A Framework for Distributed Evolutionary Algorithms*

M. G. Arenas[3], Pierre Collet[2], A. E. Eiben[1], Márk Jelasity[1], J. J. Merelo[3], Ben Paechter[4], Mike Preuß[5], and Marc Schoenauer[2]

[1] Free University of Amsterdam, Amsterdam, The Netherlands
[2] Ecole Polytechnique, Palaiseau, France
[3] Universidad de Granada, Granada, Spain
[4] Napier University, Edinburgh, Scotland
[5] University of Dortmund, Dortmund, Germany

Abstract. This paper describes the recently released DREAM (Distributed Resource Evolutionary Algorithm Machine) framework for the automatic distribution of evolutionary algorithm (EA) processing through a virtual machine built from large numbers of individual machines linked by standard Internet protocols. The framework allows five different user entry points which depend on the knowledge and requirements of the user. At the highest level, users may specify and run distributed EAs simply by manipulating graphical displays. At the lowest level the framework turns becomes a P2P (Peer to Peer) mobile agent system, that may be used for the automatic distribution of a class of processes including, but not limited to, EAs.

1 Introduction

The Distributed Resource Evolutionary Algorithm Machine (DREAM) [1] was first described in [12]. It provides a framework for the production of evolutionary algorithm systems and systems of evolving agents which use the Internet to allow distributed processing in a peer-to-peer scalable fashion. The system also allows the secure sharing of the spare CPU resources of a large number of computers. The scalability of the system will allow new types of problems to be studied which require either very large amount of processing power, or vast numbers of evolving agents competing and co-operating to find a solution to some problem together. The first public release of the software has recently been made, and this paper describes the architecture and functionality of that system.

2 System Architecture

The system architecture is split into five modules, and each provides a user interface at a different level interaction and abstraction. The architecture is shown in Fig. 1 along

* The original publication is available at www.springerlink.com. In PPSN VII, LNCS 2439, pp. 665–675, Springer-Verlag, 2002. (doi:10.1007/3-540-45712-7_64) This work is funded as part of the European Commission Information Society Technologies Program (Future and Emerging Technologies). The authors have sole responsibility for this work, it does not represent the opinion of the European Community, and the European Community is not responsible for any use that may be made of the data appearing herein. The authors are listed in alphabetical order.

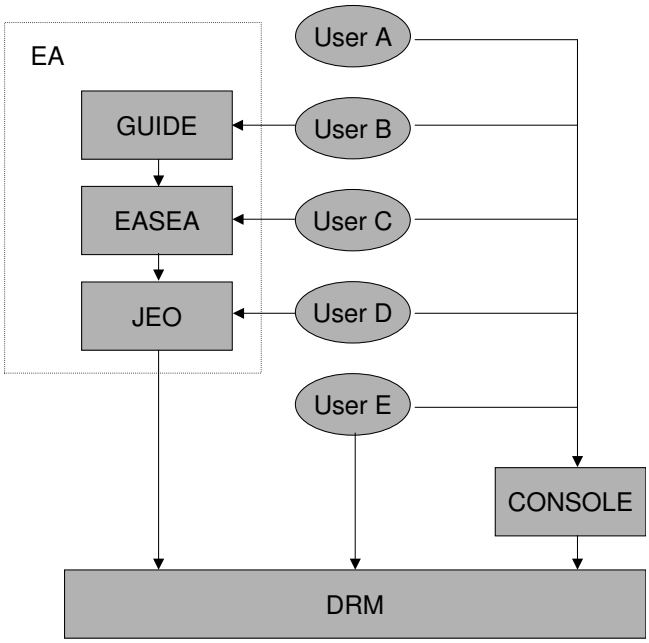

Fig. 1. The DREAM architecture and its user entry points

with the user entry points. The entry points give a variety of interfaces, with different levels of ease-of-use and power. The upper levels are easier to use, but flexibility is more limited, the lower levels require greater expertise, but give the user greater control over the system. Some of the modules or groups of modules can be used independently of the others, providing additional functionality outside the context of the integrated DREAM system. The five types of user can be categorised as follows:

