DOI: http://dx.doi.org/10.7551/978-0-262-33027-5-ch066

Wallace: An efficient generic evolutionary framework

Christopher Steven Timperley and Susan Stepney,
Department of Computer Science and York Centre for Complex Systems Analysis
University of York,
York, United Kingdom
ct584@york.ac.uk susan.stepney@york.ac.uk

Abstract

We present a novel evolutionary computing framework, Wallace, that achieves ease-of-use and genericity, via a domain-specific language, and simultaneously achieves efficiency via meta-programming, as well as supporting parallelism. Wallace also includes a novel multiple representation model of individual development, realised using meta-programming. We describe the Wallace framework, illustrating it with a number of example problems from the literature. We compare the performance of this framework to existing EC frameworks; early results show improvements in both conciseness and speed over popular alternatives. Finally, we discuss the future of EC frameworks, and the ongoing developments to the Wallace framework.

Introduction

Choosing the right evolutionary computation (EC) framework is a challenge; wherever high performance is offered, large amounts of "boilerplate" code in a low-level language seem sure to follow, whilst frameworks offering expressiveness through a simple and elegant syntax almost exclusively do so at the cost of performance, reducing their value to researchers and industrial users. By forcing users to decide between expressiveness and performance, these tools divide the research and development efforts of the community.

Our new EC framework, Wallace, achieves both ease-of-use and high performance. Wallace exploits the expressive power and conciseness of domain-specific languages (DSLs) with the process of computational reflection (Maes, 1987), the ability to write and modify parts of a program at run-time. Using the semantic information provided in human-readable descriptions, we synthesise highly optimised algorithms specific to the details of a given problem.

The rest of the paper is structured as follows. First, we review the current state of EC frameworks, examining the design decisions they present and the trade-offs they incur. We then propose how DSLs and meta-programming can be used to avoid these trade-offs, and introduce our own EC framework, Wallace. We discuss its architecture and key features, before providing a number of examples written in Wallace, then briefly examine the underlying meta-architecture used to implement it. Finally, we compare the brevity and performance of Wallace against that of some of the most popular EC frameworks.

Background

As the field of EC continues to grow in popularity, the number of tools and frameworks increases, each varying in many respects from the last, addressing the problems of a former generation, whilst creating a new set of problems for a future generation to address.

Examples of such problems include: sacrificing performance when implementing frameworks in a dynamic language; the search of brevity and ease-of-use; the introduction of overly complicated abstractions in an attempt to tailor to all known evolutionary algorithms, harming both performance and maintainability.

Where one tool appears to perform well in one respect, it often lacks in another; such compromises often determine the intended audience of the framework and ultimately constrain its applicability. Despite the constant introduction of new software in the EC community, some older frameworks, such as ECJ (Luke, nd) and Evolving Objects (Keijzer et al., 2002), have managed to maintain a lasting appeal. Why have these tools remained so popular, where have others failed, and what problems still remain to be solved?

Audience

There are three main audiences for EC tools: educational, industrial, and research. As frameworks increase in applicability, performance and genericity, they become more aligned to research and industrial users, who require the ability to write fast and highly custom algorithms for complex problems, but in the process they decrease their ease-of-use and raise their barrier to entry, making them less suited to an educational audience.

Applicability

Each framework varies in its domain of applicability; some are restricted to performing a small subset of EC, such as

Christopher Steven Timperley, Susan Stepney (2015) Wallace: An efficient generic evolutionary framework. Proceedings of the European Conference on Artificial Life 2015, pp. 365-372

GEVA (O'Neill et al., 2008), whilst others are built to facilitate all forms of EC. Each approach has its own merits and disadvantages.

By modelling only a subset of EC, one may reap the performance benefits of specificity, by significantly reducing the level of abstraction, removing inefficient generic code, and using more efficient memory allocation patterns. This improves both performance of the framework, and the maintainability of its codebase, but comes at the cost of learning a new framework for each form of EC one wishes to perform. Furthermore, creating a new framework for each field of EC involves re-implementing highly common operations, representations, and logging facilities.

