacio Hidalgo[2]

[1] C.E.S. Felipe II, U. Complutense de Madrid, 28300 Aranjuez, España
{jmcolmenar, alfredo.cuesta}@ajz.ucm.es
[2] Dept. de Arquitectura de Computadores y Automática
U. Complutense de Madrid, 28040 Madrid, España
{jlrisco, hidalgo}@dacya.ucm.es

Resumen En este artículo presentamos el sistema GOURMET (*Grammatical based Optimization Under fitness Refinement by Means of Evolutionary Techniques*). Se trata de una metodología evolutiva para la inferencia de máquinas de estado finitas basada en Gramáticas Evolutivas (Grammatical Evolution). Nuestra aproximación presenta dos claras ventajas. En primer lugar, el flujo de diseño acepta como entrada descripciones de alto nivel, haciendo más fácil y asequible el proceso a los diseñadores de sistemas. En segundo lugar, al contrario de lo que sucede con otras aproximaciones, el diseñador no necesita definir un conjunto de valores de entrada para entrenar al sistema, ya que estos valores son generados de manera automática mediante un algoritmo genético en tiempo de ejecución. Nuestros experimentos obtuvieron la máquina de estados correcta para el problema del reconocedor de secuencias en un 99,5 % de las ocasiones y en un 100 % para el problema de diseño de una máquina de vending.

Keywords: Grammatical evolution, Genetic programming, Genetic algorithms, Design/synthesis, Mathematical modeling/curve fitting

1. Introducción

Las máquinas de estados finitos (*finite state machines*, FSMs) proporcionan un mecanismo de alto nivel para la descripción de sistemas complejos, de manera que, a partir de una FSM, muchas herramientas de diseño son capaces de generar el sistema hardware subyacente. Sin embargo, acometer el diseño de una FSM desde cero requiere experiencia y conocimientos previos. En consecuencia, la creación de herramientas que ayuden en la creación automática de FSMs puede resultar muy útil, por ejemplo, para el prototipado rápido en diseño de hardware. Bajo este objetivo, y dado que la obtención de la FSM irreducible a

* Trabajo financiado por los proyectos TIN2008-00508 y MEC Consolider Ingenio CSD00C-07-20811 del Ministerio de Ciencia y Tecnología.

partir de un conjunto de entradas es un problema NP-completo [2], las técnicas metaheurísticas encajan perfectamente en esta clase de problemas de optimización. Varios trabajos de la literatura abordaron la inferencia de FSMs. En [5] se propuso un algoritmo eficiente consistente en la inferencia de un autómata finito determinista a partir de un conjunto de entradas, aplicando posteriormente un paso de minimización del número de estados. En este caso, el conjunto de entradas (conjunto de entrenamiento) provenía de la observación de sistemas existentes, y se porporcionaba como dato de entrada al inicio del algoritmo. En [4] se propone la generación de FSMs utilizando programación genética y estrategias evolutivas. Los resultados obtenidos no son demasiado buenos según los propios autores. Más recientemente, en [8] se presenta un algoritmo genético para la inferencia de FSMs, junto con un detallado estado del arte en el área bajo estudio. Sin embargo, el algoritmo requiere un conjunto de entrenamiento más una serie de reglas y fórmulas para describir las restricciones del sistema.

En consecuencia, hay dos líneas principales en la literatura. Por un lado, están aquéllas en las que se requiere un conjunto de entrenamiento de valores de entrada y sus salidas correspondientes. Sin embargo el problema de cómo seleccionar las entradas y la forma de determinar el tamaño del conjunto de entrenamiento por lo general no están cubiertos. Además, en la mayoría de las definiciones de los problemas existen algunas entradas que son más importantes que otras como, por ejemplo, los reconocedores de secuencias [3]. Por otro lado, es muy típico requerir fórmulas complejas, reglas o restricciones para describir el problema en cuestión. Creemos que una descripción de alto nivel del problema podría ser más cómodo y útil.

