Galapagos

An Advanced Evolutionary Algorithm Development Framework

Version 1.0 User & Developer Documentation

Table of Contents

1 Introduction

Galapagos is a software framework which can be used to build evolutionary algorithm applications which can range from basic genetic algorithms to advanced hybrid, distributed, parallel or multi-objective variants. Galapagos can be used in a wide variety of applications, independent of operating system, problem domain or data representation.

Using Galapagos requires at least some programming in order to define the problem and some understanding of evolutionary algorithms and of the resources available to the user. Galapagos is written in Java and can be used to solve problems in conjunction with existing code written in any language, with proper interfacing.

This manual, which is meant to be used in conjunction with the more technical JavaDocs provided with the Galapagos framework, is divided into two chapters which introduce the platform and its underlying distribution engine, followed by detailed reference appendices, as explained below:

Chapter 2 describes LightGrid, the light-weight distributed computing engine which underlies Galapagos. Galapagos was designed from the bottom up to be a distributed application, so an understanding of LightGrid is essential to any user or developer.

Chapter 3 describes the evolutionary algorithm layer of the Galapagos framework, starting with a description of how Galapagos is used, then describing the various classes that underlie the framework.

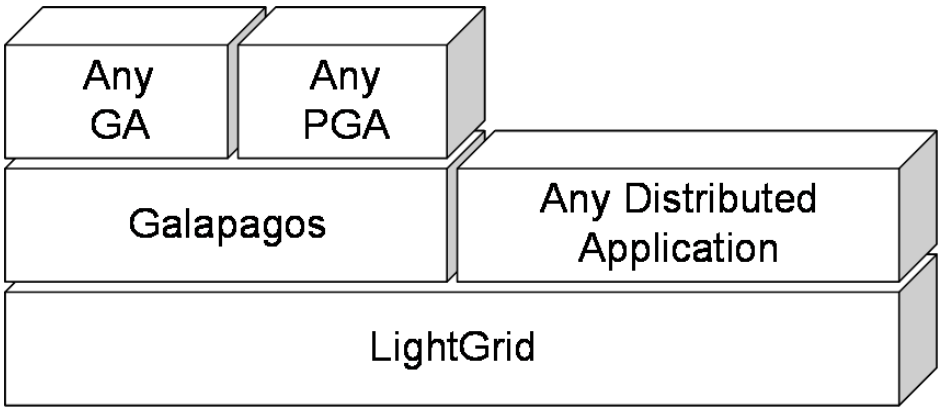
Appendix A describes the commands and sequence of operations for the various Galapagos executables.

Appendix B describes how to launch the various Galapagos executables, for those not familiar with Java.

Appendix C is a reference for the small library of evolutionary operators included with Galapagos (ComponentLib).

2 LightGrid

Galapagos was designed and built from the bottom up as a distributed application, rather than starting as an evolutionary algorithm framework to which distributed features were added. All of the distribution aspects of Galapagos' operation are handled by an original engine called LightGrid, described in this chapter. LightGrid is not fundamentally related to evolutionary algorithms, and as such can be used independently of Galapagos to develop other distributed applications. All of the features of Galapagos related to evolutionary algorithms are kept in the Galapagos layer of the framework, described in the next chapter.



2.1 Components & Terminology

The basic sequence of LightGrid's operation begins as follows: a given software process (the *client*) requires that one or more independent units of work (called *jobs*, comprising of data which must be processed, returning an output, the *result*) that can be performed in parallel. A number of independent hardware units are available to perform the work, and on each runs a process called a *resource*. The client passes jobs to a *dispatcher* process, which distributes them to the various resources, keeping track of which resource is busy, and ensuring that the jobs are processed as efficiently as possible. Resources return the results to the dispatcher, which then passes them back to the client. The dispatcher can manage jobs from more than one client, and attempts to fairly distribute the available resources between them.

2.2 Development

In code, the dispatcher is fully implemented as a Java command-line executable, as is the resource. The resource is also a complete command-line executable which uses dynamic class loading to process the jobs, using the class name provided as part of the job object sent by the client. LightGrid provides an

abstract job-processor class that must be subclassed in order to process jobs with a resource. LightGrid also provides an abstract client class which provides functions to send jobs and process results. This class is subclassed to create LightGrid application clients.

All of the information needed to build LightGrid clients and job-processors is available in the JavaDocs.

2.3 Discovery

An important issue that must be dealt with when building distributed applications is that of *discovery*, or the process whereby clients, dispatchers and resources 'find out' how to communicate with each other, by finding out each other's IP addresses.

When the dispatcher first starts up, it creates a file called 'dispatcher.txt' which contains the hostname of the current machine, as determined by the Java Virtual Machine (i.e. depending on how the computer is set up, it may not be a Fully Qualified Domain Name). It then looks for a file called 'resources.txt' which if found, should contain hostnames or IP addresses for active resources. It then tries to contact these resources. If successful, it passes them its hostname, and in turn receives their hostname. Again, these are as determined by Java and thus may not be FQDNs. Depending on the particulars of the network at hand, this may prevent the dispatcher and resources from effectively communicating and the computers may have to be slightly reconfigured or the dispatcher/resources files may have to be manually created.

Alternatively, at startup, the resources look for a 'dispatcher.txt' file and if found, attempt to contact the dispatcher, with the same FQDN caveats.

The above features make it practical to launch multiple resources in a networked file system situation where all resources and the dispatcher have access to the same directory. If some script is used to launch multiple resources remotely, this script can write the 'resources.txt' file before launching the dispatcher.

Clients must be provided with the dispatcher's hostname and contact it directly. More information on the specifics of this process are available in the JavaDocs for the client class.

For reference, the following ports are used by LightGrid:

Port Number	Usage
7070	Dispatcher to Resource

7071	Resource to Dispatcher
7072	Dispatcher to Client
7073	Client to Dispatcher

2.4 Dispatching Policy

The LightGrid dispatcher is responsible for matching up jobs to resources and getting results back to clients as efficiently as possible. This includes dynamically dealing with resources crashing or disappearing as well as dealing with heterogeneous resources (in terms of processing power).