More generic tools, such as ECJ and EO, are designed to be applicable in all realms of meta-heuristic computation, but at the cost of performance; their highly abstract frameworks add significant layers of overhead and complexity, removing much potential for optimisation. EO deals with this better than ECJ, by exploiting C++ templates, but increases the complexity of its language in the process, raising the barrier to entry. ECJ and EO possess plentiful libraries of operators, representations and more, but other high-level frameworks, such as JCLEC (Ventura et al., 2008) and Watchmaker (Dyer, 2010), are often lacking; it is easier to implement a small subset of EC well than it is to implement its entirety.

As highlighted by Gagné and Parizeau (2006), EC frameworks differ in their domains of applicability, and in the genericity of their various components, such as their representation, fitness, operations, evolutionary model, parameters, and output. Tools such as ECJ and EO have genericity in all criteria proposed by Gagné and Parizeau, and allow users to easily integrate new concepts into the framework.

Ease-of-Use

High performance frameworks are almost exclusively written in relatively low-level languages such as C, C++ and Java. Such frameworks naturally incur considerable boilerplate code and require an extensive knowledge of their underlying language. This higher level of performance comes at the cost of a reduced level of conciseness and expressiveness. By trading off these properties in search of performance, the barrier of entry to these tools is raised, and their potential as educational tools is diminished.

Frameworks written in interpreted languages, such as Python and Ruby, require fewer lines of code and permit more human readable descriptions than their compiled counterparts. Additionally, such languages allow the user to easily prototype, inspect and modify algorithms as they are running, making them well suited to the classroom. However, such advantages are attained by sacrificing the speed and optimisation opportunities afforded by faster compiled languages.

Some high performance frameworks escape the difficulties of dealing with low-level languages by bypassing them all together, and instead relying on user input to graphical user interfaces to setup and run algorithms. Whilst these GUIs often make for excellent educational tools, they are seldom useful to the industrial or research user, who almost always wishes to use representations, operations and algorithms beyond those incorporated within the tool.

Frameworks may attempt to escape these issues by allowing users to describe their algorithms in the form of a highly restricted DSL. Such descriptions serve as rigid specifications, say in the form of Java parameter files or XML files, allowing the user to specify the various settings of their algorithms from a set of predefined options. Whilst these descriptions are often more concise than their alternative, they are so at the cost of expressiveness; seldom is the user allowed to describe the behavioural aspects of an algorithm without writing in the language of the framework itself.

An Ideal Language

We believe that if a framework achieved ease-of-use, applicability, genericity *and* performance, that it might better serve all audiences, and thus become a candidate for a common language, bringing the EC audiences closer together. Yet such a framework seems impossible; surely one cannot maintain performance whilst enjoying genericity and applicability, which themselves cannot be enjoyed without compromise to ease-of-use?

Our solution is two-pronged. First, by employing a DSL, we can still enjoy ease-of-use, whilst maintain genericity and applicability. Such an approach not only allows the user to write algorithms in more natural terms, making them more amenable to communication, but also retains the expressiveness of the underlying programming language, allowing the user to integrate behaviours beyond those prescribed, unlike parameter file based approaches.

Second, in order to maintain high performance whilst achieving these qualities, we exploit meta-programming to allow us to write high level code that remains on a par with low level approaches in terms of performance. Rather than using our DSL to simply specify components within the language, we enhance it with the ability to extend the language and to dynamically synthesise new code, allowing natural high-level descriptions to be transformed into highly optimised context-aware code fragments on-the-fly, as shown in Figure 1.

Our realisation of this solution is the **Wallace** framework, named after the famous naturalist and co-discoverer of natural selection, Alfred Russel Wallace. To achieve these feats we implemented Wallace using the Julia language.

