ECJ Then and Now

Sean Luke
Department of Computer Science
George Mason University
4400 University Drive MSN 4A5
Fairfax, VA 22314
sean@cs.gmu.edu

ABSTRACT

ECJ is a mature and widely used evolutionary computation library with particular strengths in genetic programming, massive distributed computation, and coevolution. In Fall of 2016 we received a three-year NSF grant to expand ECJ into a toolkit with wide-ranging facilities designed to serve the broader metaheuristics community. This report discusses ECJ's history, capabilities, and architecture, then details our planned extensions and expansions.

CCS CONCEPTS

•Mathematics of computing → Optimization with randomized search heuristics; •Computing methodologies → Genetic algorithms; Genetic programming;

KEYWORDS

Metaheuristics, Libraries, Evolutionary Computation, Implementation

ACM Reference format:
Sean Luke. 2017. ECJ Then and Now. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082467

ECJ is an evolutionary computation system begun in the Fall of 1999, and is one of the oldest major open source toolkits (EO [17] is one year older). In the eighteen years since then, ECJ has become widely used in the field, resulting in many theses, publications, and commercial products. But as ECJ has stayed close to its evolutionary computation roots, other metaheuristics methods have since sprouted up, everything from ant colony optimization to memetic algorithms. With this has seen the arrival of many custom toolkits which have strengths in one or another of these new approaches.

I believe that a strong, unified metaheuristics toolkit will greatly help the field by providing common ground for comparing methods, mixing and matching techniques, and doing research and education with a stable and well-understood implementation. This past year GMU received a (USA) National Science Foundation grant to develop ECJ into this unified toolkit to benefit not just to evolutionary computation but the broader area of black-box stochastic optimization.

In this paper I describe why and how ECJ came to be, assess what about ECJ made it popular with the EC community, describe ECJ's capabilities (and flaws), and then detail how we will improve and extend ECJ in the near future.

1 HISTORY

ECJ's origins are in genetic programming (GP). Prior to ECJ, early genetic programming tools were limited. GP had been strongly influenced by John Koza's *Genetic Programming* [19] and subsequent texts, which included Koza's "Simple Lisp" (or "LittleLisp") genetic programming code. Various early C and C++ reimplementations of Koza's code arrived soon thereafter, the most successful probably being lil-gp [32]. Other major tools included SGPC, DGPC and GPQUICK.

In 1997 I used lil-gp to evolve team soccer softbot behaviors for RoboCup's simulator league, using a beowulf cluster at the University of Maryland [21]. This work required major modifications to lil-gp: distributed evaluation, typed genetic programming, coevolution, and various breeding and selection operators. lil-gp was not designed for this degree of modification, and so after this project I decided to build a new toolkit for future work. Many of ECJ's design decisions can be directly traced to lil-gp, including ECJ's logging facility, independent random number generation, serialization/checkpointing, and heavy use of parameter files.

Though it was meant from the start to be a full-featured evolutionary computation toolkit, ECJ drew no inspiration from early GA and ES libraries. Genetic programming toolkits were usually more complex, as their representations were nontrivial and were evaluated in simulation as if they were software. Such simulations could require multithreaded and even distributed evaluation. ECJ came from this tradition.

ECJ has always been open source, licensed under AFL 3.0, a liberal BSD-style license with a patent release [30]. As I want the code used by as broad an audience as possible, I do not permit viral licenses in the code or its dependencies. Though ECJ has had many extensions and contributions by others, in truth the core code has always been a cathedral-style effort, largely produced by myself and my students.

ECJ managed to go a full decade without a manual, offering only class documentation, tutorials, and some text files. In 2010 at the request of users, and after writing the first edition of *Essentials of Metaheuristics* [22], I finally built a

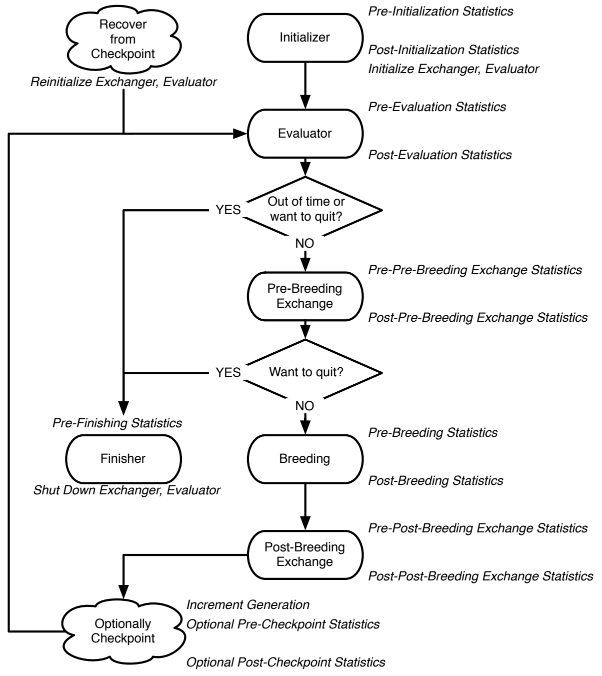
Recover from Checkpoint
Reinitialize Exchanger, Evaluator
Initializer
Pre-Initialization Statistics
Post-Initialization Statistics
Initialize Exchanger, Evaluator
Evaluator
Pre-Evaluation Statistics
Post-Evaluation Statistics
Out of time or want to quit?
YES
NO
Pre-Breeding Exchange
Pre-Pre-Breeding Exchange Statistics
Post-Pre-Breeding Exchange Statistics
Want to quit?
YES
NO
Pre-Finishing Statistics
Finisher
Shut Down Exchanger, Evaluator
Breeding
Pre-Breeding Statistics
Post-Breeding Statistics
Post-Breeding Exchange
Pre-Post-Breeding Exchange Statistics
Post-Post-Breeding Exchange Statistics
Optionally Checkpoint
Increment Generation
Optional Pre-Checkpoint Statistics
Optional Post-Checkpoint Statistics

Figure 1: Generational top-level loop.

