Desarrollo de servicios para una Arquitectura Orientada a Servicios para Algoritmos Evolutivos *

P. García-Sánchez, M. García-Arenas, A. M. Mora, P.A. Castillo, C. Fernandes, J. González, G. Romero, J.J. Merelo y P. de Las Cuevas

[1] Departamento de Arquitectura y Tecnología de Computadores, Universidad de Granada, España
[2] LaSEEB-ISR-IST, Technical University of Lisbon (IST), Lisboa, Portugal
pgarcia@atc.ugr.es

Resumen Este artículo muestra el diseño e implementación de servicios para Computación Evolutiva dentro del paradigma de la Arquitectura Orientada a Servicios. Este paradigma proporciona independencia en el lenguaje y comunicación, pero el desarrollo requiere tratar con algunas cuestiones tecnológicas y de diseño, como un diseño lo más abstracto posible o la ejecución desordenada. Para resolverlas, se utiliza OSGi-Liath, una implementacion de una Arquitectura Orientada a Servicios para Algoritmos Evolutivos, para desarrollar nuevos servicios interoperables teniendo en cuenta esas restricciones.

1. Introducción

El paradigma Service Oriented Architecture (SOA) o de Arquitectura Orientada a Servicios (AOS) [1] se está convirtiendo en una importante tendencia en el desarrollo de software. Este paradigma permite la organización y distribución utilizando el concepto de *servicio*, es decir, una interacción en la que un *proveedor de servicio* publica *descripciones de servicio* (o interfaces) en el *registro de servicios*, para que los *consumidores de servicio* puedan encontrarlo y enlazarse con los proveedores para usarlo.

SOA permite independencia en el lenguaje de programación y los mecanismos de distribución y de comunicación, centrándose en la facilidad de extensión e integración, pero cuenta con las siguientes restricciones:

- Los servicios deben ser funciones de entrada/salida.
- Los servicios no deben tener estado (es decir, no usar variables globales).
- El orden de ejecución de los servicios no es fijo.
- Los servicios deben ser diseñados con un nivel de abstracción adecuado.

* Este trabajo ha sido gracias a la beca FPU AP2009-2942 y los proyectos EvOrq (TIC-3903), CANUBE (CEI2013-P-14) y ANYSELF (TIN2011-28627-C04-02).

La computación distribuida ofrece la posibilidad de utilizar procesamiento paralelo y así conseguir un mayor poder de cómputo [2]. SOA también puede aplicarse a este área, utilizando plataformas basadas en Web Services [1], o nuevos estándares para este paradigma, como OSGi (Open Services Gateway Initiative) [3].

OSGi permite construir sistemas software de calidad considerando un alto nivel de modularidad. Además de los beneficios que los paradigmas clásicos de modularización ofrecen (como el modelado orientado a objetos) y las mejoras en testeo, reusabilidad, disponibilidad y mantenibilidad, es necesario explorar otras técnicas, como el desarrollo basado en plug-ins o el diseño SOA. Estos tipos de desarrollo simplifican aspectos como la complejidad, personalización, configuración, y coste del desarrollo. En el campo de las heurísticas de optimización los beneficios de usar este tipo de desarrollo tienen lugar en la creación de algoritmos, evaluación experimental y combinación de diferentes paradigmas de optimización [4].

En nuestro trabajo anterior [5] presentamos una Arquitectura Orientada a Servicios para Algoritmos Evolutivos (SOA-EA), junto con guías y pasos para migrar del desarrollo tradicional de Algoritmos Evolutivos (AEs) a SOA. También se presentó una implementación específica, llamada OSGiLiath (*OSGi Laboratory for Implementation and Testing of Heuristics*): un entorno para el desarrollo de algoritmos distribuidos extensible con una arquitectura basada en plug-ins y basada en una especificación software ampliamente aceptada (OSGi). En este trabajo mostramos el desarrollo completo de un servicio utilizando la tecnología específica, en lugar del diseño abstracto presentado en nuestro anterior trabajo.