Julia (Bezanson et al., 2014) is a relatively young high-level, high-performance dynamic programming language, designed for technical computing, built around multiple-dispatch, a rich type system, and a just-in-time compiler that specialises methods based upon types encountered at run-

Christopher Steven Timperley, Susan Stepney (2015) Wallace: An efficient generic evolutionary framework. Proceedings of the European Conference on Artificial Life 2015, pp. 365-372

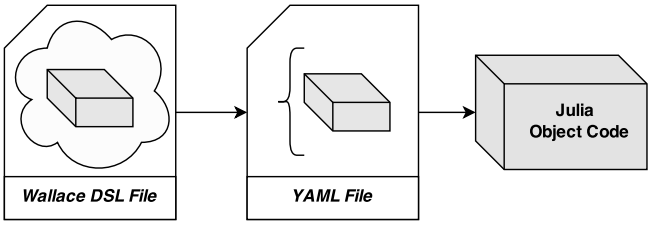

Figure 1: High-level descriptions of objects are parsed into YAML before being dynamically synthesized into optimised Julia object code.

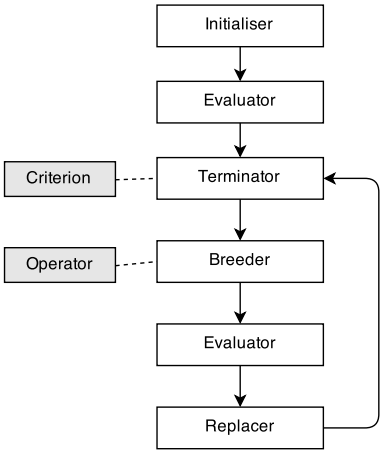

Figure 2: The flow of data within the component-based architecture of an EA in Wallace.

time. As a result of these design decisions, Julia attains a performance close to if not equal to C across a wide range of problems, without needing to resort to writing low-level code (Bezanson et al., 2012).

Architecture

Similar to ECJ and EO, Wallace employs a component-based architecture, where algorithms are described in terms of a series of customisable components, each responsible for implementing some part of the standard evolutionary loop employed by the majority of EAs; the flow of data between the components within this architecture is shown in Figure 2. Within Wallace each component is built using a provided description, and often compiled to a highly specific and optimised form through the use of run-time code evaluation.

In this section we outline how Wallace implements some of these components, and discuss its population model and its novel multiple representation model.

Population Model

Wallace uses a population model similar to that of EO and ECJ, where an algorithm operates on a single population divided into an arbitrary number of independently evolving sub-populations, or *demes*.

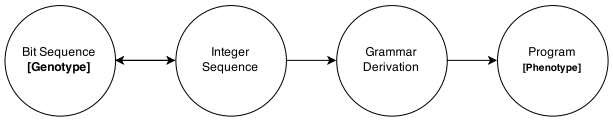

Figure 3: An example series of developmental stages for a problem using grammatical evolution. A sequence of bits is used as the genotype of an individual, from which an integer sequence may be produced; the bi-directional indicates that changes in the integer sequence may be transmitted to the bit sequence. From the integer sequence, a grammar derivation is produced, which in turn, is used to produce a program.

```
species:
  stages:
    bit_sequence:
      representation<bit_vector>: { length: 800 }
    int_sequence:
      from: bit_sequence
      representation<int_vector>: { length: 100 }
    derivation:
      from: int_sequence
      representation<string>: { frozen: true }
    program:
      from: derivation
      representation<lambda>:
        arguments: ["x::Int", "y", "z"]
```

Figure 4: A multiple representation realisation of a simple grammatical evolution (Figure 3) approach to symbolic regression.

Each deme hosts a single *species* of individuals. The population may comprise several heterogeneous demes, allowing different species to be co-evolved. Island model EAs can be realised by attaching a *migrator* component to the population, which exchanges individuals between demes, according to a *migration policy* (Whitley et al., 1998). By combining the migrator component with Wallace's multiple representation model (below), we can implement more advanced models, such as the multiple representation island model (Skolicki and De Jong, 2004), where each deme evolves a different representation of a given problem in parallel.

Multiple Representation Model

A novel feature of Wallace is its multiple representation model, which allows individuals to include an arbitrary number of linked representations, implementing a rich process of *development*. One may create a *Lamarckian* connection between certain developmental stages, allowing changes made to a later stage to be communicated back to earlier ones. An example is given in Figure 3.

