ECJ at 20: Toward a General Metaheuristics Toolkit

Eric O. Scott
George Mason University
Washington, DC
escott8@gmu.edu

Sean Luke
George Mason University
Washington, DC
sean@cs.gmu.edu

ABSTRACT

ECJ is now 20 years old. Begun as a genetic programming and evolutionary computation library in Java, it has since established itself as historically one of the most popular EC toolkits worldwide. In 2016 we received a National Science Foundation grant to improve ECJ in many ways with an eye toward making it a useful toolkit not just for EC but for the broader metaheuristics community. This paper is a report on our efforts to this end. We discuss new metaheuristics frameworks and representations added to ECJ and the design challenges that they raise for a general-purpose framework, as well as testing facilities and other support tools. We conclude with our future directions for the library.

CCS CONCEPTS

• **Computing methodologies** → **Search methodologies**; • **Software and its engineering** → **Software libraries and repositories**;

KEYWORDS

Metaheuristics, Evolutionary Algorithms, Ant Colony Optimization, Estimation of Distribution Algorithms

ACM Reference Format:
Eric O. Scott and Sean Luke. 2019. ECJ at 20: Toward a General Metaheuristics Toolkit. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3319619.3326865

1 INTRODUCTION

For 20 years, the ECJ toolkit has provided a unified and widely used framework for doing research and education with evolutionary algorithms, and for solving industrial-strength optimization problems with these methods. Over its 27 releases, ECJ has emphasized a high performance, very orthogonal architecture enabling the combination, recombination, and mutation of different aspects of evolutionary algorithms, has been relatively stable and consistent, and has had good support and documentation.

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326865

ECJ is the subject of a multi-year National Science Foundation (USA) grant to improve the toolkit in response to community feedback [20]. The goal of this grant is to convert ECJ from an evolutionary computation-oriented framework into one which could serve as a unifying nexus for the broader metaheuristics community. This paper reports on our progress so far towards that goal. Specifically we report on new or improved support for single-state optimization, Lexicase selection, Estimation of Distribution Algorithms, Ant Colony Optimization, NEAT, multi-objective optimization, and underpinnings of the system, including a testing harness.

ECJ has always emphasized a consistent, orthogonal architecture for mixing and matching many algorithms and methods; but of course this is not always possible. Some techniques are restricted to specific problem domains, genetic representations, or fitness assessment strategies. Some of the elements we have been adding to ECJ, or will be adding in the future, present these kinds of challenges to ECJ's architectural consistency.

In this paper we will first discuss the added value that general-purpose software frameworks offer the metaheuristics community, and some background on the current software landscape and ECJ's design in particular. In Section 5 we will then present the features that we have added to ECJ recently, emphasizing some of the special challenges that arise naturally in certain areas — such as code reusability in Ant Colony Optimization, or the problem of creating a test harness for highly stochastic software. The paper then ends with a discussion of future directions (Section 6), and some of the problems we are looking at tackling in ECJ's future.

2 WHY A COMMON TOOLKIT?

One consistent criticism of ECJ, and of other similar toolkits (like EO [17]), is why one should bother to learn an external toolkit at all, instead of just rolling one's own code to do experimental research.

Our response is that a unifying framework provides a number of major advantages to the community. First, while a simple (μ, λ) ES is not particularly hard to write by hand, other techniques (GP and NEAT are good examples) can be very complex. Provided that it is sufficiently easy to modify and hack — and ECJ strives to make this so — a common and well-vetted framework can reduce the potential for software errors in one's research experimental code. Second, a common framework makes it much easier to compare and contrast techniques, to introduce and popularize new approaches, and to take advantage of the products of the research coding community ecosystem. Third, given a sufficiently orthogonal framework, one gets many follow-on features for free. For instance, a researcher writing code targeting ECJ can automatically take advantage of ECJ's considerable massively distributed evaluation, island model, and asynchronous evolution facilities. Fourth, a common toolkit makes things simpler for educators who want to introduce a variety of techniques to students in a short period of time.

ECJ has another important trick up its sleeve which we think is increasingly important to good research methodology: it was designed from the start to provide not only high-quality but *replicable* or even *duplicable* experiments given the initial parameters. This stems from several factors, including ECJ's consistent low-level facilities (notably its random number generator), and the fact that it targets (intentionally primitive) Java.

Our field is heavily empirical and emphasizes random trials and statistical argument, and such research fields have increasingly come under fire for replication failure. Often researchers do not provide code to back up claims, or if they do, the code is convoluted or esoteric to the point being impossible to understand. As a field, it is imperative to provide replicable, consistent, open, and legible code to support one's research results. ECJ's replicability, and the common codebase provided by its unified framework, promise to help in this regard.

