Disambiguating Evolutionary Algorithms

Composition and Communication with ESDL

Steve Dower

A dissertation presented for fulfillment of the requirements for the award of
Doctor of Philosophy

2012

Abstract

Evolutionary Computation (EC) has been developing as a field for many years. Encompassing a range of intelligent and adaptive search, optimisation and decision-making algorithms, there is a wealth of potential for EC to be applied to problems in many domains. People who are new to the field may want to learn, understand or apply EC, while those who are more experienced are looking to extend, develop and teach. Unfortunately, it is not clear how best to approach these tasks. Ideally, students should have guidance through the field, including how EC works and approaches to algorithm design; researchers should have canonical structures, implementations, presentation formats and comparison frameworks; developers should have easy access to interested users, as well as the potential to differentiate their work in terms of performance, flexibility and aesthetics.

This thesis provides a model of the structure of Evolutionary Algorithms (EAs) based on operator composition. A small number of discrete element types and their interactions are defined, forming an algorithm architecture that supports existing concepts and provides direction for those looking to understand, use and improve EAs. A simple description language based on these elements is created to support communication between authors, readers, designers and software. Implementation concerns, ideas and potential are discussed to assist those with an interest in developing the simulation tools and frameworks used within the field. The model and description language are shown to concisely and unambiguously describe EAs in a directly publishable form.

Acknowledgements

I would like to specially acknowledge the contributions and assistance provided by people and organisations, without whom, this work could not have occurred.

My supervisory team, Tim Hendtlass, Clinton Woodward and James Montgomery, for their support, inspiration and only interfering when it was necessary; the Faculty of Information and Communication Technologies at Swinburne University of Technology for providing my candidature and support throughout my PhD; the Australian Government, for the financial support of the Australian Postgraduate Award; Jason Brownlee, Raj Vasa, Irene Moser and Naomi Parkinson for their enthusiastic support and proofreading; and finally, Kev, for keeping me focused while I worked.

Declaration

I declare that: This thesis contains no material which has been accepted for an award to myself for any other degree or diploma, except where due reference is made in the text of the thesis. This thesis, to the best of my knowledge contains no material previously published or written by another person except where due reference is made in the text of the thesis. Where the work is based on joint research or publications, this thesis discloses the relative contributions of the respective works or authors.

Steve Dower
Date: May 25, 2012

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

1.1 Motivation

When presenting an Evolutionary Algorithm (EA) in any form, whether as an approach to solving a real-world problem, the subject of a benchmarking or parameter tuning algorithm, a topic in a classroom or the result of improving an existing EA, a number of identifiable topics recur. Each publication has an overview, however brief, of what EAs are and what they are used for. Relevant aspects of related algorithms and prior work must be repeated or cited. A verification implementation is usually created, often analysed, occasionally mentioned, sometimes described and rarely shared. Finally, analysis and attribution of causality is used to prove the point that is being made.

While practically every publication on EAs reads like a success story, attempting to reproduce the results often presents considerable difficulty. With the highly segmented history of the field—most algorithms claim ancestry to Genetic Algorithms (GA), Evolution Strategies (ES), Evolutionary Programming (EP) or Genetic Programming (GP)—a range of presumptions surround the terminology used, necessitating detailed background information in each paper and precise definitions of terms such as "fitness," "population" or "proportional selection." When definitions for these terms are obtained from another work, reproduction becomes a case of citation tracing, as well as identifying any random errors introduced when passing from one author to the next (see: mutation). Provided sufficiently precise and robust definitions are available, there is a chance that a reader might fully understand what the author intended.

Given the difficulties in performing theoretical analysis of algorithms that are fundamentally driven by entropy, it is no surprise that empirical evidence is used to support most claims. Independent verification of theoretical works can be based on a single publication, provided all assumptions are valid or otherwise trusted. Verification of experimental results, however, requires a true reproduction of the

experiment. Such a reproduction is only possible where adequate information about the original experiment has been provided, and despite being included as a review criterion a quick scan of conference proceedings immediately reveals that this is not common practice. While most work includes experimental results, few also include details of the experiment beyond the hardware or programming language used. Publications based on previously used and validated experimental (in this case, software) frameworks are a minority.

Results analysis often lacks robustness. So many aspects of an EA contribute to its quality that full parameter sweeps are impractical. At best, parameter variations are controlled within an experiment in order to prove (or disprove) a hypothesis. At worst, a single instance of an experiment is cited as 'the result,' with an implication of representativeness. The historical segregation of EAs leads to performance results being associated with the entire algorithm rather than the combination of components involved, the problem being evaluated and the occasional implementation error.

The motivation for this thesis is that the understanding of EAs appears to be so heavily based in its multiple historical origins that research, design and implementation approaches mistakenly prioritise entire algorithms above the components and operators of which they are constituted. None of the originating algorithms were intended for their modern application as general-purpose optimisers—each has a specific purpose or class of target problems and any success outside of these should be attributed to luck more than design. However, each of these algorithms are instances of a general search method that can be used to design a 'new' algorithm class for optimisation that is based on, but not limited by, the last 50 years of research.

With a well-defined class of EAs, the fundamental components can be abstracted. Abstraction allows more targeted research efforts through a simpler conceptualisation of contributory components, direct composition of algorithms based on component suitability, unambiguous descriptions of algorithms and flexible implementation strategies. In short, everyone involved in the EC field, whether they are interested in theoretical validation, experimental analysis or software development concerns, can benefit from a clear description and shared understanding of EAs.

1.2 Research Statement

The central hypothesis of this research is:

> Evolutionary Algorithms are a single class of algorithms that no longer need the separation resulting from their distinct origins, and a unified model and approach will aid the understanding, development, implementation and presentation of these algorithms.

The following questions are designed to directly address the contentions of this hypothesis and are used throughout this work to explicitly identify the relevance of each contribution.

1. **Are there problems with how EAs are designed, implemented and communicated, and if so, how do these affect practitioners and researchers?** The contention that EAs should "no longer" adhere to the distinctions arising from multiple origins requires greater justification than the 'purity' of a unified approach. Identifying the problems that arise as a result of separate historical origins provides support for a different approach and a set of criteria by which the alternative may be assessed.
2. **What, fundamentally, is an EA?** Notably absent from the field is a strong understanding of what constitutes an EA, with most definitions using examples rather than independent descriptions. Formulating an EA definition that is separate from EA instances is an essential task.
3. Given answers to questions 1 and 2, **how should the design, implementation and sharing of EAs be approached?** An EA definition provides the opportunity to clearly identify the interactions, contracts and behaviours of the components and suggest ways in which they can be used, combined and described. The intent is not to prescribe a 'one-size-fits-all' piece of software, but to identify effective approaches and guidance for working with EAs.
4. **Does this model prevent existing algorithms from being described?** Despite the absence of a well-defined algorithm class, much useful research has been performed with EAs. The definition and model representing a general EA must be retrospectively applicable to prior work.
5. **Does this model solve the problems found by Question 1?** The problems identified earlier are suitable assessment criteria for the new model. While complete solutions are not strictly required, a failure to significantly mitigate known issues would suggest that the proposed approach is not beneficial.

1.3 Contributions

The specific research contributions made are as follows:

1. A conceptual model of EAs. This model defines four necessary elements, *individuals*, *groups*, *operators* and *streams*, including boundary contracts and composition restrictions. Example compositions based on well-known algorithms and a 'standard library' of components is provided.
2. Evolutionary System Definition Language (ESDL), a domain-specific description language for EAs. ESDL clearly describes the composition of an algorithm from the components of the conceptual model in a simple language that is specifically designed for EAs, allowing for concise, structured presentation of new or existing algorithms.
3. An execution model of ESDL. This model is distinct from the conceptual model and describes the interpretation and evaluation of an EA. The execution model assists developers in reliably converting ESDL systems to executable code, or producing compilers that do so automatically, and identifies opportunities for extensibility and exploitation of future developments.

The following contributions are software packages for directly supporting future work in the field:

1. `esec`, a software framework for EC experimentation. Based on the conceptual EA model, `esec` executes ESDL using its library of operators for rapid prototyping and implementation of algorithms.
2. `esdlc`, a prototype multi-targeting compiler for ESDL. Currently, `esdlc` converts ESDL systems into Python code for use with `esec` and compilable C++ code for use with C++ AMP and heterogeneous hardware platforms.

1.4 Structure

This dissertation is structured across five chapters, each of which builds significantly on the work in earlier ones. The traditional literature review appears distributed throughout each chapter due to the highly diverse range of topics covered; the relevance of work related to later chapters may not be obvious without the work of earlier chapters. The questions from Section 1.2 are each specifically addressed by one or more chapters.

Chapter 2, *Unification*, shows how the segmentation by algorithm of the EC field came about, demonstrates how it is detrimental to research, design and implementation, and introduces an application-based, rather than biologically inspired, foundation of EAs. Questions 1 and 2 are addressed by this chapter.

Chapter 3, *Model*, takes the derivation from the preceding chapter and identifies the major component types and an architecture that may be used to represent,

describe and discuss general evolutionary algorithms. Questions 2–4 are addressed by this chapter.

Chapter 4, *ESDL*, specifies a domain-specific composition language for the model in Chapter 3, providing a textual format for unambiguous description of algorithms. Questions 3 and 4 are addressed by this chapter and Question 5 is partially addressed.

Chapter 5, *Execution*, reviews existing EA software, identifies various architectural decisions and defines the architecture and execution model for ESDL. Questions 3 and 5 are addressed by this chapter.

Chapter 6, *Application*, describes the use of ESDL in supporting algorithm design, implementation and presentation, shows that existing algorithms are supported and that the issues raised in Chapter 2 are mitigated. Questions 3–5 are addressed by this chapter.

Chapter 7, *Conclusions*, reviews the goals from Section 1.2 and highlights how each was addressed throughout this work. Limitations of the proposed approach are discussed and potential further research is suggested.

All of the source code created for this work and any errata and addendums are available from `http://stevedower.id.au/thesis`.

1.5 Publications

The following peer-reviewed conference papers were based on the work conducted for this thesis.

- S. Dower and C. J. Woodward, "ESDL: a simple description language for population-based evolutionary computation," in *Proceedings of the 13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011*, ACM, 2011, pp. 1045–1052.
- S. Dower, "Automatic implementation of evolutionary algorithms on GPUs using ESDL," in *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2012*, IEEE, 2012, pp. 3356–3363.

Chapter 2

Unification

The field of Evolutionary Computation arose as the conglomeration of a range of techniques and algorithms for more efficient search. Among these are three independently developed algorithms that are considered representative of Evolutionary Algorithms (EAs): Evolution Strategies (ES), Evolutionary Programming (EP) and Genetic Algorithms (GA). Practitioners, teachers, students, researchers and others involved in the EC field are likely to be aware of, or familiar with, these algorithms. and much time has been invested in testing, tuning, adapting and otherwise trying to improve the ability of each of these algorithms to solve an ever-wider range of problems. Many of these efforts follow the traditional separation between the three algorithms, but artificially treating ES, EP and GA—and other algorithms such as Genetic Programming and Differential Evolution—as entirely distinct is unnecessary and has the potential to stifle innovation and collaboration. Unifying ES, EP and GA into variations of a general class of algorithm provides many benefits when attempting to understand, explain, implement, test and share designs and results. Enough common characteristics exist that publications often begin by generalising to a unified model of EAs that suits the intended purpose, though not necessarily consistently with other generalisations. This chapter illustrates the issues caused by the current state of segregation, and derives a general class of EAs based on an application to optimisation.

2.1 Originating Algorithms

Historically, the class of EAs derives from three algorithms: Evolution Strategies (ES), Evolutionary Programming (EP) and Genetic Algorithms (GA) [33]. Each of these was developed independently for different applications, but share sufficient

commonality to be recognisable as separate instantiations of a single class of algorithm.

This section reviews the history and background of each of these algorithms, providing the necessary context to observe how the field currently works with EAs, to observe the conflicts that result from the implicit segregation between algorithms (Section 2.2) and to identify the fundamental parts of an EA that are independent of particular instances (Section 2.3).

2.1.1 Evolution Strategies

In 1971, Ingo Rechenberg published his thesis *Evolutionsstrategie, Optimierung technischer Systeme nach Prinzipien der biologischen Evolution* [78],[1] describing a manual iterative algorithm for numeric optimisation. The technique was developed during the 1960s, in collaboration with Hans-Paul Schwefel at Technische Universität Berlin (Berlin Institute of Technology) for application to fluid dynamics design problems. Early applications sought to optimise flow through a shaped pipe [56], minimise the drag over a joint plate [77] and improve the structure of a two-phase flashing nozzle [81]. Computer simulation was not readily available and theoretical designs required construction and experimental validation. For many problems, theoretical analysis was (and still is) insufficient to determine design changes that lead to improved performance; ES was designed for problems of this nature.

As an incremental optimisation algorithm, ES seeks to make small improvements to a candidate solution over many repeated efforts. The approach attempts to balance two alternate approaches to design: deterministically finding a better solution (that is, selecting a change based on knowing the result in advance) and making purely random changes until a 'good enough'[2] solution is found. As a compromise, ES uses random changes but discards those that are regressive. These random changes are created using values selected from a Gaussian probability distribution, such as that shown in Figure 2.1, based on the observation that small variations in nature are more frequently viable (less destructive) than large variations. A presumption exists that the Gaussian distribution reflects variations to an underlying genetic sequence [33], though this interpretation has little impact on the design or application of ES and the implications of a biological inspiration could alternatively be justified by showing a correlation between the magnitude of a variation and the significance of the effect.

[1] "Evolution Strategy, Optimisation of Technical Systems according to the Principles of Biological Evolution."

[2] The concept of a 'good enough' solution in engineering is well defined, typically as a minimal (or maximal) measurement multiplied by a factor of safety. Other applications have more subjective metrics for 'good enough.'

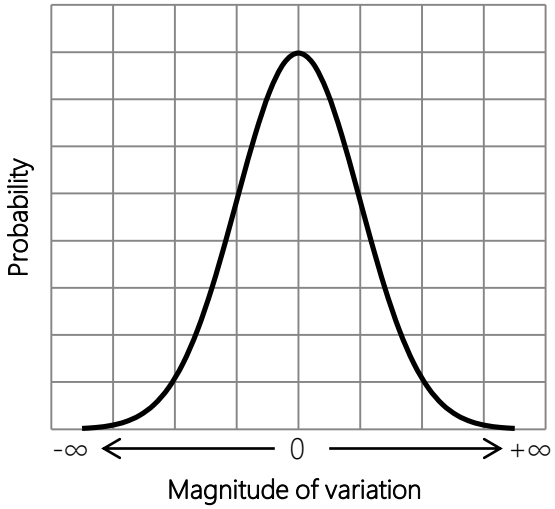

Figure 2.1: A Gaussian probability distribution, also known as a normal distribution.