To define a species, the user details the development stages of its individuals, by specifying the representation used by each stage, along with any associated parameters, and the development stage from which it should be produced. An example species definition is given in Figure 4.

Users may add arbitrary representations into Wallace by registering their implementing type and associated factory with the kernel. These representation types define the de-

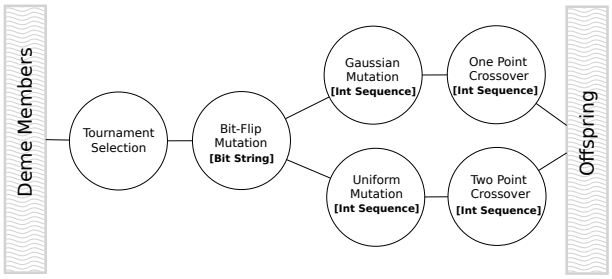

Figure 5: An example breeding setup, illustrating the chain of operations that a proto-offspring is subjected to before being inserted into the set of offspring. Each node within the graph represents a particular selection method, or a variation method, in which case it operates on the developmental stage described within.

```
breeder<breeder/fast>:
  sources:
    s<selection>:
      operator<selection/tournament>: { size: 2 }
    x<variation>:
      from: s
      stage: bits
      operator<crossover/two_point>: { rate: 0.7 }
  m<variation>:
    from: x
    stage: bits
    operator<mutation/bit_flip> { rate: 0.01 }
```

Figure 6: An example of a fast breeder setup.

```
evaluator<evaluator/simple>:
  objective: |
    SimpleFitness(true, count(i.bit_vector, 1))
```

Figure 7: An example of a simple evaluator, used to compute fitness as the number of ones within an individual's bit vector representation.

fault way to pseudo-randomly generate new individuals, and are used to describe how an instance of one representation should be transformed into an instance of another.

Breeding Model

Selecting parents and producing offspring is the responsibility of Wallace's *breeder* component, heavily inspired by ECJ's powerful breeding pipeline system. Each deme is allocated its own breeder, allowing different demes to carry out their own process of breeding. This breeder produces a set of offspring at each generation, prior to the stage of *replacement*, where the *replacement* component decides which individuals from the current members of the deme and its set of offspring should survive into the next generation.

The simplest breeder is the *fast breeder*, which takes a single breeding source and uses it to produce a desired number of offspring. This breeding source may take the form of a selection or variation method, or as a form of proxy, either returning offspring from a number of different sources based upon chance, or selecting between them based upon the state of the search. Each source may draw its inputs from other sources, or directly from the contents of a deme, as in the case of selection methods. An example breeding setup is shown in Figure 5.

Using the syntax in Figure 6, users may add new sources into the breeder by declaring their type (e.g. selector or variation operator), the name of the source from which they should draw their inputs, the name of the developmental stage upon which they should operate, which defaults to the genotype if omitted, along with any operator-specific parameters, such as mutation rate and tournament size.

Like most breeders within Wallace, the fast breeder has support for parallel breeding, which it achieves by splitting the workload at each step within the breeding process as equally as possible between all available cores.

Fitness Model

Following breeding, and prior to replacement, each unevaluated candidate solution within the population is subject to *evaluation*. This process is carried out by an *evaluator*, which accepts a population and assigns fitness values to all of its (unevaluated) individuals based on their performance. This evaluator may treat each individual in isolation, or may implement a co-evolutionary approach, where the fitness of an individual is calculated by combining or comparing it against other individuals in the same population, or another population.

The *simple* evaluator is built by being given an objective function, provided in the form of a lambda function, accepting a single input individual and returning a calculated Fitness object for that individual (Figure 7).

The concept of fitness itself is implemented using abstract Fitness objects, which only require the programmer to specify a means of comparing them, allowing multiple-objective and more advanced co-evolutionary fitness values to be used; this avoids the user becoming trapped within the limitations of a single scalar fitness value, as is the case with many other frameworks.

In a similar fashion to breeders, evaluators may also take advantage of multiple cores by splitting their workload across each of them.