3 BACKGROUND

ECJ has long served as one of the most-used frameworks for evolutionary algorithm development, but the metaheuristics software landscape is considerably more populated today than when ECJ was first introduced. Today, most leading programming languages sport at least one mature, general-purpose EA framework [11, 17], and a variety of younger frameworks routinely crop up to challenge their hegemony and to experiment with novel API designs. For Java developers, examples of newer frameworks that leverage different design philosophies or newer language features include JCLEC [32] and Jenetics.[1]

The way that programmers write and interact with their code has also changed significantly since ECJ was introduced in 1999. Java, while still a venerable industry standard, has gone through a complete boom-bust cycle in terms of industry excitement, and Python has become the new *lingua franca* of artificial intelligence programming. Some EA software maintainers have coped with this by porting their systems to Python in full. Benitez-Hidalgo et al., for example, recently took this leap with the jMetal framework [5]. Python-native packages such as DEAP [11], are increasing in popularity in no small part because of the rapidness and ease with which new algorithm prototypes can be assembled. Moreover, special-purpose modules like TPOT exploit Python's standard ecosystem of scientific tools to offer out-of-the-box EA solutions to specific data science problems (such as optimizing machine learning algorithms) [27].

We recognize the benefits that the Python ecosystem has brought, and in particular SciPy and NumPy, but Java toolkits also offer important advantages: Java has enabled ECJ's high performance, ease of massive distribution, and interoperability with the considerable Java ecosystem at large. In this way we hope ECJ strikes a "happy medium" along the performance to ease-of-use continuum.

4 ARCHITECTURE OVERVIEW

ECJ has historically supported many EA methods and representations, including most "classic" EA architectures, coevolutionary facilities, DE and PSO, multi-objective optimization, spatial EAs, many vector-based and rule-based representations, island models, massively distributed evaluation, meta-EAs, and an extensive genetic programming package. To put later discussion in context, it is worth providing a brief overview of the architecture that ECJ has used to support this diversity. For more detailed discussion, see [19] and [20].

An ECJ experiment, in the form of one or more metaheuristics runs, is contained entirely within a single top-level subclass, *EvolutionState*. This allows ECJ to serialize the entire state of an experiment to a file, move it to another machine, and then restart it as if nothing had happened. It also allows ECJ to run many experiments in parallel, or to straightforwardly perform a meta-EA by performing runs inside other ECJ experiments.

EvolutionState contains or defines two broad categories of objects which together form most of a run. First there are the *nouns*, that is, stateful objects. Specifically, a *Population* contains one or more *Subpopulations*, which each contain one or more *Individuals*. These in turn contain (in addition to their genomes) some kind of *Fitness*. Individuals' genomes may themselves contain additional nouns: for example, tree-based GP individuals contain GPTrees and GPNodes. The division of a Population into Subpopulations allows ECJ to use multiple Subpopulations as an internal island model, or as separate populations for competitive or cooperative coevolution.[2]

Some nouns share features in common: for example, many Individuals may have the same exact parameters which define them. To reduce its memory footprint, ECJ uses the Flyweight design pattern: groups of Individuals all share a common storage object, a subclass of *Species*. This is independent of the grouping class (Subpopulation). Consider a Population with three Subpopulations. Subpopulations 0 and 1 might hold tree-based Individuals which all share a single *GPSpecies* object, while Subpopulation 2 might hold vector Individuals sharing a different *FloatVectorSpecies* object. Flyweight patterns are also used with certain other nouns.

To do the actual optimization, ECJ relies on various *verb* objects. The top-level verb is the EvolutionState subclass itself, which defines the optimization loop broadly written. ECJ provides default EvolutionState subclasses for generational, steady-state, or single-state (hill-climbing etc.) optimization methods. EvolutionState contains more verbs to flesh out these details: various *Initializers* and *Finishers* to initialize or clean up the Population, *Breeders* to produce a new Population from an old one, *Evaluators* to assess the fitness of Individuals, *Exchangers* to talk to other islands either internally or over a network, and a *Statistics* object to output results.

In most EA scenarios, an ECJ Breeder can parallelize the process of producing new individuals by assigning one thread each to a *Breeding Pipeline*. This is an assembly line (a directed acyclic graph) of selection objects and modification objects (crossover, mutation, etc.). ECJ's Breeding Pipelines are highly flexible; indeed, to implement all of Simulated Annealing, ECJ simply relies on a custom set of Breeding Pipeline operators. Similarly, the Evaluator can parallelize the evaluation process through multiple *Problem* objects: one per thread. A Problem subclass defines the fitness assessment procedure for one or a group of Individuals. The Evaluator can also distribute the evaluation process among many machines by

[1] http://jenetics.io/

[2] Note that because there is only one subdivision, this means that it is awkward for ECJ to do internal island models and coevolution *at the same time*. This is an example of an orthogonality failure in ECJ's structure.

replacing the Problem with a special *MasterProblem*, which ships Individuals off to be evaluated via Problems managed by *Slaves* on remote machines.

Essentially all of these objects (all nouns and verbs, Breeding Pipeline objects, Problems) are customizable through subclassing or parameter modification. These objects and subclasses, their parameters, and indeed the entire object graph, are defined by a *Parameter Database* loaded from parameters on the command line or specified in text files. ECJ objects are generally not constructed in the Java-standard way, that is, via the "new" keyword and a constructor method. Instead, they are first built at runtime via Java's reflection facility, and then each is passed the Parameter Database as input, via a special *setup(...)* method, to construct itself. In the process of construction, objects ask the Parameter Database to load other objects specified in the database and call *setup(...)* recursively on them. The parameter database defines all the classes, parameters, and the structure of the entire ECJ experiment, and so it is common to build an entire ECJ experiment without writing *any* Java code. More often, the only Java code the experimenter must write is the class which defines his domain-specific assessment procedure.