El resto del trabajo se estructura como sigue: después del estado del arte, presentamos los principios de diseño para crear servicios para Computación Evolutiva (CE) (Sección 3). A continuación, se explica la tecnología de implementación OSGi en la Sección 4, utilizada para construir nuestro *framework* (descrito en la Sección 5). Después, se presentan los pasos para crear servicios dentro de este framework (Sección 6). Finalmente se presentan las conclusiones y trabajo futuro (Sección 7).

2. Estado del arte

Aunque SOA se usa ampliamente en el desarrollo de software no está muy aceptada en la comunidad científica de la optimización y metaheurísticas. La mayoría de los frameworks (ver [6] para un resumen) tienen carencia de la baja generalidad, ya que están enfocados en un campo específico, como por ejemplo, la Búsqueda Local. Otro problema común es que suelen ser simplemente librerías o módulos Perl [7], no tienen GUIs o son muy complicados de instalar y requieren muchas destrezas de programación.

Entre la gran cantidad de herramientas software que existen, queremos centrarnos en los frameworks de algoritmos evolutivos distribuidos más aceptados. ECJ [8], Evolutionary Computation in Java, es un conjunto de clases Java que

pueden ser extendidas e incluyen varios módulos de comunicación. MALLBA [9] se basa en esqueletos software con una interfaz común pública. Cada esqueleto implementa una técnica de resolución para optimización en el campo de optimización exacta, heurística o híbrida. Proporciona capacidades de distribución utilizando MPI. Sin embargo estos dos frameworks no se basan en el desarrollo orientado a plug-ins, por lo que no pueden tomar las ventajas de características como gestión del ciclo de vida, versionado o enlazado dinámico de servicios, como propone OSGi.

Otra plataforma importante es DREAM [10], un framework para Algoritmos Evolutivos basado en Java que define un modelo de isla y usa el protocolo Gossip y sockets TCP/IP para comunicación. Puede desplegarse en plataformas P2P y está dividido en cinco capas. Cada capa proporciona una interfaz de usuario y diferentes niveles de interacción y abstracción, pero añadir nuevas funcionalidades no es tan fácil, debido al hecho de que el sistema debe pararse antes de añadir nuevos módulos y la implementación de interfaces debe estar definida en el código fuente, por lo que necesita compilarse con cada elemento a añadir (como en ECJ). OSGi permite añadir nuevas funcionalidades solamente compilando las nuevas características y no las ya existentes. jMetal [11], también es un framework basado en Java, pero sin posibilidad de distribución y mayormente centrado en optimización Multi-Objetivo.

ParadiseEO [12] permite el diseño de AEs y Búsqueda Local con hibridación, proporcionando una gran variedad de operadores y funciones de evaluación. También implementa los modelos paralelos y distribuidos más comunes, y está basado en librerías estándar como MPI, PVM y Pthreads. Sin embargo, cuenta con los mismos problemas que los frameworks anteriores, no cuenta con gestión del ciclo de vida, ni programación orientada a servicios. GAlib [13] es muy similar y comparte las mismas características y problemas.

En el campo de los frameworks basados en plug-ins, HeuristicLab [14] es el ejemplo más avanzado. Permite además programación distribuida utilizando Servicios Web y una base de datos centralizada, pero no usa su propio diseño de plug-ins para la comunicación distribuida. Finalmente, el único framework orientado a servicios para optimización es GridUFO [15], pero sólo permite la modificación de la función objetivo y añadir nuevos algoritmos, sin poder combinar servicios existentes.

Los frameworks anteriores han sido diseñados para ser extensibles y reusables, pero sin tener en cuenta las restricciones de SOA para lograr más independencia y mejoras en el desarrollo. A continuación explicamos cuales deben ser los requisitos para poder cumplir con esas restricciones.