Around the same time as Rechenberg's first ES publication in 1965, Schwefel generalised the algorithm to operate on vectors of real values and performed the first computer simulations to compare different versions of ES [80]. The algorithm at this point derived one variation from one candidate solution, which was either retained or discarded based on the presence or absence of any observed improvement. Rechenberg extended the algorithm to keep a pool of candidate solutions and generate a single variation based on multiple random selections from the pool [78]. Each selected "parent" contributed some components to the variation before the random values were added, with the worst solution in the pool being discarded if the variation resulted in an improvement [8].

Schwefel [83] later extended the algorithm to create multiple solution variations each iteration, also introducing the modern $(\mu + \lambda)$-ES and (μ, λ)-ES notations, indicating the size of the parent pool (μ) and the number of variations (λ). Using one parent and creating one variation, a $(1 + 1)$-ES is equivalent to the original algorithm, while setting μ to a value greater than one matches Rechenberg's extension [78]. With both $\mu > 1$ and $\lambda > 1$, significant design freedom is introduced, allowing the algorithm to deviate dramatically from the original structure.

The introduction of multiple parents and offspring creates the need for mechanisms to initialise the pool of solutions, reproduce or select parents, vary the offspring, reduce the number of variations and replace either some parent (for a $(\mu + \lambda)$-ES) or all parents (for a (μ, λ)-ES). The approaches used for any or all of these aspects affect the behaviour and suitability of the algorithm for different problems; there is no known configuration suitable for all possible problems, and it is generally accepted that such a configuration does not exist (the "no free lunch" theorems [97]).

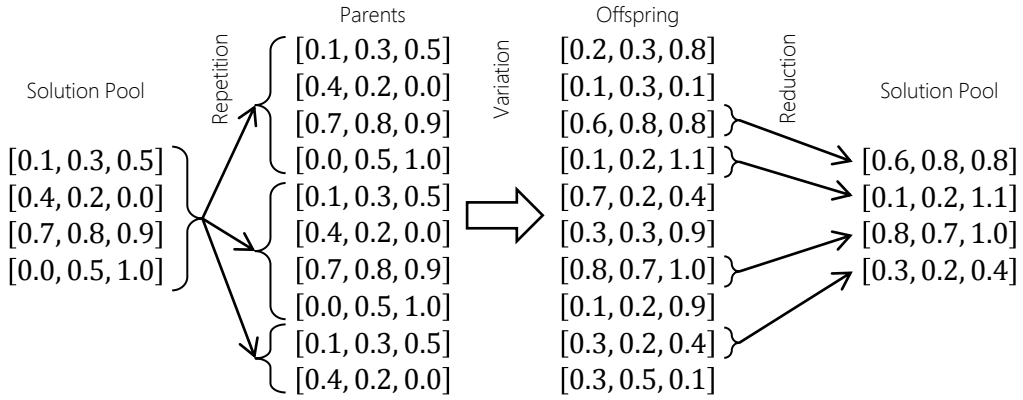

Figure 2.2: One iteration of a $(4, 10)$ -ES that reuses parents to create a larger pool of solutions.

As an illustrative example, consider a $(4, 10)$-ES using solutions represented by real-valued vectors with three components. At the beginning of each iteration, there will be a pool of four potential solutions. Initially, these are created by randomly selecting values between 0.0 and 1.0, but later they will be the result of the ES algorithm. In order to produce 10 offspring from these four parents, the group of four is repeated until 10 are obtained. (Direct repetition is one of a number of choices for obtaining a sufficiently sized group, but it is a common choice for ES with $\mu < \lambda$.) For each vector in the group of parents, an offspring is created by adding a random value to each component. This value is selected from a normal distribution with mean of zero and a variance of 0.25, giving an approximate 95 per cent likelihood of the value being between -1.0 and 1.0. Each of the ten variations is tested and the best four are retained. This process is illustrated by Figure 2.2.

In this example, the algorithm may be modified to achieve various performance characteristics. For example, parents may be reselected with a uniform probability or proportionally to their quality, rather than simply repeating each in turn. The scale of the random variation may be reduced or increased independently from the width of the distribution, resulting in more changes overall or increasing the likelihood of larger changes. Distributions other than the Gaussian may be used, since the existence of random variations is sufficient for the algorithm to make progress. A hard limit may be applied to vector components, preventing them from exceeding certain bounds, and these limits may be different for each component. Reducing the ten variations back to the pool of four may be deterministic, as in the example, or may be selected with some probability related to their quality. There are incalculable combinations of parameters that may be used.

A common extension to ES is the use of parameter adaptation, particularly of the random number distribution. Rather than fixing the standard deviation, it is allowed to vary each iteration for every solution at once, each individually or each component of a solution. Rechenberg demonstrated the ability to calculate an

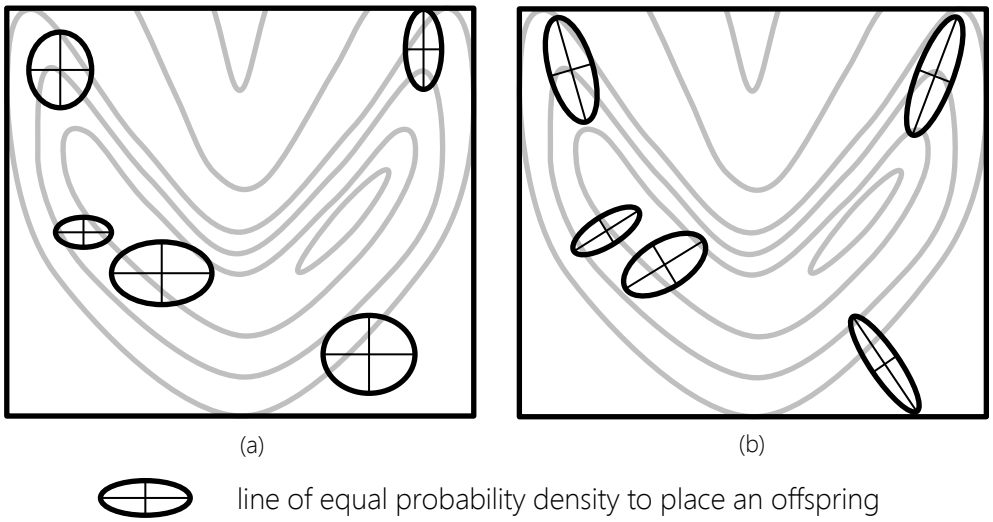

Figure 2.3: Per-component distributions (a) without and (b) with angles, to allow non-orthogonal variations (adapted from [8]).

optimum variance at each step for a given problem and a particular parent [78], though the use of self-adaptive parameters is generally preferred for ES [7, 8, 83].

Varying levels of complexity in parameter adaptation have been used with success. The simplest schemes scale the distribution variance based on the rate of successful variations. Those that are more complex introduce a "strategy parameter," which specifies the variance for each component. Values for this parameter are associated with the individuals to which they are applied and varied stochastically, the premise being that 'good' variances will produce successful variations and avoid elimination. Other approaches include rotated or correlated mutations, allowing strategy parameters to direct variations in a direction not orthogonal to the vector components, for example, as in Figure 2.3. [8, 83]

Ignoring the range of potential variations, a general ES consists of three stages: an initial collection of n real-valued vectors is either repeated or combined into a set of parents; each parent is varied to produce a child; and the set of children, or the union of children with the original set, is reduced to the best n solutions. The real-valued vector representation is considered canonical, though the use of alternative solution types with ES is also possible.

2.1.2 Evolutionary Programming

Evolutionary Programming (EP) was one of the earliest EAs, published in 1966 by Lawrence Fogel in *Artificial intelligence through simulated evolution* [34]. The intent was to replicate intelligent behaviour in a machine so that it could predict its environment and react appropriately to achieve a specific goal. In early experiments, the environment was a sequence of symbols from a finite set and finite-state ma-

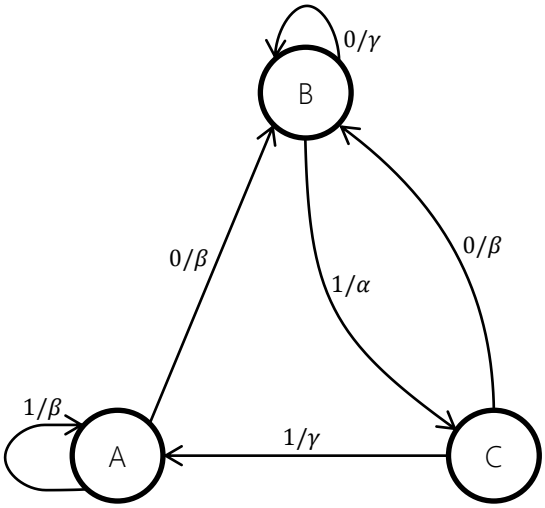

Figure 2.4: A finite-state machine (adapted from [33]).

chines (FSMs) were used to determine the reaction to each new symbol. Finite-state machines are intuitively suitable for such a task due to the memory implicit in each action's dependence on the current state. For example, the finite-state machine in Figure 2.4 consists of states A, B and C and exists in an environment with symbols 0 and 1; each response depends on the current state and is one of α, β or γ. Such a machine could be evaluated for quality by providing a sequence of symbols and measuring its predictive capability. While any particular measure depends on the specific type of symbol, a machine with better expectation of future changes in the environment generally produces a better "payoff." [33]

A finite-state machine can be designed manually for well-understood environments, in the same way that ES is unnecessary in the presence of a reliable theoretical model. However, for complex behaviours with many interacting parameters, an automatic method of design discovery greatly improves the range of applicability. Fogel's method used iterative variation of a collection of different machines. Each machine was changed using one or more out of five possible mutations: changing an output symbol (α, β or γ from Figure 2.4), changing a state transition, adding a state, deleting a state or changing the initial state. Some mutations are not valid for machines with only one state, while those that were had a uniform probability of application. Creating one variation for each machine produced a total of twice the original number; discarding the worse performing fifty per cent maintained a constant count over multiple iterations.

One early experiment sought to predict whether a natural number was prime, based on the prime-ness of all preceding natural numbers [33]. For this experiment, the environment was a sequence [0, 1, 1, 0, 1, 0, 1, ...] where 1 represented that the index was a prime number (in the short sequence shown, two, three, five

and seven). The output symbol would be either zero or one with the payoff being the number of correct predictions for the sequence up to a predetermined number. A size penalty was applied to prevent the generation of machines with as many states as there were symbols in the environment; keeping a full list of the correct answers results in good historical performance but poor predictive ability. Machines generated using EP on the first two hundred symbols (that is, numbers one through two hundred inclusive) predicted 78 per cent of the prime numbers correctly with only four states. After adjusting the payoff calculation to better reward prediction of rare events (for large ranges, always predicting non-prime gives a high success rate) the machines correctly predicted ninety-five of the first hundred primes, with 104 false alarms up to the number 547. Since the algorithm for primes cannot be represented by a finite-state machine, such a result is considered remarkable.

From the late 1980s, EP was adapted to be applicable to a variety of problems. Representations were not restricted to finite-state machines but specifically designed for particular problems, though the approach of producing a single variation for each solution was normally retained. Because changing representation necessitates changing the variation operation, EP has performed well on a range of problems, perhaps the strongest evidence that the other 'standard' EAs specify algorithms that are too constrained for general application. When using a real-valued vector representation, EP appears very similar to ES, though the latter deemphasises the importance of parent-child knowledge transfer.

A general EP algorithm consists of two steps: for each solution create one variation, and then reduce the total pool to the original size keeping better solutions. Variation schemes are representation-dependent and must be designed and tuned for a specific problem class. Such schemes often include steps to combine multiple solutions, as in ES, as well as making purely random changes. The reduction mechanism is independent of the problem provided a quality measurement is well defined, and selecting 'survivors' probabilistically rather than deterministically provides improved performance over keeping exclusively better solutions [39].

2.1.3 Genetic Algorithms

Genetic Algorithms (GA) is arguably the most influential of the three originating algorithms, particularly in terms of the terminology now in common use throughout the field. The earliest description appears to be Alex Fraser's stochastic digital simulation of nature in 1957 [36, 37], though the use of the technique as a problem solver was recognised by Hans-Joachim Bremermann [11] and formalised by John Holland throughout the 1960s and 1970s [46, 47]. Kenneth De Jong published his PhD thesis under the supervision of Holland in 1975, focusing on the "genetic adaptive model's" use in, and relevance to, search and optimisation [16]. The general

adaptation framework used by Holland and De Jong supports ES and EP as well as other algorithms, though the genetic adaptive plan that later became GA is the more widely known contribution.

Central to Holland's formulation of adaption is the notion of competition between alternative solutions to a problem. In order to have competition, some comparable measure of performance is necessary; this measure needs to be a function of the solution and the environment or problem to which it is applied. Adaptation is a strategy for using known solutions to derive some information about the environment, and then using that information to generate better performing solutions.

GA is a strategy based on a simulation of a population of biological organisms, modelled at the genetic level. A single time step in the algorithm represents one generation in this population, the time in which organisms mate and produce a replacement group of individuals. In contrast to ES and EP, variations are typically guaranteed to survive into the next generation; the quality of an individual determines the likelihood of selection as a parent, rather than its immediate survival. [16]

Solutions are represented as strings of genes, where each gene can take one of a predefined set of values or "alleles." The number of genes and the values each may take are determined by the problem, with a preference for more genes of lower arity and hence a larger search space. Mapping from genes to an individual's 'physical' manifestation in the problem domain must be explicitly defined, although this can be subsumed into the solution evaluation and comparison. Canonical GA uses binary strings—binary being the simplest useful gene—and interprets subsequences of bits to determine the value represented. Unlike ES, the underlying genetic representation in GA is explicit, rather than assumed and modelled through biased random number generation.