The policy used to do this goes as follows:

1. All jobs received from a given client are added to that client's unsent-jobs queue.
2. The dispatcher iterates over all of the clients' unsent-jobs queues in sequence, assigning jobs to the next available resource. If a job is marked as 'reassignable' by the client, it is placed in that client's reassignable-sent-jobs queue after being sent. If a client's unsent-jobs queue is empty, the dispatcher attempts to get a job from that client's reassignable-sent-jobs queue and reassigns it to the current free resource, recording this resource as also working on that job. If both queues are empty, the dispatcher looks at the next client
3. When a result is received, the dispatcher sends the results to the corresponding client and sends a 'reset' command to each resource still working on that job (if it was reassignable). The resources are not required to heed this command, and if the job is not resetable, subsequent results for this job will be ignored. When all resources working on a job have either returned a result or acknowledged the reset, the job is removed from the reassignable-sent-jobs queue.

The basic strategy to deal with errors, unreliable resources or slow resources is simply to reassign the jobs while they exist. An example of this strategy is as follows: if we have two machines (A and B), with A three times as fast as B, and two jobs, then each machine will get a job. A will finish before B does and B's job will be given to A (while B is still processing). A will then finish again before B and B will be reset. In this situation, LightGrid only really used the A machine. However, given 100 jobs to process, A will get jobs 1, 3, 4, 5, 7, 8, 9 etc while B will get jobs 2, 6, 10 etc and in the end, A will process 75 jobs and B 25. This is in

line with the two machines' processing speeds. LightGrid thus works most efficiently when it has many more jobs than resources.

2.5 Limitations

The current version of LightGrid uses a separate socket connection for each communication between components, which causes a problem with very rapid job-processors. LightGrid was initially designed with a processing time of around 4 seconds, and functions well within and above this range. Below this range, the TCP/IP connections are not torn down quickly enough and the dispatcher computer saturates with waiting connections. This issue is being looked at.

3 Galapagos

Referring to the previous chapter, Galapagos is a LightGrid application: it provides a client and two job-processor classes, as well as a large number of other classes that work in conjunction with them. This chapter will introduce these components, describe how to configure and run Galapagos, then finish with details regarding development of new Galapagos components.

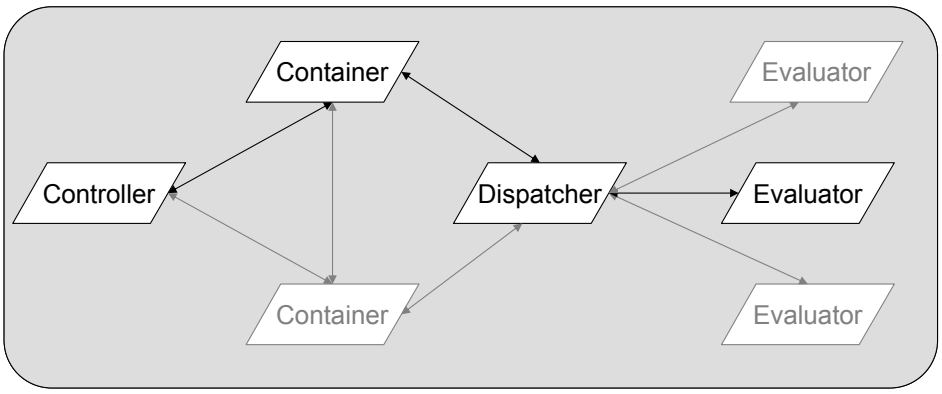
3.1 Components & Terminology

Within the context of Galapagos as a LightGrid application, the client is called a *container* and the basic job-processor is called an *evaluator*. The container process manages the EA population and all operators and is provided as a complete Java command-line executable. The evaluator is where the chromosomes are evaluated according to one or more objective functions, from which the fitness is derived (or the objective values can be used separately in a multi-objective EA). In this way, Galapagos natively implements basic master-slave distribution. The evaluator is provided as an abstract class to be subclassed.

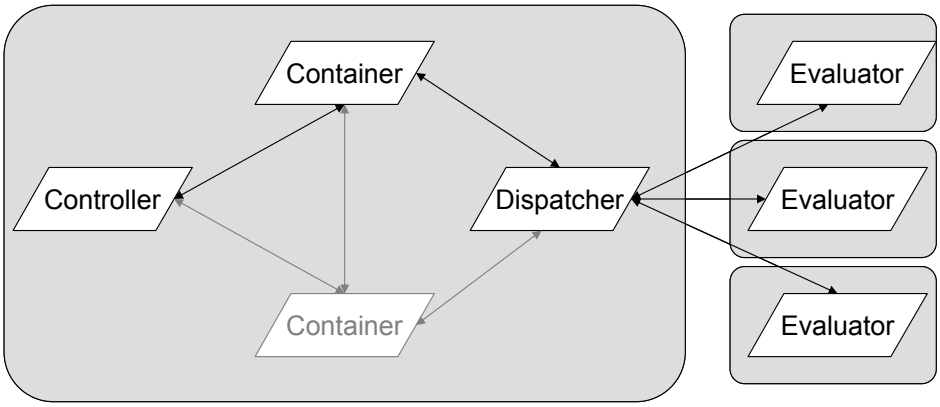
Each Galapagos container can manage multiple populations, as well as migration between them, allowing for parallel EAs. Multiple Galapagos containers can also manage populations, whose chromosomes can migrate to and fro, allowing for distribution of the populations across multiple computers as well. Each container can communicate with its own dispatcher, they could all use the same one, or any permutation in between.

In order to manage multiple independent containers, a fourth executable is provided: the *controller* (the other three being the client/container, the dispatcher and the resource). The controller is sent copies of each population every time it changes and is where control decisions (EA convergence, user communication etc) occur. Galapagos provides an abstract controller class as well as subclasses for abstract CLI (command-line interface) and GUI (graphical user interface) controllers. Finally, fully functional generic CLI and GUI executables are provided (see the Advanced Development section for more information).

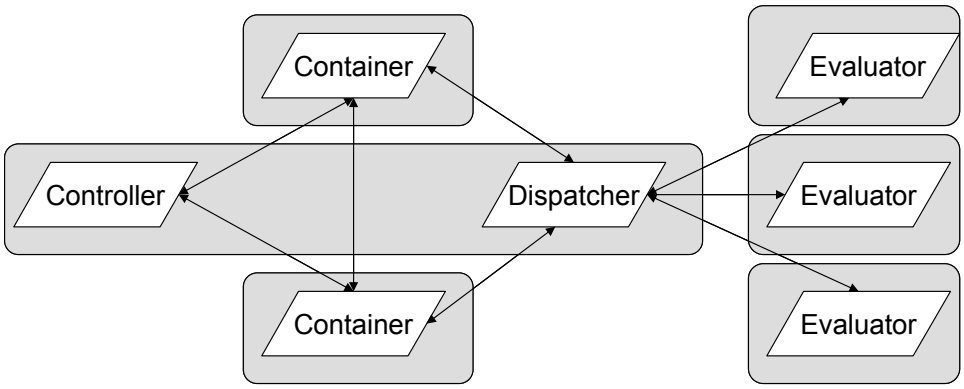
With these four components, Galapagos can be used to implement a huge variety of parallelism and distribution schemes, a few of which are presented below (the computers are the grey rounded boxes):