Examples

To illustrate the use of Wallace, we give two examples: implementing the OneMax problem using a simple genetic algorithm (Figure 8), and evolving a Java AI controller for the Robocode (Nelson et al., 2014) tank game via grammatical evolution (Figure 9).

We begin the algorithm description for a problem by starting with the algorithm keyword, followed by the name of our particular algorithm. We specify that our algorithm extends the base evolutionary algorithm; were we to imple-

```
type: evolutionary_algorithm

_my_breeder<breeder/fast>:
  sources:
    s<selection>:
      operator<selection/tournament> { size: 2 }
    x<variation>:
      from: s
      stage: bits
      operator<crossover/uniform>: { rate: 0.7 }
    m<variation>:
      from: x
      stage: bits
      operator<mutation/bit_flip>: { rate: 0.01 }

_my_species:
  stages:
    bits:
      representation<bit_vector>: { length: 100 }

replacement<generational>: {}

population:
  demes:
    - capacity: 100
      species:  $(_my_species)
      breeder:  $(_my_breeder)

evaluator<evaluator/simple>:
  objective: |
    SimpleFitness(sum(i.bits), true)
```

Figure 8: An implementation of OneMax within Wallace.

ment some other form of meta-heuristic we would extend its base type instead.

We then describe the species of individuals that we wish to evolve, outlining each of its stages of development and the representations they use, as well as the type of fitness object used to compare the fitness of individuals within the same species. For OneMax, the species definition says that our individuals have one stage of development, that being their bit-string, and that we use a simple scalar-based measure of fitness. For the more complex Robocode problem, we setup the bit-string, codon sequence, and grammar derivation, representing the source code for our robot controller, as commonly used when performing GE, along with an additional controller stage that we have added, which uses a custom representation to simplify compilation of robocode controllers and communications with the robocode platform.

Next, we outline the specifics of our breeder, detailing each of its operators, along with their respective sources and the name of the development stage on which they operate, in the case of variation methods.

Finally, we specify our method of replacing individuals following the breeding phase, and specify the size, species and breeder used by each of the demes within the population, before providing an evaluation function for our problem, responsible for measuring the quality of potential solutions.

Wallace takes these descriptions and using meta-programming, compiles them into highly efficient object-code, optimised to the specifics of the given problem.

```
type: evolutionary_algorithm

termination:
  iterations<criterion/iterations>: { limit: 1000 }

_my_species:
  stages:
    bits:
      representation<bit_vector>: { length: 800 }
    codons:
      from: bits
      lamarckian: true
      representation<int_vector>: { length: 100 }
    source_code:
      from: codons
      representation<grammar_derivation>:
        root:  s
        rules: ...
    controller:
      from: source_code
      representation<robocode_controller>: {}

_my_breeder<breeder/fast>:
  sources:
    s<selection>:
      operator<selection/tournament>: { size: 5 }
    x<variation>:
      from:   s
      stage:  bits
      operator<crossover/two_point>: { rate: 0.7 }
    m<variation>:
      from: x
      stage: codons
      operator<mutation/gaussian> { rate: 0.02 }

replacement<generational>: { elites: 1 }

population:
  demes:
    - capacity: 100
      species: $(_my_species)
      breeder: $(_my_breeder)

evaluator<evaluator/simple>:
  objective: |
    score = execute(i.controller)
    SimpleFitness(score, true)
```

Figure 9: An implementation of Robocode controller evolution within Wallace.

Meta-Architecture

The Wallace meta-architecture provides a process responsible for transforming concise high-level component descriptions, provided in a DSL, into fragments of highly optimised code, using meta-programming. It is through the reflective capabilities of Julia that Wallace manages to perform such transformations on-the-fly, as the program is running.

Wallace Description Language

Rather than interacting directly with the Julia programming language, Wallace users provide natural and concise high-level descriptions of their algorithms and their various components using a specialised DSL built upon an extended YAML, which retains the ability to provide code fragments and to define new behaviours.