"Crossover" is used as the main source of variation; the purely random changes, "mutation," as used in ES and EP are relegated to a "background operator" [16]. As in nature, the genetic information present in a particular generation is predominantly a mixture of what existed in the previous generation. The crossover operator emulates this by exchanging randomly selected subsequences of two individuals to create one or two offspring, as shown in Figure 2.5. Many different approaches to crossover exist with varying numbers of parents, numbers of "crossover points," blending or additive behaviours, segment reordering and others described throughout the literature [7, 16, 46].

Crossover alone is the unable to introduce gene values that do not exist in the current population. Values may disappear due to selection pressures or may never have existed in the initial population. To mitigate this limitation, a very low frequency mutation operator is applied. This operator is intended to maintain a high similarity between the individual and its mutant by randomly replacing a small

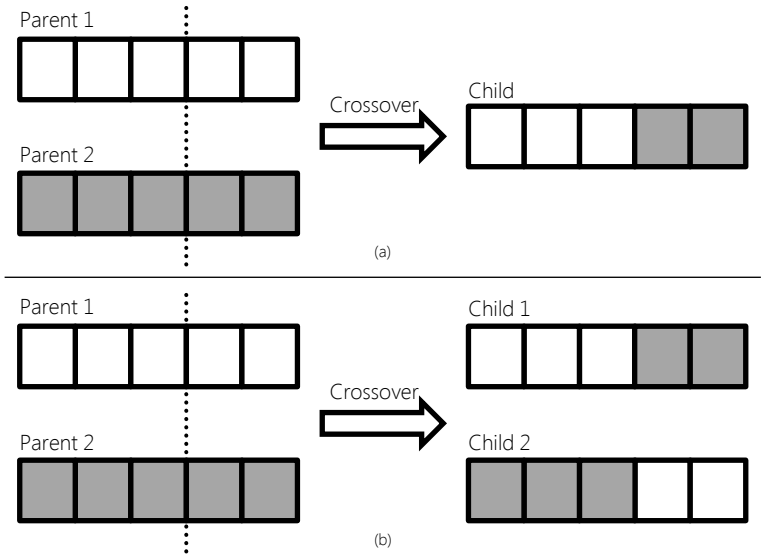

Figure 2.5: Diagrammatic representation of crossover in GA creating (a) one or (b) two new offspring.

number of genes with new values (assuming the decoding method used assigns an equivalent weight to each gene regardless of position [7]). Early suggestions ranged between a 0.1 to 1 per cent chance of mutating any particular individual, compared to a very high and often guaranteed likelihood of performing crossover [16, 43, 46].

Repeating these two variation operations provides a computationally efficient method for sampling points within a large search space. However, a sampling plan without any pressure towards creating better solutions is not useful. The pressure in GA is derived from deliberately biasing the selection of parents, applying the observation that organisms less suited to their environment are less likely to reproduce. Suitability in GA is called "fitness" and is typically a positive real number with larger numbers associated with fitter individuals. Fitness values are used when selecting to change the distribution of a population to include a higher proportion of fit individuals.

Multiple approaches to selection in GA have been established. The original approach was to normalise fitness values and treat them as selection probabilities, known as fitness-proportional selection [46]. Normalisation requires positive fitness values where more positive represents more fit ("maximising"), which is why this convention is preferred. Since scaling and shifting may be unsuitable for achieving this constraint, other selection techniques use a probability proportional to the number of less fit individuals; operators that compare to all other individuals are known as rank-proportional, while those that compare to a random subset are tournament selectors [7].

A standard GA generation begins with the selection of parents from the existing population. Depending on the crossover operation, there are either as many parents

as members of the population or double the amount. Typically 90 to 100 per cent of the parents are combined using crossover and mutation is then applied to the new individuals to produce the next population. In contrast to ES and EP, the reduction stage in GA is at the beginning of the cycle; as long as the number of offspring does not exceed the desired population size, there is no need for a final reduction step. Some forms of GA retain a subset of the original population, bypassing variation and allowing the individuals to compete with the newly created ones; algorithms retaining few of the most-fit individuals are said to be "elitist," while those that keep most or all are "overlapping" or "steady-state" algorithms, depending on the design [7].

2.1.4 Other Algorithms

As well as those in the previous sections, there are algorithms that were either developed as part of or later included into the EC field. Genetic Programming (GP), popularised by John Koza [54], bears great similarity to GA applied to a different solution representation; the names and intuitive behaviours of the operators used are identical, despite differences in implementation. Similarly, algorithms such as Grammatical Evolution [70], Cartesian Genetic Programming [65] and Linear Genetic Programming [10] use operators comparable to GA's but with different representations. Differential Evolution (DE) [76, 89] uses a combination of selection, crossover, mutation and elitism to target real-valued optimisation problems. Algorithms such as CCGA [75] and more than twenty others [3] consist of multiple collaborating (either coevolutionary or parallel) instances of other algorithms, commonly variations of GA. The area of swarm intelligence includes algorithms such as Ant Colony Optimisation [24] and Particle Swarm Optimisation (PSO) [52] which, while usually not considered to be EAs, have similar population-based iterative structures and are often applied to similar problems.

This list is far from exhaustive; there are hundreds of EC algorithms, each with a distinct name, body of research and set of problems for which they are better suited. Inspiration for these algorithms can typically be traced back to one of ES, EP, GA or another natural metaphor. Unfortunately, these origins are often used discriminatorily, with research work frequently able to be applied to a broader category of algorithms than those at which it is explicitly targeted. These issues also arise when designs are linked to one of ES, EP or GA; the following section discusses the implications of this in detail.

2.2 Algorithm Cliques

Despite the similarities, ES, EP and GA were considered distinct algorithms for many decades with separate conferences and journals. Discoveries, experimental

results, design concepts and software implementations often reinforced the assumption that their independent discovery implied some incompatibility between these algorithms. Despite amalgamation of the field into EC, attempts to treat ES, EP and GA as variations of a single class of algorithm have met with limited success and much work in the field continues to identify as relating to only one of them.

This section considers the issues that result from the assumption of incompatibility between algorithms in the context of research and experimental work, algorithm design and development, and software implementations. Works cited as displaying issues are intended as illustrative rather than personal criticism; they are examples of publications that support the view that EAs are not efficiently or unambiguously described as a derivation of another algorithm such as ES, EP or GA.

2.2.1 Segregation in Research

Since their inception, many improved versions of the three originating algorithms have been created, implemented, published and tuned for various applications. However, much of this work was originally developed for only one of the algorithms and later independently duplicated for the others, particularly throughout the 1980s when function optimisation became a more important focus than biomimicry. Only since the early 1990s have ES, EP and GA really benefited from sharing in each other's work. Thorough histories of EAs through this period can be found in [19] and [32].

One indicator of the growing collaboration between these algorithms comes from conference amalgamations. Throughout the 1990s, conferences such as the *International Conference on Genetic Algorithms*, *Evolutionary Programming* and *Genetic Programming* merged into broader EC conferences, such as the *Genetic and Evolutionary Computation Conference* (GECCO) and the *Congress on Evolutionary Computation* (CEC). This indicates a level of acceptance that the algorithms are fundamentally related; however, even within these conferences, contributions are streamed based on algorithm 'ancestry.' While streaming by algorithm is a convenient and often orthogonal categorisation, it is not conducive to exposing ideas and developments that span multiple applications. Worse, such streaming encourages new research to target a specific algorithm, which reinforces the orthogonality of the category.

Despite the practical separation, there is recognition of the distinction being purely theoretical. Fogel identifies that ES, EP and GA have blended to the point where the "practical utility" of the terms is minimal [33], while De Jong has been actively encouraging unification for decades [18–21]. However, despite an apparent acceptance of the central underlying algorithm class, the tendency to adhere to historical boundaries is still pervasive within the field. For example, the predisposition

to select an entire algorithm for a problem, particularly by those not familiar with the field, is an ongoing effect of the strong identity of the original algorithms [20,53]. This may simply result from an unawareness of the ability to tailor an algorithm to the problem being targeted, or the plausible—though incorrect—assumption that EAs have simple and universal performance characteristics, in the same way that deterministic algorithms are often compared using big-O expressions [66].

Because each algorithm supports so many parameter combinations, as well as canonical descriptions implicitly being targeted to particular problem characteristics, simply deciding to 'use a GA' is practically always deciding to use an inefficient approach. Unfortunately, the entry point to the EC field is typically through a particular algorithm, due to the lack of any structured, problem-centric approach to design [20]. Attempting to avoid this issue, introductory textbooks by Eiben and Smith [31] and De Jong [19] illustrate and derive EAs from common features such as reproduction, crossover and mutation, and subsequently describe specific algorithms in terms of these.

An improved situation for the EC field would see less emphasis on algorithms and encourage the combination of components based on an understanding of their effect on the search. Much work has been conducted into the effects of the operators of particular algorithms, such as recombination [66, 71, 86], though many experiments simply vary an operator within a particular algorithm rather than evaluating its effectiveness or behaviour independently. The resulting observations then describe a feature of an algorithm on a given problem without necessarily controlling for other components or problem characteristics.

The problem-centric view of design has gained traction in the field, observable in De Jong's "unified approach" [19], parameter control and tuning work (both independently and collaboratively) by Eiben and Smit [29, 84], Moraglio's efforts to design problem-specific crossover operators [66], Aydt et al.'s algorithm-independent representation [6] and numerous others. Reducing the importance of the entire algorithm in favour of general theories, problem characterisation and structured operator design provides a more robust scientific platform on which to work and allows research to better target specific hypotheses and outcomes.

2.2.2 Segregation in Design

The preoccupation with distinct algorithms is reflected in the development of derivations of existing algorithms. While inspiration is useful and necessary to the creation of new approaches, many novel ideas suffer from tight coupling to the early descriptions of one of ES, EP or GA. Rank-based selection methods (as opposed to those based on absolute or scaled fitness) were unappreciated within GA research for years because they appeared to be inconsistent with the existing schema theorem,

despite being standard for ES and EP [7]. While taking inspiration from existing algorithms is entirely acceptable, a valid concern can be raised about the extent to which unnecessary constraints are retained.

For example, canonical GA necessitates that fitness is represented by a single real value. Research into multi-objective problems (such as [79]) has shown that this is undesirable, and while many actively justify using this convention, its use by default persists even into recent algorithm specifications [59, 67]. A simple numeric fitness can be overly limiting, and a more flexible definition provides a better abstraction in the general case. For specific theoretical analyses, such as [66], explicitly defining fitness values as real numbers may be necessary or convenient, though the application of such analysis is then restricted to those applications where the constraint is valid. Many useful applications require higher complexity fitness, but when a designer believes they are limited to a single real value, they will fall back on a clumsy device such as weighted sums or a simplified model of the problem.

A greater issue than fitness representation is the residual structure of another algorithm. Using crossover followed by mutation or representing solutions in binary just because GA was the inspiration is lazy design—copying, rather than inspiring. Selecting a good representation requires an understanding of the nature of the problem being solved and selecting good operators further depends on the chosen representation. Assumptions proven for a different type of problem cannot necessarily be transferred to another application [66]. The no free lunch theorems support this: in order to achieve better than average performance on a particular problem, an algorithm must be specialised for that problem [97]. The general lesson is that a representation that is used by one algorithm, no matter how successful, cannot be seen as an endorsement of that representation for all problems.

Comparisons to the originating algorithms risk implying a greater association than may be intended. For example, early presentations of PSO state that "the adjustment … by the particle swarm optimizer is conceptually similar to the *crossover* operation utilized by genetic algorithms" [52], despite the availability of (and earlier reference to) more appropriate analogies. The metaphorical benefits of such an association appear limited to justifying PSO as an EA, which given the proper historical context was probably necessary, but it is not a simile that aids in the understanding or use of the algorithm.

The issue of structure can also be seen in existing abstractions and interpretations of ES, EP and GA, particularly those in published software libraries. These abstractions allow the substitution of particular operators; for example, GA may use an alternative crossover operator, though changes that are more dramatic typically require the design of a 'new' algorithm. Frameworks with wide use tend to include significant lists of operators and conventions for implementing new ones, though

few provide sufficient abstraction to change a GA into something different without library-level programming. Constructing a unique algorithm represents progress and novelty within the field, and yet the first design decision is often "which algorithm should I base mine on?"

Moraglio's apt response is that a "more rational classification of evolutionary algorithms would be based on the properties of the solution space being actually explored" [66, p. 12]. Such an approach is relatively unsupported by whole-algorithm design work and requires a deep understanding of the field, existing approaches, algorithm and software design, creativity and imagination. While there is some value in determining and understanding the performance of particular algorithms on particular problems, recognition that both problem and algorithm contribute to such results is not common and the use of this fact to design more appropriate benchmarks is even rarer [48].

2.2.3 Segregation in Implementation

As mentioned in Section 2.2.2, software implementations are often targeted at a particular algorithm or algorithms. Code reuse is a perpetual concern throughout software engineering [85], and yet a recent survey suggests that a significant proportion of EC researchers believe they are "first to release code in [their] area" and there is little "other code that might substitute" [88]. While it is likely that novel research has involved writing some code that is not already available, there is evidence suggesting that researchers actually write a majority of their code themselves [30, 53]. Some reasons suggested in [53] for this bias towards creation rather than reuse include a low perceived implementation difficulty, bad past experiences with others' code and an enjoyment of programming. Other possibilities are that innovation encourages a perception that existing code is not useful and that there is a sense of illegitimacy in publishing work based on code not written entirely by the named authors.

When software is written for a single purpose, and particularly when that purpose is to produce a single result set, it is less likely to be thoroughly tested and verified against requirements other than those of the immediate problem [53, 88]. Specifically targeting the algorithm currently under investigation means that even the original developer may have difficulty reusing components of their code for future work. Libraries with wide use are often more flexible and reliable in this regard, though the list of widely used libraries for EC is short. ECJ [60] is one library with significant usage but is most accessible as a collection of parameterised algorithms rather than a set of reusable algorithm components. EO (Evolvable Objects) [42] uses a composition-based design based on operators, but still encourages full encapsulation of an algorithm in its own class, limiting modifiability. A detailed survey of

these libraries is deferred to Chapter 5, but most of the other libraries reviewed were either not actively maintained (updated in the last twelve months), were highly specific (targeting at most two specific algorithms) or are apparently unused by anyone beyond the original developer.