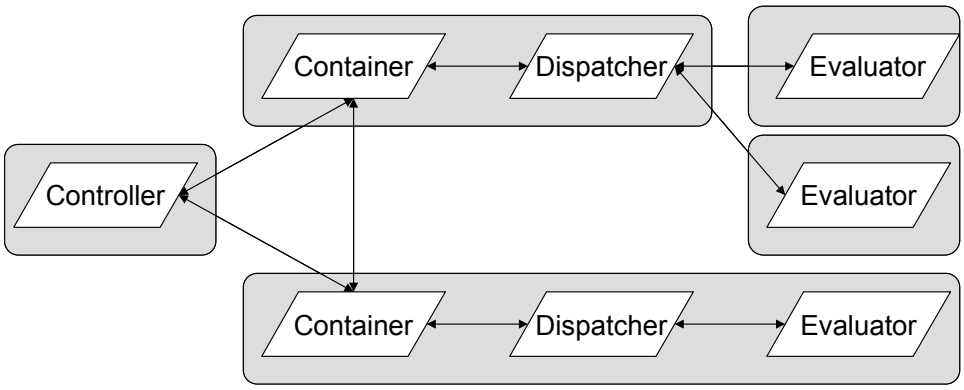
Single-computer mapping



Master/Worker Mapping



Peered Mapping (Controller & Dispatcher collocated)



Possible highly tailored mapping

The second job-processor defined in Galapagos is a generic *local searcher*. Local searchers are used in hybrid EAs and define the other search method(s) which are to be used to support the EA. Local searchers take in chromosomes as inputs and return chromosomes, which presumably are better than the inputs used to seed the local search. With local searchers implemented as job-processors, hybridization in Galapagos is automatically distributed among the available resources, for speed and efficiency.

For reference, the following ports are used by Galapagos, in addition to the ones used by LightGrid:

Port Number	Usage
7074	Container to Container
7075	Controller to Container
7076	Container to Container

More information on Galapagos development is provided in the latter sections of this chapter, but for context, the next section describes the steps of a basic Galapagos run.

3.2 A Basic Galapagos Run

In order to provide context for the rest of this chapter, this section describes the steps a user must go through in order to perform a Galapagos run.

1. Design the desired EA: make decisions regarding the number of computers to use, the population structure of the EA, if hybridization is possible, whether the problem is single- or multi-objective, the nature of the operators desired, etc. The run described here will be the simplest

possible: a non-distributed EA (one container, one resource, thus one dispatcher, and of course only one controller).

2. Write the configuration XML file which defines the above EA (this is explained in the following section).
3. Launch a dispatcher process and a resource process on the computer you have chosen, managing discovery as appropriate (see above chapter). Ensure that the job-processor class in the configuration file for the EA is on the resource's classpath (for more information on launching processes and classpath issues, please refer to the relevant appendix).
4. Launch a container process, ensuring that all of the operators chosen and defined in the configuration file are on the classpath.
5. Launch a controller process (CLI or GUI).
6. Load the configuration file (type or click 'load): the controller will process the configuration file, and pass the relevant portions to the controller, which will then contact the dispatcher and configure itself.
7. Start the run (type or click 'start'): the controller will signal the container to send the first generation to the dispatcher, which will pass them one by one to the resource, which will evaluate the chromosomes, returning the objective function values to the container via the dispatcher. This will continue until stopped (see next step)
8. The EA run will stop either when the run converges (as defined by the termination criteria in the configuration file) or when you stop it manually (type or click 'stop').
9. You now have the option to save the population to disk, save any graphs that were generated etc by using the corresponding commands or buttons. You may then reset the controller (type or click 'reset') to prepare it for another run (by returning to step 6) or you may shut down all four processes.

3.3 Configuration Files

The Galapagos run described in the previous section was largely dominated by the configuration file which defines the EA to be run down to the smallest detail. This file is loaded by the controller, which passes the relevant portions to each container to be used, telling them how many populations to create, how big to make them, which operators to use etc. The configuration file is also how the

Galapagos layer manages discovery: all controller, container and dispatcher hostnames are contained in this file.

The configuration file is an XML (eXtensible Markup Language) file, which is simply a text file whose contents must adhere to certain simple rules. XML is a way of 'marking up' or delimiting data within a file. The following could be the contents of very simple XML file:

```
<?xml version="1.0" ?>
<A>
   <B>b-data</B>
   <C>c-data</C>
   <D>
      <E>e-data</E>
   </D>
</A>
```

In the above, each token of the format "<token>" is known as a *tag*. A pair of *open-* and *close-tags* ("<token>data</token>") as well as the data or other tag-pairs contained within is known as an *element* or *node*. Clearly, as shown above, elements can be nested, with the tree-representation of the above being:

- A
    - B: b-data
    - C: c-data
    - D
        - E: e-data

The Galapagos configuration file must contain certain elements in order to be valid. The required elements are listed in a tree on the next page.

- parameters
    - controllerhostname
    - chromosome
        - numobjectives
        - genome
            - name
            - [class-specific parameters]
        - fitness
            - name
            - [class-specific parameters]
    - evaluator
        - name
        - [class-specific parameters]
    - terminator
        - name
        - [class-specific parameters]
    - container
        - containerid
        - containerhostname
        - dispatcherhostname
        - population
            - populationid
            - populationsize
            - generationsize
            - sendatonce
            - priority
            - initializer
                - name
                - [class-specific parameters]
            - assembler
                - name
                - [class-specific parameters]
            - generator
                - name
                - [class-specific parameters]
            - migrator
                - name
                - [class-specific parameters]
            - hybridizer
                - name
                - [class-specific parameters]
        - population
        - population, etc
    - container
    - container, etc.