3. Diseñando servicios para una Arquitectura Orientada a Servicios para AEs

En [5] demostramos que es posible crear una arquitectura orientada a servicios para AEs utilizando una tecnología SOA específica. Esta arquitectura utiliza las capacidades que SOA ofrece. Para hacer esto, se diseñaron servicios

débilmente acoplados para EAs (SOA-EA), y se implementaron utilizando una tecnología SOA y se compararon con otros frameworks. Estos servicios pueden combinarse de varias formas para obtener diferentes algoritmos (por ejemplo, de un Algoritmo Genético (AG) canónico se puede crear un NSGA-II simplemente añadiendo nuevos servicios). También se presentaron varias técnicas para combinar servicios de forma flexible.

3.1. Principios de diseño

Una de las restricciones principales de SOA, además de el enfoque en servicios abstractos es la naturaleza sin estado de los servicios. Por lo tanto, en SOA el diseño de servicios debe seguir algunas pautas.

Para empezar, debido a que los servicios no tienen por qué tener conocimiento de otros servicios, no debe haber variables globales en ninguna parte del código. Los servicios están escuchando y esperando a ser ejecutados. Por ejemplo, debe evitarse una función fitness con un contador que se incremente cada vez que se llama (para parar el algoritmo al llegar a un límite). Si varios (y diferentes) algoritmos están trabajando en paralelo, y llamando a esta función a la vez el contador no distinguiría entre algoritmos, ofreciendo resultados erróneos. Sin embargo, un servicio que mantiene algún tipo de estado se permite, por ejemplo, un servicio de estadísticas que lee eventos de todos los algoritmos que se ejecutan a la vez, pero debe ser administrado para evitar errores.

Además, un servicio debe ser indistinguible de ser ejecutado de forma local o remota en otro nodo de la red. Por lo tanto, cada etapa del algoritmo debe ser tratada como servicio a ser ejecutado en local o en remoto, incluso un servicio de *Población* o *Parámetros*. Deben proveerse mecanismos correctos para el intercambio de datos. Además, muchas implementaciones del mismo servicio pueden existir a la vez (por ejemplo, diferentes implementaciones del servicio *Crossover*), y deberían ser manejados y usados correctamente.

Un servicio es siempre una función solicitud-respuesta. Por ejemplo, el cálculo del fitness no debe ser un método de la implementación *Individuo*, como en la mayoría de los frameworks, sino una función que recibe una lista de individuos y devuelve una lista de fitness de esos individuos. Esto permite opciones como cálculo remoto del fitness y balanceo de carga distribuido, más difícil de realizar si el fitness es un método de la clase Individuo.

Pensar lo más abstractamente posible requiere separar conceptos como el orden de recombinación, y el propio crossover. Normalmente, después de la selección de los padres, los individuos se cruzan en orden. Sin embargo, si necesitamos un mecanismo de emparejamiento distinto (por ejemplo, usar más de un padre, o seleccionar un padre más de dos veces) se necesita una duplicación de esfuerzo para integrar nuevo código. Esta es la razón por la que se deben separar conceptos como *recombinación* de *crossover*.

Finalmente, no debemos asumir el orden de los servicios a ejecutar. Por ejemplo, servicios como *Recombinador* or *Mutator* deberían devolver los individuos con el fitness ya calculado. Normalmente este paso se realiza en la última etapa de la generación, pero podría ser necesario obtenerlo anteriormente para ser usados

en otras tareas, por ejemplo una búsqueda local o un recolector de estadísticas para guiar al algoritmo.

3.2. Otras restricciones tecnológicas

En [5] presentamos las ventajas de usar SOA en algoritmos Evolutivos: para empezar, SOA encaja con las ventajas de generalidad en el desarrollo de AEs explicado en [16], pero añade nuevas características, como independencia del lenguaje y mecanismos de distribución. También permite añadir y eliminar servicios en tiempo de ejecución sin alterar la ejecución general del algoritmo (es decir, no es obligatorio pararlo o añadir código extra para soportar nuevos operadores). Esto permite incrementar la interoperabilidad entre diferentes elementos software. Además, esto permite distribución del código de manera fácil: SOA no requiere de una implementación o librería de distribución concreta.