Parameterised algorithm implementations allow ES, EP and GA to be created by substituting operators without changing the fundamental structure. These approaches normally require a user to choose selection, crossover and mutation operations. Such implementations acknowledge the similarities between the algorithms, but treat them as a literal fact and provide a minimum level of flexibility. The ability to restructure the algorithm—even something as trivial as performing mutation prior to crossover—is rarely provided or considered as an option. The limited potential sometimes supports implementations in providing other features, such as distributed processing [5], GPU processing [62], network structures [98], statistical analysis [4, 95] and easy specifications [14]. However, the range of available implementations for EAs, combined with most providing no more than one significant feature beyond basic algorithms, continues to prevent any single piece of software becoming standardised within the EC field.

The need for a standard software package for the field has been raised many times in the past, whether by those seeking to improve reproducibility [29, 30, 48] or to promote their own development efforts [4, 14, 28, 38], with few resounding successes. This is largely due to the established distinction between algorithms based on ancestry. With a complete redefinition of EAs, a class of algorithms can be created to support unification and problem-centric design. Furthermore, this will give practitioners a useful model for research, development, communication, presentation and related work.

2.3 'Inventing' algorithms for optimisation

In order to provide a useful abstraction for EAs, many current understandings must be reviewed and revised. Solution representations and behaviours need to be reduced to the simplest specification necessary to produce algorithms with optimising characteristics without restricting generality.[3] Specifying "crossover" or "real-valued vectors" is far too limiting, both in terms of what can be expressed and what designers can imagine. This approach is similar to that of De Jong [18], obtaining a subtly different result for the same purpose of unification.

The omission of biological inspiration is deliberate, but is not intended as a rejection of its relevance. Many useful algorithms have been explicitly inspired

[3] Applications such as search and learning can be viewed as optimisation problems consistent with our definition; for simplicity, the term "optimisation" is used as a synonym for "optimisation, search, learning and other applications that may be treated as optimisation" throughout this work.

by observations in genetics, materials science or animal communities. However, embedding such disparate ideas in a single abstraction is complicated and easily avoided at the cost (or benefit) of a lack of explicit biomimicry. Specific algorithms are not forced to surrender their inspiration and analogy will no doubt remain central to the design and justification of novel approaches; common structural terminology allows metaphorical terminology to be more clearly defined and used.

To approach a fundamental definition of EAs, this section makes full use of hindsight while avoiding the burden of history to 'invent' a class of algorithms suitable for solving complex optimisation problems. The proposed class is, or could be, what is currently understood to be EAs, but developing the general algorithm from scratch eliminates (or at least mitigates) potential misunderstandings due to overloaded and inherited terminology. Mathematical and algorithmic notations are avoided in favour of extended description; the purpose is to investigate design and intent rather than mechanical verifiability.

2.3.1 What are we looking for?

Computation is fundamentally concerned with solving two problems: given a function f and values x, what is $f(x)$; and, given a function f and result y, what values for x fulfil the equality $y = f(x)$. A function in this context is far broader than a mathematical expression, despite the notation, though many may be expressed in a mathematical form. Examples of functions include "what is the sum of these numbers?", "how far does one travel when visiting these cities in this order?" or "what is the speed and power consumption of this electronic circuit?". For all cases, the function uses known methods to determine answers to a question.

However, even given a well-defined function, the reversal of this function is rarely straightforward. While many arithmetic functions can be inverted, the same cannot be said of functions more generally. Inverting the descriptions of the above examples is easy—"which numbers sum to this total?", "which order should we visit these cities to have travelled this distance?" and "which electronic circuit has these speed and power characteristics?"—but the function itself is not. Many discussions of black-box versus white-box functions essentially reach the same conclusion, though the result here is subtly different, in that a black-box function has a transform that cannot be seen. The focus here is on transforms that cannot be inverted.

This is a similar problem to those observed in engineering control systems: given some system and controllable inputs, how can it be ensured that the output behaves in the desired way? In the engineering case, a complex or dynamic external environment is a typical cause for an unpredictable output from an otherwise well understood and predictable system. However, for optimisation problems, high dimensionality and interactions between parameters produce systems with responses

that are difficult or impossible to predict. For example, while it is easy to calculate the distance travelled when visiting each of n cities in a given order, selecting an order to visit n given cities to travel a particular distance is intractable. This problem has $n-1$ variables, where the contribution of each to the total distance depends on the value of another. The resulting interactions mean that for any significant value of n, it is difficult to manually infer the visitation order required for a specific result.

(As a brief aside, De Jong argued in the provocatively titled *Genetic Algorithms are NOT Function Optimizers* [17] that equating the two is incorrect, artificial and unhelpful. This is entirely true concerning the behaviour and analysis of the algorithm when compared to dedicated function optimisation algorithms; however, the title should not be interpreted as discouraging the application of EAs to function optimisation. Rather, De Jong encourages an interpretation of the algorithm as a sequential decision-maker—choosing which set of input values to try next—that is entirely compatible with the approach developed here.)

In order for the derivation and construction of engineering control systems to be achievable, a number of simplifying assumptions are necessary. The most common of these are the existence of some error function (the ability to quantify how good particular inputs are), linearity (changes in the output are directly related to changes in the input) and time-invariance (a given input has the same effect at all times). Similar, though less restrictive, assumptions are also useful here. Comparability between output values is essential in order to infer rankings for inputs—a better input is one that produces a better output—otherwise improvements towards optimality cannot be observed. Binary (that is, one of only two possibilities), low-cardinality discrete and symbolic outputs are generally unsuitable since they are less orderable, though a separate error or confidence function may be useful. It is convenient to assume that functions are continuous, such that small changes to inputs produce small changes in the outputs and hence good inputs are similar to other good inputs, though this is not a strict requirement. Finally, assuming time-invariance is a handy simplification, and is more often valid for optimisation than control engineering, though problems may include time as an independent and uncontrollable variable.

The algorithms we are concerned with here are required to discover a set of system inputs that are in some way optimal. Optimality is not a universal characteristic, but one that must exist in any system where results are qualitatively comparable. When specifying our class of algorithms, we restrict their application to optimisable problems, that is, those with one or more optima.

2.3.2 Where do we look?

Since optimality is the primary attribute we are interested in, it is logical to consider existing approaches from other fields. Control systems are very similar, in that they typically attempt to match an output signal to a known reference, the optimal case being where the output matches the reference exactly. For example, a power supply has an internal signal that is the desired voltage, though at a much lower power level. The regulator uses feedback from its own output to determine how to adjust the input voltage to the transformer. Similar feedback loop systems exist in many industrial processes.

For optimisation, there is less interest in matching a reference signal than in attempting to minimise or maximise an output relative to other possible outputs. Knowing the shortest path to visit all of the cities is more useful than just one path for an arbitrary distance. (Many problems, particularly function optimisation benchmarks, have known optima, though an algorithm designed to take advantage of this knowledge will exhibit reduced performance on other configurations.) Without a fixed reference, there is no meaningful proportional response to an error function as is often used in engineering control systems. Adjustments must be based solely on history, that is, how much improvement was observed from previous changes to the inputs. If a particular change improved the output, it is likely that an optimal solution is more similar to the new inputs than the previous ones.

One advantage optimisation has over a traditional control system is that problems are usually simulated. Computer simulations allow many possible adjustments to the input values to be tested without the reset times (or clean-up times, if the inputs were *really* bad) of physical systems.[4] Rather than making one adjustment and waiting for feedback, ten, one hundred or more can be tried in a reasonable amount of time, and many possible solutions can be 'live' without needing a separate physical system for each. This parallelisation allows the test-keep or test-discard cycle from ES and EP to be generalised as follows:

- From a pool of m 'live' solutions, generate n variants, expanding the number of possible solutions to $m + n$.
- Reduce the pool of $m + n$ candidates to a pool of size m.

Even within this simple framework, there are a huge range of possibilities. How are variants created? Is reduction based on solution quality or age? Must n be greater than m? Does the reduction necessarily keep only the best solutions or should some lesser solutions be retained? Is the implicit memory of previous attempts sufficient or is an explicit history beneficial? Can m and n change over time? Each of

[4] Worth noting is that many engineering control systems can also be simulated and designed using stochastic optimisation techniques. However, due to the complexity of real-world effects, these simulations are not as accurate or reliable as physical experiments.

these questions may be answered differently without departing from the fundamental concepts of exploring possible solutions (creating variations) and focusing on the better solutions (reducing the pool). The general process can be viewed as moving a window of interest within a space of potential solutions: a repeated expansion and narrowing process to find increasingly better inputs to an uninvertible function.

2.3.3 Is this an Evolutionary Algorithm?

Those familiar with EAs in their present form may have noticed a number of well-established concepts have been omitted or dealt with very briefly. These omissions are deliberate; some will be defined later (and may have been defined differently in other work), while some concepts are simply not as relevant to this model as they are historical artefacts of the development of EAs. "Fitness" was referred to without using the term, since it will be explicitly defined in the next chapter. "Problems" and problem landscapes will also be defined in the next chapter.

Representation is considered fundamental to many algorithms. However, in the model described above, representation is simply not relevant. In part, this is because the model is higher level than most specific algorithms, but also because representation-dependent concepts are subsumed into the creation of variations or determining the function output. Whether the underlying solutions use binary strings, executable programs or vector-based images has no bearing on the reduction step, and as will be shown in the following chapters, this abstraction supports a useful and generalisable model.

Stochastic processes have not been mandated at any point, which are normally considered a defining characteristic of EAs. This model and thesis use the original definition of the term evolution as "the gradual development of something," [72] rather than the natural process. For similar reasons, the terms "population," "offspring," "reproduction," "generation," "genotype" and "phenotype" were not used. These terms are due to the biological inspiration of some algorithms, particularly GA, rather than the fundamental structure of the algorithms. They are, however, convenient and generally accepted metaphors, and will be used later as descriptors of algorithm components.

2.4 Chapter Summary

This chapter has presented the motivation for this work in detail. Section 2.1 provided the historical background required to assess the current approach to EAs, Section 2.2 identified issues arising from adherence to the historical separation and Section 2.3 began approaching a general definition of EAs.

Despite their independent origins, the ES, EP and GA algorithms share many common characteristics with each other, engineering control systems and other opti-

misation algorithms. These similarities are often ignored, resulting in research work being unnecessarily linked to one but not the others, or taken literally, resulting in software libraries that lack the ability to change anything but a few parameters. Both of these approaches ignore that ES, EP and GA, along with other algorithms such as GP and DE, are specific instantiations of a more general class of algorithm.

Problems resulting from algorithm segregation include:

- Difficulty in sharing advances between distinct algorithms
- Limited guidance in designing (rather than tuning) an algorithm to suit the targeted problem
- Adherence to conventions for historical reasons rather than reassessing their suitability for a new context
- Replication of software development because of perceived or actual unsuitability for reuse

The definition of a general algorithm class enables the development of mitigations for these problems. Section 2.3 outlined the basis of an iterative algorithm for optimisation applications in preparation for Chapter 3, which expands it into a detailed model. The fundamental algorithm is based on incrementally improving a collection of potential solutions by repeatedly applying these two steps:

- Expand the collection of m solutions by creating n variations
- Reduce the $m + n$ solutions to a collection of size m

All of the algorithms discussed are shown to be instances of this EA in Chapter 3, as are other algorithms that are normally not considered EAs. Taking the view that an EA evolves solutions rather than simulating nature allows for a more inclusive class that encompasses algorithms where m and n are both 1, or where the "collection" is represented as a model or distribution rather than a set (for example, Estimation of Distribution Algorithms). Chapters 3–5 develop this algorithm class into a complete model and description language, while ensuring that existing work is supported, and Chapter 6 demonstrates the practicality and benefits of the approach with respect to software implementations, research experimentation and sharing of algorithm designs.

Chapter 3

Model

The previous chapter showed that early EAs demonstrate similarities in their structure and application to solving optimisation problems despite being independently developed. A well-defined, unified model should assist all users and developers of EAs with understanding and implementation. Students and researchers in particular benefit from a broader framework for learning, teaching and discussion. A suitable model should define the smallest set of components necessary to represent current algorithms, as well as being extensible for future developments. Clearly specified interactions and well-defined contracts allow reuse and sharing of algorithm components. This chapter finds the minimum set of components required to represent existing EAs and to provide a separation of concerns that simplifies description and implementation based on the structure of the 'reinvented' EA from the previous chapter. A representative algorithm selection is shown as compositions of these components to demonstrate the model is inclusive of existing work.

3.1 Defining what to solve

One of the limiting factors in the unification of EAs is the lack of a common model. Each originating algorithm has its own structure and terminology, which requires adaptation or translation in order to be applied to others. This chapter defines a new model and terminology (though many of the terms will be familiar) based on the problems and algorithms described in Section 2.3. The model can be shared consistently between existing EAs and applied to new algorithms that are yet to be created.

3.1.1 Problems

Defining the problem to solve is not a trivial exercise. A functional description may have a significant effect on the experimental setup, the complexity of the actual so-

lution or the size and comprehensiveness of the search space.[1] In order to provide a robust and useful model, a clear definition of the intended target problems is necessary, or else any effort risks suffering from a lack of direction and overgeneralisation.

The discussion in Section 2.3 about the necessary simplifying assumptions showed that EAs are intended for selecting inputs to a well-defined, but not necessarily well-understood, function. Algorithms that use representations other than simple numbers are not excluded; executable code, car parts and food recipes could all be used as function inputs, producing outputs of a set of state modifications, a vehicle and a cake, respectively. This definition is intended solely to delineate, name and specify that component which is to be 'solved.'

For the purposes of this work, this component is named the *problem*. It may be viewed as a map from input values to output values. This mapping is deterministic and many-to-one; a given set of input values will always produce the same output, though more than one set of input values may produce that output. For example, the input to a jigsaw puzzle may be a sequence of movements from a collection of unplaced pieces into one of the available slots. Each possible sequence maps to some final arrangement of the puzzle and some distinct sequences may map to the same arrangement; however, given a consistent initial state and process, the result will always be identical. Similarly, a function optimisation problem is a mathematical expression, with arguments forming the set of inputs and the result of the expression being the output. The expression x^2, to use the mathematical term, is not one-one, but a given value for x always produces the same result.

It may appear that this determinism constraint excludes dynamic problems. A dynamic problem intentionally produces different outputs for the same set of inputs to represent some real-world applications more accurately than with a static calculation. Dynamic problems can be treated as having an extra input in addition to the solution inputs, such as the current time. This input must be consistent throughout a set of calculations in order to generate comparable results, effectively producing a sequence of similar problems. Within each time-step, the problem is deterministic, while 'incrementing' time creates a new problem.