- *User A* does not wish to use the DREAM for conducting experiments, but simply wishes to donate their spare CPU time or monitor the experiments of others. This type of user, interacts with the DREAM only through the Console, other types of user use the console and (normally) one other user entry point.
- *User B* is either not an experienced programmer or wishes to rapidly prototype a system without the need for textual programming. This type of user interacts with the GUIDE layer which allows distributed evolutionary algorithms to be defined using a fully graphical interface. The GUIDE interfaces with the EASEA layer through the use of the EASEA language. This user should have a knowledge of the workings of evolutionary algorithms.
- *User C* programs the system through the EASEA layer, which provides a high level textual language in which to program distributed evolutionary algorithms. This layer produces Java code through a compiler. The code it produces uses the objects and methods of the JEO (Java Evolutionary Object) library.

– *User D* programs directly in Java and makes use of the JEO library. The library not only provides useful objects and methods for evolutionary computing, but also provides an API (Application Programming Interface) to the DRM (Distributed Resource Machine) core layer. This layer is intended for users with a knowledge of evolutionary algorithms and the Java language.
– *User E* is an expert user, who programs using the DRM API directly. This level of user can use the DRM for many useful distributed processing purposes beyond evolutionary computing, but is not fully protected from the complexities of this type of programming.

The following sections describe each of the architecture layers in turn.

3 GUIDE Layer

The Graphic User Interface for DREAM Experiments is the entry point at the highest possible level of interaction and abstraction. The idea is that even a non-expert programmer should be able to use the DREAM through GUIDE, specifying the algorithm by means of point-and-click in a number of panels referring to different parts of the evolutionary algorithm.

An Evolutionary Algorithm is made of two parts that are almost completely orthogonal: on the one hand are the problem dependent components, including the *genotype structure*, its *initialization*, the *variation operators* (crossover, mutation and the like) that will be applied on the genotypes and of course the *evaluation* (computation of the fitness value). On the other hand, the *evolution engine* implements the artificial Darwinism part of the algorithm and should be able to handle any population of objects that have a fitness, regardless of the actual genotype. Evolution engines are made up of two steps, the *selection* of some parents to become actual *genitors* and generate offspring, and the *replacement* of some individuals by some offspring to build up the next generation.

The structure of GUIDE reflects this point of view, and offers four panels to the user: the Problem Specification panel to define the problem dependent components, the Evolution Engine panel for the Darwinian components, the Distribution Control panel to define the way the different islands will communicate (see section 5) and the Experiment Monitor panel, from where the user can run his experiment and view the temporary and final results.

3.1 Evolution engine specification

As far as the ultimate end user is concerned, writing the evolution engine part has nothing to do with the problem being solved – and this is where GUIDE can handle the work completely.

Any decent book about EAs describes the main evolutionary engines, namely *Generational* or *Steady-State GA*, *Plus* or *Comma Evolution Strategy* selection, ... It should be possible for a user to say "I want to solve my problem with Generational GA evolution, using 100 generations of 50 infohabitants, initially evenly distributed in 5 different islands, with such and such parameters."

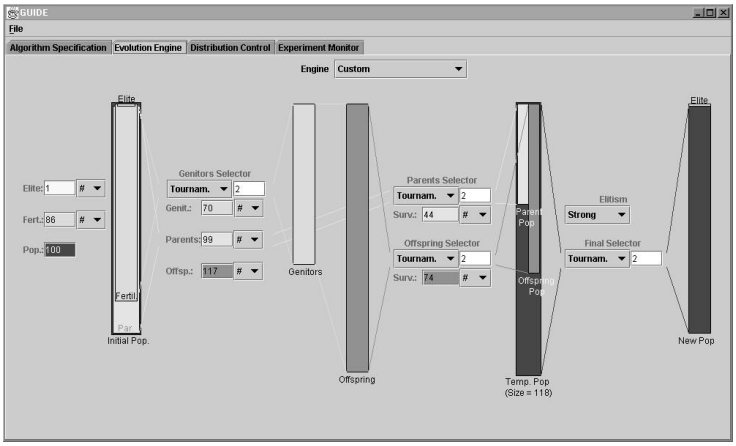

Fig. 2. The Evolution Engine panel of the GUIDE.