Each of these is described in detail in the following pages. “a/b” refers to the “b” element within the “a” element. All of the elements listed here are children of the top-level “parameter” node.

controllerhostname: the hostname of the machine the controller will be running on, in a format such that the containers are able to contact it, given the network setup (FQDN is best, IP address may also work).

chromosome: all of the data within this element is used to configure the chromosome for this EA. This is how a user specifies whether to use a real-coded or binary-coded representation etc as well as how the objectives vector is used to create a fitness value. (see rest of this chapter for info on existing classes or developing your own)

chromosome/numobjectives: the number of objective values returned by the evaluator.

chromosome/genome: all of the data within this element is used to configure the genome for this EA. (see rest of this chapter for info on existing classes or developing your own)

chromosome/genome/name: the fully qualified class name of the genome class to be used for this EA. This class must subclass the ca.utoronto.civ.its.galapagos.templates.Genome class.

chromosome/fitness: all of the data within this element is used to configure how the objectives vector components scale and/or combine to form a scalar fitness value to be maximized

chromosome/fitness/name: the fully qualified class name of the fitness class to be used for this EA. This class must subclass the ca.utoronto.civ.its.galapagos.templates.Fitness class.

evaluator: all of the data within this element is used to configure the evaluator job-processor class. (see rest of this chapter for info on existing classes or developing your own)

evaluator/name: the fully qualified class name of the evaluator class to be used as the basis of the problem definition for this EA. This class must subclass the ca.utoronto.civ.its.galapagos.templates.Evaluator class. *Note: the class chosen must be compatible with the genome chosen.* (see rest of this chapter for info on existing classes or developing your own)

terminator: all of the data within this element is used to configure the component which defines when the EA run is to stop. (see rest of this chapter for info on existing classes or developing your own)

terminator/name: the fully qualified class name of the terminator class to be used. This class must subclass the ca.utoronto.civ.its.galapagos.templates.Terminator class.

container: all of the data within this element is used to configure the corresponding container (as determined by the container/containerhostname element).

container/containerid: must be a unique string, not shared with another container. This string can be used elsewhere in configuration files to refer to this container and is also used internally within Galapagos.

container/containerhostname: the hostname of the machine this container will be running on. Again, this hostname must be such that the controller can see it given the network setup in operation.

container/dispatcherhostname: the hostname of the machine this container's dispatcher is running on. This could be the same dispatcher as the one used by other containers, if desired.

container/population: all of the data within this element is used to configure a particular population managed by a particular container.

container/population/populationid: must be a unique string, not shared with another population. This string can be used elsewhere in configuration files to refer to this population and is also used internally within Galapagos.

container/population/populationsize: the number of chromosomes in this population which is to be maintained throughout the EA run.

container/population/generationsize: the number of chromosomes generated at each generation for this population.

container/population/sendatonce: experimental parameter, keep set to generationsize unless the dispatcher has memory problems, at which point it may be reduced, but no lower than 1.

container/population/priority: the priority the dispatcher assigns to jobs from this population, usually set to 1 for all populations, but could be changed to favour some populations in job assignment to resources.

container/population/initializer: all of the data within this element is used to configure the component which defines how to initialize the population. *Note: the*

class chosen must be compatible with the genome chosen. (see rest of this chapter for info on existing classes or developing your own)

container/population/initializer/name: the fully qualified class name of the initializer class to be used. This class must subclass the ca.utoronto.civ.its.galapagos.templates.Iinitializer class.

container/population/assembler: all of the data within this element is used to configure the component which defines how to assemble a generation into the current population to obtain the next state of the population. *Note: the class chosen must be compatible with the genome chosen.* (see rest of this chapter for info on existing classes or developing your own)

container/population/assembler/name: the fully qualified class name of the assembler class to be used. This class must subclass the ca.utoronto.civ.its.galapagos.templates.assembler class.

container/population/generator: all of the data within this element is used to configure the component which defines to generate new chromosomes at each generation. *Note: the class chosen must be compatible with the genome chosen.* (see rest of this chapter for info on existing classes or developing your own)

container/population/generator/name: the fully qualified class name of the generator class to be used. This class must subclass the ca.utoronto.civ.its.galapagos.templates.Generator class.

container/population/migrator: all of the data within this element is used to configure the component which defines if and how this population handles migration in parallel EA setups. *Note: the class chosen must be compatible with the genome chosen.* (see rest of this chapter for info on existing classes or developing your own)

container/population/migrator/name: the fully qualified class name of the migrator class to be used. This class must subclass the ca.utoronto.civ.its.galapagos.templates.Migrator class.

container/population/hybridizer: all of the data within this element is used to configure the component which defines if and how this population uses a local searcher to hybridize the EA. *Note: the class chosen must be compatible with the genome chosen.* (see rest of this chapter for info on existing classes or developing your own)

container/population/hybridizer/name: the fully qualified class name of the hybridizer class to be used. This class must subclass the ca.utoronto.civ.its.galapagos.templates.Hybridizer class.

3.4 Development

As outlined in the previous section, the Galapagos configuration file specifies what sort of EA is to be run by specifying a particular set of basic parameters (genome, fitness, evaluator, terminator) and operators (initializer, assembler, generator, migrator, hybridizer) as particular Java classes which are then dynamically loaded at run-time. Each of these classes must subclass the corresponding abstract class provided in the Galapagos framework. In addition to the above classes, Galapagos provides a number of other useful abstract class templates which can be used in conjunction with the required ones to develop re-usable operators, to ease mixing and matching EA configurations. Included with Galapagos is a small library of already-implemented, commonly used operators which can be used right away (ComponentLib, see the relevant appendix for details). The JavaDocs will also be useful in understanding how these templates are to be used to develop new Galapagos components.

All of the abstract classes listed below are in the ca.utoronto.civ.its.galapagos.templates.* package.

3.4.1 Core Classes

Genome: defines the number and datatype of genes as well as accessors for them. Genome also defines file output formats and other such practical features.

Fitness: defines how the objectives vector values are combined into a scalar fitness value. Galapagos Evaluators (below) return a vector of values, which must be converted to a fitness value, the Chromosome class does this. Galapagos will always maximize fitness, so the Chromosome class should be used to scale or invert objective function values into a fitness.

Evaluator: defines how to evaluate a chromosome based on one or more objective functions. Constraints on the EA can be implemented in this component by evaluating chromosomes as 'not feasible' or invalid.

Terminator: defines when the EA run must stop (based on time, number of generations, statistical condition etc.)

LocalSearcher: defines how to perform a local search within the solution space, starting at a given seed chromosome (point in that space).

3.4.2 Operators

Initializer: defines how to generate the initial population for a given EA run. Depending upon the problem, this may be using some sort of seed or simply

randomly generated. Also can generate 'replacement' chromosomes in case any in the initial population are not feasible points.

Assembler: defines how to update the population given a set of evaluated chromosome from the latest generation. The most common ways to do this usually involve some sort of selection operation, also used in other common EA tasks, so this is another templated operator.