287-page manual. In addition to the obvious goal of providing comprehensive documentation of all of ECJ's features, this exercise also resulted in a top-to-bottom reexamination and revision of ECJ's code, parameters, and documentation.

2 WHY HAS ECJ BEEN POPULAR?

ECJ has proven popular. Christian Gagné and Mark Parizeau (authors of OpenBEAGLE [8]) described ECJ as "probably the most popular public EC system coded in Java" [10]. J. J. Merelo *et al* (EO, JEO [1], and Algorithm::Evolutionary [25]) describe ECJ as "probably the most widely-used general-purpose evolutionary computation library" [25].

Publications using ECJ are not required to cite it, but a 2012 hand search from Google Scholar yielded 413 publications which had used, compared against, or otherwise cited ECJ, beyond those from my institution. A more recent search revealed an additional 424 publications since 2013.

I think that there are four reasons why ECJ was successful:

- ECJ showed up at the right time, and rode the wave of Java popularity. As Java progresses from exciting new language to boring enterprise tool, ECJ's strength here may wane.
- ECJ has over time acquired many capabilities, most of which can be used in almost any combination. ECJ has proven good for large, complex projects.
- ECJ could acquire these capabilities because it was over-architected from the start, enabling me and other researchers to easily extend and customize it. ECJ's architecture has, I think, proven to be one of its biggest advantages.

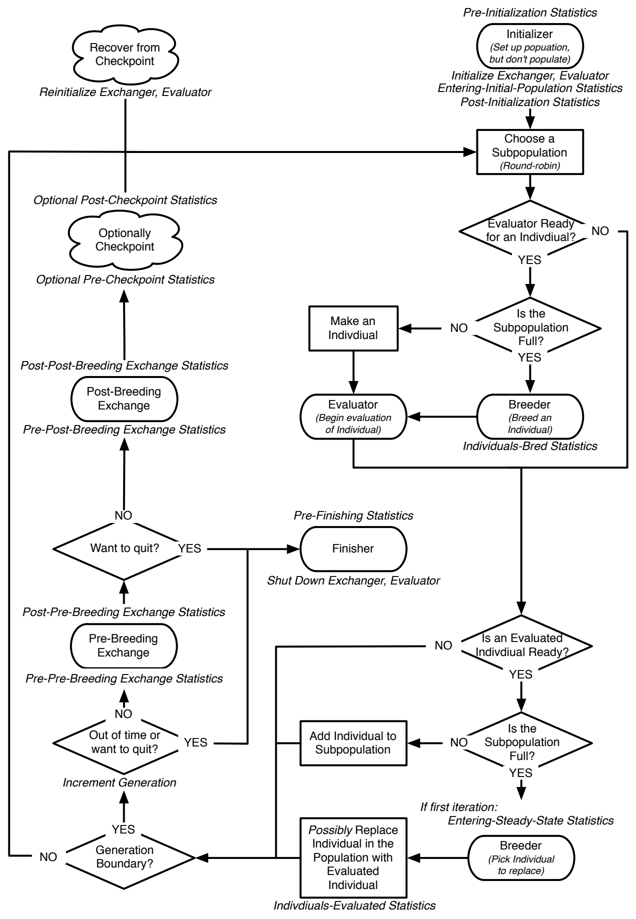
Pre-Initialization Statistics
Initializer (Set up popuation, but don't populate)
Initialize Exchanger, Evaluator
Entering-Initial-Population Statistics
Post-Initialization Statistics
Recover from Checkpoint
Reinitialize Exchanger, Evaluator
Choose a Subpopulation (Round-robin)
Optional Post-Checkpoint Statistics
Optionally Checkpoint
Optional Pre-Checkpoint Statistics
Evaluator Ready for an Indivdiual?
NO
YES
Make an Indivdiual
NO
Is the Subpopulation Full?
YES
Post-Post-Breeding Exchange Statistics
Post-Breeding Exchange
Pre-Post-Breeding Exchange Statistics
Evaluator (Begin evaluation of Individual)
Breeder (Breed an Individual)
Individuals-Bred Statistics
NO
Want to quit?
YES
Pre-Finishing Statistics
Finisher
Shut Down Exchanger, Evaluator
Post-Pre-Breeding Exchange Statistics
Pre-Breeding Exchange
Pre-Pre-Breeding Exchange Statistics
NO
Is an Evaluated Indivdiual Ready?
YES
NO
Out of time or want to quit?
YES
Add Individual to Subpopulation
NO
Is the Subpopulation Full?
YES
Increment Generation
If first iteration: Entering-Steady-State Statistics
YES
NO
Generation Boundary?
Possibly Replace Individual in the Population with Evaluated Individual
Breeder (Pick Individual to replace)
Indivdiuals-Evaluated Statistics

Figure 2: Steady-state and asynchronous distributed evolution top-level loop.

- Perhaps more than anything else, ECJ has had good support. ECJ comes with many tutorials, class documentation, and a very large manual. It has been stable and has had few showstopper bugs. We have supported ECJ via email for two decades.