This is part of the functionality offered by the GUIDE. However, restricting the choice to the five mainstream paradigms cited above would have been very restrictive: experience shows that most real-world EAs in fact deviate one way or another from the strict historical paradigms.

The Evolutionary Engine panel of GUIDE (Figure 2) therefore offers a generic set of primitives from which the user can define a huge variety of implementation artificial Darwinism implementations. The already-mentioned "classical" evolution engines are simple instantiations of GUIDE parameters. The pedagogic interest is immense, as one can see the parameters changing when selecting any of the predefined engines. Alternatively, thousands of unexplored engines can be tested, simply by arranging the parameters in an original way.

3.2 Problem-dependent components

When using the DREAM, the user refers to a specific problem to be solved. This means that the notion of programming cannot, unfortunately, be completely removed from the description of experiments – at least the computation of the fitness of infohabitants has to be typed in textually. At the moment, this is also true in GUIDE for all problem-dependent parts.

GUIDE therefore provides a panel that allows the user to type in the specificities of his problem in terms of genome structure, genome initialiser, genome recombinator, genome mutator and genome evaluator. This is done thanks to a very high level language with a C/C++/Java like syntax that completely hides the very complex notion of classes/objects that is required to handle populations of genomes. The user can hence concentrate on his problem, not on making the whole thing work in some complex library.

4 EAsy Specification of Evolutionary Algorithms

GUIDE turns mouse driven diagrams representing some complex evolutionary algorithm into runnable code using the genome structure, genetic operators and evaluator specified by the user thanks to a powerful intermediate language associated with a compiler.

The fact that at some point, some representation must be used to, at least, save and reload user experiments, implies that this same representation should be capable of describing virtually any kind of evolutionary algorithm. Rather than designing some obscure internal representation, specific to the DREAM, the decision was taken to create a fully independent human-readable language that could have an existence of its own.

Among other advantages, this meant that EASEA [5, 11] could be developed completely independently from the evolutionary library specific to the DREAM project, simply by using already existing off the shelf evolutionary libraries.

As a result, an evolutionary algorithm specified in the EASEA language (or specified and saved by GUIDE with an EASEA syntax) can be compiled in a C++ source file using the GALib evolutionary library[13], or the EO [2, 10] fully templatised object oriented library; or of course in Java source files using the JEO library of the DREAM.

As a matter of fact, the EASEA language compatible with GALib or EO has been downloaded more than one thousand times in the last 12 months, with the result that even before the DREAM was released, many users around the world have been unknowingly developing potential DREAM applications, that would only need a fully working DREAM system to exploit the distributed resources of the Internet.

The genome structure, as well as its initialiser, recombinator, mutator, and evaluator described in the GUIDE are compiled by the EASEA compiler into a pure Java classes meeting the JEO requirements. The resulting files are ready for compilation by the JAVA compiler. Compiling as well as launching the experiment can be carried out from within the GUIDE Experiment Monitor panel as well as from the command line, depending on the type of user (B or C).

5 JEO Layer

This section explains the Evolutionary Computation layer in DREAM. This layer is called Java Evolutionary Objects, JEO for short.

5.1 JEO from the User's Point of View

JEO is a framework for building evolutionary computation experiments, which sits on top of the DRM layer, but can be used independently from it if no parallel execution is required. It provides a DREAM entry point to *User D* so it is flexible, powerful and extensible enough to allow the users to design, develop and control their experiments in the easiest way. JEO output is an experiment specification that can be run in distributed fashion using the DRM module (see section 6). JEO can also act as the bridge between the EASEA and DRM modules. It hides the physical DRM details, like communication protocols or threads, letting the user concentrate on evolutionary computation concepts.