En este trabajo se propone un flujo de optimización que, por medio de la evolución gramatical (GE), en colaboración con un algoritmo genético (AG), se obtiene la descripción FSM que resuelve un problema de descrito a alto nivel verificando además un conjunto de valores de entrada generado automáticamente. Hemos llamado a este esquema de colaboración GOURMET (*Grammatical based Optimization Under fitness Refinement by Means of Evolutionary Techniques*). GOURMET genera automáticamente el conjunto de entrenamiento de los entradas que intervienen en el cálculo de la función de aptitud o fitness a través de un AG que se encarga de obtener el conjunto de entradas representativas. Este AG trabaja en colaboración con un bucle GE que busca la FSM concreta que resuelve un problema que se describe con un lenguaje de alto nivel. Hemos aplicado GOURMET a dos tipos diferentes de problemas: el detector de secuencia de 3 bits y el problema de la máquina expendedora. Los resultados mostraron que con el uso de una gramática sencilla y una breve descripción de alto nivel del problema, se encuentra con éxito y en un tiempo razonable, una FSM solución para casi el 100 % de las ejecuciones.

El resto del artículo está organizado como sigue. En la sección 2 se da una visión general del sistema. En la sección 3 se describe el espacio de búsqueda y la codificación. La sección 4 explica el proceso evolutivo con la colaboración del AG y la GE. Los resultados experimentales se recogen en la sección 5 y las conclusiones en la 6.

2. GOURMET, visión general

Cualquiera de nosotros es capaz de calificar o medir algún producto o resultado, pero no es capaz de producirlos. Incluso los expertos a veces carecen de la habilidad pura y, a veces la falta de conocimiento. Muchos ejemplos ilustran esta situación. Por ejemplo, un piloto de motocicletas puede ser capaz de determinar cuán bueno es el rendimiento de una moto sin necesidad de tener ningún conocimiento de mecánica o ingeniería. Por supuesto que esta información le ayudaría a ser más preciso, pero no es una característica obligatoria, como lo demuestran algunos campeones del mundo. Del mismo modo, un gourmet podría no ser capaz de cocinar, ni siquiera una tortilla, pero sus desarrollados sentidos le permiten apreciar todos los sabores en un plato.

De esta manera, cuando a un cocinero se le solicita cocinar un plato nuevo por primera vez, podría utilizar las habilidades del gourmet para mejorar su resultado hasta adivinar la receta. El gourmet podrá degustar el plato y asignarle una calificación. La evaluación propuesta por el gourmet será una realimentación que permite al chef mejorar el plato. Además, cuanto mayor sea el número de platos probados, mayor será la experiencia adquirida por el gourmet. El gourmet puede incluso recordar los platos del pasado y cambiar su calificación para ser más preciso y justo con el chef.

Desde el punto de vista algorítmico, el chef representa un algoritmo de búsqueda sobre la mejor solución, mientras que el gourmet representa el objetivo o función de aptitud para ser optimizado. De esta manera, la función de aptitud también mejorará a medida que el número de iteraciones del algoritmo de búsqueda crece.

Hemos transformado la analogía en un esquema de colaboración, como se muestra en la Figura 1, donde se definen tres módulos:

- GE: Evolución gramatical. Se corresponde con el chef, y genera las FSMs candidatas a ser evaluadas por el gourmet.
- FIT: Fitness. Pertenece a la alta cocina, y produce el resultado esperado para una entrada dada de acuerdo a una especificación de alto nivel del problema. Recibe como parámetros de entrada la FSM para su evaluación y un conjunto de entradas a procesar. Devuelve el fitness, calculado como el número de diferencias entre los resultados esperados y los dados por la FSM.
- AG: Algoritmo genético. También pertenece al gourmet y genera el conjunto de entradas que se evalúa para cada FSM candidata.