Descriptions are realised as objects within Wallace by passing them to the make function. Following a pattern sim-

```
type: evolutionary_algorithm

evaluator:
  type: evaluator/simple
  objective: |
    ...

replacement:
  type: generational
...
```

Figure 10: An example algorithm description translated to YAML.

```
> julia
Julia...

julia> using Wallace

wallace> alg = load("max_ones.wlc")
<EvolutionaryAlgorithm:...>

wallace> run!(alg)
...
```

Figure 11: Loading and executing an algorithm description via the REPL.

ilar to that of the Factory pattern (Freeman et al., 2004), inspired by the popular Ruby test data generation tool, FactoryGirl (Ferris, 2014), descriptions are forwarded to the factory responsible for unfolding them; this may be done automatically by the Wallace kernel, by inspecting the type attribute of the description, or by manually specifying which factory should be used.

Once a description has reached a factory, it is subject to the processes of *preparation*, *validation* and *composition*, before it is transformed into an object of the desired type (which may itself be a description).

Preparation Stage The description is transformed into a valid YAML object (Figure 10), by first removing all comments, before extracting each type tag from within the document and inserting it into its associated object, and finally handling all insertion point operations.

Validation Stage Next, the YAML description is passed to the validation stage, where it is checked against a series of rules for legality, provided in the form of a function, before it passed onto the final stage to be transformed into a concrete object.

Composition Stage Finally, the validated description is used to construct the concrete object it describes. Usually this stage involves recursively constructing the concrete objects of the desired object's individual components, by invoking each of their associated factories with their descriptions, before composing them into a single object. Rather than provide a concrete object, it may be the case that certain *abstract* factories are used to construct partial objects, or to further detail given descriptions.

Performance Optimisations

In addition to using reflection to transform descriptions into object code, Wallace exploits reflection to implement optimised data structures and algorithms, highly specific to the details of a given problem.

To realise its multiple representation model without incurring a performance hit by storing each stage of an individual's development in an abstract container, Wallace uses the information provided by a description of a given species to create a memory-optimised individual type, specific to that species; this approach substantially reduces look-up time and memory consumption. This technique is also applied to the fitness model used by a given species of individuals; by specifying the type of fitness used by individuals within that species, Wallace can create a faster and more compact individual type definition by stating it specifically.

Wallace also uses reflection to implement a more efficient mechanism for converting between representations. Rather than dynamically calculating whether a given stage is out of sync with the genome and then determining the series of conversion steps that should be performed, Wallace determines the state of each of the development stages following each operation and hard-codes the minimal number of conversion operations into a highly specific (and thus optimised) breeder.

Typically, these conversion operations are applied to each individual within a group in sequence, and written in terms of a mapping operation on a single individual. However, the user may elect to implement a *batch conversion* mechanism for their representation, defining how a group of individuals should have representations converted collectively, allowing costly overheads to be minimised, such as compiling external code, proving useful when performing grammatical evolution in an external language.

Framework Interaction

Currently, Wallace lets its users interact in three different ways, allowing it to be used in a variety of contexts and for a multitude of purposes.

Read-Eval-Print-Loop (REPL) The quickest way to get up and running with Wallace is through Julia's REPL interface, augmented with additional functionality, allowing users to quickly perform experiments and prototype simple code. Component descriptions can be loaded into Wallace through calling the load or build commands with the location of a valid component description file (Figure 11). Once loaded, objects may be treated as any other kind of object within Julia, allowing them to interact with external libraries.

The REPL also provides a number of usability-driven fea-

Framework	Language
ECJ	Java
JCLEC	Java
DEAP	Python
Pyevolve	Python
inspyred	Python

Table 1: Evolutionary computation frameworks to perform performance and brevity comparisons.

tures, some of which are listed below, which allow users to quickly assimilate themselves with the framework with minimal effort.

- help(type), used to provide descriptions of the various components available within Wallace, along with their inputs and modes of operations, in a neat and user-friendly manner.
- list_subtypes(type), provides a list of all known subtypes of a given type, each of which can be further inspected using the help command.

Script Execution One may also write conventional Julia scripts to compose and execute their algorithms, by simply calling "using Wallace" at the top of the script, and executing the file as standard from the command line.