User D must be familiar with EC concepts. This type of user prefers to work with Java classes to have direct control over the experiment. The procedure to build an experiment is simple. The user implements a Java class to create the objects that will be placed in each Island. The set of Islands that constitutes the experiment will be launched in DRM to distribute the experiment.

For each task to be performed on an island, the user must create one specific object. The requirement on those objects is to meet the corresponding JEO interface, e.g. an *operator* must implement the operators interface, an *individual* the individuals interface... Moreover, JEO provides several classes that actually implement each interface and can be used as examples. However, JEO provides no implementation for the *evaluator* interface, as it is totally problem-dependent.

5.2 Islands and Island Components

JEO provides a general Island class: an island holds one or more *environments*. An environment groups one population and the objects that will be used to evolve that population (i.e. a complete evolutionary algorithm). The different environments possibly interact through the *assessor* objects, that perform the evaluation of all environments simultaneously.

First of all, all populations of the island are initialised using the corresponding initialisers. The following cycle is the run:

1. *Stopping test and statistics calculation* are carried out by *Terminators*: different stopping criteria are available, as well as different on-line or cumulated statistics. These values are sent to the DRM console, passed to GUIDE Experiment Monitor panel, or simply displayed on the screen in the case of JEO being used stand-alone.
2. Each island population then undergoes a single step of evolution through the corresponding *Breeder*. This includes (fitness-based) *selection* and *variation operators*.
3. The *assessor* is then called upon all environments (i.e. all populations). This step can be a simple evaluation in the case of evolutionary optimisation experiment, or can correspond to a few life steps in the case of artificial life simulations. Nevertheless, at the end of this step, all infohabitants are *Rewarded* . This reward mechanism leaves room for forthcoming large-scale simulation of sociological and economical evolution, where every operation (from mating and mutating to being evaluated) will have a cost, and infohabitants will simply fight for survival, without explicit fitness function being defined.
4. *Migration* is then performed by two objects, *Immigrator* and *Emigrator*. *Immigrator* reads from the input immigrant's buffer the recently arrived individuals from some neighbouring island and decides how to include them into the Island populations. *Emigrator* selects the individuals that will emigrate, and selects the neighbour target island.
5. *Clean* is the method that eliminates the "poorest" infohabitants (in term of the rewards mentioned above), the remaining infohabitants becoming the initial population of the next generation.

The experiment can be submitted to the DRM module using the DREAM Console or DRM command line and subsequently the DRM module distributes the experiment

through the DREAM machine. The distribution mechanism is completely transparent to the user. The user identifies the experiment using an experiment name and identifies each island using the island name set into the experiment specification.

Parallel evolution is performed asynchronously; every island writes to another island's buffer at any time it is required to; every island reads from its own buffer. There is a thread that receives immigrants and places them in a buffer, from where the thread that runs the evolutionary algorithm can read them.

5.3 JEO as Java Tool

JEO is developed using *jdk 1.3*, like the rest of the DREAM project. JEO classes and interfaces are organized into packages [4, 3]. The main one is the `dream.evolution` which includes the main classes and interfaces as Island class or operators interface. Other important packages are for example `dream.evolution.genomes`, including interfaces and classes to build genomes as linear, tree or graph structures, and `dream.evolution.operators`, that includes variation operators and selectors.

All classes have `javadoc` comments to help the user to understand all variables and methods. Moreover each package includes a `package.html` file where package elements are described.

JEO today includes some basic evolutionary algorithms examples, that solve easy toy problems (e.g. OneMax and Symbolic Regression).

6 DRM Layer

The distributed resource machine (DRM) is composed of (a possibly very large number of) machines all over the Internet forming an environment for distributed applications. At this level of abstraction it is no longer assumed that the application is from the field of evolutionary computation. This section summarizes the basic concepts and the algorithms which implement these concepts. For a more detailed description of the different aspects of the DRM please refer to [8, 9].