El módulo GE realiza la búsqueda de una solución FSM y, una vez encontrada, el módulo AG se utiliza para encontrar la secuencia de entrada para la que el FSM genera la peor salida. Este valor de entrada se incorporará al módulo FIT, y tomará parte en la evaluación de la aptitud de las FSM posteriores. A continuación se describe un ejemplo que corresponde a la búsqueda de una FSM que permite el reconocimiento de la secuencia 01 a partir de una entrada en serie de diez bits (con solapamiento). Inicialmente, el módulo FIT no tiene ninguna información previa, por lo que comienza con una entrada generada al

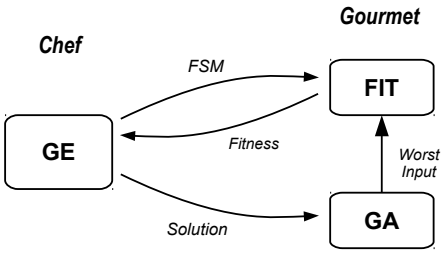


Figura 1. Principales módulos de la optimización con la analogía de chef y gourmet.

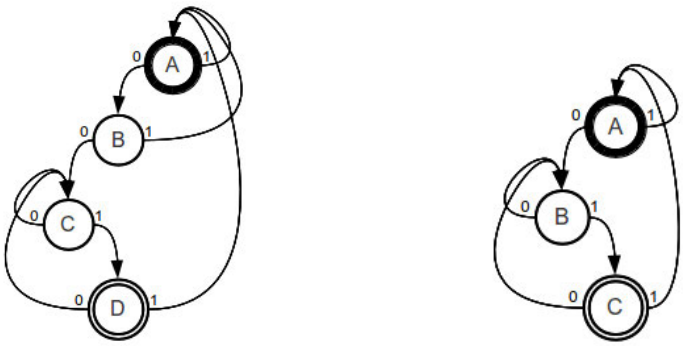
azar. Para este simple reconocedor, supongamos que genera 1001001010 como cadena de entrada, cuya cadena de salida correcta es 0001001010. Con el fin de encontrar la mejor solución, el módulo GE explora el espacio de búsqueda FSM mediante la evaluación de cada solución candidata con el módulo de FIT, que en este punto sólo tiene una entrada para la prueba. Después de cierta exploración, GE genera una solución que produce la salida correcta dada la entrada inicial (0 diferencias). Vamos a llamar a esta solución sol_1 . La Figura 2(a) muestra la primera FSM obtenida para esta muestra. Cada nodo del grafo tiene dos entradas posibles: 0, representado en la rama izquierda, y 1, representado en la rama derecha. Probablemente, esta primera solución no será la FSM *golden*, es decir, la FSM que produce la salida correcta para cualquier entrada dada. A continuación, el módulo de AG se ejecuta tomando sol_1 y trata de encontrar la peor entrada posible. Este valor será la secuencia de entrada para la que el número de bits de salida incorrectos es máxima. En el caso de que estemos frente a una solución final, el AG no encontrará ninguna cadena de entrada peor, pues ya se habrá conseguido una correcta. En el segundo paso, para nuestra sol_1 , el AG devuelve como cadena de entrada peor 0101010101. Su cadena de salida correcta es 0101010101, mientras que la cadena de salida generada por sol_1 es 0000000000. A continuación, se añade la nueva entrada al módulo FIT, que ahora cuenta con dos valores de entrada.

La utilización del módulo AG tiene dos beneficios diferentes. Por un lado, se comprueba si las soluciones candidatas propuestas por GE son *golden*, en cuyo caso, el AG no encontraría entrada alguna que produjese fallos. Por otro lado, el AG contribuye a la mejora de la función de aptitud añadiendo entradas que serán comprobadas en las siguientes iteraciones de GE.

Para el ejemplo comentado, al terminar las iteraciones definidas para el módulo GE, se obtuvo la FSM que se muestra en la Figura 2(b).

3. Espacio de Búsqueda y Codificación

Desde un punto de vista general, el trabajo presentado define un marco que es independiente del problema concreto que se quiere abordar. De esta manera, se definirá el espacio de búsqueda en términos de FSM y se explicará la codificación.



(a) Primera FSM. (b) FSM final.

Figura 2. FSMs para el detector de secuencia 01.

