GENERACIÓN DE HOJAS DE ESTILO XSLT MEDIANTE PROGRAMACIÓN GENÉTICA

Daniel Merino, Néstor Zorzano, M. G. Arenas, Juan Julián Merelo Guervós

Resumen
En este trabajo se presenta un procedimiento, basado en programación genética, para hacer evolucionar programas que transforman documentos XML, denominados hojas de estilo XSLT. Este problema se plantea en el contexto de acceso y de presentación de información almacenada en documentos XML. La solución propuesta usa una representación en árbol para las hojas de estilo, así como diversos operadores específicos, para obtener, en los casos estudiados y en un tiempo razonable, una hoja XSLT que lleva a cabo la transformación.

Palabras clave: programación genética, XML, XSLT, JEO.

I. PLANTEAMIENTO DEL PROBLEMA

El problema que se desea abordar consiste en la creación automática de scripts que nos permitan realizar transformaciones sobre documentos XML. En nuestra aplicación dichas transformaciones lo que pretenden conseguir es la obtención de ciertas partes de información del documento XML original. Estos scripts se denominan hojas de estilo XSLT (*XML Stylesheet Language for Transformations*); las hojas XSLT son programas que, aplicados a un documento XML, producen otro documento XML.

Las hojas de estilo se usan hoy en día en problemas en los que hay que transformar de un formato a otro, agregar resultados procedentes de diferentes documentos, o extraer información de un documento en XML. Un ejemplo típico puede ser extraer los titulares de un periódico en internet que use XHTML (una versión XML del lenguaje de páginas web HTML), o bien convertir de XHTML a otro formato final, tal como WML (*Wireless Markup Language*, el lenguaje que usan los navegadores de los teléfonos móviles).

El escribir hojas de estilo es un problema que se escala linealmente con el número de formatos iniciales y finales; sin embargo, teniendo en cuenta que cada conversión representa un programa escrito a mano, y que tanto los formatos iniciales como los finales pueden variar con cierta frecuencia, cualquier automatización del proceso significa un ahorro considerable de esfuerzo por parte de los programadores.

El objetivo de este trabajo es encontrar la hoja de estilo o documento XSLT que, a partir del documento XML de entrada, sea capaz de obtener un documento XML de salida (ambos suministrados por el usuario), que contendrá exclusivamente la información que se considera relevante del XML original, pudiéndose situar además dicha información en cualquier orden posible en el documento de salida.

A partir de estos dos archivos, el programa presentado en este trabajo, buscará el XSLT que transforma el archivo de entrada en el de salida, usando programación genética.

XML [1,2] (*eXtensible Markup* Language) es un lenguaje de anotación extensible, es decir, un lenguaje que permite definir elementos (etiquetas) y la gramática que siguen. XML está basado en la noción de encapsulamiento. Cada porción del documento está encapsulada en un área designada por dos tags o etiquetas.

Todos los documentos XML tienen una estructura de árbol en la que debe haber una etiqueta raíz que encierra (encapsulada) todos los contenidos del documento. Las etiquetas XML, pueden además llevar atributos, los cuales podrían contener otras informaciones necesaria para el procesamiento del documento.

XSLT [3] se utiliza para escribir y procesar hojas de estilo. Es un lenguaje de programación para conversión de documentos XML, de un tipo a otro. XSLT proporciona un mecanismo para la asociación de patrones en el documento XML original y para la aplicación de formato en estos datos. En este trabajo se utilizarán sólo tres instrucciones de XSLT: `template`, `apply-templates` y `value-of`.

XPath [4] define cómo localizar un elemento específico dentro de un documento XML, mediante referencias a nodos específicos en el documento, análogamente a como se accede a ficheros dentro de un árbol de directorios. En la especificación XPath, un documento se considera un árbol de estos nodos, accesibles por posición. XPath permite además seleccionar grupos de elementos (llamados *nodesets*) y filtrarlos usando predicados.

Todos estos elementos se combinan en este trabajo, que se puede esquematizar tal como se muestra en la figura 1. El resto del trabajo se estructura de la forma siguiente: expondrá brevemente el estado del arte, para pasar, en la sección III, a describir la solución presentada en este trabajo. Se presentarán dos ejemplos en la sección IV, para finalizar con una sección donde se presentan las conclusiones y posibles líneas de trabajo futuro

II. Estado del arte

Hasta el momento el único trabajo publicado relacionado con el tema es el desarrollado por Scott Martens [5], que presenta una técnica para encontrar hojas XSLT que transformen un archivo XML en otro archivo HTML usando programación genética. Martens trabaja sobre documentos XML sencillos, como el que muestra en su artículo, y como base para su función fitness, Martens utilizaba, la función diff de UNIX. Su conclusión es que la programación genética puede servir para dar soluciones a ejemplos sencillos del problema, aunque para ejemplos complejos se precisa un tiempo excesivo de ejecución, y tal vez la programación genética podría no ser un método del todo recomendable para este tipo de problemas.