Los servicios a desarrollar deben cumplir las siguientes restricciones tecnológicas:

- Estos servicios pueden ser enlazados dinámicamente para cambiar los aspectos necesarios del AE.
- El código fuente de los servicios básicos no debe ser reescrito o recompilado para conseguir esta tarea.
- Nuevos servicios pueden añadirse en tiempo de ejecución.
- No hace falta código fuente específico para distribución, ni el código fuente de los servicios debe modificarse para este propósito. Es decir, cambiar las librerías de distribución no debe añadir código extra en los servicios existentes.

4. Tecnología de implementación

Esta sección explica algunas características técnicas de la tecnología escogida para implementar una SOA para AEs y así guiar al lector para entender el framework OSGiLiath y poder evaluar las ventajas de utilizar estas características en el desarrollo de algoritmos distribuidos que cumplan las restricciones anteriores.

La tecnología utilizada, OSGi, define una especificación para SOAs en máquinas virtuales. Proporciona características muy deseables, como abstracción de paquetes, administración del ciclo de vida o versionado, permitiendo reducción en el tiempo de desarrollo, soporte, y complejidad en el despliegue de las aplicaciones.

Esta tecnología permite descubrimiento dinámico de nuevos componentes, para incrementar la colaboración y minimizar y administrar el acoplamiento entre módulos. Además, ya existen interfaces estándar para patrones muy usados, como servidores HTTP, configuración, registros, seguridad o administración de XML.

OSGi cuenta con un modelo orientado a capas, que explican su funcionalidad. Cada una se basa en la capa siguiente:

- *Bundles*: son los componentes OSGi creados por los desarrolladores (aplicaciones).
- Servicios: Esta capa conecta los bundles de forma dinámica ofreciendo un modelo de publicación-búsqueda-enlace.
- Ciclo de vida: La API para instalar, iniciar, parar, actualizar y desinstalar *bundles*.
- Módulos: Esta capa define cómo un bundle puede importar y exportar código.
- Seguridad: los aspectos de seguridad se administran en esta capa.
- Entorno de ejecución: Define qué metodos y clases están disponibles en una plataforma específica. Por ejemplo, los dispositivos móviles cuentan con menos clases Java debido a restricciones de rendimiento.

Para cumplir la restricción de que en un buen framework para EAs los servicios deben de ser indistinguibles de ser locales o remotos se han utilizado otras características OSGi. Se ha elegido ECF (Eclipse Communication Framework)[3] por ser la implementación más madura y aceptada [17], y porque soporta el mayor número de protocolos de transmisión, incluyendo comunicación síncrona y asíncrona. Proporciona una implementación modular del estándar de Servicios Remotos de OSGi V4.2 [4]. Esta especificación utiliza el registro de servicios de OSGi para exponer servicios como remotos (siendo indistinguibles de los locales). ECF también separa el código fuente del mecanismo de descubrimiento y transmisión, permitiendo que los usuarios apliquen la tecnología más adecuada a sus necesidades y proporcionando integración con aplicaciones existentes.

5. OSGiLiath

Utilizando tecnologías como OSGi o ECF se puede crear un entorno orientado a servicios. Esta sección explica la funcionalidad y diseño del entorno OSGiLiath, presentado en [5]. Este entorno es un framework para el desarrollo de aplicaciones de optimización utilizando metaheurísticas, no centrándose en un paradigma concreto, y cuyo principal objetivo es promover el uso de SOA y OSGi, y ofrecer a los programadores las siguientes características:

- Interfaces fáciles. Después del estudio de los frameworks anteriores se ha desarrollado una jerarquía de interfaces para usar.
- Envío/recepción de datos asíncronos. Gracias a las posibilidades de distribución con OSGi, este framework cuenta con distribución fácil de servicios, sin implementar funciones específicas para esta tarea, al contrario que otros frameworks de distribución, como MPI.
- Programación orientada a componentes. El framework está orientado a plugins, por lo que se pueden añadir nuevas mejoras de forma fácil sin modificar los módulos existentes. Añadir o modificar implementaciones de servicios se puede realizar sin recompilar el código fuente.