Siguiendo un esquema típico de Máquina de Moore, hemos definido una FSM como un grafo dirigido con las siguientes propiedades:

- Sólo hay un estado inicial, representada en las figuras por bordes en negrita.
- Cada estado produce una sola salida.
- A partir de un determinado estado existen tantas transiciones diferentes como valores de entrada podría recibir.
- Aquellas transiciones que conducen al estado inicial se denominan "Reset".

En un primer enfoque, teniendo en cuenta las salidas binarias, si un estado produce 1 como valor de salida, se denota el estado con un borde de doble línea. Si se produce un estado 0 como valor de la salida, se utiliza una línea simple. Como se ve en la Figura 2(b), el estado inicial de esa FSM es A, mientras que el único estado que genera 1 como salida es de C. Dado que el conjunto de posibles FSM para un problema dado tiene un tamaño infinito, es necesario limitar el espacio de búsqueda. En nuestra propuesta se aplican Gramáticas Evolutivas (GE) para explorar el espacio de búsqueda, por lo tanto, se utiliza una gramática para limitar el espacio de búsqueda.

Las Gramáticas Evolutivas (GE) [7] son una forma de programación genética basada en gramáticas. La Figura 3 muestra la gramática que hemos utilizado para en el espacio de búsqueda de FSMs. Como se puede ver, el símbolo no terminal `<FSM>` representa el estado inicial, que es siempre un estado, `<State>`. Cada `<State>` corresponde a un identificador, `<StateId>` asociado a un valor de salida (0 o 1, ver regla **IV**). Desde cualquier `<State>`, salen dos transiciones que se corresponden con `<Trans>`. La primera corresponde con el valor de entrada 0, mientras que la segunda corresponde con el valor de entrada 1.

Por lo tanto la gramática de la Figura 3 permite una FSM con un máximo de 8 estados que se corresponden con las etiquetas (`<StateId>`) A a D combinados con ambos valores de salida. El espacio de búsqueda queda limitado de esta forma a todas las FSMs que se pueden obtener con todas las posibles combinaciones de transiciones entre, como másimo, estos 8 estados.

```
N = {<FSM>,<State>,<StateId>,<Output>,<Trans>}
T = {Reset,0,1,A,B,C,D}
S = <FSM>

I    <FSM>      ::= <State>
II   <State>    ::= <StateId><Output><Trans><Trans>
III  <Trans>    ::= <State> | Reset
IV   <Output>   ::= 0 | 1
V    <StateId>  ::= A | B | C | D
```

Figura 3. Gramática que describe una FSM de hasta 8 estados: de A a D con salida 0 más A a D con salida 1.

4. Proceso Evolutivo

Como se ha explicado en la sección 1, proponemos un esquema evolutivo llamado GOURMET donde dos elementos principales colaboran: una gramática evolutiva (GE) y un algoritmo genético (AG). Por un lado, la gramática evolutiva intenta encontrar la máquina de estados finitos con el mejor fitness y, por otro lado, evolucionamos la función de fitness mediante un algoritmo genético. El módulo de GE recibe una población inicial de máquinas generadas de forma totalmente aleatoria. Este módulo detiene su ejecución cuando se encuentra una solución, esto es, cuando se encuentra una máquina de estados que no contiene errores, o cuando se llega al máximo número de generaciones. Hay que tener en cuenta que la gramática evolutiva utiliza el módulo de evaluación de fitness (FIT) para probar cada una de las máquinas candidatas.

El módulo de GE termina su ejecución cuando se encuentra una solución, es decir, al encontrar una FSM sin errores o cuando GE alcanza su máximo número de generaciones. Una vez que termina la GE, se comprueba el número de iteraciones, que se corresponde con el número de veces que se ejecuta el módulo de GE. Si no se alcanza este límite, el módulo de algoritmo genético (AG) realiza una búsqueda sobre la mejor solución proveniente de GE. Además, si GE encuentra más de una solución, el AG se ejecuta una vez para cada una de ellas. Por el contrario, Si la GE anterior no ha encontrado una solución, el AG se ejecuta para los mejores individuos.