En este artículo se trata de resolver el mismo problema, pero usando técnicas de programación genética diferentes. En vez de usar primitivas adaptadas al problema, que más adelante se convierten en hojas XSLT, en este trabajo se usan hojas XSLT representadas como un árbol, con diferentes estructuras. La función de fitness es similar, aunque se tienen en cuenta otros factores como el tamaño de la hoja XSLT obtenida.

III. Solución aportada

Se ha usado un enfoque similar a la programación genética: las hojas XSLT se han introducido en estructuras de datos de tipo árbol, que se han hecho evolucionar mediante operadores de variación; también corresponde, en sentido estricto, a programación genética puesto que lo que se está haciendo evolucionar son, en realidad, programas. Cada hoja XSLT nueva generada es evaluada mediante una función fitness que indica el grado de ajuste del XML generado por dicha hoja, con el XML de salida.

La solución se ha programado usando JEO, [6,7,8] la librería de algoritmos evolutivos en Java desarrollada en la Universidad de Granada dentro del proyecto DREAM [9,10], y que está disponible, junto con el resto del proyecto, en http://www.dr-ea-m.org

1) Operadores comunes

Los operadores se clasifican en dos tipos diferentes: el primer tipo consta de operadores que son comunes a las tres estructuras y cuyo cometido es modificar las rutas XPath que contienen los atributos de las instrucciones XSLT. El segundo tipo de operadores sirve para modificar la estructura en sí de los árboles XSLT y es específico de cada estructura.

Los operadores comunes son:

- **XSLTreeMutatorXPathAddFilter:** Añade en una ruta XPath un filtro de cardinalidad a alguna de las etiquetas de dicha ruta XPath que lo permitan, etiqueta e índice son aleatorios. Por ejemplo: **/libro/capítulo → /libro/capítulo[4]**
- **XSLTreeMutatorXPathMutateFilter:** Cambia aleatoriamente en la ruta XPath un filtro de cardinalidad ya existente en ella seleccionado aleatoriamente. Por ejemplo: **/libro/capítulo[2] → /libro/capítulo[4]**
- **XSLTreeMutatorXPathRemoveFilter:** Elimina de una ruta XPath alguno de los filtros de cardinalidad que contenga, escogido aleatoriamente. Por ejemplo: **/libro/capítulo[2] → /libro/capítulo**
- **XSLTreeMutatorXPathAddBranch:** Incorpora a una ruta XPath una nueva

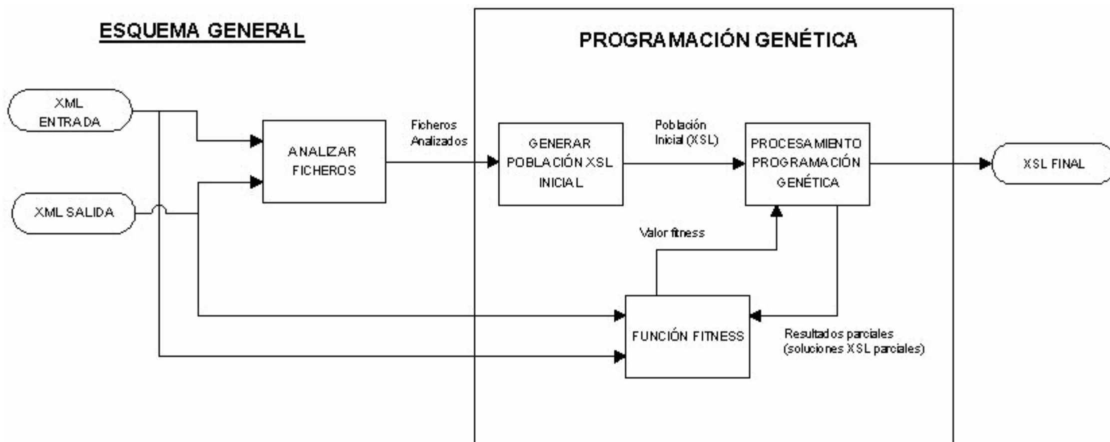

Figura 1 Esquema general de la generación automática de hojas XSLT

A. Estructura de las soluciones y operadores

Los XML generados, se encapsula en una etiqueta XML cuyo nombre es la etiqueta raíz del XML de entrada.