Selector: defines how to choose a given number of chromosomes from a given pool.

Generator: defines how to create new chromosomes from the current population. Selectors usually used here to pick parents, which are then recombined into children using recombination operators, then mutated with mutation operators, both of which are templated in Galapagos. Constraints on the EA can be implemented in this component.

Recombiner: defines how to generate children from parents. Constraints on the EA can be implemented in this component.

Mutator: defines how a chromosome is to be mutated. The preferred way to do this in Galapagos, but by no means the only way, is to use the Mutator to define which genes are to be mutated, but to use another operator (GeneMutator, see below) to define how they are to be mutated.

GeneMutator: defines how a given gene is to be mutated.

Migrator: defines if, when and how to perform migrations from one deme to another. The timing operator requirement is shared with the hybridizer (below) so a template exists for this. The target demes for a given migration is defined in a Topology operator. The rest of the operations handled by Migrators are usually passed off to Selectors and Assemblers.

Epoch: defines a given length of time, based on real time or EA conditions (number of generations, statistical measures etc).

Topology: defines if and how many chromosomes should migrate to a given population from a given population.

Hybridizer: defines if and how to use local search job-processors to hybridize an EA. Hybridizers usually rely on Epoch, Selector and Assembler operators.

3.4.3 Building an Application

The first step to building a Galapagos application is to make basic design decisions regarding your evolutionary algorithm:

- What objective(s) will your EA search with? Is this a single- or multi-objective EA? How will these combine to form a fitness?
- How will you represent your solutions (what kind of chromosome will you use)? Real-coded, binary-coded, some mix thereof?
- What constraints are there on the feasible space?
- What sort of operators will you use for recombination and mutation, selection etc?
- What sort of population structure will this EA have?
- Are you building a hybrid EA?
- How many computers are available & what does the network look like?

Once these questions have been answered, classes must either be built or found which match these design criteria. Most EAs will use traditional genome encodings, so this developing a custom Genome class will likely not be an issue. Each EA solves a different problem, and so will require that a custom Evaluator class be written, one that matches the chosen Genome.

Depending on the constraints and operators chosen, custom Operator classes may have to be written. The same goes for the population structure chosen, if a new Topography class is needed. If hybridization is desired based on problem-domain heuristics or other external knowledge, a LocalSearcher job-processor must be written.

Once all the required classes are available, the configuration file must be written. After this, the application may be run with one of the provided controller executables, or more development may take place to enhance the generic user input/output functions of the controller, as described in the following section.

3.5 Controller Development

As outlined in the introduction to the Galapagos components, the Galapagos controller comes in a few different forms. At the root, there is an abstract controller class which provides hooks for subclasses to capture various EA events. This class is subclassed by two more abstract classes: a CLI wrapper and a GUI wrapper, which wrap the hooks, while providing basic user I/O features. These are then further subclassed to form two generic (empty event hooks) controller executables: a CLI and a GUI.

Any application-specific data logging or processing or user I/O can be implemented by subclassing one of the interface wrappers and capturing the

events. The GUI wrapper provides facilities for adding Swing tabs to the tab-pane, so a developer can easily add controls or graphs to the GUI.

The ‘headless’ abstract controller class could be used internally by some other application as well, by subclassing it directly.

Appendix A: Launching the Executables & Dealing with Classpaths

This appendix explains how to go about launching the LightGrid and Galapagos executables, and is intended for those not familiar with running Java code. This brief lesson is not meant as a complete how-to regarding Java, and assumes that the Java Virtual Machine (JVM, also known as the Java Runtime Environment or JRE) is installed on the machine in question.

When Java source code is compiled, the result is a class file. If the class contains a 'main' method, it is executable, and can be run from the command line by typing:

java <fully qualified classname>

if this class file is on the classpath. The classpath is a set of directories that the Java Virtual Machine will look through to find any classes it needs to run. By default, the current directory is on the classpath, so typing

'java ca.utoronto.civ.its.galapagos.container.Container'

will cause the JVM to try to execute the class Container.class in the following subdirectory: ca/utoronto/civ/its/galapagos/container/.

Another way to run executables is to save the class file into a Java archive (jar file). Jar files must be explicitly placed on the classpath, as follows:

'java –classpath thejar.jar ca.utoronto.civ.its.container.Container'

Now this is somewhat tedious, so it is common practice to store these long-winded commands in files which can be called easily, so creating a text file called 'container.bat' on Windows systems or 'container.sh' on Unix or Linux systems and storing the command in there is much more practical. (Note: in '.sh' files, the first line must be '#!/bin/sh') When this is complete and the permissions on the resulting file are set such that it may be executed, simply calling that file from the command line (or double-clicking on it in a window) will launch the executable, if the classpath is set correctly.

By default, Galapagos comes in the format of 5 jar files: lightgrid.jar, galapagos.jar, componentlib.jar, jcommon-0.9.0.jar and jfreechart-0.9.15.jar (the last two being open-source graphing libraries from http://www.jfree.org/). Galapagos also comes with '.bat' and '.sh' files ready to execute.

Things become more complicated when Galapagos is to be used with code written by you, the developer. Your code must be on the classpath of the relevant executable (usually, your code is an Evaluator, and thus is used by the resource). You are free to do this via the class or jar method, as appropriate.

Appendix B: Executable Commands

LightGrid Dispatcher

resetall: this will clear all job queues and issue reset commands to all resources

killresources: this will instruct all resources to terminate

quit: this will cleanly exit the dispatcher

LightGrid Resource

Resources have one purpose: to process jobs. They are meant to be remotely controlled through the dispatcher, and can be cleanly terminated from there. They do not respond to user commands. An idle resource can be contacted at any time by any dispatcher that knows its hostname.

Galapagos Container

reset: this will clear out all populations and issue a client reset to the dispatcher, clearing out this client's job queues and issuing a reset to resources processing this client's jobs. Containers may also be remotely reset via the controller.

quit: this will cleanly terminate the process, but the container must be idle (reset) before this will work.

Galapagos Generic Controllers

The controller command sequence is essentially: load, start, stop, (optional savepop etc), reset. The generic controllers save a log file to the /logs directory in a self-explanatory XML file which also saves a copy of the configuration file, for easy reference.

load: this will load/parse a given configuration file and send the relevant portions to the specified containers. At this point the containers are no longer idle, but waiting for a start command

start: issues the start command to all containers, prompting them to send their first batches of jobs to their dispatchers.

stop: this stops the EA run by issuing a reset command to all containers (see above section for more info). This will automatically happen when the terminator indicates that the run is over. At this point, the controller still holds all data from the run just ended, and the user may save any data or graphs they wish

savepop: this will dump the population to a directory of the user's choice. This command can be issued anytime between start and reset, as may any other graph-saving commands in the GUI.

reset: this command will return the controller to a pre-load-command condition, ready for another run.