5 RECENT FEATURES

The ECJ NSF grant [20] proposed a variety of changes to the system in response to community feedback. At this point we have implemented a significant number of these changes, plus others. In this section we discuss changes made to ECJ during this period.

5.1 Single-State Optimization

This is the term we use for methods in which a single individual is updated and maintained at a time. Techniques in this vein include hill-climbing variants, simulated annealing, tabu-search, and iterated local search. In the past, ECJ could do implementations of hill-climbing with or without replacement by performing $(1, \lambda)$, $(1 + \lambda)$, or of course $(1 + 1)$. This required use of ECJ's evolution strategies toolkit, and consequently all of the overhead of ECJ's generation-based population facility and statistics output.

ECJ features a dedicated single-state optimization framework complete with its own specialized Breeder and EvolutionState subclasses designed to significantly speed up the performance of these algorithms. The package contains examples for several hill-climbing variants and for simulated annealing.

5.2 Estimation of Distribution Algorithms

ECJ now sports four Estimation of Distribution Algorithms: PBIL [2], CMA-ES, iAMaLGaM-ID𝔼A [6], and DOvS; and so its package structure has been made more abstract to accommodate them.

At first glance it would seem that EDAs pose a small challenge to ECJ, as most have no "population" at all during breeding, and so would have no use for ECJ's Breeding Pipelines. In fact this is not problematic: it simply means that each such EDA must provide its own custom Breeder which takes an existing Population, revises the EDA's internal model, and then resamples a new Population from this model. Everything else can use standard ECJ facilities, including massively distributed evaluation.

An EDA does need to maintain model parameters: these are stored in a custom Species object. Indeed, as Species objects can produce new random individuals, they hold nearly the entirety of an EDA's resampling procedure: in each case, the EDA's Breeder is largely syntactic sugar.

5.3 Ant Colony Optimization

All of the metaheuristics historically supported by ECJ have relied on operations that work with complete candidate solutions: whole solutions are mutated, mated, or sampled from a distribution, and only whole solutions are evaluated for their quality. The beauty of this class of metaheuristics is that they are easy to decouple from the specifics of a particular application domain. Code for the evolutionary population model, performance metrics, selection and variation, architectures for parallelization, etc., can be reused across myriad applications and recombined in new ways with little to no modification. The majority of the time, the only custom code a user needs to write for a new application is the fitness function.

Algorithms based on *constructive* heuristics, by contrast, pose interesting design challenges for a general-purpose framework. Constructive methods work with partial solutions by adding to them component-by-component, and they often rely on lower-level, problem-specific heuristic information and neighborhood structure at every step of this process. This makes it difficult to describe the details of an Ant Colony Optimization (ACO) algorithm, for example, independently of the structure of the specific problem it is being applied to. Of the many dozens of ACO implementations publicly accessible on GitHub at the time of writing, in fact, the vast majority are hard-coded to operate on Traveling Salesmen Problems, and most of the remainder are hard-coded for some other domain (such as a particular data mining task, etc.). One exception is the Java Ant Colony Optimization Framework (JACOF),[3] which reuses code across algorithms that solve TSP, knapsack, quadratic assignment, and next release problem instances. JACOF turns out to be the exception that proves the rule, however: it still makes the strong assumption that every problem's structure can be modeled as a graph, and that solutions are constructed by traversing (and laying down pheromones along) edges that connect pairs of nodes on a graph structure. This assumption, which comes out of ACO's strong tradition as a TSP solver, is not well-suited to say, knapsack tasks — where it is individual *components*, rather than pairwise links between them, that have heuristic value.

Our objective in writing an ACO framework within ECJ is to provide a framework that is as generally useful as possible, which performs well, and which plays well with ECJ's existing parallelization and distribution facilities. To achieve this within a system that was originally built for population-based methods, we follow a high-level design that resembles an EDA architecture: we again create a custom *Ant Breeder* implementation that takes a Population at each step, updates its internal model (now a pheromone table), and then resamples a new Population using the updated pheromone concentrations. The basic similarity between our ACO and EDA implementations is no accident — it has long been observed that these two algorithm families have fundamental structural commonalities, wherein ACO's table of pheromones act much like the parameters of a probability distribution [9, p. 57]. In both cases, individuals in the Population are made up of whole solutions (i.e. the solutions

[3] https://github.com/thiagodnf/jacof

that are constructed by ACO's "ants"). This Population object is compatible with ECJ's evaluation system, which permits fitness evaluation across multiple cores or a distributed network via a master-slave model.

A graphical overview of our experimental ACO architecture is shown in Figure 1. These components are ultimately governed by the aforementioned Ant Breeder object. Most members of the ACO algorithm family can be defined by implementing concrete versions of the *Update Rule* and *Construction Rule* classes. ECJ offers an *Ant System Update Rule* implementation, for example, that implements the well-known ANT-CYCLE, ANT-DENSITY, and ANT-QUANTITY pheromone deposition strategies.