2.1 Java

ECJ was developed when Java was in its infancy, and it was far from clear that Java would become efficient, popular, or well-supported. But I think that the choice of Java for ECJ proved prescient. Java originally had an undeserved reputation for being slow: but modern Java VMs can often approach C/C++ code in speed. Java's primary bottlenecks are due to its poor libraries and convenience mechanisms (iterators, certain generics). By not using them, ECJ has remained efficient while still providing much of the portability and consistency that are Java's hallmarks. ECJ has heavily emphasized speed over memory efficiency: and so one of ECJ's fundamental problems is its large memory footprint.

ECJ is written in an archaic dialect of Java: when it was being designed, Java was at 1.1. Early versions of Java

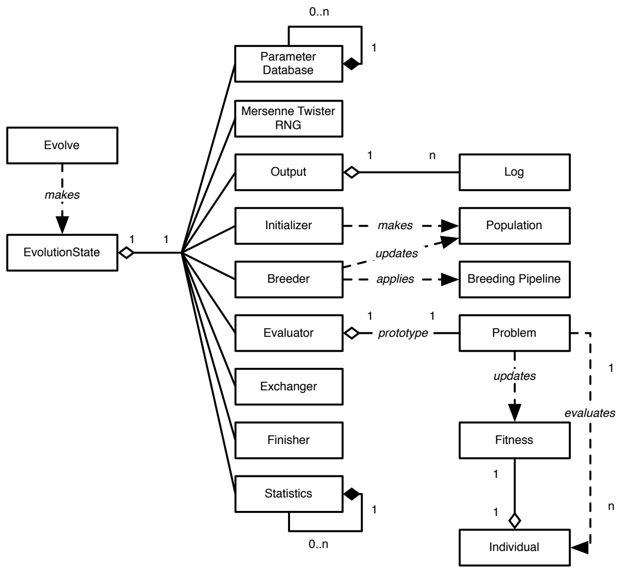

Figure 3: EvolutionState and its "Verb" Singletons.

were missing facilities for logging, good random number generation, sorting, selection from distributions, and so on, which explains why ECJ has its own versions of these in its code. Even now ECJ maintains its own versions of some elements (such as thread pools and logging) because the current standard Java versions of these utilities are poor.

ECJ's Java style is peculiar. Because ECJ is old, and because it is often used on systems (notably Macs) whose Java versions are usually behind the cutting edge, ECJ has always been backward compatible with *very* old versions of Java; indeed right now ECJ is intentionally compatible with Java 1.5. This is why ECJ to this day has no Java annotations, for example. Other modern features are eschewed by ECJ because they are inefficient: for example, ECJ does not use generics in many places because they can incur a dramatic boxing and unboxing penalty. Similarly, ECJ has traditionally used arrays rather than ArrayLists (which until recently have been 5× slower). But as Java has improved, some of these justifications have progressed from sound efficiency considerations to pure cargo cult programming: and so will be jettisoned going forward, as discussed in Section 3.1.

2.2 Capabilities

ECJ was meant to serve my research goals for ten years from its creation and to grow into a full-fledged evolutionary computation toolkit. To this end, ECJ is designed first and foremost to be easily hacked: it is heavily over-architected, with many hooks where you can greatly modify the system. As a result, at present, ECJ has many capabilities, including:

Evolutionary Computation Methods. The genetic algorithm; elitism; the steady-state genetic algorithm; (μ, λ) and $(\mu+\lambda)$; particle swarm optimization; differential evolution; NSGA-II and SPEA2; one-population competitive, two-population competitive, and N-population cooperative coevolution; spatially distributed evolution; CMA-ES; many genetic programming evolution methods; grammatical evolution; Push; many selection, breeding, and parsimony pressure techniques; multithreaded evaluation and breeding; internal and external distributed island models; generational, opportunistic, and asynchronous distributed evaluation (tested on hundreds of thousands of machines); random restarts; and meta-evolution.

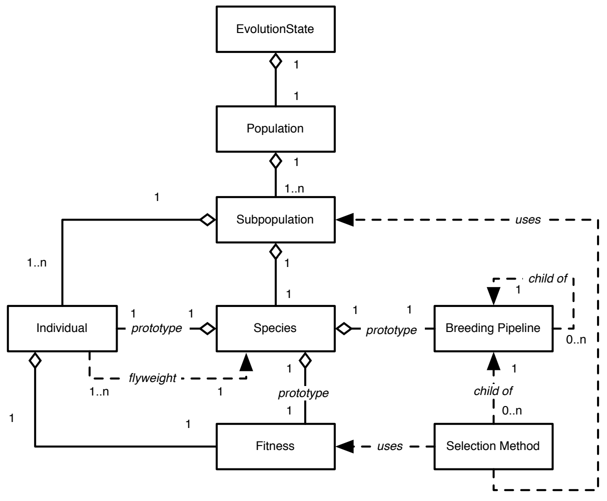

Figure 4: ECJ's "Nouns": Populations, Subpopulations, Species, Individuals, and Fitnesses.

Representations. Fixed-length vectors of various boolean, numerical, and arbitrary data types; strongly-typed tree-style genetic programming; arbitrary-length lists (also used for grammatical evolution); rulesets.

Utilities. Job handling, checkpointing, multithreaded evaluation and breeding, significant statistics and logging facilities, good random number generation and replicability standards, a GUI with charting.

Extensions. ECJ has had many extensions by contributors, including an ALPS/FSALPS package, Cartesian GP, GEP, ECJ output parsers, GPU and CUDA extensions, additional coevolutionary algorithms, and ports to massive distributed facilities (DR-EA-M and Parabon/Frontier).