As a concrete example, the shortest time to travel to each of a number of locations in a city will vary from day to day. Road works, accidents, traffic and other closures mean that the same sequence of visits may take different times on different days. However, all travellers taking the same route at the same time will require approximately equal amounts of travelling time. The problem is deterministic at time t as well as at $t + \Delta t$, though the problem at $t + \Delta t$ is not the same as it was

[1] For example, despite $\left(x^3 - 6x^2 + 11x - 6\right)$ and $(x-1)(x-2)(x-3)$ being equivalent, the values of x that result in zero are considerably more obvious with the latter form.

at t. It is not logical to compare a trip starting at 9am with one starting at 2am: dynamic problems are distinct when their parameters differ.

An algorithm intended solely for static problems can safely assume that all changes in an output are due to its own changes to the inputs. However, an algorithm that is intended to be robust for dynamic problems cannot make this assumption. As a result, the algorithms have a different design emphasis, but entirely distinct models are not necessary.

3.1.2 Evaluators

Fitness represents the suitability of a set of input values for the problem to which they are applied. In trivial cases the output value from the problem may be a suitable metric, though it is more likely that suitability will require some combination of the output, the expected/desired result and other efficiency measures. Determination of the fitness of a set of inputs against a particular problem is conceptualised as an *evaluator*.

Unlike many other works, fitness here is not strictly defined as a real number. For a fitness definition to be useful it must provide a partial ordering across solutions, that is, comparing one fitness to another should result in 'fitter,' 'less fit' or 'equally fit,' where more-fit fitnesses imply that the associated input values produce a more desirable output. The actual output values are of no relevance to the EA beyond their contribution to fitness, and the actual fitness values are rarely of interest after solution ordering has been determined.

The usual aim of an EA is to find input values that result in the highest fitness, these being optimal by whatever quality measures are defined for the problem. In an extremely large search space, it is near impossible to know when the fittest values have been found. An assumption of continuous fitness is necessary to exclude significant areas of poor fitness based on sampling rather than enumeration. (Local smoothness is often beneficial, since this reduces the amount of sampling necessary, but all continuous functions will exhibit local smoothness when viewed at a small enough scale.) EAs are normally intended for problems where the size of a sufficient sample set is unknown or too large to enumerate. However, discontinuities in the fitness plane are deceptive rather than fatally problematic.

Because problems are deterministic, evaluators are referentially transparent.[2] Evaluations on previously evaluated solutions can be skipped because in the absence of a solution modification, the fitness is guaranteed not to change. However, this guarantee is not directly applicable in the case of dynamic problems. Dynamic problems typically define evaluators as parameterised against time or another vari-

[2]In programming, a function is referentially transparent if it can be replaced by its result without affecting the program's behaviour, which implies that for constant parameters its result does not change.

able; treating such parameters as part of an evaluator's identity or parameters, rather than its definition, allows referential integrity to be maintained. Effectively, an evaluator $f_t(x)$ is not equivalent to $f_{t+\Delta t}(x)$. Including a varying parameter as an independent variable is equivalent—$f(t,\ x)$ rather than $f_t(x)$—though the association between the parameter and the evaluator is less obvious; this model deliberately prefers treating independent parameters as part of the identity of the evaluator (that is, $f_t(x)$).

This does not restrict in any way the points within an algorithm step at which evaluations can occur. Intermediate solutions can be generated, evaluated and discarded freely because fitness can be treated as an immutable property of the solution itself (or more precisely, a property of the tuple of the solution and an evaluator).

An alternative approach is to allow evaluator results to change while maintaining their identity, meaning that $f(x)$ is no longer referentially transparent and could produce different values at t and $t+\Delta t$. This prevents (in the general case) the ability to cache fitnesses, but allows dynamic evaluators to more closely match current implementations (such as ECJ, which provides options for multiple re-evaluations [60]) and conceptualisations that have the problem varying completely independently from the algorithm. However, since fitness caching can often reduce execution time significantly and without user intervention, as well as the non-caching behaviour being obtainable where desired and the risk of subtle errors caused by mutable evaluators, this model considers an evaluator to always be a deterministic map from one set of solution parameters to one fitness value. Where the result of the mapping should change, an explicit change of evaluator is required.

3.2 How to 'have' a population

3.2.1 Individuals

In this model, each set of input values to a problem is identified as an *individual*. The representation of an individual is irrelevant to the definition of an algorithm, despite many existing algorithms specifying it as a distinguishing factor. For most applications, the only requirement is that individuals can be ordered: given two individuals, one can consistently be determined to be better or equivalent. The usual comparison attribute is fitness, discussed in Section 3.1.2, though *age* (the time since the individual was first observed) and similarity are sometimes useful traits.

An individual is identified by the input values it represents, which implies that each exists only once and may not be modified (just as the value of a particular number cannot be modified). This does not prevent the model from faithfully representing existing algorithms, as most variations to individuals can be implemented

with a substitution of one individual for another. The benefits of immutable data structures to understanding and reasoning about algorithms are well known [1]. Conceptualising individuals as immutable values simplifies aliasing—where multiple references are held to the same storage location—in contrast to models used by existing software frameworks that allow in-place modification of individuals. Typically intended as a performance optimisation, in-place modification of groups and individuals may improve execution time in some cases, particularly on systems with very limited memory, but always increases implementation complexity [1].

Individuals are the atomic (indivisible) data element in this model—all operations are performed using at least one individual. The layout and contents of an individual are deliberately unspecified, requiring operator implementations to actively support particular representations. Individuals are not explicitly identifiable by name, reference or index; they exist only as members of groups (Section 3.2.2) and streams (Section 3.2.3). Evaluating an individual determines its fitness value, and since individuals cannot change and evaluators are deterministic, evaluation is a mapping from one individual and one evaluator to precisely one fitness. Using a different evaluator means that the fitness value may change and is the intended use of evaluators.

3.2.2 Groups

Much of the power of EAs comes from their simultaneous use of multiple solutions. Parallel exploration followed by exploitative convergence is observable in useful EAs. Typical EAs contain a "population" of individuals from which "parents" are selected and "offspring" are created; the generalisation of this concept is a *group*.

A group is a list[3] containing a subset of all possible individuals. Groups are used to identify, store and associate individuals but provide no direct functionality. Multiple groups may contain the same individual simultaneously and each group may contain an individual multiple times; the *size* of the group includes repetitions and is always finite. Groups have an associated name and are immutable: they can be 'modified' only by creating a new group and changing the name associations.

Figure 3.1 shows three groups, with some individuals appearing in more than one and some appearing outside of any group. The main pool of individuals is stored in the `population` group, with `parents` containing a subset. The `offspring` group was generated by stochastically modifying some members of the `parents` group; since not every individual has changed, `offspring` shares some members with `parents`. Group names are for convenience and descriptiveness only; they have no influence on the behaviour or use.

[3] An array or ordered multiset; this is not an implementation requirement for a particular data structure.

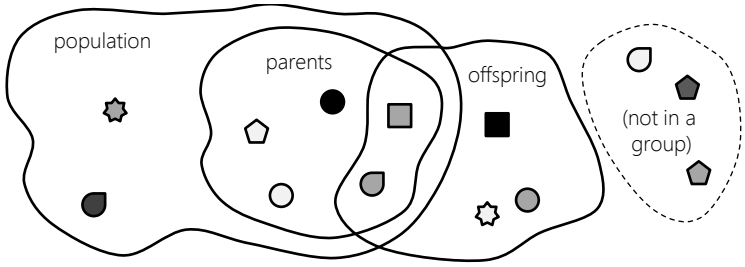

Figure 3.1: Three groups containing subsets of all known individuals.

Individuals cannot be directly accessed at the algorithm level—the appropriate mechanisms are described in Section 3.2.3—and all operations are applied to entire groups. A group of size one may be used to retain a single individual, for example, when treating the fittest individual separately from the rest. Individuals that are not members of groups cannot be accessed except through exploration; those that have never been part of a group are the unexplored area of the search space.

3.2.3 Streams

Algorithms are intended to be concerned solely with groups and their interactions rather than the behaviour of particular individuals. However, groups are merely storage mechanisms for individuals, which are the actual computational element. In order to define consistent interactions, the contents of a group need to be accessed in a general but efficient manner. *Streams* are temporary sequences of individuals that are produced as the result of applying *operators* (Section 3.3) to groups or other streams. Streams are "delayed lists" [1, p. 316] that may be queried to obtain ("take") the next individual. Groups may be implicitly converted to streams, and streams may be stored in a group by fully enumerating all individuals (or potentially, all remaining individuals) and explicitly creating a group to contain them.[4]

Operators are applied to a stream to make use of the individuals it provides. Streams are consumed on use and cannot be reused by another operation without either recreating the stream or by using a group as temporary storage. Groups implicitly treated as streams are not consumed and can be reused.

There is no requirement for a stream to be stored in memory, which allows a potentially infinite number of individuals to be made available. Streams of infinite length require a *partition* operator (see Section 3.3.3) to limit the group to a finite size.

For example, randomly selecting individuals from a group with replacement produces an infinite stream. The result of this cannot be stored in finite memory, nor

[4]The parallel with programming languages is that groups are named variables (sometimes called an *lvalue*) while streams are temporary expression results (sometimes called an *rvalue*). Variables (groups) can always be used as an expression (stream), but an expression must be evaluated before storing it into a variable.

does it need to be, and so partitioning is used to create a group containing only the first N individuals. Selecting individuals without replacement can never produce a stream with more individuals than the source: if the source stream was finite then the result will also be finite. Partitioning is then only needed to produce a group smaller than the original.

Both individuals and groups are persistent and continue to exist beyond the scope of a given operation. The 'global' scope in which groups exist is not intended to have the intricacy of modern programming languages—one name always represents the same group—which removes a source of complexity and subtle logic errors. Listing 3.1 shows a C++-style analogy to these scoping rules, specifically that each iteration accesses and updates a single set of groups.

Generators are special streams that provide infinite individuals. They are necessary to create initial groups and can produce randomly generated individuals or some subset of all potential individuals. Other uses of generators include replacement of invalid or corrupt individuals or as sources of diversity. For example, a generator `Random Real Values` may produce every number an infinite number of times in non-sequential order—effectively, a random number generator. Partitioning `Random Real Values` by storing only the first fifty values may provide an initial set of individuals for an algorithm.

3.3 How to 'improve' a population

3.3.1 Operators

Operators represent the two fundamental operations that may be performed on collections of data: *transformation* and *filtering* [9,93]. In this context, transformations represent the creation of one or more output streams by combining or modifying individuals. Input individuals are sourced from groups or a stream—the result of another operator. Operators are components, in that they have "been designed to be used in a compositional way together with other components" [68, p. 4], and composing operators using groups as storage points defines a network that traces the flow of individuals through an algorithm. This network is an oriented digraph without loops,[5] where each source of the graph is a generator or a group from the previous iteration and each sink is a group. Multiple sources and sinks are permitted.

The two fundamental operations are further divided to give five classes of operators, shown in Figure 3.2 with examples:

- *Stream operators*, which merge (see Section 3.3.2) and partition (see Section 3.3.3) streams,

[5]Strictly, loops are permitted, provided they begin from a group and incur a delay of one algorithm iteration. A perhaps more intuitive interpretation is that the contents of a group persist between iterations and those from the previous iteration may be used as sources.

```
int main()
{
    // "global" groups
    Group population;
    Group parents;
    Group offspring;

    // initialisation phase
    population = part(100, random_real(0.0, 1.0));

    // iteration phase - one hundred iterations
    for(int i = 0; i < 100; ++i)
    {
        // parents = part(100, tournament(2, 0.9, population))
        {
            Stream stream1 = population.asStream();
            Stream stream2 = tournament(2, 0.9, stream1);
            Stream stream3 = part(100, stream2);
            parents.replaceWith(stream3);
        }
        // offspring = mutate(0.1, crossover(0.9, shuffle(0.05, parents)))
        {
            Stream stream1 = parents.asStream();
            Stream stream2 = shuffle(0.05, stream1);
            Stream stream3 = crossover(0.9, stream2);
            Stream stream4 = mutate(0.1, stream3);
            offspring.replaceWith(stream4);
        }
        // population = part(100, sort_descend(merge(population, offspring)))
        {
            Stream stream1 = population.asStream();
            Stream stream2 = offspring.asStream();
            Stream stream3 = merge(stream1, stream2);
            Stream stream4 = sort_descend(stream3);
            Stream stream5 = part(100, stream4);
            population.replaceWith(stream5);
        }
    }
}
```

Listing 3.1: Analogy of scoping rules using C++.

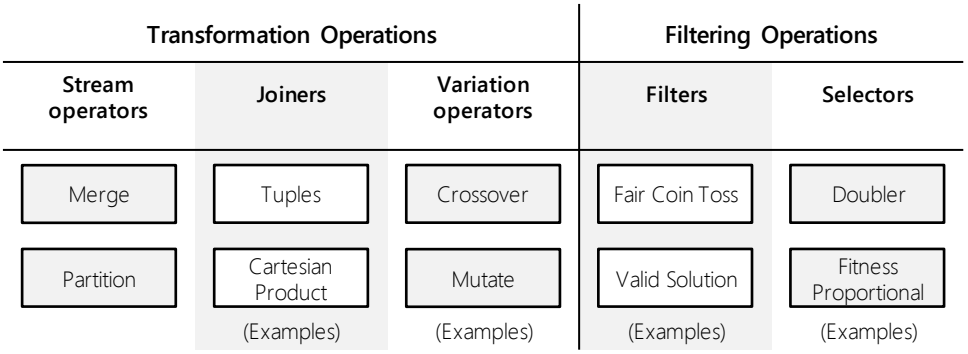

Figure 3.2: Operator classifications and examples.

- *Joiners*, which create associations between individuals in streams (see Section 3.3.4),
- *Filters*, which remove individuals from streams based on simple predicates (see Section 3.3.5),
- *Selectors*, which choose and order individuals from streams based on more complex mechanisms than filters (see Section 3.3.6); and
- *Variation operators*, which produce new individuals based on those in an existing stream (see Section 3.3.7).

These classes are orthogonal, such that knowing the class of operator allows behavioural assumptions to be made safely. For example, a filter can never produce a stream that is longer than its source and variation operators always maintain order, despite potentially changing, inserting or omitting individuals. EC already has a number of sub-classifications for variation operators, such as sexual versus asexual, though for this abstraction the greater level of distinction is not important. A general variation operator creates one or more individuals based on one or more individuals in the source; strictly defining the number of individuals involved is less important than guaranteeing preservation of order.