Galapagos Generic GUI Controller

The generic GUI controller displays graphs of useful EA status data. This data is logged to an XML file, as with the generic CLI controller, but right-clicking on the graph allows a user to 'save data' to a CSV file for further plotting/manipulation. Right-clicking on a graph also exposes some other useful formatting/printing/saving commands.

Appendix C: ComponentLib Reference

One of the jar files provided with Galapagos is componentlib.jar which contains a number of common genetic operators based on the templates described in chapter 3 of this manual. The following pages will list all of them, describe what they do and what sort of EA they might be used for, as well as how to configure them within the configuration file (i.e. what to put in the [class specific parameters] placeholder in the configuration files section of chapter 3).

These can also be used as inspiration or as a guide to developing your own custom operators, or implementing other common operators. If you do and wish to share them with others, please email the developer of this package, Nicolas Kruchten, at nicolas@kruchten.com. Please refer to the JavaDoc to understand the method signatures for each of the templates.

RealGenome

extends Genome

```
<genome>
   <name>ca.utoronto.civ.its.galapagos.genomes.RealGenome</name>
   <numgenes>[number of genes]</numgenes>
   <geneminvalues>
      <range>
         <to>[extent of range]</to>
         <value>[min value over this range]</value>
      </range>
      <range>
         <to>[extent of range]</to>
         <value>[min value over this range]</value>
      </range>
      [etc]
   </geneminvalues>
   <genemaxvalues>
      <range>
         <to>[extent of range]</to>
         <value>[max value over this range]</value>
      </range>
      <range>
         <to>[extent of range]</to>
         <value>[max value over this range]</value>
      </range>
      [etc]
   </genemaxvalues>
</genome>
```

This implementation of Genome is for real-coded chromosomes, where each gene is a real number. Each gene is constrained to lie within some range of values, specified above. The ‘range’ elements start at zero, and sets the boundary value for every gene up to the one specified in the ‘to’ element. The final ‘range’ element must thus go ‘to’ the value of ‘numgenes’.

WeightedObjectiveSumFitness

extends Fitness

```
<fitness>
   <name>ca.utoronto.civ.its.galapagos.fitnesses.WeightedObjectiveSumFitness</name>
   <weights>
      <weight>[weight of first objective]</weight>
      <weight>[weight of second objective]</weight>
      [etc]
   </weights>
</fitness>
```

This Fitness is composed of a weighted sum of the objective values, specified simply above.

SingleObjectiveFitness

extends Fitness

```
<fitness>
   <name>ca.utoronto.civ.its.galapagos.fitnesses.SingleObjectiveFitness</name>
</fitness>
```

This Fitness is simply the first objective value, unmodified.

StdTerminator

extends Terminator

```
<terminator>
   <name>ca.utoronto.civ.its.galapagos.terminators.StdTerminator</name>
   <minstd>[the standard deviation which defines convergence]</minstd>
</terminator>
```

This terminator will halt the EA when the standard deviation of the fitness across the global population drops below some value ‘minstd’.

RandomInitializer

extends Initializer

```
<initializer>
   <name>ca.utoronto.civ.its.galapagos.operators.initializers.RandomInitializers</name>
</initializer>
```

This basic initializer returns Chromosomes with RealGenomes. Their genes are uniformly randomly chosen such that they lie between the min and max for each gene.

CannonicalAssembler

extends Assembler

```
<assembler>
   <name>ca.utoronto.civ.its.galapagos.operators.assemblers.CannonicalAssembler</name>
   <elitenumber>[number of top chromosomes to keep]</elitenumber>
</assembler>
```

This assembler can be used to build so-called 'cannonical' EAs, in which each successive generation replaces the original population. The 'elitenumber' best chromosomes from the initial population are reinserted into the resulting set, if they are better than any of the new chromosomes, displacing the least fit.

If there are less chromosomes in the incoming set, the difference is treated as elite chromosomes. If there are more chromosomes in the incoming set, only the best are let into the population.

CrowdingAssembler

extends Assembler

```
<assembler>
   <name>ca.utoronto.civ.its.galapagos.operators.assemblers.CrowdingAssembler</name>
   <elitenumber>[number of top chromosomes to keep]</elitenumber>
   <selector>
      <name>[name of Selector class to use]</name>
      [class-specific parameters]
   </selector>
</assembler>
```

This assembler builds a temporary population from the current population (minus the elites) and the incoming set, then applies a the chosen Selector to it. If they have a higher fitness than any in the resulting set, the top 'elitenumber' chromosomes from the initial population are then reinserted, displacing those with the smallest fitness.

IntermediatePopulationCrowdingAssembler

extends Assembler

```
<assembler>
   <name>ca.utoronto.civ.its.galapagos.operators.assemblers.IntermediatePopulationCrowdin
gAssembler</name>
   <elitenumber>[number of top chromosomes to keep]</elitenumber>
   <popselector>
      <name>[name of Selector class to use]</name>
      [class-specific parameters]
   </popselector>
   <newselector>
      <name>[name of Selector class to use]</name>
      [class-specific parameters]
   </newselector>
   <selector>
      <name>[name of Selector class to use]</name>
      [class-specific parameters]
   </selector>
</assembler>
```

This assembler generalizes the crowding assembler to allow canonical, full-replacement assembler behaviour and other intermediate behaviours.

It operates identically to the CrowdingAssembler, except that rather than combining the entire existing population (minus elites) and incoming set, it uses a subset of each (using 'popselector' and 'newselector', respectively). By selecting 0 from the existing population and the entire incoming set, canonical behaviour is possible, and by selecting the entire existing population as well as the entire incoming set, crowding behaviour is possible.

StandardGenerator

extends Generator

```
<generator>
   <name>ca.utoronto.civ.its.galapagos.operators.generators.StandardGenerator</name>
   <recombiner>
      <name>[name of Recombiner class to use]</name>
      [class-specific parameters]
   </recombiner>
   <mutator>
      <name>[name of Mutator class to use]</name>
      [class-specific parameters]
   </mutator>
</generator>
```

This generator is straightforward and should allow almost any type of EA development. The chromosomes are generated via the chosen Recombiner, then the Mutator is applied.