The intent is that users will select the algorithm components that they desire and then, at a minimum, define problem-specific attributes of their application in a *Constructive Problem Form* object. A Constructive Problem Form defines a fitness function for the application (as usual), but it also defines additional kinds of information that are required by constructive heuristics: a pool of *Component* objects that solutions may be built from, a neighborhood function and further constraints that define the set of Components that are permissible to add to any given partial solution, and a predicate function that indicates whether a solution is "complete" (as opposed to partial). Depending on the application, Components may be complex objects with many attributes — but at a minimum, every component has a *cost( )* attribute which conveys its heuristic value.

On a knapsack problem, for example, the Components in play are items that can be added to the knapsack — defined by a size attribute and a *cost( )* attribute. A knapsack Problem implementation would define a fitness function to maximize the total values of the items in a solution, a neighborhood function that allows any item to be added to the sack at any time so long as it doesn't cause the solution to exceed to the total sack capacity, and an *isCompleteSolution( )* predicate that returns true when no more items can fit in the sack. Most applications will proceed along similar lines, but it will sometimes be necessary to also augment the representation of whole or partial solutions with some additional auxiliary information. Implementing a TSP problem, for example, may require extending the *Constructive Individual* representation to include an attribute that remembers, say, the most recently visited city in a partial solution.

All this problem-specific information is exploited by a Construction Rule, which is responsible for executing "ants" — i.e. constructing whole solutions out of basic Components. ECJ offers a *Simple Construction Rule* that should be suitable for most purposes, which applies a *Component Selector* strategy to incrementally build solutions. For example, ECJ's *Proportionate Component Selector* implements the classic probabilistic transition function used by the Ant System algorithm:

$$p_{ij} = \begin{cases} \frac{\tau_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{u\in\mathcal{N}_i}\tau_{iu}^{\alpha}\eta_{iu}^{\beta}} & \text{if } j \in \mathcal{N}_i \\ 0 & \text{if } j \notin \mathcal{N}_i \end{cases} \quad (1)$$

This textbook-style equation brings together problem-specific *cost( )* and neighborhood information from the Constructive Problem Form (η_{ij} and $\mathcal{N}_i$, respectively) with data from the Pheromone Table (τ_{ij}) to define the probability that a Component represented by the ordered pair (i, j) is added to a solution.

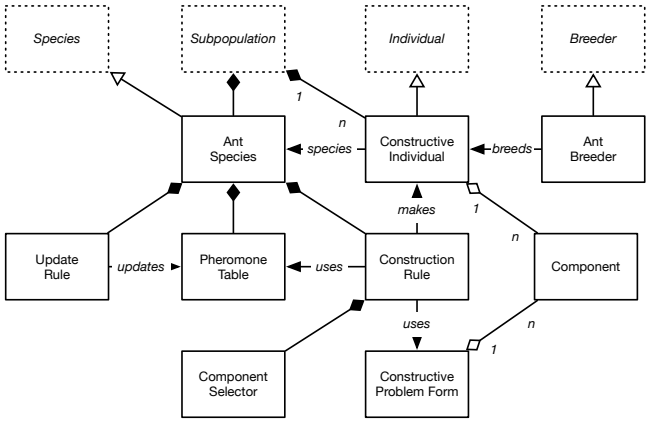

Figure 1: UML class diagram of the ECJ's basic ACO objects. Standard ECJ objects are in dashed boxes.

Altogether, while constructive heuristics do take considerably more problem-specific investment to get off the ground, this framework seems to do a good job of decomposing domain-specific implementations from more reusable, meta-level components. We reached out to a number of researchers who have published papers on ACO algorithms within the past three years, and our informal survey suggests that the classic three methods — Ant System, Ant Colony System, and Max-Min Ant System [7, 8, 30] — are still among the most important and current methodologies in the field. To date we have implemented Ant System, and we aim to implement the other two soon, tweaking the framework's design as necessary to maximize the flexibility of the architecture.

5.4 NEAT

NeuroEvolution of Augmenting Topologies, or *NEAT*, is a popular method for developing basic graph structures for neural networks and other applications [29]. ECJ has a basic implementation of NEAT, but not its sibling, HyperNEAT.

The basic NEAT genotype consists of two variable-length vectors of "genes": *node genes* and *connection genes* (which define edges between nodes) respectively. During fitness assessment, a graph structure is assembled from the instructions encoded in these genes and is then evaluated.

Figure 2 shows the basic structure of ECJ's NEAT classes. ECJ implements NEAT individuals by borrowing from ECJ's *vector* package, which contains Individuals and Species for a variety of traditional vector genotypes, such as vectors of integers, boolean, floats, and so on. One particular Individual, *Gene Vector Individual*, holds a vector of arbitrary objects which subclass from the abstract class *Gene*. This Individual is used when more per-gene flexibility is needed than simple numbers. NEAT subclasses Gene Vector Individual to form *NEATIndividual*, and uses the superclass's original vector to store connection genes (as *NEATGene*), then adds an additional vector to store node genes (as *NEATNode*). An appropriate Species, *NEATSpecies*, handles many of the sundry NEAT parameters required by the Individuals.

NEAT explicitly requires a particular form of speciation to encourage diversity in its individuals. In NEAT this is handled using

a special object called (not surprisingly) a "species." This is called a *NEATSubspecies* in ECJ, it already uses the term "species" in a different way. NEAT also guarantees diversity among its connection genes by assigning each an *innovation number*. ECJ combines this number, plus certain other factors of the connection used to determine similarity, in an object called a *NEATInnovation*.