[3] `http://www.eclipse.org/ecf/`
[4] `http://www.osgi.org/Release4/Download`

- Cliente/servidor o modelo distribuido. Todos los componentes del framework pueden comunicarse de forma bidireccional, por lo que no hace falta un administrador central si no se requiere.
- Independiente del paradigma. Este framework no está enfocado en un tipo de metaheurística.
- Servicios declarativos. Enlazar interfaces a distintas implementaciones se puede realizar sin modificar el código fuente existente. Los programadores no necesitan instanciar implementaciones de los servicios.
- Manejo de eventos remoto: utilizando las ventajas de OSGi, los usuarios pueden usar una herramienta muy potente para sincronizar o compartir datos entre servicios.

El código fuente está disponible en `http://www.osgiliath.org`, bajo una licencia LGPL.

6. Desarrollo de servicios en OSGiLiath

Esta sección explica los pasos para añadir servicios al núcleo de OSGiLiath. En esta sección se explica cómo añadir el problema del Vehicle Routing Problem (VRP).

6.1. Creación de un bundle

En OSGiLiath, los servicios se pueden añadir a bundles existentes o crear nuevos. Cada *bundle* incluye un fichero MANIFEST.MF . En este fichero se seleccionan los paquetes que importar (incluyendo interfaces y clases del núcleo de OSGiLiath) y exportar. La sección *Service-Description* en este fichero muestra la localización de Component Definitions para describir los servicios. En este caso se implementan dos interfaces: *TransportData* y *FitnessCalculator*. Otras clases relacionadas con el VRP se añaden, como *Route* o *Shop*.