BestSelector

extends Selector

```
<selector>
   <name>ca.utoronto.civ.its.galapagos.operators.selectors.BestSelector</name>
</selector>
```

This selector returns chromosomes starting from the best and working down to the worst.

RandomSelector

extends Selector

```
<selector>
   <name>ca.utoronto.civ.its.galapagos.operators.selectors.RandomSelector</name>
</selector>
```

This selector returns random chromosomes, with a uniform probability for each.

RankBasedSelector

extends Selector

```
<selector>
   <name>ca.utoronto.civ.its.galapagos.operators.selectors.RankBasedSelector</name>
</selector>
```

This selector returns random chromosomes, with a probability distribution defined by:

P[a given chromosome] = (popsize-rank)/(popsize)

RouletteWheelSelector

extends Selector

```
<selector>
   <name>ca.utoronto.civ.its.galapagos.operators.selectors.RouletteWheelSelector</name>
</selector>
```

This selector returns random chromosomes, with a probability distribution defined by:

P[a given chromosome] = (fitness)/(sum of fitnesses across population)

TournamentSelector

extends Selector

```
<selector>
   <name>ca.utoronto.civ.its.galapagos.operators.selectors.TournamentSelector</name>
   <tournamentsize>[number of chromosomes in each tournament]</tournamentsize>
</selector>
```

This selector repeatedly uniformly randomly selects ‘tournamentsize’ chromosomes and returns the best one.

RealAlphaBlendCrossover

extends Recombiner

```
<recombiner>
   <name>ca.utoronto.civ.its.galapagos.operators.recombiners.RealAlphaBlendCrossover</nam
e>
   <alpha>[see formula]</alpha>
   <sendnumber>[1 or 2]</sendnumber>
   <selector>
      <name>[name of Selector class to use]</name>
      [class-specific parameters]
   </selector>
</recombiner>
```

This recombineruses the specified Selector to select two parents, and sets each gene of the child chromosome to:

*(parent1gene * gamma) + (parent2gene *(1-gamma))*
*where gamma = ((1 + (2 * alpha)) * [randomnumber 0-1]) – alpha*

Clearly, two possible complimentary chromosomes can be generated this way, and either one or both can be used, depending on the value of 'sendnumber'.

This recombiner operates on Chromosomes with RealGenomes.

RealBlendRecombiner

extends Recombiner

```
<recombiner>
   <name>ca.utoronto.civ.its.galapagos.operators.recombiners.RealBlendRecombiner</name>
   <selector>
      <name>[name of Selector class to use]</name>
      [class-specific parameters]
   </selector>
</recombiner>
```

This recombiner uses the specified Selector to select two parents, and sets each gene in the child to some value uniformly randomly uniformly distributed between the corresponding gene values in the parents.

This recombiner operates on Chromosomes with RealGenomes.

RealMultiCrossover

extends Recombiner

```
<recombiner>
   <name>ca.utoronto.civ.its.galapagos.operators.recombiners.RealMultiCrossover</name>
   <sendnumber>[1 or 2]</sendnumber>
   <selector>
      <name>[name of Selector class to use]</name>
      [class-specific parameters]
   </selector>
</recombiner>
```

This recombineruses the specified Selector to select two parents, and sets each gene of the child chromosome to the corresponding gene in one or the other of the parents, with equal likelihood.

Clearly, two possible complimentary chromosomes can be generated this way, and either one or both can be used, depending on the value of ‘sendnumber’.

This recombiner operates on Chromosomes with RealGenomes.

RealSimpleCrossover

extends Recombiner

```
<recombiner>
   <name>ca.utoronto.civ.its.galapagos.operators.recombiners.RealSimpleCrossover</name>
   <sendnumber>[1 or 2]</sendnumber>
   <selector>
      <name>[name of Selector class to use]</name>
      [class-specific parameters]
   </selector>
</recombiner>
```

This recombineruses the specified Selector to select two parents, and sets each gene of the child chromosome to the corresponding gene in one of the parents until some random gene, after which it uses the other parent's genes.

Clearly, two possible complimentary chromosomes can be generated this way, and either one or both can be used, depending on the value of 'sendnumber'.

This recombiner operates on Chromosomes with RealGenomes.

AllMutator

extends Mutator

```
<mutator>
   <name>ca.utoronto.civ.its.galapagos.operators.mutators.AllMutator</name>
   <genemutator>
      <name>[name of GeneMutator class to use]</name>
      [class-specific parameters]
   </genemutator>
</mutator>
```

This mutator simply applies the chosen GeneMutator to all genes.

It operates on Chromosomes with RealGenomes, explicit min/max clipping is done.

SingleGeneUniformMutator

extends Mutator

```
<mutator>
   <name>ca.utoronto.civ.its.galapagos.operators.mutators.SingleGeneUniformMutator</name>
   <genemutator>
      <name>[name of GeneMutator class to use]</name>
      [class-specific parameters]
   </genemutator>
</mutator>
```

This mutator simply applies the chosen GeneMutator to a single uniformly randomly selected gene.

This gene mutator operates on Chromosomes with RealGenomes, explicit min/max clipping is done.

StaticMutator

extends Mutator

```
<mutator>
   <name>ca.utoronto.civ.its.galapagos.operators.mutators.StaticMutator</name>
   <genemutator>
      <name>[name of GeneMutator class to use]</name>
      [class-specific parameters]
   </genemutator>
   <mutationprobabilities>
      <range>
         <to>[extent of range]</to>
         <value>[probability of genes in range being mutated]</value>
      </range>
      <range>
         <to>[extent of range]</to>
         <value>[probability of genes in range being mutated]</value>
      </range>
      [etc]
   </mutationprobabilities>
</mutator>
```

This mutator applies the chosen GeneMutator to each gene with the specified probability. The first ‘range’ starts at the first chromosome and applies until the ‘to’ chromosome, the last ‘range’ must apply ‘to’ ‘numgenes’ in the chromosome.

This gene mutator operates on Chromosomes with RealGenomes, explicit min/max clipping is done.

BoundaryGeneMutator

extends GeneMutator

```
<genemutator>
   <name>ca.utoronto.civ.its.galapagos.operators.genemutators.BoundaryGeneMutator</name>
</genemutator>
```

This gene mutator sets the gene to either its minimum or maximum value.

This gene mutator operates on Chromosomes with RealGenomes.

CreepGeneMutator

extends GeneMutator

```
<genemutator>
   <name>ca.utoronto.civ.its.galapagos.operators.genemutators.CreepGeneMutator</name>
   <mutationrange>[range of mutation]</mutationrange>
</genemutator>
```

This gene mutator a uniformly distributed random number of plus or minus ‘mutationrange’ to the gene.