2.3 Architecture

ECJ's most common top-level evolutionary loops are prosaic; they're shown in Figures 1 and 2. The loop is contained in a singleton *EvolutionState* object as shown in Figure 3. This object shepherds ECJ's primary "noun" (a *Population*) through various "verb" stages: population initialization, evaluation, breeding/resampling, exchange with other ECJ processes, and destruction. These "verbs" are handled by the *Initializer, Evaluator, Breeder, Exchanger,* and *Finisher* respectively. Throughout, the EvolutionState maintains random number generators and calls hooks on *Statistics* objects, which output to *Logs* managed by an *Output* facility. Subclasses of these "verb" singletons implement various metaheuristics.

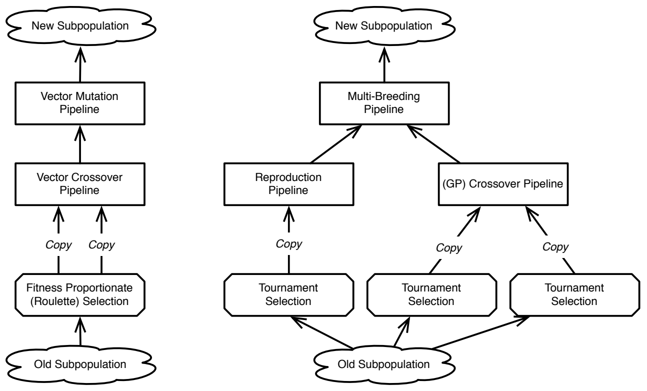

Figure 5: Two Common Breeding Pipelines. At left is a common pipeline for genetic algorithms. At right is a common pipeline for genetic programming.

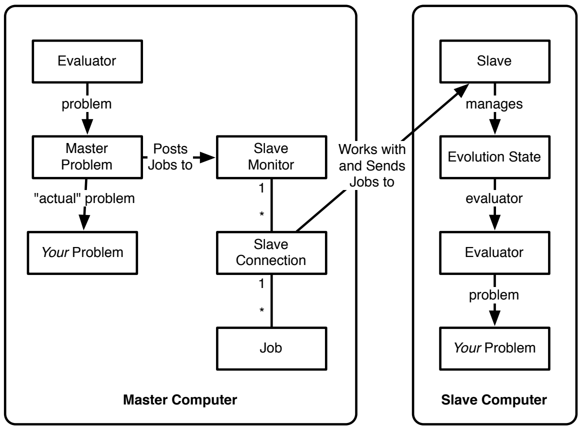

Figure 6: Distributed Evaluation Architecture.

ECJ is serializable, and so can save its state to a checkpoint file, then be moved to a different computer or operating system, and restarted from there. The EvolutionState is also entirely self-contained. Multiple ECJ evolutionary runs can be performed side by side in the same Java Virtual Machine; or ECJ can be started inside another Java application.

Parameters. ECJ's parameter files are certainly its most notorious feature. They dictate the entire structure and operation of an experiment, except (typically) the Problem subclass written by the researcher. Their disadvantage is that parameter constraints cannot be checked by the compiler; parameters are also verbose. Even so, I think this approach has proven wise: you don't need to rebuild the entire system just to change the architecture, which greatly assists scripting.[1]

ECJ's parameters come from a hierarchy of parameter files, or from the command-line. When ECJ is launched, a bootstrap class (usually *Evolve*) loads the parameters into a *ParameterDatabase* and then, using this database, builds the EvolutionState. The EvolutionState then recursively uses the ParameterDatabase to construct and set up all of its subsidiary singleton objects, which in turn set up *their* subsidiary objects from parameters, and so on.

One important side effect of this process is that ECJ contains almost no uses of the `new` keyword! Instead, singleton objects are loaded from the ParameterDatabase via `Class.newInstance()`, and *Prototype* instances are loaded from the same and then later are `clone()`ed to create large numbers of Individuals, Fitness objects, and so on.

Dynamic loading like this has big advantages. ECJ's Meta-EA facility can easily fire up *other* ECJ processes within the same VM, or remotely on other machines, simply by constructing a new parameter database, all from a single class (*MetaProblem*). We used the Meta-EA facility to perform one of the largest EC experiments in the literature, involving over 14.2 million separate evolutionary runs [23].

[1]Indeed, I have found that when metaheuristics toolkits lack runtime parameterization, researchers inevitably build in their own ad-hoc version anyway, thus fulfilling a form of Greenspun's Tenth Rule.

Populations, Initialization, and Breeding. Populations are created via a subclass of Initializer, which normally works by asking a Species to generate an Individual. As shown in Figure 4, a Population contains a set of *Subpopulations*, each of which contains a set of *Individuals*. Subpopulations are evolved independently and potentially asynchronously. This two-level hierarchy makes possible things like coevolution (a subpopulation of sorting networks versus a subpopulation of unsorted vectors, say), or internal island models (subpopulations of migrating Individuals). However it can only be used for one of these at a time, a minor flaw.

An Individual contains a *Fitness* and has some kind of representation. ECJ provides subclasses for many representations ranging from bit-vectors to GP style trees. Fitnesses may also be customized, enabling single- and multiobjective fitness measures, among others.

Resampling is handled by a subclass of Breeder. To do (the default) EC-style generational breeding, the experimenter specifies one or more *Breeding Pipelines* — DAGs of selection methods, mutation procedures, and recombination procedures — which define how an Individual is to be selected, copied, modified, and added to the new population. Pipelines can be cloned and used per-thread in multithreaded breeding. Figure 5 shows two example pipelines.

Other resampling methods, such as CMA-ES, Differential Evolution, or Particle Swarm Optimization, ignore Breeding Pipelines and resample or modify Subpopulations directly.