Se han desarrollado tres diferentes estructuras para las hojas XSLT que, cada una de las cuales parece adaptarse mejor a ciertos tipos de problemas; las tres estructuras se prueban secuencialmente en cada experimento

etiqueta, escogida aleatoriamente entre las posibles, respetando la jerarquía del árbol XML de entrada: **/libro/capítulo → /libro/capítulo/título**

- **XSLTreeMutatorXPathSetSelf:** Sustituye la etiqueta nodo más profunda de una ruta XPath por el nodo self: **/libro/capítulo → /libro/.**
- **XSLTreeMutatorXPathSetDescendant:** Elimina uno de las etiquetas intermedias de una ruta XPath, quedándose un nodo de tipo Descendant: **/libro/capítulo/título → /libro//título**
- **XSLTreeMutatorXPathRemoveBranch:** Elimina la etiqueta elemento más profunda de una ruta XPath, ascendiendo un nivel en el árbol XML. Por ejemplo: **/libro/capítulo/título → /libro/capítulo**

2) . Estructura árboles XSLT

a) **Primera estructura.**

- Los árboles XSLT tendrán una profundidad de tres niveles, contando la etiqueta raíz `<xsl:stylesheet>` que es la misma para todos los XSLT.
- De la raíz siempre colgarán un número indeterminado de instrucciones `<xsl:template match=...”>`.
- El valor del atributo `match` para el primer template que cuelga de la raíz será “/”. Ni este `template` ni su contenido se verán en ningún momento modificados por la aplicación de operadores. La única instrucción dentro de este elemento será `apply-templates`, que poseerá un atributo `select` cuyo valor será una barra “/” seguida del nombre de la etiqueta raíz del XML. De esta forma haremos que se procesen el resto de los *templates* que se incluyan en la hoja de estilo.
- Los valores para los atributos `match` para el resto de los *templates* a partir del segundo serán simplemente nombres de etiquetas del XML de entrada. Cada uno de ellos tendrán un numero indeterminado de hijos, que serán instrucciones `apply-templates`, o `value-of`. Estas instrucciones llevarán atributos `select`, cuyos valores serán rutas XPath relativas, construidas a partir de la ruta del template de las que son hijas. Estas rutas podrán incluir todas las cláusulas XPath posibles.
- Solamente se utilizarán `value-of` en lugar de `apply-templates` en caso de que la ruta XPath de su atributo select acabe en un predicado de tipo *self*, es decir en un punto “.”.

b) Segunda estructura. Las principales diferencias con la anterior son las siguientes:

- El valor del atributo `match` para el primer template que cuelga de la raíz será “/”. Este template tendrá un número indeterminado de hijos, que serán todos instrucciones `apply-templates`, cuyos valores para el atributo `select` serán rutas XPath absolutas válidas en el XML de estrada, que incluyan sólo nombres de etiquetas separadas entre sí por una sola barra.
- Los valores para los atributos match para el resto de los *XSLT*:*template* que cuelgan de la raíz del XML, a partir del segundo hijo, serán los mismos valores que tenían los atributos select de las instrucciones *XSLT*:*apply-templates* que colgaban del primer hijo de la raíz.

Habrá pues tantas instrucciones *XSLT:template* como instrucciones *XSLT:apply-templates* había colgando del primer hijo de la raíz, y se situarán en el mismo orden de estas, atendiendo a la ruta XPath que procesan.

- Cada uno de los template del apartado anterior, tendrán un número indeterminado de hijos, que serán todos ellos instrucciones *XSLT:value-of*, en las que el valor para el atributo select serán rutas XPath relativas a la ruta XPath absoluta del template del que son hijas. Estas rutas podrán incluir todos los mecanismos de XPath que permiten los operadores diseñados.
- Si la ruta absoluta de un *template*, tuviera un grado de profundidad máximo dentro de la estructura del XML su único hijo *XSLT:value-of* tendrá como valor de su select la ruta self: "."

c) Tercera estructura. Esta estructura para los árboles XSLT es idéntica a la anterior con la diferencia de que los hijos de las instrucciones `template` serán instrucciones `apply-templates` en lugar de `value-of`, excepto cuando el XPath del atributo select sea el ".".