El módulo GE dentro de GOURMET se divide en dos etapas consecutivas. El primer paso es una GE clásica donde se realiza la exploración sobre el espacio de búsqueda determinado por la gramática. Los parámetros para esta primera evolución se muestran en la Tabla 1, columna GE-1. Los valores de estos parámetros se eligieron tras un estudio experimental. El segundo paso del módulo de GE consiste en una búsqueda adicional sobre la población que viene de la primera GE con el objetivo de aumentar el número de soluciones o, al menos, aumentar el número de individuos similares a la solución obtenida. Por lo tanto, una búsqueda corta (100 generaciones) se ejecuta con una probabilidad alta de mutación, mientras que la probabilidad de cruce es baja. La Tabla 1, columna GE-2, muestra los valores de los parámetros para la segunda fase de la GE. Como resultado,

se realiza una búsqueda local que permite la mejora de las soluciones obtenidas, que se utilizarán más adelante en el módulo del AG.

Cuadro 1. Resumen de los parámetros utilizados en las dos fases de GE y en el AG.

Parámetro	GE-1	GE-2	AG
Tamaño de la población	500	500	200
Generaciones	6000	100	1000
Probabilidad de cruce	0.85	0.2	0.8
Probabilidad de mutación	0.2	0.95	0.02
Tamaño de cromosoma	100	100	100
Max. número wraps	0	0	-
Fin si encuentra solución	true	false	false

Tanto las fases de GE como el AG utilizan cruce basado en un solo punto, mutación entera uniforme y selección por torneo. También se implementa elitismo manteniendo los 10 mejores individuos.

La función de aptitud en GOURMET es una medida de la distancia, en número de valores diferentes, entre la salida generada por la FSM en evaluación y la salida esperada (correcta). Por lo tanto, se evalúa cada individuo teniendo en cuenta el conjunto actual de valores de entrada. Entonces, para cada entrada, la salida propuesta por el individuo se compara con la salida correcta esperada. El número de diferencias entre las dos salidas se acumula, obteniendo así un valor de fitness. Los individuos que no tienen errores recibirán el mejor valor de aptitud (el más bajo), que es de 0. La aptitud se obtiene así:

$$Fitness = \sum_{i=1}^{N} \sum_{j=1}^{L} \mathit{diff}\,(i,j)$$

donde N es el número de cadenas de entrada actual, L es la longitud de la secuencia de entrada, y *diff* (i,j) devuelve 1 si el bit de salida j es incorrecto para la entrada i, y 0 en caso contrario.

La función de aptitud depende, por tanto, de las cadenas de entrada de las que se disponga en cada momento. Estas cadenas son seleccionadas por el AG tras la búsqueda de la peor entrada para cada una de las soluciones candidatas en cada iteración de GE. En consecuencia, se produce una coevolución entre GE y AG. La Tabla 1 muestra también los parámetros seleccionados para el AG.

Tras cada iteración, las mejores soluciones (élites) obtenidas en la ejecución de ambas fases GE se reinsertan en la población aleatoria que se crea para la siguiente iteración sobre GE. Esto permite conservar la información genética de los mejores individuos a la vez que permite a las soluciones correctas pervivir a lo largo de las iteraciones previstas para GE.

Hemos implementado GOURMET en Java. El módulo de GE fue codificado utilizando GEVA [6], una conocida herramienta de GE basada en Java, mientras que el resto del código fue desarrollado por nuestro grupo de investigación. Los algoritmos, programas y pruebas están a disposición del público en [1].

```
N = {<FSM>,<InitState>,<State>,<StateId>,<Output>,<Trans>}
T = {Reset,0,1,A,B,C,D,E}
S = <FSM>

I    <FSM>       ::= <InitState>
II   <InitState> ::= <StateId>0<Trans><Trans>
III  <State>     ::= <StateId><Output><Trans><Trans>
IV   <Trans>     ::= <State> | Reset
V   <Output>    ::= 0 | 0 | 0 | 1
VI  <StateId>   ::= A | B | C | D | E
```

Figura 4. Gramática para el reconocedor de secuencia de 3 bits.