The Breeding Pipelines, standard initialization procedures, and various features common to a given type of Individual are stored in a *Species* object shared by multiple Individuals via the Flyweight pattern. Each Subpopulation contains Individuals sharing the same Species in common. One weakness in this approach is that while ECJ can have different Species from Subpopulation to Subpopulation, and thus different Breeding Pipelines, it cannot have multiple heterogeneous Initializers, Breeders, or Evaluators in the same process. Thus for example ECJ cannot simultaneously coevolve two Subpopulations, one using a Genetic Algorithm and the other using CMA-ES, without very significant customization.

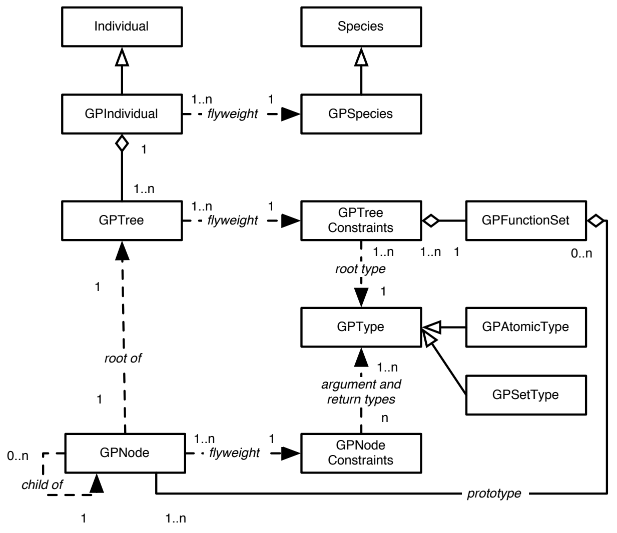

Figure 7: Tree-Style Genetic Programming.

Evaluation and Exchange. The Evaluator makes clones of an experimenter-supplied *Problem* subclass with which to test Individuals in a Subpopulation. The Problem takes an Individual, tests it, and assigns it a fitness. Problem is typically the one place where an experimenter writes in Java.

Individuals may be tested singly or in groups: for example, a cooperative coevolutionary Evaluator may gather some N Individuals from various Subpopulations to be assessed together in a single test. Individuals may also be evaluated with some T tests before being assigned a final Fitness; and Fitnesses can be single or multi-objective.

An Evaluator can also test Individuals in parallel in multiple threads, and can ship Individuals off to remote machines to be tested. ECJ's distributed evaluation support is strong: it has traditional master-slave evaluation, Opportunistic Evolution, and Asynchronous Evolution (a parallel extension of steady-state evolution) built in. Distributed evaluation is achieved by communicating with one or more slave processes, as shown in Figure 6. Each slave is also a full ECJ process, so an experimenter's Problem code need not distinguish between running on a slave and running on the master.

ECJ also supports distributed island models through an *Exchanger* subclass, which migrates Individuals to and from other machines at appropriate times in the evolutionary cycle. Island models may be combined with distributed evaluation.

Genetic Programming and Other Representations. Given ECJ's origins is not surprising that GP is one of its strengths. ECJ implements Koza-style GP using trees of nodes rather than packing trees into arrays as was done in DGPC and lil-gp. This approach is less memory efficient but is much easier to debug and extend. As shown in Figure 7, ECJ's *GPIndividuals* contain forests of *GPTree* objects, each of which holds a tree of *GPNodes.* Just as Individuals have a flyweight relationship with Species, GPNodes share information such as function sets in a common *GPNodeConstraints* and likewise GPTrees share common *GPTreeConstraints.* ECJ supports

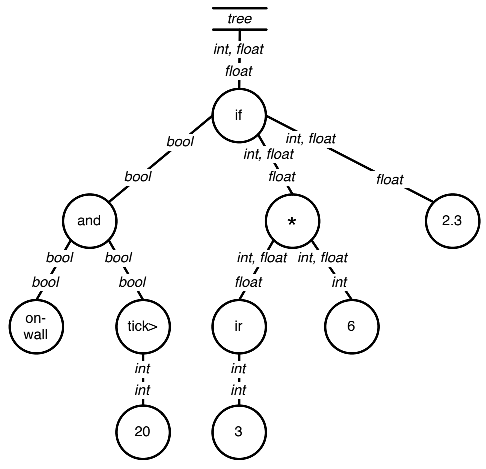

Figure 8: A Strongly-Typed Genetic Programming Tree. Type names shown (*int, float,* etc.) are convenient symbols and need not refer to a data type.

Automatically Defined Functions and Macros, Ephemeral Random Constants, per-tree modification constraints, and many mutation, crossover, and parsimony operations.

ECJ uses set-based Strongly-Typed Genetic Programming. Each node's parent slot and child slots are augmented with a set of types, for example $\{T_a, T_b, T_c\}$. The tree's root slot is also augmented with a set of types. Parent slots may attach to other node's child slots, or to the tree's root slot, if the intersection among their two sets is nonempty. Figure 8 shows an example of typing in action. This approach permits constraints such as subtypes and subclasses, is fast, and is easy to integrate with traditional GP tree generation, crossover, and mutation operators. However, it does not have the full power of polymorphic typing. Consider the if node in Figure 8, which represents the typed expression $\{\mathit{float}\} \leftarrow \mathsf{if}(\{\mathit{bool}\}, \{\mathit{int}, \mathit{float}\}, \{\mathit{int}, \mathit{float}\})$. ECJ cannot constrain the second and third arguments such that they are both *int* or both *float.* For this reason, and because its types are finite, ECJ cannot implement an arbitrary matrix multiplication operator as a GPNode, for example.

ECJ also provides packages to support friends of the GP family, such as Grammatical Evolution (GE), and basic forms of Push. Though it has no package for Linear GP, ECJ's variable-length list representation makes it easy to implement it. Pitt-approach classifier systems may be straightforwardly implemented with ECJ's ruleset package.