This gene mutator operates on Chromosomes with RealGenomes, no explicit min/max clipping is done.

FitnessDifferenceAdaptiveCreepGeneMutator

extends GeneMutator

```
<genemutator>
   <name>ca.utoronto.civ.its.galapagos.operators.genemutators.FitnessDifferenceAdaptiveCr
eepGeneMutator</name>
   <mutationrange>[range of mutation]</mutationrange>
</genemutator>
```

This gene mutator adds a uniformly distributed random number to the gene between plus or minus (bestfitness-worstfitness)*('mutationrange'). This operator is thus adaptive in that its behaviour changes depending on the state of the population, when all the chromosomes have similar fitnesses, the mutation is small and vice versa.

This gene mutator operates on Chromosomes with RealGenomes, no explicit min/max clipping is done.

FitnessDifferenceAdaptiveGaussianGeneMutator

extends GeneMutator

```
<genemutator>
   <name>ca.utoronto.civ.its.galapagos.operators.genemutators.FitnessDifferenceAdaptiveGa
ussianGeneMutator</name>
   <mutationrange>[std of Gaussian]</mutationrange>
</genemutator>
```

This gene mutator adds a Gaussian distributed random number to the gene with a mean of 0 and a standard deviation of (bestfitness-worstfitness)*('mutationrange'). This operator is thus adaptive in that its behaviour changes depending on the state of the population, when all the chromosomes have similar fitnesses, the mutation is small and vice versa.

This gene mutator operates on Chromosomes with RealGenomes, no explicit min/max clipping is done.

GaussianGeneMutator

extends GeneMutator

```
<genemutator>
   <name>ca.utoronto.civ.its.galapagos.operators.genemutators.GaussianGeneMutator</name>
   <mutationrange>[std of Gaussian to be added]</mutationrange>
</genemutator>
```

This gene mutator adds a Gaussian distributed number to the gene, with a mean of 0 and a standard deviation of 'mutationrange'.

This gene mutator operates on Chromosomes with RealGenomes, no explicit min/max clipping is done.

PercentageGeneMutator

extends GeneMutator

```
<genemutator>
   <name>ca.utoronto.civ.its.galapagos.operators.genemutators.PercentageGeneMutator</name
>
   <mutationrange>[percentage as decimal (ie 0.1 = 10%)]</mutationrange>
</genemutator>
```

This gene mutator adds to the gene:

(genevalue)*('mutationrange')*(uniform random number between -1 and 1)

This gene mutator operates on Chromosomes with RealGenomes, no explicit min/max clipping is done.

RandomGeneMutator

extends GeneMutator

```
<genemutator>
   <name>ca.utoronto.civ.its.galapagos.operators.genemutators.RandomGeneMutator</name>
</genemutator>
```

This gene mutator sets the gene to a random number within the feasible range for that gene, with a uniform distribution.

This gene mutator operates on Chromosomes with RealGenomes.

NGenerationsEpoch

extends Epoch

```
<epoch>
   <name>ca.utoronto.civ.its.galapagos.operators.epochs.NGenerationsEpoch</name>
   <numgenerations>[the number of generations between each epoch]</numgenerations>
</epoch>
```

This epoch will expire every ‘numgenerations’ generations.

NullMigrator

extends Migrator

```
<migrator>
   <name>ca.utoronto.civ.its.galapagos.operators.migrators.NullMigrator</name>
</migrator>
```

This migrator will never cause a migration to occur, meaning it is the one to use when a non-parallel EA (a panmictic EA, or one with a single deme or population) is desired. It has no parameters because it doesn't do anything.

StandardMigrator

extends Migrator

```
<migrator>
   <name>ca.utoronto.civ.its.galapagos.operators.migrators.StandardMigrator</name>
   <assembler>
      <name>[name of Assembler class to use]</name>
      [class-specific parameters]
   </assembler>
   <epoch>
      <name>[name of Epoch class to use]</name>
      [class-specific parameters]
   </epoch>
   <selector>
      <name>[name of Selector class to use]</name>
      [class-specific parameters]
   </selector>
   <topology>
      <name>[name of Topology class to use]</name>
      [class-specific parameters]
   </topology>
</migrator>
```

This standard migrator should be enough to build most basic parallel EAs. Whenever the chosen Epoch expires, the chosen Selector selects migrants to send to each population, the number and destination of which are specified by the Topology. The Assembler is used to determine which, if any, incoming migrant chromosomes, to allow into the population.

StaticTopology

extends Topology

```
<topology>
   <name>ca.utoronto.civ.its.galapagos.operators.topologies.StandardTopology</name>
   <target>
      <populationid>[the populationid of the target]</populationid>
      <nummigrants>[the number of chromosomes to send each epoch]</nummigrants>
   </target>
   <target>
      <populationid>[the populationid of the target]</populationid>
      <nummigrants>[the number of chromosomes to send each epoch]</nummigrants>
   </target>
   [etc]
</topology>
```

This basic topology operator should suffice for most basic parallel EAs. It basically exhaustively lists each migration path from each population, specifying the target and the number of migrants to send there each time migration is called for.

NullHybridizer

extends Hybridizer

```
<hybridizer>
   <name>ca.utoronto.civ.its.galapagos.operators.hybridizer.NullHybridizer</name>
</hybridizer>
```

This hybridizer will never trigger, meaning it is the one to use when a non-hybrid EA is desired. It has no parameters because it doesn't do anything.

StandardHybridizer

extends Hybridizer

```
<hybridizer>
   <name>ca.utoronto.civ.its.galapagos.operators.hybridizers.StandardHybridizer</name>
   <numhybrids>[number of chromosomes to use to seed local searches]</numhybrids>
   <jobprocessor>
      <name>[name of LocalSearcher class to use]</name>
      [class-specific parameters]
   </jobprocessor>
   <selector>
      <name>[name of Selector class to use]</name>
      [class-specific parameters]
   </selector>
   <assembler>
      <name>[name of Assembler class to use]</name>
      [class-specific parameters]
   </assembler>
   <epoch>
      <name>[name of Epoch class to use]</name>
      [class-specific parameters]
   </epoch>
</hybridizer>
```

This hybridizer will probably suit most users' needs. It's pretty straightforward and reuses a few other common operators. Each time the Epoch chosen expires, 'numhybrids' chromosomes will be sent out to the resources via the dispatcher for processing using the LocalSearcher chosen. These chromosomes are selected from the current population using the Selector, and the results of the local searches are put back into the population using the specified Assembler.