5. Resultados experimentales

GOURMET se ha probado en dos problemas diferentes: los detectores de secuencia y el problema de la máquina expendedora. En ambos usamos cadenas de entrada de 100 bits de longitud, que corresponde a una enorme espacio de búsqueda para el conjunto de entrada (2^{100} combinaciones).

Aplicamos GOURMET con el objetivo de encontrar las ocho FSM que implementan todos posibles reconocedores de secuencia de 3 bits. Para ello utilizamos la gramática de la Figura 4, presentando un estado inicial con un valor de salida fijo de 0. Además hemos reducido la probabilidad de tener 1 como salida modificando la regla **V** para `<output>`.

Para cada uno de la ocho patrones diferentes del reconocedor de 3 bits, lanzamos GOURMET 30 veces, iterando 50 veces el bucle principal de la GE. Después de cada ejecución, se obtuvo una FSM como resultado. Se encontró que las FSMs resultantes podrían ser de tres tipos diferentes: soluciones *golden*, es decir, FSM no reducible que resuelve correctamente el problema; soluciones reducibles, es decir, soluciones equivalentes a la correcta, pero con exceso de estados (reducible); y FSMs que no resultaban ser solución puesto que obtuvieron valores de fitness superiores a 0.

La Tabla 2 resume los resultados en seis columnas. En primer lugar, se indica el patrón de 3 bits, seguido, respectivamente, por el porcentaje de soluciones *golden* y reducibles encontradas. La cuarta columna muestra el número de ejecuciones que no encontraron ninguna solución. Las columnas quinta y sexta indican el tiempo medio de ejecución de un ciclo de optimización, y el número total de de entradas procesadas teniendo en cuenta todas las iteraciones para cada patrón.

Como se ve en los resultados, nuestro algoritmo obtiene soluciones correctas para todas las ejecuciones, excepto para una del patrón de 011 que no encontró ninguna solución. Tenga en cuenta que sólo estamos interesados en soluciones sin fallos. Por lo tanto, FSMs parcialmente correctas no son consideradas como soluciones. Por otra parte, todas los FSM obtenidas fueron validadas, y se ha verificado que eran exactamente la misma que la solución correcta minimizado FSM para cada patrón, o cualquier máquina reducible equivalente.

A partir de los resultados que se muestran en la Tabla 2 se puede observar que es más difícil de encontrar la FSM solución en algunos de los patrones.

Cuadro 2. Resultados para los reconocedores de secuencia de 3 bits.

Patrón	Golden	Reducible	NO sol.	Avg. T. (secs)	# Inputs
000	23.33 %	76.66 %	0	10735.78	3336
001	65.52 %	34.48 %	0	6465.56	1756
010	93.10 %	6.89 %	0	8860.21	1456
011	93.10 %	3.45 %	1	17995.69	1529
100	92.86 %	7.14 %	0	6685.07	1568
101	96.55 %	3.45 %	0	6012.92	1428
110	63.33 %	36.66 %	0	6298.69	1833
111	23.33 %	76.66 %	0	9862.46	3211

En concreto, en aquellos en los que todos los bits son iguales: 000 y 111. Para ambos, el porcentaje de soluciones golden es bajo y el tiempo de ejecución es alto. El tiempo de ejecución está directamente relacionado con el rendimiento de la búsqueda. Si una iteración sobre GE encuentra un FSM correcta, ya sea golden o reducible, el AG no obtendrá una cadena de entrada que provoque fallos. Por lo tanto, dado que las elites se reinsertan en la población de la próxima iteración de GE, la solución será un inmigrante. Entonces, GE se ejecutará sólo una generación, ya que está configurado para detenerse cuando se encuentra una solución, reduciendo el tiempo total de ejecución. Por lo tanto, cuanto más corto el tiempo de ejecución, más pronto se encontró la solución.