Interactive Graphical Notebook As Wallace seamlessly integrates into the Julia language, one may also exploit Julia's powerful browser-based interactive environment, IJulia, built upon the popular IPython/Jupyter interactive computational environment. By combining IJulia with other packages, such as Gadfly and Interact.jl, one may also create a large variety of highly detailed and customisable plots and graphs.

Together, these tools form a powerful package for evolutionary computation, both in a research, industrial and classroom environment, allowing experienced users and newcomers alike to quickly and empirically prototype solutions and to visualise the evolutionary process, whilst retaining the performance benefits shared by lower level languages.

Comparison

Here we compare the brevity and performance of Wallace to a selection of the most popular EC frameworks used within the literature, listed in Table 1, in order to assess its standing. Due to time constraints and compilation issues, we were unfortunately unable to test performance against other C/C++ options.

Brevity

In order to compare the conciseness of Wallace descriptions to those of other frameworks, we repeated the study carried out by Fortin et al. (2012), where the number of lines required to implement an algorithm to solve the OneMax benchmark problem is counted and compared for each framework. The results are given in Table 2.

Framework	Type	Config.	Alg.	Example	Total
Wallace	34	29	22	0	85
ECJ	308	34	88	26	456
Pyevolve	59	0	261	16	336
inspyred	0	0	330	30	360
DEAP	0	0	0	59	59
JCLEC	198	15	192	29	434

Table 2: Number of lines required to implement OneMax problem using different EC frameworks.

	Benchmark
GA1	OneMax ($n = 100$)
GA2	Rastrigin ($n = 100$)
GP1	Artificial Ant (Santa-Fe Trail, 400 moves)
GP2	Symbolic Regression ($x^4 + x^3 + x^2 + x$)
GP3	Boolean Circuit (8x3 multiplexer)

Table 3: Benchmark problems used to compare performance of different EC frameworks. Each had a population size of 100, and ran for 1000 generations.

Whilst it took more lines in Wallace to implement the OneMax problem than it did using DEAP, a framework firmly established for its clarity and brevity, the example file used to enter the specifics of the algorithm setup and the OneMax problem was smaller and considerably simpler than that used by DEAP, involving only the specification of problem details in a simple and compact structure.

Furthermore, Wallace required only that the user describe the setup of their algorithm in terms of its components, allowing them to choose from a range of operators and representations from its standard library, whereas the example file used by DEAP required the user to write the EA from scratch in Python, using its framework to skip boilerplate definitions.

Performance

In order to fairly assess the performance of Wallace to that of other frameworks, we compared execution times across 100 runs for a number of common benchmark functions on a single thread, listed in Table 3, under the same conditions and using the same algorithm setup (or as close as the underlying framework would permit).

An overview of the results from our experiment are given in Table 4; the full experiment setup and raw results data are online at https://github.com/ChrisTimperley/EC-Software-Benchmarks. Where it was not possible to implement a solution to a given problem using the standard library of a framework, that benchmark was skipped, and is denoted within the table by *n/a*.

Christopher Steven Timperley, Susan Stepney (2015) Wallace: An efficient generic evolutionary framework. Proceedings of the European Conference on Artificial Life 2015, pp. 365-372

Framework	GA1	GA2	GP1	GP2	GP3
Wallace	0.329	0.453	0.938	0.294	15.971
ECJ	0.502	1.070	1.338	0.791	31.759
JCLEC	0.386	0.431	n/a	0.430	n/a
DEAP	5.357	12.530	105.683	53.949	59.056
Inspyred	5.343	11.679	n/a	n/a	n/a
Pyevolve	2.977	5.011	n/a	n/a	n/a

Table 4: Mean time taken to perform each benchmark, averaged over 100 runs. Recorded using the @time macro in Julia for Wallace, and using time in the shell for all others.

As one might predict, the results show a marked difference between the relatively slow performance of the Python-based frameworks and the significantly faster Java-based frameworks.

However, Wallace, like the Python-based frameworks, is written in a high-level dynamic language, yet it shows a performance similar to, and in some cases beyond that of, the the lower-level Java-based frameworks. Wallace attains the best performance on 4 out of the 5 benchmarks tried, with only a narrow gap to the winner of GA2, JCLEC. Although difference in performance between Wallace and JCLEC is relatively small across each of the benchmarks, the difference between Wallace and each of the other languages tested is far more noticeable.