6.2. Implementando servicios

Para implementar un servicio se crea una clase que implementa una interfaz. Por ejemplo, *VRPFitnessCalculator* implementa la interfaz *FitnessCalculator*. La relación entre estos dos elementos se hace en el Component Definition de la Figura 1. De esta forma, la implementación se anuncia a los otros servicios en el entorno, que pueden enlazarse o desenlazarse. Por ejemplo, la implementación *VRPInitializer* (implementando *Initializer*) requiere esta implementación para crear los individuos. Los servicios pueden enlazarse automáticamente con otros servicios con los métodos set/unset en el Component Definition. Otros servicios fuera del AE pueden añadirse (por ejemplo, en este bundle el servicio *TransportData*, que incluye información sobre distancias y tiempo a los nodos). Finalmente, se añaden *VRPMutation* and *VRPCrossover* siguiendo los consejos proporcionados en la Sección 3.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="VRPFitnessCalculator">
   <implementation class="es.ugr.osgiliath.vrp.VRPFitnessCalculator"/>
   <service>
      <provide interface="es.ugr.osgiliath.evolutionary.elements.FitnessCalculator"/>
   </service>
   <reference bind="setTransportData"
    unbind="unsetTransportData"
    cardinality="1..1"
    interface="es.ugr.osgiliath.vrp.TransportData"
    name="TransportData"
    policy="static"
   />
   <property name="name" type="String" value="vrpfitnesscalculator"/>
</scr:component>
```

Figura 1: Service Description. Este documento indica que la implementación del servicio *FitnessCalculator* es *VRPFitnessCalculator*, pero no se puede activar hasta que todas sus referencias (otros servicios) se activen.

6.3. Añadiendo comunicación

Gracias a la especificación OSGi 4.2, los servicios pueden y deben ser indistinguibles de los existentes en un entorno OSGi respecto a uno externo (en la misma máquina o en la red). Para conseguir esto, se utiliza ECF para exportar los servicios. En este caso, hemos creado un servicio de migración. Este servicio tiene dos operaciones: enviar y leer. El primero se usa para enviar los individuos al exterior y el segundo para leer los recibidos. Normalmente cada isla deberá tener un migrador para recibir individuos, y referencias a los migradores de otras islas. En nuestro caso, la implementación de *Replacer* enlaza el *Migrator* local para escribir en él los individuos a enviar. Un ejemplo de implementación del migrador es *MigratorRingBuffer*: esta clase implementa esa interfaz y automáticamente enlaza con todos los migradores disponibles en el entorno (cuenta con un vector de referencias) gracias a los métodos *bind* y *unbind* de los servicios declarativos. Por lo tantos, los migradores pueden ser añadidos en tiempo de ejecución y no pasa nada si un nodo cae. El *MigratorRingBuffer* envía individuos al Migrador remoto cuya ID es inmediatamente superior que la ID local (un anillo). La Figura 2 muestra esta configuración. Se pueden añadir algunas propiedades al servicio para que ECF automáticamente anuncie la implementación a todos los nodos de la red, por lo que no hace falta cambiar el código para cambiar de un mecanismo de distribución a otro.

7. Conclusiones

Este trabajo muestra los requisitos para crear un framework orientado a servicios para AEs y la tecnología escogida para cumplir esos requisitos. Service Oriented Architecture (SOA) ofrece independencia del lenguaje, mecanismos de distribución o incluso sistemas operativos, permitiendo la integración de diferentes elementos. Sin embargo, algunos elementos deben considerarse en el desarro-

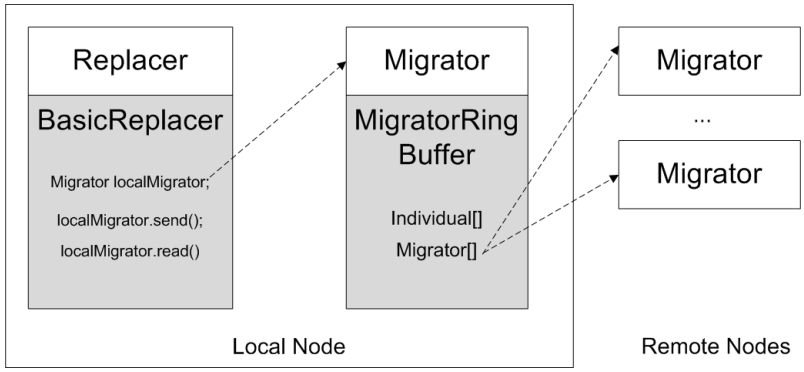

Figura 2: Utilizando el servicio *Migrator* para crear un AE distribuido basado en islas con topología en anillo (las cajas blancas son interfaces de los servicios y las grises son las implementaciones).

llo: los servicios son funciones de entrada/salida sin estado, los servicios pueden aparecer o desaparecer en tiempo real, y el orden de ejecución no debe estar fijo. En el campo de los AEs los servicios deben desarrollarse teniendo en cuenta estas restricciones. Este trabajo muestra el diseño abstracto de los elementos de un AE, junto con los requisitos tecnológicos a resolver. Se ha utilizado la tecnología OSGi como ejemplo. Los elementos para crear una SOA para AEs han sido presentados junto con un ejemplo de desarrollo.