While it is possible to encapsulate an entire algorithm within a single operator, this is not the intended use of the model. A well-designed algorithm will abstract as little as possible into many operators, allowing the composition to define the algorithm: composing simple components promotes greater code reuse than using few monolithic operators [68].

The behaviour of an operator may be parameterised, but the class of the operator does not change. For example, a selection operator may act like a filter when certain parameters are specified but it is still classed as a selector, and a 'filter' that performs reordering with certain parameters is always a selector. A variation operator may not modify any individuals if its application rate is very low, but since it retains the potential to create modifications, it is classed as a variation operator, as are operators that aggregate an entire group into a single representative individual.

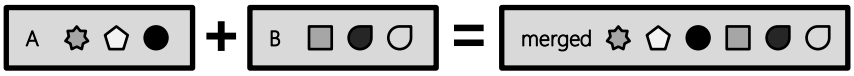

Figure 3.3: Merging streams `A` and `B` into group `merged`.

Despite the clear delineation of selection and variation into separate steps in many existing algorithms, where all selection must be performed before any variation, the two concepts often intermingle. For example, a variation phase may include implicit filtering of degenerate combinations or invalid results. In the model presented here, which does not enforce a strict structure, such tight coupling is unnecessary; a filtering operator can be included either before or after the variation operator. Further, multiple filters can be used to apply different variation operators to parts of the same group. The general nature of graph-style operator composition allows a wide range of novel algorithms to be described that are not possible with typical 'select–crossover–mutate–survive' structures.

3.3.2 Merging

The merge operator produces an output stream by concatenating multiple groups or streams. Merging is a stream operator that does not modify or reorder individuals. In the simplest case, applying the merge function to two groups produces a new group that contains all individuals from both. Since order is preserved, given streams $A = [1\ 2\ 3]$ and $B = [4\ 5\ 6]$, $\text{merge}\,(A, B) = [1\ 2\ 3\ 4\ 5\ 6]$ and $\text{merge}\,(B, A) = [4\ 5\ 6\ 1\ 2\ 3]$ (shown in Figure 3.3).

Merging is an operation that is not explicitly specified for many existing algorithms; rather, it is implied. For example, algorithms that include competition between parents and offspring implicitly merge the two groups before performing some type of selection.

While not theoretically relevant to most algorithms, the guarantee of order preservation allows operator composition to create complex expressions without the need to restate ordering assumptions. Consider a case where each individual in a group `parents` is mutated and stored in a group `offspring`. For each individual in `offspring`, the individual at a matching offset into `parents` is known to be the one on which it was based.

Order preservation also allows streams to be merged as well as groups. Streams do not support any sensible concept of a set-based union, but provided only the final stream is infinite, concatenation is well defined. For example, appending a generator or infinite selector to a group and partitioning can be used to ensure that the result is never shorter than the desired size. Without the guarantee of order, however, ensuring the infinitely long stream is not concatenated prior to finite-length groups would become complicated.

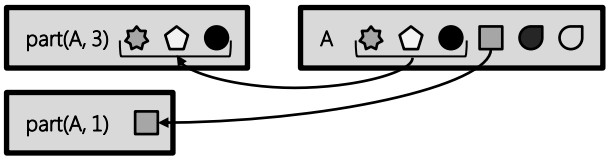

Figure 3.4: Partitioning stream `A` into two finite streams.

Figure 3.5: Joining groups `A` and `B` by index.

3.3.3 Partitioning

The partition operator takes a specified number of adjacent individuals from the front of the input stream. Repeated applications of the partition function to the same source stream continue taking individuals, but individuals can only be taken once. If the size requested exceeds the number of available individuals, all individuals are taken and the result is smaller than the request. If the size is omitted, every remaining individual is returned; if the source stream is infinite and the size is omitted, the resulting stream is also infinite. Infinite streams cannot be stored in groups.

Partitioning preserves order, such that an infinite stream $A = [10\ \ 11\ \ 12\ \ 13\ \ \ldots]$ is partitioned as $\text{part}\,(A, 3) = [10\ \ 11\ \ 12]$ and then $\text{part}\,(A, 1) = [13]$ (shown in Figure 3.4). The order of application is important, since elements of the stream may only be taken once.

Partitioning is required frequently in this model; at an algorithmic level, it is the only way to separate individuals for separate treatment. For example, an algorithm may require replacing one individual in a group `population` with one individual from a group `offspring`. Applying a partition operator to `population` twice, the first taking one element and the second taking the rest, produces a finite stream (the result of the second application) that is identical to `population` but missing the first individual. The first individual from `offspring` is obtained in a similar fashion; a subsequent merge produces the required group. (To remove a specific individual, rather than whichever happens to be located first, a selector can be used to reorder the stream.)

3.3.4 Joining

Merging allows multiple individuals to undergo the same operations as part of a coherent group, and variation operators may use adjacencies within a group to infer associations such as a pair of parents for recombination. However, a more general

mechanism for arbitrarily associating individuals is also required, particularly in order to handle algorithms with multiple distinct individual types.

For example, an algorithm that optimises coordinate values in a two-dimensional plane might evolve x and y coordinates independently in separate groups. If these two groups were to merge, there may be no way to determine which value was for which axis and no way to evaluate them as pairs. If the groups `Xs` contain [1 2] and `Ys` contain [3 4], then taking the Cartesian product of these should produce a group containing $[(1,3)\ (1,4)\ (2,3)\ (2,4)]$—the coordinates of interest—while the merge operator would produce $[1,2,3,4]$, which is not useful in this situation.

A *joiner* is an operator that takes one or more streams and produces a stream of *joined individuals*. Each joined individual is a tuple referencing one individual from each source in the order the sources were provided. References to the original streams or groups are not necessary, since immutability means that changes made to the joined group cannot affect any others. Joined individuals are treated identically to regular individuals: an evaluator may assess fitness, selectors and filters may be applied and groups of joined individuals may be created, merged and partitioned. A variation operator may be used to extract the component individuals and specially designed evaluators can distribute the joined individual's fitness among its components.

Joining comprises elements of merging and selection but in a form that cannot be specified without a separate operation. For example, where individuals in a `parents` group are known to have variations in `offspring` at matching positions, joining provides a mechanism to associate each varied individual with its unvaried counterpart, while merging creates a group that simply contains both. More precisely, *merging* group P with the varied group $\left[p' : p \in P\right]$ produces a group M such that

$$\forall p \left(p \in P \Rightarrow (p \in M) \bigwedge \left(p' \in M \right) \right) \tag{3.1}$$

while *joining* the same groups by position produces J such that

$$\forall p \left(p \in P \Rightarrow \langle p, p' \rangle \in J \right). \tag{3.2}$$

As another example, let group A contain [1 2 3], group B contain [4 5 6], tuple (x, y) be a joiner producing index-associated pairs and cart (x, y) be a joiner producing the Cartesian product. Applying tuple (A, B) gives $[(1,4)\ (2,5)\ (3,6)]$ (as shown in Figure 3.5) and cart (A, B) gives $[(1,4)\ (1,5)\ (1,6)\ (2,4)\ (2,5)\ (2,6)\ (3,4)\ (3,5)\ (3,6)]$. Either of these streams may be stored in a group, evaluated or otherwise manipulated as regular streams, though operator implementations need to be aware of any distinction between joined and non-joined individuals; by design, the model treats all groups identically and without regard to the individual representation.

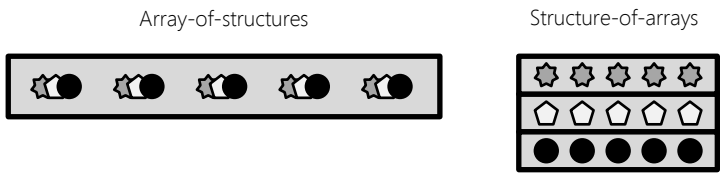

Figure 3.6: Array-of-structures and structure-of-arrays representations.

When a source group contains already joined individuals they are treated as regular individuals. For example, the result of applying tuple (x, y) to $[(1, 4)\ (2, 5)\ (3, 6)]$ and $[7\ 8\ 9]$ is $[((1, 4),\ 7)\ ((2, 5), 8)\ ((3, 6), 9)]$. Joining does not extend or modify the existing individuals in any way.

Joining allows the resulting group to be stored using either the array-of-structures or structure-of-arrays models, shown in Figure 3.6. One or the other of these representations will typically be more efficient depending on the operations to be applied. For example, array-of-structures suits coevolutionary algorithms that interpret joined individuals as distinct individuals, each having a distinct fitness and identity. In contrast, Differential Evolution associates a base vector with a target and two mutation vectors; structure-of-arrays provides simpler construction while allowing a traversal of the group's elements to represent the correct associations. Both representations are equivalent, and it is left to performance-conscious implementations to determine which to use.

Joiners may be limited, with a loss of convenience but not generality, to a single fixed operator that joins individuals in multiple streams by position. This restriction, however, complicates the specification of joiners that select individuals that are compatible in some way (as required in DE [76]). In these cases, selection is inseparable from joining; including the selection logic as part of the joiner simplifies expression and reasoning. As with selectors, joiners may be parameterised or restricted (on a per-operator basis) to finite streams or groups.

Co-evolutionary algorithms, such as CCGA-1 and CCGA-2 [75], require joining in order to evaluate a joined individual against the problem and distribute the fitness amongst the component individuals ("credit assignment"). When a joined individual is evaluated, the resulting fitness is associated with it and not the individuals that form the association. Such behaviour would interfere with most credit-assignment schemes, which have to account for repetitions and weighting between components. The general solution to credit-assignment is to model the evaluation as multiple problems: the primary problem evaluates a joined individual and assigns a single fitness, and credit-assigner evaluators assign fitness to the source individuals based on their contribution to members of the joined group. This maintains the rule of one fitness value per individual per evaluator, while allowing credit assignment to

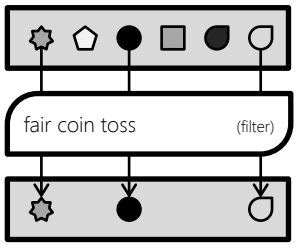

Figure 3.7: A filter that removes approximately half of the individuals from a stream.

account for individuals that form part of multiple joined individuals (which would not be possible if fitness was directly assigned to a member of a joined individual).

There is no requirement in either merging or joining for all individuals in a group to be of the same underlying representation or use the same evaluator. This allows very distinct individuals, for example, an executable program and a set of constants, to be joined and evaluated as a single element, provided the evaluator is aware of this representation. Merging presents a more complicated scenario; mixing different types of individuals implies some form of operator overloading or type aware implementations that are unlikely to be easily implemented or analysed.

3.3.5 Filtering

Filters apply a predicate to each individual in a stream, producing a substream that only includes those individuals satisfying the predicate. Filters can be applied to an infinite stream, in which case the returned stream must be assumed to be infinite. A filter never produces a stream that is longer than, in a different order to, or that contains individuals that were not in the original stream.

Filters implement functionality such as removing invalid individuals from groups or dividing groups based on an attribute of each individual. Predicates cannot refer to individuals from the source stream other than the one being tested, which is necessary to ensure that they may be applied to infinitely long streams. For example, the predicate "Are the components of this individual within the valid range?" can be applied to a finite or an infinite stream. Figure 3.7 shows a filter that randomly removes individuals with a probability of 50 per cent, which can be independently applied to each member of the source stream.

If a predicate needs to compare against individuals that may come later in the stream, a *selector* must be used instead. For example, "Is this the fittest individual in this group?" depends on knowing information about other individuals in the stream, which filters do not. However, "Is this individual fitter than x?" is a valid predicate for a filter, since it only depends on a specific constant value. ("Is this the fittest individual so far?" is also a suitable filter, though probably not particularly useful to an algorithm.)

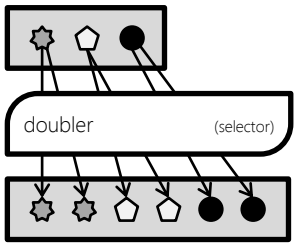

Figure 3.8: A selector that creates a stream containing each individual twice.

In essence, a filter is a validation or constraint satisfaction operation, in that it reduces a stream to ensure that some constraint is met by all individuals, while a selector is an algorithmic operator with its own non-trivial analysis that is independent from the complete algorithm. Software implementations of this model can use the distinction to improve execution performance and simplify coding requirements; stateless filters are safely parallelisable and require only the predicate to be specified, for example.

3.3.6 Selection

Selectors produce a stream by choosing individuals from a source stream. Selectors are not required to preserve order and the output stream may be longer than the source if individuals are chosen more than once. Unlike filters, selectors may cache the entire source stream to allow selections based on information about other individuals, such as selecting in order of fitness. Roulette-wheel style selection requires the sum of all fitness values to allow normalisation and must return individuals more than once in order to affect the resulting fitness distribution.

As an option, selectors may be suitable for use with infinite streams. Figure 3.8 shows a selector that expands a stream by returning each individual twice. Because the length of the stream is increased, even though order is maintained and no aggregate information is required, this operation is classed as a selector and not a filter. Selectors may return infinite streams even if the source is finite.

Although selectors can produce identical results to filters, where possible, the use of filters is preferred since the tighter restrictions make them easier to understand. Filters only ever produce subsequences of the source, while selectors may perform complicated actions to determine which individuals are returned. Common tournament selection implementations (for example [7, 31]) choose a number of individuals from a source group (commonly two or seven) and return the fittest of this pool. This process is not a simple predicate and necessitates a thorough description and analysis of its behaviour. In contrast, determining whether the components of an individual are within a constant range can be expressed as a simple predicate; classifying this as a filter effectively communicates both the behaviour and the intent

to a reader. For pure theoretical work, filters allow invariants to be more easily determined, whereas selectors introduce a source of complexity into the analysis.

3.3.7 Variation

Variation operators produce a stream of individuals based on those in a source stream. Since individuals are immutable, each variation is a copy with modifications and the original individual is unchanged. A common parameterisation for variation operators is the probability of each individual being modified, which may result in some of the source individuals appearing in the new stream.

For example, a variation operator `Mutate Random` might produce a stream of individuals where some have one component replaced with a completely random value. The operator could copy the entire source stream before mutating individuals in place, or may replace components as part of copying each, but the original individuals are not modified and, once the operator has added the new individuals to the output stream, these cannot be modified again.