3 WHERE IS ECJ GOING?

In 2016 we received a 3-year National Science Foundation grant to extend ECJ into a full-fledged metaheuristics toolkit. Our goal was to create a popular common benchmark metaheuristics facility to serve as a nexus for software development. Such a library would make it easier to compare methods, to build on stable and debugged algorithms, and to teach classes in these areas. We took inspiration from Weka [13], a tool popular in the data mining community. Weka has many general purpose data mining and machine learning methods, is easy for novices to use, serves as an excellent education tool, and has dozens of public extensions. Weka was deemed important enough that in 2005 it received the SIGKDD Data Mining and Knowledge Discovery Service award.

To achieve such a tool one would need to start with an existing library that is stable, efficient, and feature-rich, and which has a strong community behind it. We think that ECJ is best situated for this; two other obvious contenders being EO [17] and OpenBEAGLE [8]. To transform ECJ into this toolkit would require several things:

- Add quality control tests and modern Java usage.
- Add many new metaheuristics approaches.
- Integrate ECJ with Eclipse/NetBeans plugins and improve its GUI to make it easy to use by beginners.
- Integrate ECJ better with statistical analysis tools and add more benchmarks.

A few changes will require some re-architecting of ECJ, but I cannot yet detail plans for a revised architecture, simply because we haven't decided on them yet. Refactoring will occur gradually and on-demand over the next three years as we consider each module in turn.

The list of changes above, and indeed many of the modifications to ECJ over the last five years, were inspired in part by occasional community surveys begin in Summer of 2010.

3.1 Making ECJ More Robust

As discussed earlier, ECJ is written in an old, idiosyncratic Java style, with no annotations, generics, closures and iterators, or many common utility classes. Part of this is intentional: for example, if we used generics, ECJ's vector-representation package could be dramatically simplified, but at a significant speed penalty due to boxing and unboxing. Similar efficiency arguments can be made for other recent Java features, such as iterators. However there are a multiple areas where ECJ could be improved in this regard: many classes could be made generic without efficiency concerns, ArrayLists could replace arrays in many places, and so on. One of our primary goals will be to modernize ECJ's Java style as much as possible while keeping an eye towards efficiency.

Architecture. There are weaknesses in ECJ's architecture which ought to be remedied. Beyond those already discussed, one major orthogonality weakness is ECJ's assumption of a single Breeder, Evaluator, Initializer, and indeed a single EvolutionState. For example, PSO has its own special Breeder; so if one wanted to coevolve a PSO subpopulation and a GA subpopulation, this wouldn't be possible without significant modification. Similarly some representations (like GP) have historically used the Initializer as a source of global storage, meaning that a CMA-ES subpopulation (which uses its own Initializer) and a GP subpopulation would be incompatible.

ECJ also has lots of boilerplate. Dynamic loading from parameter files has advantages, but it also means that ECJ must do runtime checks for compatibility when there are constraints on usage. For example, CMA-ES may only be used with real-valued vector Individuals: and so it must check to verify this. This results in a great deal of redundant code in setup methods which we hope to eliminate or simplify.

Tests. This is a glaring flaw. ECJ was originally built with few tests or automated quality control checks. ECJ has exhibited few serious bugs over the years, but this is no excuse: it badly needs a testing facility. To this end we have recently built a testing harness for ECJ and are building both unit and system tests.

Some kinds of unit and system tests are deterministic: ECJ must produce identical statistics output when refactored, or else a flag is raised. Except in certain parallel programming situations, ECJ is fully replicable, and we can use this fact to help us identify errors in our refactoring of its architecture.

However, what about testing whether a slight change in the software has introduced an error? When results differ from expected results, is this because of a bug (or bug-fix) or is it due to the random number generator? Testing for nondeterministic cases such as this presents a difficult and interesting challenge for stochastic optimization software because of the semi-random nature of the results generated. The obvious approach in this second case is to build tests based on distributions and derived statistics: our plan is to run many combinations of algorithms, representations, and problems for a large number of random number generator seeds. If changes in ECJ cause distribution metrics (mean, variance, etc.) to deviate significantly, this will raise a flag.

3.2 Adding Metaheuristics

ECJ lacks framework support for several major and standard areas of metaheuristics: particularly single-state optimization, combinatorial optimization, optimization by model fitting, and hybrid/memetic architectures. Additionally, ECJ is missing at least one major representation (NEAT). Building this support is not just a matter of adding the relevant algorithms: it may require a some re-architecting of ECJ to retain its orthogonality while incorporating sufficient support for these diverse and (in the case of memetic algorithms) vaguely-defined techniques. Here are some of our plans:

Efficient Single-State Optimization. ECJ has no independent framework for doing single-state optimization: at present such optimization is done by treating the candidate solution as a population of size 1 or 2. This is inefficient, as it comes with the statistical baggage associated with population-based optimization. We have built a new abstract module for single-state optimization, and have implemented several basic algorithms in that framework, notably: stochastic and steepest-ascent Hill-Climbing, and Simulated Annealing [18]. We plan on also including variations on Tabu Search [12].

Combinatorial Optimization. ECJ also has no abstract framework for combinatorial optimization. The primary challenge in building one is that combinatorial optimization tends to require much domain-specific user-customization of the evaluation and resampling mechanisms (as in AntNet [5], for example). It is usually not sufficient to describe the components of a possible solution: one must also describe the different ways by which they may be assembled. This will require revisiting how evaluation is performed in ECJ, breaking it into separate evaluation and solution-construction mechanisms, for example. It will also require us to emphasize low-level representational manipulation much more than ECJ normally affords. We will add GRASP [9] and at least the common Ant System [6] and Ant Colony System [7] versions of ACO, and plan to implement more recent ACO algorithms. ACO and GRASP both dovetail well with the ECJ framework.