Conclusion

We have presented Wallace, a new EC framework, possessing both high performance *and* a high degree of conciseness and clarity, through the novel combination of DSLs and computational reflection. The results of our comparison show that Wallace shares a similar performance with the fastest existing frameworks that we analysed, and manages to do so whilst requiring substantially fewer lines of code than its nearest competitors.

The source code for Wallace, available under the LGPL license, along with its latest binaries, documentation and bug reporting can be found online at: https://github.com/ChrisTimperley/Wallace.jl.

Future Work

Our primary focus for the future remains on improving and supporting the Wallace framework so that users can easily write and share their own algorithms, representations and extensions. In the short term, we intend to do this by creating a dedicated website, complete with in-depth documentation and a series of tutorials for a wider range of users. In the further future, we hope to build upon Julia's package system and provide our own online repository, allowing users to share their creations for the benefit of the EC community.

We also intend to explore the wider possibilities for metaprogramming within Wallace, and how our techniques might be applied to provide similar performance boosts to other large frameworks.

Acknowledgments

We thank Dan Franks, for his advice and support, and for trusting us to use Wallace for his Masters-level evolutionary computation module, and the patient students who provided us with invaluable feedback. We also thank Paul Andrews and Simon Hickinbotham, for numerous discussions about the ideas and direction of the framework. This work was partly funded by an EPSRC DTG.

References

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2014). Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607.

Bezanson, J., Karpinski, S., Shah, V. B., and Edelman, A. (2012). Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145.

Dyer, D. (2010). The Watchmaker Framework for Evolutionary Computation. http://watchmaker.uncommons.org/. Accessed: 2015-03-05.

Ferris, J. (2014). FactoryGirl. https://github.com/thoughtbot/factory_girl. Accessed: 2015-03-05.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., and Gagné, C. (2012). DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175.

Freeman, E., Freeman, E., Bates, B., and Sierra, K. (2004). *Head First Design Patterns*. O' Reilly & Associates, Inc.

Gagné, C. and Parizeau, M. (2006). Genericity in evolutionary computation software tools: Principles and case-study. *Int. J. on Artificial Intelligence Tools*, 15(02):173–194.

Keijzer, M., Merelo, J., Romero, G., and Schoenauer, M. (2002). Evolving Objects: A General Purpose Evolutionary Computation Library. In *Artificial Evolution*, volume 2310 of *LNCS*, pages 231–242. Springer.

Luke, S. (n.d.). ECJ 22: A Java-based Evolutionary Computation Research System. http://cs.gmu.edu/~eclab/projects/ecj/. Accessed: 2015-03-05.

Maes, P. (1987). Concepts and Experiments in Computational Reflection. In *OOPSLA '87*, pages 147–155. ACM.

Nelson, M. A., Larsen, F. N., and Savara, P. (2014). Robocode. http://robocode.sourceforge.net/. Accessed: 2015-03-05.

O'Neill, M., Hemberg, E., Gilligan, C., Bartley, E., McDermott, J., and Brabazon, A. (2008). GEVA: Grammatical Evolution in Java. *SIGEVOLution*, 3(2):17–22.

Skolicki, Z. and De Jong, K. (2004). Improving Evolutionary Algorithms with Multi-representation Island Models. In *PPSN VIII*, volume 3242 of *LNCS*, pages 420–429. Springer.

Ventura, S., Romero, C., Zafra, A., Delgado, J. A., and Hervás, C. (2008). JCLEC: a Java framework for evolutionary computation. *Soft Computing*, 12(4):381–392.

Whitley, D., Rana, S., and Heckendorn, R. B. (1998). The Island Model Genetic Algorithm: On Separability, Population Size and Convergence. *J. Comp. Info. Tech.*, 7:33–47.

Christopher Steven Timperley, Susan Stepney (2015) Wallace: An efficient generic evolutionary framework. Proceedings of the European Conference on Artificial Life 2015, pp. 365-372