Variation operators must always be usable with infinite streams and the size of the resulting stream is not required to match the size of the source. While one or more individuals may be used to produce one or more variations, they must be adjacent in their streams and the resulting individuals appear in the order of the originals. If this were not the case, it would be impossible to associate specific offspring with their parents, as is required in some parameter adaptation schemes.

For example, a recombination operator `Uniform Crossover` might take adjacent pairs of individuals from the source stream. Components are randomly selected from either individual and combined to produce a new individual. The resulting stream will be half the length of the source (assuming the source has an even length), and an individual located at index i in the result is known to have parents at $2i$ and $2i + 1$ in the source.

While creating offspring is a common application, it is also possible for variation operation to produce individuals that are not directly related to the source. For example, an operator `Bit Probability` may use n binary individuals to produce one real-valued individual, with each element containing the rate of 1 bits occurring in the source stream. This real-valued individual may then be varied by a `Bit Selector` operator that produces infinite binary-valued individuals based on the probabilities in the source. Using individuals with different representations as intermediate values allows many interesting algorithms to be described, and in particular supports the creation of EDAs.

3.3.8 Termination

A difficulty with applying EAs to real-world problems is the lack of estimable or repeatable running times. Purely deterministic search algorithms have well understood best, average and worst case performance analyses. For example, linear search has constant best running time (in the relatively unlikely case that the first value tested is the search target) but a worst case proportional to the size of the search space. However, the search spaces that require EAs are not as well understood, making analysis of average and worst-case times difficult or impossible. Even best case timings cannot be assumed to be constant, since many problems require an extensive search to confirm optimality, even if the first solution tested appears adequate. Worst-case times may be infinite because of the possibility of the algorithm becoming 'stuck' at an insufficiently fit solution (often called "premature convergence").

Without any obvious halting condition other termination criteria are used, most of which can be categorised as measuring effort, diversity or quality. Effort measurements count iterations, evaluations, seconds or the number of individuals created, terminating (or restarting) the algorithm after a predetermined amount. These measurements have the most predictable running time, though there is no guarantee of a quality solution being found. Comparative research often runs multiple algorithms and compares the best fitness found after a fixed amount of time or effort, though there is no universally accepted and used effort measurement and this "competitive testing" of algorithms has been criticised as unscientific [48].

Diversity measurements allow an algorithm to continue running until the search process stalls. Such a condition may be detected by comparing the similarity of individuals, by measuring the improvement over time of the best-known individual or the success rate of individual variation operators. Terminating based on diversity is often an appropriate compromise when a good solution is required, but "good" is poorly defined and time is not a limitation. Some algorithms will very quickly abandon diversity in order to exploit a section of the search space, while others maintain a higher diversity to achieve wider exploration. In general, however, EAs converge to the best solution they find.

Quality measures terminate an algorithm when a solution is found that is 'good enough.' Typically, a fitness value is specified, where the first observed individual that is as fit or fitter is considered the final solution. Quality may be based on more criteria than simply fitness, though this may be seen as an indication that fitness is poorly defined, since it is supposed to represent solution quality. Measuring quality guarantees that an algorithm will continue running until a useful solution is found.

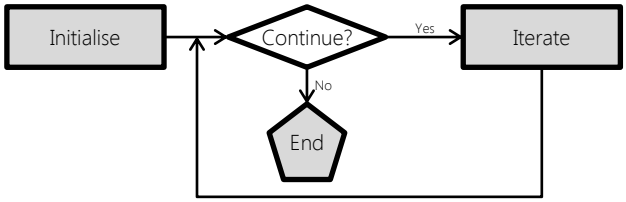

Figure 3.9: General structure of iterative algorithms.

However, if no solution exists or the algorithm converges to a suboptimal solution, termination may never occur.

Combining multiple termination conditions usually produces the most practical limits. For example, a constant fitness limit and a fitness gradient diversity measure will run an algorithm until a good solution is found or the algorithm stops finding improvements. The reaction to each case may be different: for example, termination due to lack of diversity may restart the algorithm with different initial conditions, while a fitness-based termination indicates a successful search.

In this model, termination criteria are assessed between each iteration, as part of the "Continue" block shown in Figure 3.9. The general structure consists of two phases: initialisation and iteration, each with a different operator network but sharing groups and other state. The initialisation phase generates the initial groups, allowing the iteration phase to assume that all necessary groups already exist. In some cases, the initialisation phase is not required and may be omitted, though since such algorithms then cannot use results from previous iterations they are better described as repetitive rather than iterative. Any algorithm that may use values from earlier iterations requires initialisation. No information is discarded between iterations, all groups and other context remain unchanged, and for the purposes of statistics collection and termination, the initialisation phase is considered the first iteration; if a termination criterion is met after initialisation, the iterating phase never occurs.

Given the range of options for termination criteria and reactions, and the potential for new developments in this area, it is not appropriate to mandate or specify a particular functionality or model. Further, since termination criteria does not define an algorithm—GA is still GA whether it runs for 50 generations or until fitness reaches a certain value—specification is outside the scope of this model. The use of arbitrary groups rather than specifically designated populations, as in other abstractions, prevents general statements regarding the source of relevant individuals. Co-evolutionary algorithms in particular may create and evaluate groups that are not intended to be considered for termination. Termination conditions are experimental parameters, rather than algorithm features, and their specification is best left to implementations.

3.4 How to share an algorithm

One of the purposes of the conceptual model presented here is to simplify presenting and sharing EAs with the research community. At present, algorithms are often described in literature in an imperative manner, that is, 'first A happens, then B, followed by C,' (for example, as in [31]). Descriptions that break an algorithm into component parts can suffer from complexity when recombining each part into a clear structure (as in [7]). Despite the complexity, modularity is recognised as "the key to successful programming" and the ability to "glue" components together efficiently is considered the greatest benefit a programming language provides [50].

The examples in Section 3.5 use English prose and diagrams to connect the components into a recognisable system, though this is far from an ideal format. Ambiguities within the English language cause practically every published specification to be open to (mis)interpretation.[6] For an algorithm description, this results in multiple, differing implementations that all claim to represent the same algorithm. This is not the intent of the original authors, who do not want incorrect implementations of their algorithm in use. Equally, implementers may be frustrated when they encounter an ambiguity in a description that cannot be resolved using readily available information [53].

A common mitigation is the provision of pseudo or executable source code, either as supplementary material (as in [19] and [76]) or an integral part of the description (as in [12]). However, despite the availability of source code, it may remain difficult or impossible to reproduce the original behaviour (as evidenced by [73]), particularly when insufficient time was available for "tidying up" the code before publication, or when translation into a more familiar programming language is required [53, 64].

With the model described in this chapter, the identification and abstraction of component parts of an algorithm become available to authors without having to define interfaces, general behaviours and terminology. Direct citation of an operator in another publication is possible, since its behaviour is decoupled from the context, the contractual obligations of the algorithm can be clearly defined and abstraction from the application is easily achievable.

In Section 3.5, diagrams show the connections between operators (figures 3.11–3.16). However, despite providing a useful and intuitive overview of the system, these diagrams lack the specificity needed to actually implement the algorithms. Some form of programming or specification language is necessary for a full implementation.

At a minimum, a specification language requires the ability to specify operators, named groups and the connections between them. Other useful features may include

[6] A problem that many professional organisations attempt to mitigate by providing standard definitions of selected words, for example, in IETF RFC2119 (http://www.ietf.org/rfc/rfc2119.txt).

```
# initialisation phase
population = (part 100 (random_real 0.0 1.0))

# iteration phase - fifty iterations
repeat 50
  parents = (part 100 (tournament 2 0.9 population))
  offspring = (mutate 0.1 (crossover 0.9 (shuffle 0.05 parents)))
  population = (part 100 (descending (merge population offspring)))
end repeat
```

Listing 3.2: Example of a possible specification language inspired by functional programming.

parameterised operators, arithmetic and numeric variables, external functions, termination conditions, user interaction, tools for statistics collection, loops, textual macros and conditional execution. Inclusion of any extra features risks increasing the complexity of the language and thus potentially offsetting the gains provided by the conceptual model's abstractions. There is no benefit in defining another general-purpose programming language.

Due to the functional nature of the operators described in Section 3.3.1, use of a functional programming style would appear to be suitable. An example of how such a language may look is shown in Listing 3.2, with a visual representation of the iteration part shown in Figure 3.10. The features shown in this example are named groups, parameterised operators and a parameterised generator. Some restructuring would be necessary for execution in pure functional programming languages, primarily due to the iteration phase requiring a recursive definition. However, by substituting the transient groups a single (recursive) expression is found, shown in Listing 3.3, that produces the final group after 50 iterations.[7]

With a suitable library of functions, even without following the concepts described here, Listing 3.3 can be written without need for a customised, domain-specific language. In many less 'pure' functional languages, Listing 3.2 is also possible. However, in both cases the simplicity of the code decreases dramatically with any increase in the complexity of the algorithm; the amount of explanation required for both examples negates any modularity advantage.

A second source of inspiration comes from the diagrams that have been used (such as Figure 3.10). Rather than a functional notation, which encloses each previous calculation as a parameter of the next, a pipelined structure treats the stream of individuals as separate from operator parameters. Such a notation may appear as shown in Listing 3.4. This example is similar to the binding operators in some functional programming languages, for example, Haskell's `>>=` operator. The flow

[7] Referential transparency is essential for Listing 3.3 to execute correctly, due to `population n-1` being invoked twice for each `n` and the stochastic nature of the operators being used.

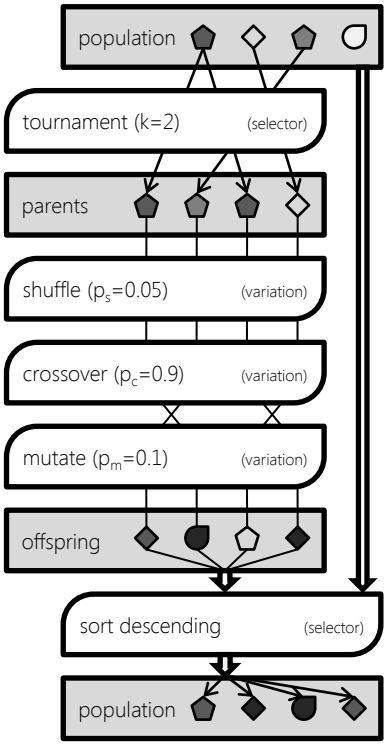

Figure 3.10: Iteration phase of the algorithm in Listing 3.2.

```
population 0 = (part 100 (random_real 0.0 1.0))

population n = (part 100
        (descending
            (merge
                (mutate 0.1 (crossover 0.9 (shuffle 0.05
                    (part 100 (tournament 2 0.9 (population n-1)))
                )   )   )
                (population n-1)
    )   )   )

final_population = (population 50)
```

Listing 3.3: Listing 3.2 adapted to use recursion.

```
# initialisation phase
random_real(0.0, 1.0) -> part(100) -> population

# iteration phase - one hundred iterations
repeat 100
  population -> tournament(2, 0.9) -> part(100) -> parents
  parents -> shuffle(0.05) -> crossover(0.9) -> mutate(0.1) -> offspring
  merge(population, offspring) -> descending -> part(100) -> population
end repeat
```

Listing 3.4: Example of a possible specification language inspired by a pipeline structure.

of individuals from one or more groups, through some operators and into another group is shown clearly.

The design of a language for the purpose of specification, description, definition and execution is given in the Chapter 4, though this represents only one possible use of the model. Even without the use of a specific description language, the model serves as a useful breakdown of EAs into easily defined, combinable, explainable and researchable components.

3.5 Example Algorithm Descriptions

This section describes seven established algorithms by identifying their components in terms of the concepts described in this chapter. The range of algorithms is intended to be representative of algorithmic development within EAs and widely used throughout the literature. As it is only a sample, this section cannot be taken as complete proof of the model's suitability for all EAs. However, since most related algorithms are based on one of the algorithms presented here, there are unlikely to be such significant structural differences that representation is impossible.

The combination of components is described briefly in text and diagrammatically and references to more detailed descriptions of the algorithms are provided. Prior familiarity with the algorithms may assist in understanding, although some of the terms used are not necessarily standard for each algorithm but mirror the concepts already described here. All of these algorithms have variations that perform better in general or for specific applications; the examples shown are adapted from [12].

3.5.1 Evolution Strategies

ES[8] [19,33,82] consists of a single variation operator and one of two selection operators. Individuals are represented as vectors of real numbers. Initialisation generates a pool of individuals from random values. Each iteration, λ individuals are gen-

[8] In this case, the classical Evolution Strategies is used, rather than the self-adaptive varieties that are considered state-of-the-art. [7]

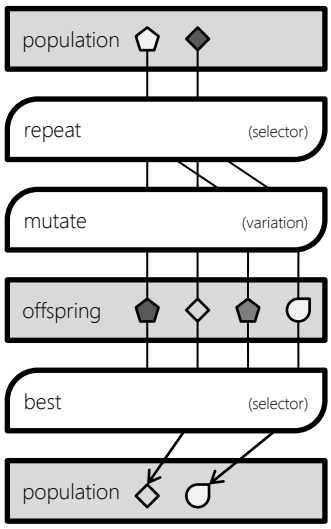

Figure 3.11: Evolution Strategies evolutionary algorithm.

erated from the pool, reusing source individuals if required, by mutating one or more components by a random amount. The best μ variations are retained, optionally making the original individuals available for selection. ES parameters are summarised as a $(\mu + \lambda)$-ES if the sources and variations are merged or a (μ, λ)-ES if only variations are retained. For example, a $(1, 10)$-ES produces ten variations of an individual and retains the best one. A $(1 + 10)$-ES also produces ten variations, but if none of the variations is an improvement, the original individual is retained.

Figure 3.11 shows a (μ, λ)-ES represented as groups and operators. The two groups are the `population`, consisting of two individuals, and a group `offspring` that is generated anew each iteration. The `Repeat`, `Mutate` and `Best` operators are used to implement the ES algorithm described above.

Repeat selector: Creates an infinite stream containing all of the individuals in the source group, repeating the group if necessary. In this instance, the partition operator is applied after the `Mutate` operator.

Mutate operator: Add a normally distributed random variable to each component of each individual. Here, it is applied to an infinite stream, but since the result is partitioned to λ individuals, only λ mutations are created.

Best selector: Returns the source group in descending fitness order (fittest first). In this instance, partitioning after μ individuals produces the next `population` group from the most fit individuals.