Model Fitting. ECJ has a package for CMA-ES: but it does not have an abstract framework for generalizing to other EDAs. We will first implement this framework as an extension of the lightweight framework originally built for CMA-ES. We will then implement two basic univariate methods, PBIL [2] and UMDA [28], and plan to implement at least one additional well-regarded CMA-ES-like method, such as BIPOP-CMA-ES [14] or AMaLGaM IDEA [4]. We have already begun implementing a space-partitioning search metaheuristics package for algorithm families such as DOvS [15].

Hybrid Architectures. There are lots of hybrid architectures, ranging from multi-level optimization methods such as Iterated Local Search (ILS) [3, 20], to techniques which jump back and forth between resampling methods (such as the Learnable Evolution Model (LEM) [26]), to performing sub-optimization within an individual's assessment procedure (the hallmark of so-called *memetic algorithms* [27]) The number of possible hybrid architecture combinations is innumerable, and ECJ cannot implement all of them, but we *can* provide building blocks with which researchers may assemble many common types. In some cases we may be able to take advantage of ECJ's ability to recursively launch self-contained sub-processes within the same Java VM to provide optimization at different levels; but many other scenarios may require serious reconsideration about ECJ's architecture.

Representations. NEAT is the primary major representation which is not present in ECJ and which has a large, established community: we are currently adding it to ECJ.

Multiobjective Metaheuristics. ECJ has a stable multiobjective optimization framework and two of the most popular algorithms (NSGA-II and SPEA2), but multiobjective heuristics have expanded considerably since then. For example, the jMetal multiobjective optimization package sports at least twenty such algorithms [29]. We plan to extend ECJ to include a much larger multiobjective optimization collection.

3.3 Making ECJ Easier to Use

ECJ is neither small nor simple. It has extensive documentation, tutorials, and support, but is still a daunting system. To make it more useful to the broader community, we need to make it easier to use, particularly for educational purposes.

IDEs. We are building programmer support for the Eclipse and NetBeans IDEs. Of particular interest is the development of wizards for setting up ECJ: normally Eclipse and NetBeans employ these to construct class files, and we will do exactly that to help walk the experimenter through the process of building a Problem. However of more interest may be wizards which walk the experimenter through the (sometimes laborious) process of setting one or more parameter files and actually performing a run.

GUI. ECJ has a disused GUI developed around 2005, capable of generating limited charts and graphs, but desperately in need of improvement. To date we haven't done so because our own research needs have not required it: ECJ runs happily on the command line. However to make ECJ accessible to a wider audience requires a serious reconsideration of the GUI, particularly aimed at teaching. We will require a GUI which makes it easy to set up parameters, chart statistics, and launch remote jobs on back-end machines.

3.4 Assisting in Analysis of Results

Statistics Utilities. ECJ has extensive support for adding probes into the optimization run to dump statistics regarding the performance of candidate solutions as they are being tested. However it has no facilities for analyzing these statistics. This is an extremely common need.

To this end we first intend to add options to ECJ to dump its statistics into files designed to be directly entered into *R* [16], either as prepared data or as actual *R* code ready to be interpreted. Second, we may directly integrate some statistical analyses, such as nonparametric, unknown variance T-tests, Bonferroni adjustment, and computation of confidence intervals.

We have traditionally rolled our own code for nearly all of ECJ: but will not do so here. Statistics code is *really easy to get wrong* and is best left to experts: and so we will rely on well-vetted statistical packages to do this work. There are however certain statistical measures special to metaheuristics which we will need to implement ourselves. These include relative quality measures for coevolutionary techniques; generalization methodologies; and hyper-volume for multiobjective optimization.

Benchmarks. ECJ has long had significant involvement in benchmark and standardization efforts: for example, the Genetic Programming Benchmarks working group chose ECJ as its "baseline toolkit" [24, 31]. ECJ has many benchmarks implemented, but we need to implement many more. A lot of common benchmarks are aimed at numerical vector spaces, and these are generally trivial to implement (we have in the past focused on the BBOB Benchmark series [11]). But other benchmarks, particularly for more peripheral techniques such as GP, ACO, or NEAT, can be much more complex as they often involve some sort of simulation. For these, we have to work carefully with the communities in question to identify the (hopefully small) set of benchmarks to focus on.

3.5 Development Plan

In the first year we have focused on NEAT, Single-State Optimization, ACO, certain model-fitting methods, and a test harness. We plan to begin work on hybrid, multiobjective, and additional combinatorial optimization and model-fitting methods in the second year, along with statistics utilities and additional tests. We will focus on tools (IDEs, GUIs) and benchmarks in the final year. Re-architecting of ECJ, and refactoring it with more modern Java, will be gradual and will be spread over all three years. We hope to have at least one major release every six months: but we're already behind. The team consists of myself and two PhD students (Ermo Wei and David Freelan): we would welcome contributions and testing in the ECJ proper or as external packages.

4 CONCLUSION

ECJ was originally built to last ten years, but it is now almost twenty years old. I think it has been a very successful open source toolkit, and now sports a sophisticated architecture and a wide range of tools. ECJ has primarily focused on evolutionary computation, but with under new NSF grant we aim to convert ECJ into a full-scale metaheuristics toolkit of interest to a broad community. This will include significant revisions to ECJ's code and architecture, major new packages and facilities, and example applications and test facilities.