NEAT comes with an elaborate breeding procedure which is difficult to orthogonalize with classic breeding methods. At present, ECJ provides a *NEATBreeder*, which does some minor bookkeeping, then generates individuals via NEATSpecies (which does the heavy lifting). We hope to work to eliminate the need for NEATBreeder in the future, but it may prove difficult.

5.5 Lexicase Selection

ECJ now offers a Lexicase selection operator for use in problems where an Individual's fitness can be decomposed into multiple components [14]. In the past, ECJ had used a variety of ad-hoc techniques to assemble a fitness value out of multiple trials. This need has surfaced in Meta-EAs, coevolution, and certain GP test problems, for example. To develop Lexicase we have developed a *trials* attribute as part of the Fitness object; trials can themselves have subtrials, and so on. We are now working to convert our former ad-hoc approaches to all use *trials* instead.

5.6 Multiobjective Optimization

We have begun work on adding additional multi-objective optimization algorithms to ECJ. To this end, we have added NSGA-III to the existing NSGA-II and SPEA2. We plan to add more algorithms still. ECJ's multi-objective optimization facility also has its own Statistics subclass: we have augmented this class to compute and export multi-objective hypervolume statistics.

NSGA-III does not differ in major ways from NSGA-II: it largely sports an improved sparsity facility. However, it is worth mentioning that the process of making it consistent with NSGA-II, and making both of them more consistent with ECJ's elitism facility, required a tedious refactoring process which introduced a number of bugs identified by our user community.

5.7 Dependency Management

While much of AI research has seen a trend toward making experiments more reproducible in recent years, some authors continue to float the idea that we suffer from a general "reproducibility crisis" [15, 16]. Even outside the scientific community, the importance of reproducibility is increasingly becoming recognized as an essential component of effective software deployment. Ensuring that software is run with the specific stack of dependencies it was constructed for makes up a big part of that goal [10].

ECJ has historically instructed users to manually install a handful of dependencies into their Java `CLASSPATH` environment variable before compiling the software with GNU Make — not an unusual request for Java veterans, but a somewhat tedious task for new students. We now offer Apache Maven as a build tool in addition to GNU Make in an effort to make this process more streamlined. Maven automatically downloads project dependencies from established repositories — namely the Maven Central Repository — and ensures that the correct versions of dependencies are configured for your individual application. Much like the "virtual environments" that are popular among Python developers, Maven helps bolster one aspect of reproducibility by ensuring that correct libraries are available at compile time, and that we don't fall into versioning conflicts with other application stacks installed on the same system. Dependency management of this kind is not a complete solution to reproducibility by itself (as we will discuss below), and it introduces a strong reliance on access to external package repositories over the Internet, but it offers one step in the right direction.

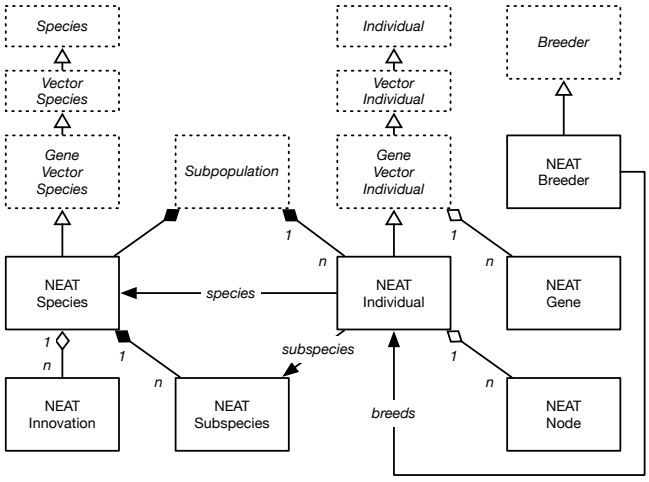

Figure 2: UML class diagram of ECJ's basic NEAT objects. Standard ECJ objects are in dashed boxes.

5.8 Test Harness

ECJ was originally developed with little to no automated testing facility. This oversight grew into a glaring weakness as automated testing became established as an industry norm in the early 2000s. Testing methodology is often overlooked by the scientific community as a whole, with the result that code may easily be "riddled with tiny errors that do not cause the program to break down, but may drastically change the scientific results that it spits out" [24]. While ECJ has been fortunate enough to exhibit few major bugs over the years, due diligence is called for, and a test harness will be necessary for ECJ to expand its feature set, integrate third-party code contributions, and to continue to be of value to the community. Adding a test harness to a large, pre-existing software project can be challenging, and our efforts here are ongoing. We have approached this from several directions, but two in particular: unit testing and a system test approach.

5.8.1 Unit Tests. First, we are writing unit tests for new functionality, and also when we modify old code or otherwise revisit it to check for bugs. This is our first and arguably best line of defense against faults in metaheuristics software. It seems that unit testing occupies an especially pivotal position, moreover, in contexts where complex numerical calculations are the norm. Many of the errors that most often creep into scientific work involve silent mistakes — an EA with a bug won't always crash outright or give any obvious indication to the user that something is amiss. Such

mistakes in other fields have occasionally led to high-profile work being retracted [26]. Moreover, because evolutionary algorithms are nonlinear dynamical systems, it is not typically possible to characterize and verify their overall expected behavior at a high level (i.e. at the level of system tests). The only way to ensure that a complex algorithm is working correctly is often to break it down into its subroutines (data structures, selection operators, breeding loops, etc.) and to verify these individually on example data — which is to say, to write a unit test suite.