3) Operadores sobre árboles XSLT.

Los operadores para la manipulación de la primera estructura XSLT propuesta son estos

- **XSLTreeCrossoverTemplate**: Es el único operador de cruce desarrollado. Intercambia dos subárboles de instrucciones `template` entre los dos árboles.
- **XSLTreeMutatorAddTemplate**: Este y los siguientes son todos operadores de mutación, y por lo tanto actúan sobre un solo árbol. El objetivo de este, es insertar un nuevo `template` colgando de la raíz del árbol, y darle un contenido inicial al nuevo subárbol. Para escoger que etiqueta se le da más prioridad a las etiquetas menos profundas La posición del nuevo `template` dentro del árbol será seleccionada aleatóriamente y su contenido serán etiquetas `apply-templates` o `value-of` con el atributo `select` conteniendo rutas XPath relativas a la ruta XPath del `template` padre generadas aleatoriamente mediante los operadores de XPath
- **XSLTreeMutatorMutateTemplate**: Este operador es muy similar al anterior. También crea el contenido para un nodo *template*, pero en vez de generar un nuevo subárbol, realiza el proceso sobre un de los ya presentes en el árbol, escogiéndolo aleatoriamente
- **XSLTreeMutatorRemoveTemplate**: Se escoge un *template* del árbol aleatóriamente y se elimina, así como todos los hijos que este contuviera. Este operador no se aplica sobre árboles que aparte del *template* principal cuya ruta XPath es "/", sólo tuviesen un *template* más.
- **XSLTreeAddApply**: Se añadirá un nuevo hijo, a un *template* escogido aleatóriamente de entre los presentes en el árbol. La posición de la nueva hoja dentro del subárbol que representa el *template* también será aleatoria. La etiqueta relativa de la nueva hoja se genera aleatóriamente partiendo de la ruta que contiene su instrucción *template* padre.
- **XSLTreeMutateApply1**. Este operador es muy similar al anterior, la diferencia es que en lugar de crear un nuevo hijo para el

template seleccionado, lo que hacemos es cambiar la ruta XPath de uno de los hijos ya presentes, escogido aleatóriamente.

- **XSLTreeMutateApply2**. Aquí también pretendemos modificar la ruta de uno de los hijos de uno de los *XSLT:template*. En este caso creamos una ruta XPath relativa a partir de no sólo la que contiene el *XSLT:template* padre sino también del XPath que ya contiene la hoja que vamos a modificar.
- **XSLTreeRemoveApply**: Eliminamos un hijo de un *template*, escogidos ambos altatoriamente. En caso de que el hijo del *template* eliminado, fuese el único hijo que le quedaba a este, se elimina también el propio *template* de la jerarquía. Lógicamente este operador no se aplicará si existe solamente un *template* con un único hijo.
- **XSLTreeSetTemplateNull:** Escoge una de los subárboles *template* del árbol XSLT y sustituye todo su contenido por una única instrucción *<XSLT:value-of select=”.”>*.

  Para las segunda y tercera estructura de árbol XSLT existen otros nueve operadores equivalentes, con la salvedad de que deben mantener la estructura.

B. *Función Fitness:*

Cada hoja XSLT se evalúa aplicándola al documento XML de entrada, y comparándo el resultado con el documento XML objetivo. La función de evaluación, que se tratará de minimizar, es la siguiente:

Fitness = (D / L1) + (S / 2)^2 + (L2 / 10000)

Donde:

D= Numero de líneas en las que se diferencia el XML obtenido y el XML de Salida Modelo

L1=Nº de líneas del XML obtenido

L2=Nº de líneas de la XSLT utilizada para la transformación

S=L1 – Nº de líneas del XML de Salida Modelo ó 0 si el número de líneas del XML de Salida Modelo – L1 < 0.

Para calcular el fitness, la hoja XSLT se aplica en memoria al documento XML, también representado en memoria, usando el procesador XSLT Xalan (http://xml.apache.org/xalan-j)

IV. EJEMPLOS

Se han usado diversos ficheros XML de entrada y de salida modelo, de los cuales mostraremos aquí alguno de los más simples. Uno de ellos contiene información sobre una biblioteca musical, y de ella, trataremos de obtener el título de cada disco, el nombre del autor, y a continuación las canciones del disco. Se usa una población de 500 individuos, y 200 generaciones. La primera estructura encuentra la solución en la generación número 15, lo cual tarda menos de 10 minutos en un AMD a 1.1GHz.La curva que representa el fitness de la mejor solución encontrada según el número de generaciones es la que se muestra en la **figura 3**.