6.1 Application Model

In a traditional single-machine environment, an application is composed of one or more threads which are run by an operating system (OS). The OS controls the threads, it assigns resources and takes care of different security aspects. When adapting this approach to very large scale distributed environments, not all aspects can be implemented in exactly the same way due to the relatively high costs of information exchange.

A key feature of our model is that we think of an application as a set of cooperating autonomous agents. For instance, an evolutionary algorithm is implemented in this framework by a set of agents which all host an island. An agent is analogous to a thread in an OS: running an application is done through launching one or more agents that can communicate with each other, can make decisions based on the state of the system as a whole (e.g. available computational resources) or based on the state of each other.

The agents can also launch new agents themselves. The DRM controls the agents, the resources they have access to, security, etc.

However these agents have much more freedom: they are also mobile, they can change their physical location while performing computations. On the new location they can continue their task.

Despite the similarities, the applications we have in mind are rather special. Usual things like quickly accessible shared memory are a luxury in the world of large distributed systems. Another problem is the unreliable nature of both the communication channels between the nodes of the environment and the unreliability of the nodes themselves. This restricts the application area to tasks that are robust (not sensitive to loosing a subset of the agents they are composed of) and massively parallel (i.e. only little communication is necessary between the agents). Fortunately evolutionary computation fulfills these requirements and there are additional possible applications as well.

6.2 Implementation

An important implementation decision was that we do not use central servers at all. This is to maximize scalability and robustness. With this restriction even maintaining the connectivity of the network becomes a major challenge. This problem and also the problem of information distribution through the DRM is solved using *epidemic protocols* [6, 7].

In our system this protocol works as follows: each node in the DRM has an *incomplete* database which contains entries on other nodes. These entries contain information over the available resources and the agents there are running on the node. Each node chooses a random node from its database regularly to exchange information. If the size of the database exceeds a given limit, randomly chosen entries are removed to keep the communication costs affordable.

Theoretical and practical results show [7–9] that this protocol is a very effective and scalable way of distributing information over the network. For example if the database of each node contains only 100 random entries then a network of 10^{33} such nodes is partitioned only with a probability of at most 10^{-10}.

The question of mobility and security was solved by choosing the Java environment to implement the DRM. This environment offers a natural solution for moving executable code as well as data between hosts and it offers a rich set of security features.

7 DREAM console

The DREAM console is the primary tool for managing a computer connected to a DRM. It utilises the metaphor of a file manager view onto the DRM to present its known components and the output they produce (see Figure 3) . The list of components is mainly retrieved from the incomplete database kept by the underlying DRM layer (see section 6.2), so that the console usually only listens to the ongoing communication rather than inducing network traffic of its own. Different component types include nodes, experiments, users, islands and agents. Although the console is designed to help the user during execution of an experiment, it does not completely hide DRM layer information.

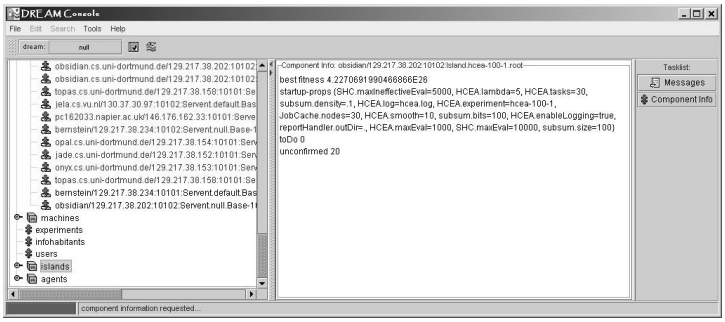

Fig. 3. The Console View.

As stated before, agents are analogous to threads in an operating system and islands are always implemented as agents (see section 6.1). However, there are agents performing operating system tasks like shutting down experiments or searching for a piece of information specified by the user. These are visually separated from the island agents.