The ubiquity of stochastic behavior in EA components, however, often poses an impediment to unit testing. Compared to other software domains (and even other sub-fields of artificial intelligence), evolutionary algorithms present an unusually high frequency of methods that make use of pseudo-random numbers to generate complex output distributions. These stochastic procedures then tend to be chained together into even more complex pipelines. While a great deal of research has investigated stochastic methods for *generating* test cases for software systems [12, 28, 31, 33], as well as how to handle so-called "flaky tests" which sometimes fail probabilistically [23], the problem of testing stochastic software proper has received relatively little attention from the software engineering research community.

We have adopted the straightforward approach of using statistical distribution tests (such as Pearson's χ^2 test) in our unit tests where necessary. For example, to test our implementation of Lexicase selection [14], we run the selection operator 1000 times on a test population, count the number of times that each individual was selected, and then apply a χ^2 test to determine whether the distribution of counts deviates significantly from the theoretically expected distribution.

Statistical tests of this kind are costly (the unit under test must be executed hundreds of times to ensure sufficient statistical power), and they are flaky: type I and type II errors may occur. To minimize false alarms, we have opted to use relatively low p values as our threshold for detecting test failures (ex. $p > 0.01$ or even lower). This has served us well to date, but as the number of these flaky tests grows, the chance of at least one test exhibiting a type I error (i.e. a false test failure) increases rapidly. EA projects with high test coverage and many stochastic units of code may benefit from using more advanced techniques for detecting true test failures [4].

To date, our unit testing efforts have reached only about 10% statement coverage (9% branch coverage) — see Figure 3. While statement coverage is far from a perfect measure of test suite quality [1], it is a valuable rough indicator, and we hope to considerably increase our coverage ratio as this work continues.

5.8.2 System Tests. While many of the most pernicious errors in scientific software fail silently and are difficult to identify, plenty of bugs *do* cause systems to crash, and these easily sneak into unexercised code. While our unit test coverage is still getting off the ground, then, we've turned to system-level tests as a quick and effective way to help ensure that changes that we introduce into one part of ECJ don't inadvertently cause some applications to completely fail. Our system test harness is very simple: we take every complete parameter file from ECJ's 39 built-in example applications and execute it for exactly two generations. The test is successful if and only if this two-generation experiment completes

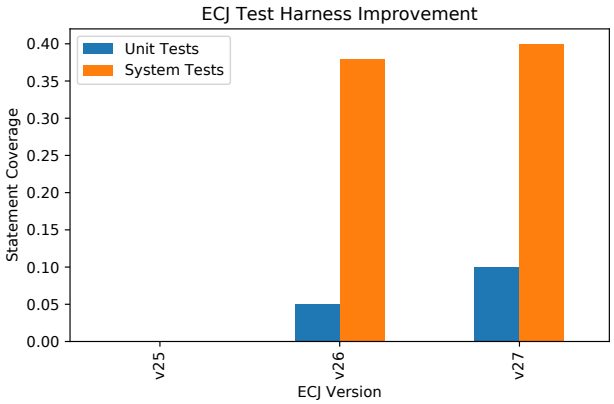

Figure 3: ECJ's test coverage has improved incrementally over the past two years.

successfully (without crashing or throwing an exception). These simple system tests are surprisingly effective at helping catch major bugs that might otherwise go unnoticed for months until a user observes that a particular application is no longer functioning. The examples alone exercise approximately 40% of the code base, giving us surprisingly comprehensive coverage for minimal test construction effort. This knowledge will incentivize us to construct new examples going forward to improve our system test coverage.

5.9 Parameter Macro Expansion

ECJ is infamous for its parameter files. These files free the experimenter from having to define (and compile) experiments as Java code, which in turn allows a high degree of flexibility in scripting ECJ experiments. However this results in a *lot* of parameters. ECJ's parameters are also organized in a namespace hierarchy of the form *foo.bar.baz... = ...* , resulting in tedious parameter declarations like *pop.subpop.0.species.construction-rule.component-selector.alpha = 1.0*. ECJ has a number of basic facilities designed to reduce the tedium and wrist strain of repetitive, long parameter strings. However, we have recently introduced a new parameter macro expansion facility to dramatically reduce this work.

Specifically, a experimenter can introduce an *alias macro parameter*, such as *pop.subpop.0.species.alias = foo* which permits her to rewrite the above example as *foo.construction-rule.component-selector.alpha*; and similarly any other related parameters. A related macro parameter enables wildcards: *pop.subpop.n.default = bar* would permit *bar.species.construction-rule.component-selector.alpha* regardless of the the value of *n*. These expansion rules are applied recursively, allowing for dramatic reductions.