In this report I discussed ECJ's background, its capabilities and architecture (and problems), and our intended future directions in the next three years. Our goals are ambitious but I think that if we can achieve them, ECJ stands a chance at providing a common benchmark tool for the metaheuristics research, education, and application community as a whole.

ACKNOWLEDGMENTS

ECJ is due to grants and gifts from DARPA, NASA, NSF, SAIC, SRA International, General Electric, and the Naval Research Laboratory, most recently NSF CRI grant 1629850.

ECJ was co-developed with Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Rafal Kicinger, Elena Popovici, Keith Sullivan, Joseph Harrison, Jeff Bassett, Robert Hubley, Ankur Desai, Alexander Chircop, Jack Compton, William Haddon, Stephen Donnelly, Beenish Jamil, Joseph Zelibor, Eric Kangas, Faisal Abidi, Houston Mooers, James O'Beirne, AKM Khaled Ahsan Talukder, Sam McKay, James McDermott, Ermo Wei, David Freelan, and Steven Armentrout.

Package contributors include David Oranchak, Alberto Cuesta, Bob Orchard, Ken Webb, Anthony Awuley, Olly Oechsle, Robert Baruch, Denis Robilliard, Virginia Marion-Poty, Cyril Fonlupti, Mehran Maghoumi, Marcin Szubert, and Ben Stabile. Thanks also to our many bug reporters.

Thanks to Ken De Jong, J. J. Merelo, Marc Schoenauer, David Andre, Bill Langdon, Bill Punch, and Doug Zongker for their recollections about early GP and GA implementations. And to Ermo Wei and David Freelan for assisting in the development of this paper.

REFERENCES

[1] M. G. Arenas, et al. A framework for distributed evolutionary algorithms. In *PPSN*, pp. 665–675. 2002.
[2] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In *International Conference on Machine Learning (ICML)*, pp. 38–46. 1995.
[3] J. Baxter. Local optima avoidance in depot location. *Journal of the Operational Research Society*, 32:815–819, 1981.
[4] P. A. N. Bosman, J. Grahl, and D. Thierens. Enhancing the performance of maximum-likelihood Gaussian EDAs using anticipated mean shift. In *PPSN*, pp. 133–134. 2008.
[5] G. Di Caro and M. Dorigo. Antnet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, pp. 317–365, 1998.
[6] M. Dorigo. *Optimization, Learning and Natural Algorithms.* Ph.D. thesis, Politecnico di Milano, Milan, Italy, 1992.
[7] M. Dorigo and T. Stützle. *Ant Colony Optimization.* MIT Press, 2004.
[8] R. Feldt, et al. GP-Beagle: A benchmarking problem repository for the genetic programming community. In *Late Breaking Papers, GECCO*, pp. 90–97. 2000.
[9] T. A. Feo and M. G. C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.
[10] C. Gagné and M. Parizeau. Genericity in evolutionary computation software tools: Principles and case study. *International Journal on Artificial Intelligence Tools*, 15(2):173–194, 2006.
[11] Genetic and Evolutionary Computation Conference. *Black-Box Optimization Benchmarking (BBOB) Workshop*, 2009–2016.
[12] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 5:533–549, 1986.
[13] M. Hall, et al. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.
[14] N. Hansen. Benchmarking a BI-population CMA-ES on the BBOB-2009 function testbed. In *GECCO Companion*, pp. 2389–2396. ACM, 2009.
[15] L. J. Hong, B. L. Nelson, and J. Xu. Discrete optimization via simulation. In M. C. Fu, ed., *Handbook of Simulation Optimization*, chap. 2. 2015.
[16] R. Ihaka and R. Gentleman. The R project for statistical computing. http://www.r-project.org/, 1995.
[17] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving Objects: a general purpose evolutionary computation library. In *Evolution Artificielle (EA)*, pp. 231–242. 2002.
[18] S. Kirkpatrick, C. D. G. Jr., and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
[19] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.
[20] H. Lourenço, O. Martin, and T. Stützle. Iterated local search. In F. Glover and G. Kochenberger, eds., *Handbook of Metaheuristics*, pp. 320–353. Springer, 2003.
[21] S. Luke. Genetic programming produced competitive soccer softbot teams for RoboCup97. In *GP98*, pp. 204–222. 1998.
[22] S. Luke. *Essentials of Metaheuristics.* 2009. Open Text. Available at http://cs.gmu.edu/∼sean/book/metaheuristics/.
[23] S. Luke and A. K. M. Khaled Ahsan Talukder. Is the meta-EA a viable optimization method? In *GECCO*. 2013.
[24] J. McDermott, et al. Genetic programming needs better benchmarks. In *GECCO*, pp. 791–798. 2012.
[25] J. J. Merelo Guervós, P. A. Castillo, and E. Alba. Algorithm::evolutionary, a flexible perl module for evolutionary computation. *Soft Computing*, 14(10):1091–1109, 2010.
[26] R. Michalski. Learnable evolution model: Evolutionary processes guided by machine learning. *Machine Learning*, 38(1–2):9–40, 2000.
[27] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report 158–79, Caltech Concurrent Computation Program, California Institute of Technology, 1989.
[28] H. Mühlenbein. The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5(3):303–346, 1997.
[29] A. J. Nebro, J. J. Durillo, and M. Vergne. Redesigning the jMetal multi-objective optimization framework. In *GECCO Companion*, pp. 1093–1100. 2015.
[30] L. Rosen. Academic Free License 3.0. https://opensource.org/licenses/AFL-3.0, 2005.
[31] D. R. White, et al. Better GP benchmarks: Community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14:3–29, 2013.
[32] D. Zongker and B. Punch. lilgp 1.01 user's manual. Technical report, Michigan State University, USA, 26 March 1996.