Como trabajo futuro, se pretende realizar un estudio sobre escalabilidad utilizando otros algoritmos (como GRASP, Scatter Search, Ant Colony Optimization y otros). Además está planeado incrementar el uso de las ventajas de OSGi, como la administración de eventos o la gestión de servicios de forma más profunda. Finalmente, se plantea desarrollar un portal o un repositorio Maven[5] para centralizar todas las implementaciones de problemas y algoritmos para permitir la distribución junto con la plataforma base. También se plantea un estudio sobre portar software existente (especialmente los frameworks escritos en Java, como DREAM o ECJ) a nuestro framework. Además, debido a la facilidad de enlazar implementaciones a interfaces, se plantea desarrollar la funcionalidad de elegir una u otra implementación dependiendo de varios parámetros, o por ejemplo, utilizar Programación Genética para evolucionar e hibridizar algoritmos.

Referencias

1. Papazoglou, M., van den Heuvel, W.J.: Service oriented architectures: approaches, technologies and research issues. The VLDB Journal **16** (2007) 389–415 10.1007/s00778-007-0044-3.

[5] `http://www.maven.org`

2. Altunay, M., Avery, P., Blackburn, K., Bockelman, B., Ernst, M., et al.: A Science Driven Production Cyberinfrastructure-the Open Science Grid. Journal of GRID Computing **9**(2, Sp. Iss. SI) (2011) 201–218
3. OSGi Alliance: OSGi service platform release 4.2 (2010) Available at: `http://www.osgi.org/Release4/Download`.
4. Wagner, S., Winkler, S., Pitzer, E., Kronberger, G., Beham, A., Braune, R., Affenzeller, M.: Benefits of plugin-based heuristic optimization software systems. In Moreno Díaz, R., Pichler, F., Quesada Arencibia, A., eds.: Computer Aided Systems Theory - EUROCAST 2007. Volume 4739 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2007) 747–754
5. García-Sánchez, P., González, J., Castillo, P.A., Arenas, M.G., Guervós, J.J.M.: Service oriented evolutionary algorithms. Soft Comput. **17**(6) (2013) 1059–1075
6. Parejo, J.A., Cortés, A.R., Lozano, S., Fernandez, P.: Metaheuristic optimization frameworks: a survey and benchmarking. Soft Comput. **16**(3) (2012) 527–561
7. Merelo Guervós, J., Castillo, P., Alba, E.: Algorithm::evolutionary, a flexible Perl module for evolutionary computation. Soft Computing - A Fusion of Foundations, Methodologies and Applications **14** (2010) 1091–1109
8. Luke, S., et al.: ECJ: A Java-based Evolutionary Computation and Genetic Programming Research System (2009) Available at `http://www.cs.umd.edu/projects/plus/ec/ecj`.
9. Alba, E., Almeida, F., Blesa, M., Cotta, C., Díaz, M., Dorta, I., Gabarró, J., León, C., Luque, G., Petit, J., Rodríguez, C., Rojas, A., Xhafa, F.: Efficient parallel LAN/WAN algorithms for optimization. The MALLBA project. Parallel Computing **32**(5-6) (2006) 415–440
10. Arenas, M., Collet, P., Eiben, A., Jelasity, M., Merelo, J.J., Paechter, B., Preuß, M., Schoenauer, M.: A framework for distributed evolutionary algorithms. In: Parallel Problem Solving from Nature, PPSN VII. (2002) 665–675
11. Durillo, J.J., Nebro, A.J., Alba, E.: The jmetal framework for multi-objective optimization: Design and architecture. In: IEEE Congress on Evolutionary Computation. (2010) 1–8
12. Liefooghe, A., Jourdan, L., Talbi, E.: A software framework based on a conceptual unified model for evolutionary multiobjective optimization: ParadisEO-MOEO. European Journal of Operational Research (2010)
13. Wall, B.M.: A genetic algorithm for resource-constrained scheduling, Ph.D. thesis, MIT. (1996) Available at: `http://lancet.mit.edu/ga`.
14. Wagner, S., Affenzeller, M.: HeuristicLab: A generic and extensible optimization environment. In Ribeiro, B. and Albrecht, R.F. and Dobnikar, A. and Pearson, D.W. and Steele, N.C., ed.: Adaptive and Natural Computing Algorithms. Springer Computer Science (2005) 538–541 7th International Conference on Adaptive and Natural Computing Algorithms (ICANNGA).
15. Munawar, A., Wahib, M., Munetomo, M., Akama, K.: The design, usage, and performance of gridufo: A grid based unified framework for optimization. Future Generation Computer Systems **26**(4) (2010) 633 – 644
16. Gagné, C., Parizeau, M.: Genericity in evolutionary computation software tools: Principles and case-study. International Journal on Artificial Intelligence Tools **15**(2) (2006) 173
17. Petzold, M., Ullrich, O., Speckenmeyer, E.: Dynamic distributed simulation of DEVS models on the OSGi service platform. Proceedings of ASIM 2011 (2011)