Apart from this macro facility, we considered moving to a JSON file format to offer a more user-friendly representation of hierarchical parameters, and developed a prototype of the same. We found, however, that JSON representations of deep parameter hierarchies tend to be difficult to read and considerably less compact than ECJ's existing parameter language. This and other complications — such as JSON's lack of support for comments — persuaded us against making JSON configurations a standard part of ECJ's interface.

ECJ's parameter database facility is independent of the rest of the code, and can (and has) been used for many unrelated open-source projects, such as MASON [21]: we hope these changes will be of broad benefit.

6 DISCUSSION AND FUTURE WORK

There are certain portions of the proposed work which have not yet been completed, and are worth discussion here, along with other future directions in which we hope to move the framework.

6.1 Hybrid Metaheuristics

Hybrid metaheuristics, sometimes called *memetic algorithms*, encompass a broad array of approaches to hybridization of different optimization strategies. One classic approach is to perform local improvement during fitness evaluation via hill-climbing, gradient ascent, etc. Another approach is to switch between different methods depending on generation: for example, the Learnable Evolution Model (LEM) switches back and forth between a classic genetic algorithm and a technique more resembling an estimation of distribution algorithm [25]. Still another approach involves a meta-optimizer of some sort, such as a meta-EA, or iterated local search (ILS) [18].

There are still more approaches beyond this, and their combinations are many. ECJ cannot directly support all of these techniques, but we hope to provide hooks for many of them, including the three discussed above. To effect many of these techniques we can take advantage of ECJ's self-contained nature to provide EAs within EAs. We note that meta-EAs are already well-supported in ECJ, and have been stress-tested in an experiment with over 14.2 million separate evolutionary runs [22]. However, running ECJ runs within ECJ runs is an inefficient way to do ILS, LEM, or local improvement: these will require dedicated methods and possibly significant restructuring of ECJ's core architecture.

6.2 User Environment

We also plan on improving ECJ's user support. ECJ has historically been a command-line tool with a limited GUI. Our immediate goal is to improve this in four ways. First we will provide integration with Eclipse in the form of wizards to walk the experimenter through the task of setting up common metaheuristics methods and their related parameters, ultimately resulting in a set of parameter files and a skeleton for a Problem class to fill out. Second, we intend to significantly revise the GUI to provide a fuller set of automatically-generated publication-quality charts and graphs, and to improve the GUI's job-handling facilities to make large runs easier to perform.

Third, we will add facilities to dump statistics to files that can be easily parsed by or run in *R*, and add various statistical analyses special to metaheuristics and machine learning, including coevolutionary relative-quality measures and generalization methods (such as K-fold validation). We also may directly integrate well-vetted Java implementations of common statistical tests (T-tests, Bonferroni adjustments, etc.), but we are hesitant to do this given the high-quality statistics facilities available elsewhere.

Fourth, and perhaps most importantly, we intend to add more benchmark applications. ECJ already has a large collection of benchmarks and test problems, but it could use more for two reasons. First, many benchmarks have been established since our last significant effort in this direction, including a great many problems for floating-point vector representations. One obvious source here is the BBOB Benchmark Workshop [13], which we have borrowed from in the past. Second, we have introduced techniques (ACO, NEAT) which come with their own classic benchmarks and demos.

6.3 Containerization

ECJ's use of Maven ensures that the correct dependencies are on hand for the software at compile time (provided that access to the Maven Central Repository remains reliable). But other problems with the deployment environment can still arise over time: Oracle's recent release of JDK 11 temporarily broke ECJ's build, for example, because a version of the code coverage measurement library that we use for our test suites was only compatible with Java versions 10 and lower. The software development industry is rapidly moving toward *containerization* as a means of mitigating these problems and improving reproducibility. As industrial norms settle around deploying production applications inside Docker containers managed by cluster services like RedHat OpenShift, the scientific community (and thus ECJ and/or its users) should also consider using containerization to guarantee reproducibility of algorithmic experiments [3].

6.4 Cluster Computing

ECJ's distributed evaluation facilities are scalable and well-used, but require manual effort to set up master and slave instances on a variety of machines. Most users today access high-performance computing resources either through cloud services like Amazon Web Services, or through batch-cluster resources that are available through academic networks. At a minimum, we should like to provide tutorials or example scripts to help users deploy ECJ in these environments. If larger modification to ECJ's features will prove useful here, we will explore that too.

6.5 Language and Interface

Another significant trend is the widespread adoption of executable notebooks (such as Jupyter Notebook and R Markdown) for both exploratory experimentation and the presentation of polished scientific analysis. These environments combine a convenient REPL interface for interpreted languages with commentary, LaTeX equations, and inline plots, collecting what used to be several different steps of the experimental process into a single unified interface. While nothing prevents ECJ from being launched externally as part of a custom notebook workflow, its reliance on standalone parameter files and compiled executables — rather than interpreted, programmatic instantiation of algorithms — makes it slightly more cumbersome to incorporate into a Jupyter session. One option for future work might be to write a Python wrapper for launching and parameterizing ECJ sessions and retrieving their output — making it easier for students to get up and running with a complete ECJ experiment-analysis cycle. This approach would fit comfortably in the tradition of using Python as language for "glue code" that holds a heterogeneous project together.