7.1 Supported user tasks

User actions supported by the console include, but are not limited to, component inspection and search; experiment startup and control; and visualization and analysis of experiment results. Regardless of the way an experiment has been constructed[6], it can be started using the console. Thereafter, the user may watch creation and evolution of the islands belonging to the new experiment. Actual status or output of a known component can be retrieved by simply clicking on it. If an island is able to change its behaviour during runtime, the experimenter may use the console to send new setup information, for example parameter changes. Finally, it is also possible to shutdown one or multiple components via the console. There are issues relating to the access rights policy for different users, but these are beyond the scope of this paper.

7.2 Interfaces to user/DRM code

For component information retrieval or messaging, the console uses interfaces defined by the DRM layer that are implemented by the experiment code. Depending on the user entry point, this implementation is done with varying degrees of automation.

8 Conclusion

We have described a system for the easy specification and implementation of highly distributed evolutionary algorithms. This system has already been implemented and the

[6] The procedure of experiment construction is different for users of types B-E, depending on the tools and libraries they chose.

first version released. It includes several independent modules which can be used in an integrated fashion or as stand-alone units. The system allows for the sharing of CPU resources in a secure and scalable fashion. Future work will concentrate on developing algorithms and applications that work particularly well with this architecture. This will not only include applications that require vast amounts of CPU time, but is expected to include solutions to problems that can more easily be partitioned in some way (for example some data mining or scheduling problems). Systems which involve the co-operating and competing of evolving agents, either to solve real world problems, or to model aspects of society, will also be studied.

Acknowledgments

The authors would like to thank the other members of the DREAM project for fruitful discussions, the early pioneers [12] as well as Hans-Paul Schwefel, Emin Aydin and Daniele Denaro.

References

1. http://www.dcs.napier.ac.uk/ benp/dream/dream.htm
2. http://eodev.sourceforge.net.
3. M. Arenas, L. Foucart, J. Merelo, and P. A. Castillo. Jeo: a framework for evolving objects in java. In *Actas Jornadas de Paralelismo*. Universidad Politécnica de Valencia, 2001.
4. M. Arenas, L. Foucart, M. Schoenauer, and J. Merelo. Computacin evolutiva en java: Jeo. pages 46–53. Universidad de Extremadura, Febrero 2002.
5. P. Collet, E. L. M. Schoenauer, and J. Louchet. Take it easea. In *Parallel Problem Solving from Nature VI*, pages 891–901. Springer Verlag, LNCS 1917, 2000.
6. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database management. In *Proc. 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, Vancouver, Aug. 1987. ACM.
7. P. T. Eugster, R. Guerraoui, S. B. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probablistic broadcast. In *Proc. International Conference on Dependable Systems and Networks (DSN 2001)*, Göteborg, Sweden, 2001.
8. M. Jelasity, M. Preuß, and B. Paechter. A scalable and robust framework for distributed applications. In *CEC2002*, pages 1540–1545. IEEE, IEEE Press, 2002.
9. M. Jelasity, M. Preuß, M. van Steen, and B. Paechter. Maintaining connectivity in a scalable and robust distributed environment. In H.E. Bal et al., eds, *Proc. 2nd IEEE Intl Symposium on Cluster Computing and the Grid (CCGrid2002)*, Berlin, Germany, pages 389–394, 2002.
10. M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving objects: a general purpose evolutionary computation library. In P. Collet et al., eds, *Artificial Evolution'01*. pages 229–241, Springer Verlag, LNCS 2310, 2002.
11. E. Lutton, P. Collet, and J. Louchet. Easea comparisons on test functions : Galib versus eo. In P. Collet et al., eds, *Artificial Evolution'01*. pages 217–228, Springer Verlag, LNCS 2310, 2002.
12. B. Paechter, T. Bäck, M. Schoenauer, M. Sebag, A. E. Eiben, J. J. Merelo, and T. C. Fogarty. A distributed resoucre evolutionary algorithm machine (DREAM). In *CEC2000*, pages 951–958. IEEE, IEEE Press, 2000.
13. M. Wall. http://lancet.mit.edu/ga/.