Además, definimos un valor umbral para las iteraciones de GE donde no se encontró una solución. En estos casos, el AG se ejecuta sobre estas FSMs con el fin de obtener sus peores entradas. De esta manera, se contribuye a la mejora del conjunto de entradas para FIT, de modo que no se desperdicia el trabajo realizado por la iteración sobre GE. Sin embargo, en esos casos, no hay inmigrantes enviados a la siguiente iteración de GE. Este hecho es el que hace que las ejecuciones para los patrones 000 y 111 sean más lentas puesto que cada GE ejecuta todas sus iteraciones.

El problema de la máquina expendedora representa un escenario diferente. Podemos describirlo con las siguientes reglas: (1) La máquina dispensa un producto después de haber recibido una cierta cantidad de dinero. (2) Después de la distribución del producto (salida 1), la máquina vuelve al estado inicial. (3) Se aceptan dos tipos de monedas: 5 ¢, y 10 ¢, y no devuelve cambio. A pesar de la simplicidad de la descripción de alto nivel de este problema, la búsqueda de una FSM reducida que resuelva el problema es más difícil que en el problema reconocedor de secuencia. Hemos abordado el problema de la máquina expendedora para un importe de 15 ¢. La configuración experimental y la gramática fueron los mismos que en el problema de reconocimiento de patrones. La optimización se ha realizado correctamente porque todas las 30 ejecuciones obtuvieron la FSM *golden*. El tiempo de ejecución promedio fue de 25.330,45 segundos, y 1.274 entradas fueron procesadas. En consecuencia, todos los problemas fueron resueltos con éxito por GOURMET obteniendo la FSM no reducible.

6. Conclusiones y trabajo futuro

Las FSM son descripciones útiles de bajo nivel que solucionan muchos tipos de problemas basados en estado. Encontrar una implementación correcta de FSM es un problema que se complica cuando el número de estados y transiciones crece. En este trabajo, proponemos un flujo evolutivo donde se obtienen, de manera automática, implementaciones de FSM a través de gramáticas evolutivas. Utilizando un algoritmo genético, el flujo genera automáticamente el conjunto de entrenamiento de las entradas que se probarán por la función de fitness, descubriendo las entradas con la mayor tasa de fracaso. Además, no se requieren reglas o fórmulas complejas, sino que sólo se requiere un código Java de alto nivel que resuelva el problema abordado. Hemos llamado a este flujo evolutivo GOURMET (*Grammatical based Optimization Under fitness Refinement by Means of Evolutionary Techniques*). Nuestros experimentos mostraron muy buenos resultados para el problema del reconocedor de secuencia de 3 bits, donde se obtuvo la FSM solución no reducible en el 68,38 % de las ejecuciones, una solución reducible equivalente en el 31,20 % de los casos, y simplemente no encontraron una solución en una sola ejecución para la secuencia de 011. Además, el problema de la máquina expendedora para 15 ¢ también fue resuelto con éxito, obteniendo la FSM reducida en el 100 % de las ejecuciones.

La fase de evaluación es el cuello de botella en GOURMET, por lo que el aumento de paralelismo reducirá el tiempo de ejecución, lo que permitirá intensificar la exploración para poder hacer frente a futuros problemas más complejos.

Referencias

1. JECO: Java Evolutionary Computation library, 2013. `http://code.google.com/p/paba/`.
2. E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302 – 320, 1978.
3. S. Goren and F. Ferguson. Checking sequence generation for asynchronous sequential elements. In *International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034)*, pages 406–413. Int. Test. Conference.
4. C. Johnson. Genetic programming with fitness based on model checking. In M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, and A. Esparcia-Alcázar, editors, *Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 114–124. Springer Berlin / Heidelberg, 2007.
5. A. L. Oliveira and J. P. Silva. Efficient algorithms for the inference of minimum size DFAs. *Machine Learning*, 44:93–119, 2001.
6. M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon. GEVA - grammatical evolution in Java. *SIGEVOlution*, 3(2):17–22, 2008.
7. M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Trans. Evolutionary Computation*, 5(4):349–358, 2001.
8. F. Tsarev and K. Egorov. Finite state machine induction using genetic algorithm based on testing and model checking. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, pages 759–762, New York, NY, USA, 2011. ACM.