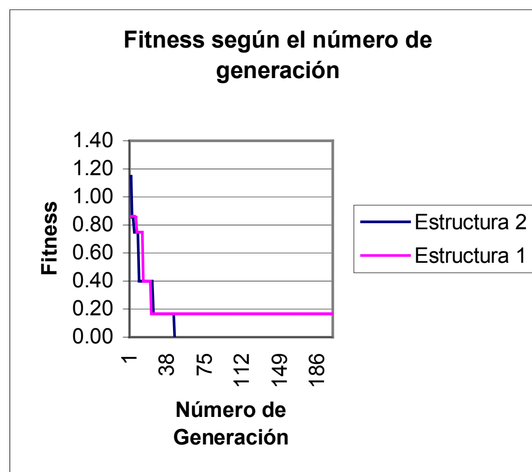

Figura 4 : Evolución del fitness en el segundo ejemplo de generación de hojas XSLT

En un ejemplo un tanto más complejo, la primera estructura no encontró la solución

idónea, mientras que la segunda sí la encontró en la generación número 42, lo cual indica la adecuación de cada estructura a un tipo de problema: en este ejemplo se pretende obtener el autor de un disco específico (concretamente el primero), el nombre de dicho disco y los nombres de las tres primeras canciones del disco. Este ejemplo, aunque quizás algo carente de sentido y utilidad, puede servirnos para mostrar un ejemplo algo más complejo, en el cual hay que usar, por ejemplo, corchetes en la ruta XPath para seleccionar el primer disco o las 3 primeras canciones.

El documento de salida modelo empleado para este ejemplo es el que se muestra en la **figura 3**. Como se puede apreciar muestra lo comentado anteriormente, el autor del primer disco, el nombre del disco, y las tres primeras canciones puestas por orden.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<biblioteca_musical>
            I
            Led Zeppelin
            God Times, Bad Times
            Babe, I'm Gonna Leave You
            You Shook Me
</biblioteca_musical>
```

Figura 4: XML de salida modelo 2

La curva que representa el fitness según el número de generación para este segundo ejemplo, para las dos estructuras empleadas, es la que se muestra a continuación, en la **figura 4**.

En dicha curva se aprecia que en un principio las dos estructuras obtenían un fitness similar, pero llega un momento en que la primera estructura se estanca, mientras que la segunda encuentra la solución rápidamente. Es por esta razón por la que usamos tres diferentes estructuras, porque hemos comprobado con numerosas ejecuciones que ciertos tipos de ejemplos se resuelven bien con cierta estructura, pero dicha estructura puede no servir para encontrar otros tipos de problemas, ya que la población se queda estancada sin mejorar el fitness, mientras que a lo mejor ese tipo de problema se puede resolver más fácilmente con otra de las estructuras. Ninguna estructura es adecuada parar todos los tipos de
problemas diferentes que pueden encontrarse, y por ello hemos seleccionado tres estructuras que, aunque similares entre sí, pueden complementarse, encontrando una las soluciones que otra no encuentra.

En la figura 5 se muestra la hoja XSLT generada por la primera estructura para el primer ejemplo expuesto.