7 CONCLUSION

We have offered a justification for ECJ as a common toolkit, have placed it in the context of other current systems, and have provided a brief overview of its architecture. We have also offered an update on the features added during the NSF grant period so far, and some of the general design challenges that our attempts to create orthogonal frameworks for metaheuristics work have raised. We welcome collaboration and user feedback going forward with this effort to create a unifying platform for quality metaheuristics research work.

ACKNOWLEDGMENTS

Thanks to Lilas Dinh, David Freelan, Ermo Wei, Sunil Rajendran, Ananya Dhawan, Ben Brumbac, and Anson Rutherford for their recent contributions to ECJ's codebase. This work is supported by the National Science Foundation (USA) under Grant No. 1629850.

REFERENCES

[1] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing.* Cambridge University Press.
[2] Shumeet Baluja. 1994. *Population-based incremental learning. a method for integrating genetic search based function optimization and competitive learning.* Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Dept Of Computer Science.
[3] Brett K Beaulieu-Jones and Casey S Greene. 2017. Reproducibility of computational workflows is automated using continuous analysis. *Nature biotechnology* 35, 4 (2017), 342.
[4] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: automatically detecting flaky tests. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 433–444.
[5] Antonio Benitez-Hidalgo, Antonio J. Nebro, Jose Garcia-Nieto, Izaskun Oregi, and Javier Del Ser. 2019. jMetalPy: a Python Framework for Multi-Objective Optimization with Metaheuristics. *arXiv:1903.02915 [cs]* (March 2019). http://arxiv.org/abs/1903.02915 arXiv: 1903.02915.
[6] Peter AN Bosman, Jörn Grahl, and Dirk Thierens. 2013. Benchmarking parameter-free amalgam on functions with and without noise. *Evolutionary computation* 21, 3 (2013), 445–469.
[7] Marco Dorigo and Luca Maria Gambardella. 1997. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation* 1, 1 (1997), 53–66.
[8] Marco Dorigo, Vittorio Maniezzo, Alberto Colorni, et al. 1996. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 26, 1 (1996), 29–41.
[9] Marco Dorigo and Thomas Stützle. 2004. *Ant Colony Optimization.* MIT Press.
[10] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. 2014. Virtualization vs containerization to support PaaS. In *2014 IEEE International Conference on Cloud Engineering*. IEEE, 610–614.
[11] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research* 13, Jul (2012), 2171–2175.
[12] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 416–419.
[13] Nikolaus Hansen, Anne Auger, Raymond Ros, Steffen Finck, and Petr Pošík. 2010. Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*. ACM, 1689–1696.
[14] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (2015), 630–643.
[15] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*.
[16] Matthew Hutson. 2018. Artificial intelligence faces reproducibility crisis. *Science* 359, 6377 (FEbruary 2018), 725–726.
[17] M. Keijzer, J. J. Merelo, G. Romero, and Marc Schoenauer. 2002. Evolving Objects: a General Purpose Evolutionary Computation Library. In *Evolution Artificielle (EA)*. 231–242.
[18] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. 2003. Iterated local search. In *Handbook of metaheuristics*. Springer, 320–353.
[19] Sean Luke. 2012. *Multiagent Simulation and the MASON Library.* Updated regularly. Over 350 Pages. Available at http://cs.gmu.edu/~eclab/projects/mason/.
[20] Sean Luke. 2017. ECJ then and now. In *Companion Proceedings to the Genetic and Evolutionary Computation Conference*. 1223–1230.
[21] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. 2005. Mason: A multiagent simulation environment. *Simulation* 81, 7 (2005), 517–527.
[22] Sean Luke and A. K. M. Khaled Ahsan Talukder. 2013. Is the Meta-EA a Viable Optimization Method?. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation (GECCO)*.
[23] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 643–653.
[24] Zeeya Merali. 2010. Computational science: Error, why scientific programming does not compute. *Nature* 467, 7317 (2010), 775–777.
[25] Ryszard Michalski. 2000. Learnable Evolution Model: Evolutionary Processes Guided by Machine Learning. *Machine Learning* 38, 1–2 (2000), 9–40.
[26] Greg Miller. 2006. A scientist's nightmare: software problem leads to five retractions. (2006).
[27] Randal S Olson and Jason H Moore. 2016. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*. 66–74.
[28] Velur Rajappa, Arun Biradar, and Satanik Panda. 2008. Efficient software test case generation using genetic algorithm based graph theory. In *2008 First International Conference on Emerging Trends in Engineering and Technology*. IEEE, 298–303.
[29] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation* 10, 2 (2002), 99–127.
[30] Thomas Stützle and Holger H Hoos. 2000. MAX–MIN ant system. *Future Generation Computer Systems* 16, 8 (2000), 889–914.
[31] Kuo-Chung Tai and Yu Lei. 2002. A test generation strategy for pairwise testing. *IEEE transactions on software Engineering* 28, 1 (2002), 109–111.
[32] Sebastián Ventura, Cristóbal Romero, Amelia Zafra, José A Delgado, and César Hervás. 2008. JCLEC: a Java framework for evolutionary computation. *Soft Computing* 12, 4 (2008), 381–392.
[33] James A Whittaker and Michael G Thomason. 1994. A Markov chain model for statistical software testing. *IEEE Transactions on Software engineering* 20, 10 (1994), 812–824.