```
<?xml version="1.0"?>
<XSLT:stylesheet                                                    version="1.0"
xmlns:XSLT="http://www.w3.org/1999/XSLT/Transform">
  <XSLT:template match="/">
    <biblioteca_musical>
      <XSLT:apply-templates select="/biblioteca_musical"/>
    </biblioteca_musical>
  </XSLT:template>
  <XSLT:template match="disco">
    <XSLT:apply-templates select="titulo"/>
    <XSLT:apply-templates select="autor"/>
    <XSLT:apply-templates select="cancion"/>
  </XSLT:template>
```

Figura 5: hoja XSLT generada para el primer ejemplo

V. Conclusiones y trabajo futuro

El campo de investigación relacionado con la conversión de formatos de documentos es muy fértil y extremadamente complejo. Cualquier nueva aportación al mismo requerirá un esfuerzo humano y computacional importante. Algo tan aparentemente sencillo como lo que se ha planteado aquí, la extracción de contenido de un documento XML, ha supuesto un esfuerzo considerable, y aún así los resultados obtenidos no han sido todo lo satisfactorios que hubiéramos deseado. Por ello, más que una aplicación al uso, consideramos que nuestro programa es un arduo trabajo de investigación en un campo hasta ahora muy

poco explotado, y que merece ser valorado como tal.

En el trabajo también se han utilizado una representación y unos operadores adaptados al problema, de forma que el mismo DOM (Document Object Model) que se usaba para aplicar transformaciones se usaba también para aplicarle operadores.

Este trabajo ha usado la librería JEO, disponible en http://www.dr-ea-m.org , una librería general y extensible de algoritmos evolutivos en Java, que forma parte del proyecto DREAM.

Nuestra aportación es mejorable y ampliable en muchos aspectos, aunque podría ser una buena base para proyectos más ambiciosos. Estas son algunas de las cuestiones más importantes que han quedado pendientes:

- Uso de atributos de los XML de entrada como elementos discriminatorios a la hora de extraer contenido, y de otras características tales como espacios de nombres.
- Aprovechamiento más amplio de las características de XPath como lenguaje de acceso a contenidos, uso de las funciones XPath de manejo de cadenas y de predicados, por ejemplo.
- Utilización de DTD asociado a XML como posible fuente de información para llevar a cabo conversiones entre formatos XML y para restringir las posibles variaciones de los elementos XML.
- Inserción de etiquetas en las hojas XSLT como base para poder generar documentos en formatos finales como HTML o WML.
- Uso de varios documentos de entrada y de salida, para que la hoja XSLT resultante sea más general.
- Ejecución de muchos más experimentos cuyos resultados nos permitiesen sacar conclusiones claras acerca de la validez de la utilización de un método como la programación genética en la conversión automática de formatos. También nos permitiría investigar qué clase de problemas requieren un tipo u otro de estructura, y qué parámetros del algoritmo evolutivo y qué tasas de aplicación de los operadores son los más adecuados para cada tipo de conversión.
- Evaluación del procesador XSLT más adecuado para la tarea. En esta aplicación se usaba Xalan (disponible en http://xml.apache.org/xalan), pero generalmente se considera que otros procesadores, tales como el Saxon o XT, son más rápidos.

VI. BIBLIOGRAFIA

[1] XML.com, sitio dedicado a XML de O'Reilly y asociados.

[2] Elliotte Rusty Harold, *XML Bible, 2nd Edition*, Wiley and Sons.,

[3] *XSLT Programmer's reference*, Michael Kay, Wrox Press.

[4] James Clark, Steve deRose, *XML Path* Language *(XPath), W3C Recommendation 16 November 1999*, disponible en http://www.w3.org/TR/xpath

[5] Scott Martens, *Automatic Creation of XML Document Conversion Scripts by Genetic Programming*, "Genetic Algorithms and Genetic Programming at Stanford 2000", pag. 269.

[6] M. G. Arenas, Brad Dolin, Juan Julián Merelo Guervós, P. A. Castillo, I. Fernández De Viana and Marc Schoenauer. *JEO: Java Evolving Objects*, In GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference. W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, N. Jonoska editors , pp 991, New York, 2002.

[7] M.G. Arenas, L. Foucart, J. J. Merelo y P. A. Castillo, *JEO: A Framework for Evolving Objects in Java*, In XII Jornadas de Paralelismo. Universidad Politécnica de Valencia, pp 185-191, 2001, Valencia.

[8] M.G Arenas, L. Foucart, M. Shoenauer and J.J. Merelo. *Computación Evolutiva en Java: JEO*, Actas del I Congreso Español de Algoritmos Evolutivos y Bioinspirados, Universidad de Mérida, pp 46-53, Merida, Badajoz, isbn: 84-607-3913-9, 2002.

[9] Ben Paechter and Thomas Baech and Marc Schoenauer and Michèle Sebag and A. E. Eiben and J. J. Merelo Guervós and T. C. Fogarty, *DREAM Distributed Resource Evolutionary Algorithm Machine*, In Proceedings of the Congress on

Evolutionary Computation 2000 ,pp 951--958, Available from http://dr-ea-m.sourceforge.net,

[10] M. G. Arenas and P. Collet and A.E. Eiben and M. Jelasity and J.J. Merelo and B. Paechter and M. Preub and M. Schoenauer, *A Framework for Distributed Evolutionary Algorithms*. In Proceedings of the Seventh Conference on Parallel Problem Solving from Nature, pp 665-675. Springer-Verlag, Vol 2439 series Lecture Notes in Computer Science. Granada September 2002.