3.5.2 Evolutionary Programming

EP [33] has changed considerably throughout its lifetime, though the creation of one variation for each source individual and competition between parents and offspring has been consistent throughout. Each individual is mutated to produce a child, effectively doubling the population size. The population is reduced to its original size

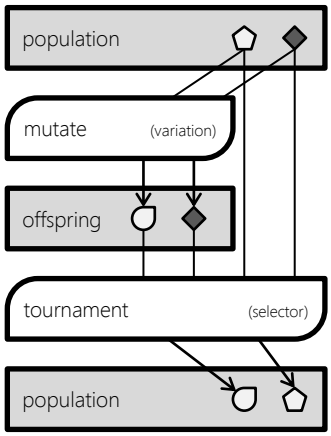

Figure 3.12: Evolutionary Programming evolutionary algorithm.

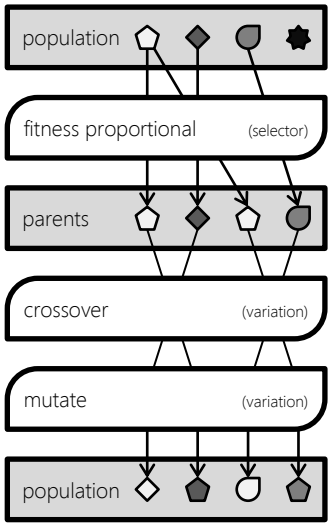

Figure 3.13: Genetic Algorithms evolutionary algorithm.

by selecting better performing individuals using competitive tournament selection. [7, 12]

Figure 3.12 shows EP represented as groups and operators. The two groups are `population` and `offspring`, as in ES, and the operators used are `Mutate` and a selector `Tournament`.

Mutate operator: Create one variation for each of the n individuals. The nature of the variation depends on the representation used for individuals.

Tournament selector: Compares each individual to a random pool of individuals and orders the resulting stream by the number of 'wins' each individual has. In this instance, the result is partitioned to n individuals, which make up the next `population` group.

3.5.3 Genetic Algorithms

GA [47] is distinguished from ES and EP by the use of a sexual variation operator, that is, multiple individuals are combined to produce each variation. The usual operator for this process creates a variation by taking components from each of a pair of parent individuals to produce a child individual. Selection is used to determine the individuals that become parents, rather than to reduce the population after reproduction. In the simplest case, all variations are retained, though some extensions to the algorithm include a second selection step allowing one or more fitter individuals to survive without modification. Individuals are traditionally represented using a binary digit for each component, though it is possible to use a different representation with suitable changes to the variation operators.

Canonical GA places an added restriction that is not a general necessity of the model: fitness proportional selection requires that each fitness value is a single, real number, such that a simple transform may be applied to calculate selection probabilities. In many cases, this is not a significant restriction, but for situations where a simple fitness value is not possible, other forms of selection (such as tournament or rank-proportional) may be substituted.

Figure 3.13 shows GA represented as groups and operators. The groups used are `population` and `parents`, though if some cross-generational survival were to be included there would also likely be an `offspring` group. The `Fitness proportional` selector and `Crossover` and `Mutate` operators implement the remainder of the algorithm.

Fitness proportional selector: The relative proportion of each individual's fitness determines its likelihood of being selected, such that the fittest individual has the highest chance of being selected. The resulting stream is infinite, since individuals may be selected more than once, and hence partitioning to a predetermined size is required for the `parents` group.

Crossover operator: Takes pairs of individuals and exchanges one or more randomly located contiguous strings of bits between them. For example, combining an individual `00000` with an individual `11111` may result in individuals `00111` and `11000`.

Mutate operator: Replaces a random selection of zero or more bits in each individual with their inverted values. For example, an individual `00000` might be mutated to `01001` by inverting two bits.

3.5.4 Differential Evolution

DE [76] was designed specifically for numerical optimisation problems. Each individual is represented as a real-valued vector with the primary group of individuals being called `population`. One variation is generated for each vector based on the values of

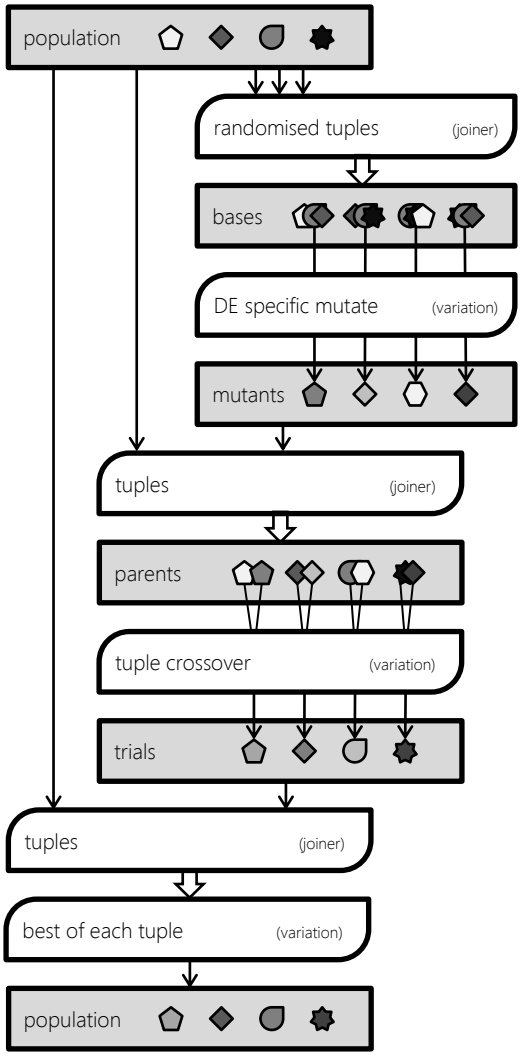

Figure 3.14: Differential Evolution evolutionary algorithm.

other vectors in `population`, with the effect of scaling adjustments in proportion to the distribution. As the range of input vectors converges towards optima, adjustments are smaller, resulting in fine-grained search. Although the variation strategy bears some similarity to EP (each vector produces one variation), selection differs: EP retains all the best results, while DE only retains those that are better than their "target" vector.

Figure 3.14 shows DE represented using groups, joins and operators. The groups of single individuals used are `population`, `mutants` and `trials`, while `parents` contains pairs of individuals joined and `bases` contains triples. Two joiners are used, `Tuples` and `Randomised tuples`, as well as a `Crossover` operator for joined individuals, a `Best of each tuple` selector and a DE-specific operator for the scaled mutation operation.

Randomised tuples joiner: Creates a stream of joined individuals for each individual in the first specified group, associated with randomly selected individuals from each other source. In this instance, `population` is provided three times, which produces a joined individual with the base and two others used by the mutation operator.

DE mutation operator: Takes joined individuals containing three real-valued vectors and combines them by adding the difference vector between the second and third to the first, specifically:

$$\vec{out} = \vec{in}_0 + F\left(\vec{in}_1 - \vec{in}_2\right)$$

A scaling factor F is applied to the difference vector to control the rate at which the algorithm converges.

Tuples joiner: Creates a stream of joined individuals associated by position within each source group. In this instance, each individual originally selected as a base is associated with the result of the mutation operator, and later with the result of the crossover operator.

Tuple crossover operator: Performs a crossover operation between the two individuals making up the joined individual. This operator and the preceding joiner are necessary to ensure the base vectors and mutated vectors are correctly associated, rather than being merged into a single stream.

Best of each tuple operator: Creates a stream containing the best individual for each joined individual in the source. Because this operator is returning different individuals to those in the source stream, it is classed as a variation operator rather than a selector.

3.5.5 Genetic Programming

GP [54] is used to find results that are executable computer functions or programs. For example, the symbolic regression problem [54] attempts to find a suitable ex-

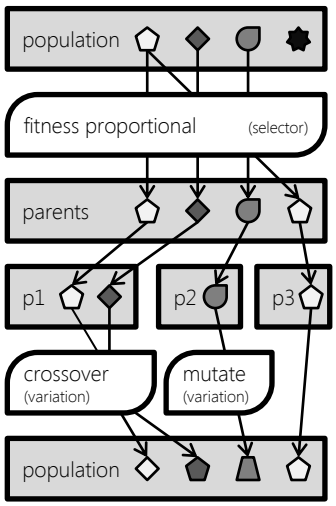

Figure 3.15: Genetic Programming evolutionary algorithm.

pression to map one set of data points to another. Program trees are constructed out of basic arithmetic operators, an input variable and some constants. Each program is used to calculate points based on the target data with the accumulated error used as fitness. Other applications of GP include Boolean function derivations, hierarchical system designs and game strategies. In each case, different tree nodes are used, though the structure and process are retained.

An initial pool of programs is generated randomly by selecting a root function and filling its parameters with functions and terminals. New programs are created by replacing a branch of a tree with randomly generated nodes, exchanging branches between two trees or sometimes by simplifying or adjusting the tree structure without modifying the behaviour. Programs are selected for modification, and in some cases reproduction without modification, in proportion to their fitness (as in GA).

Figure 3.15 shows the GP algorithm as groups and operators. The `population` group contains the main pool of individuals, with `parents` being those selected to create the next generation. The parents group is partitioned into `p1`, `p2` and `p3`, which are varied using different operators.

Fitness proportional selector: Identical to the selector used for GA in Section 3.5.3.

Crossover operator: Randomly selects one function node in each of two individuals and exchanges them, and all their children, between individuals.

Mutate operator: Replaces one function or terminal node with a randomly generated tree.

3.5.6 Steady-State Genetic Algorithms

The GA described in Section 3.5.3 uses what is known as a *generational model*, in that it takes only one cycle for every individual to be replaced. The *steady-state*

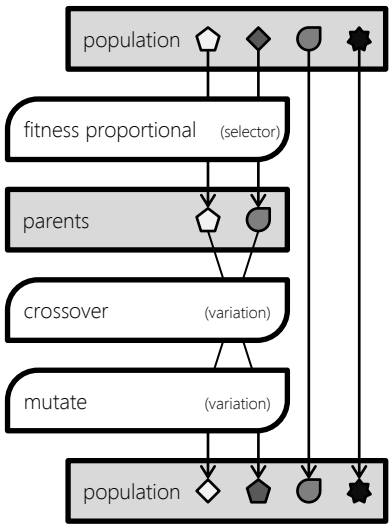

Figure 3.16: Steady-State Genetic Algorithms evolutionary algorithm.

model differs in selection: rather than using every single input, a (usually small) percentage known as the *generational gap* are varied and retained in favour of less-fit, older or randomly selected solutions. [31]

Figure 3.16 shows SSGA as groups and operators. Each component is identical to those used in Section 3.5.3, but a small number of individuals are selected and varied, with the rest of the `population` group being partitioned and included in the next `population` group unchanged.

3.5.7 Particle Swarm Optimisation

While generally not considered an evolutionary algorithm, PSO fits the model described here. A swarm takes the place of a population, consisting of a set of individuals that each have a position and a velocity within the search space. Each iteration, the velocity of each particle is updated based on a personal best-seen location (attracting the particle back to where it has already been) and a neighbourhood best-seen location (attracting the particle to other good locations).

Figure 3.17 shows an algorithm structure that handles the main `swarm` group as well as maintaining a group `p_bests` with the personal best locations for each particle. Storing these values as separate individuals reflects their separate identities: two particles at the same location have the same fitness, regardless of their personal bests. Neighbourhoods are handled as part of the update velocity operator. Variation operators `Update velocity` and `Update position` provide the primary search behaviour, while `Tuples` and `Best of each tuple` (described in Section 3.5.4) assist with maintaining the separate list of best positions.

Update velocity operator: Adjusts the velocity of an individual based on the individuals personal best position (provided as a component of the joined in-

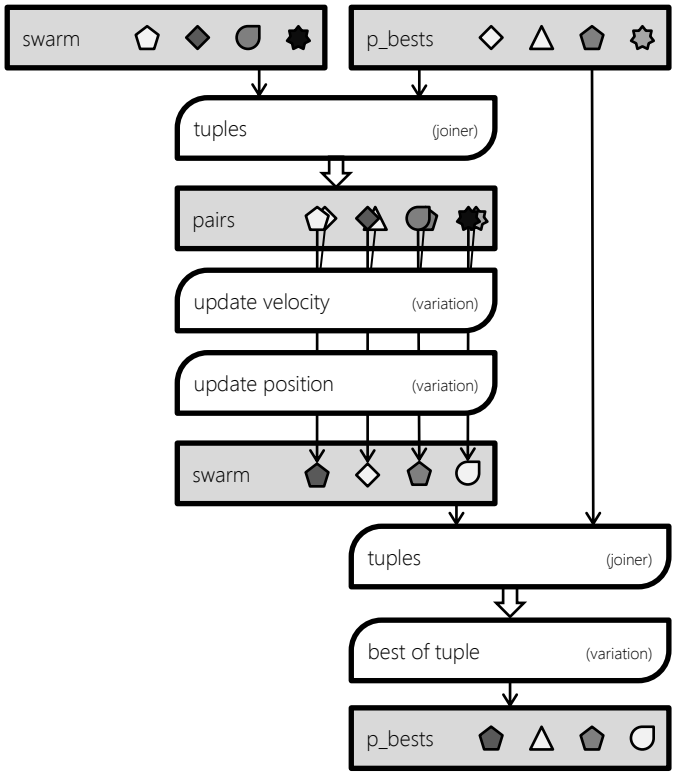

Figure 3.17: Particle Swarm Optimisation algorithm.

dividual), the best position amongst a "neighbourhood" of individuals and other parameters simulating inertia or repulsion.

Update position operator: Adjusts the position of an individual based on its velocity.

3.6 Chapter Summary

This chapter has detailed the model of EAs that was started in Chapter 2. The model defines algorithms as a process of 'having' a collection of potential solutions (Section 3.2), a method to improve this collection (Section 3.3) and a well-defined problem to assess the solutions against (Section 3.1).

In this model, the basic algorithm elements are *individuals*, *groups*, *operators* and *streams*. Individuals represent potential sets of input values to a problem function with a fitness representing the quality of those values. Operators are the processing elements that perform merging, partitioning, joining, filtering, selection and variation, while groups are intermediate storage nodes and streams represent the edges linking storage and processing. Operator graphs created from these components represent a complete description of an EA.

Algorithms composed using this model can be depicted graphically or textually without the complexities that arise when all configuration aspects, such as population sizes and termination conditions, are integrated into a single presentation. By

identifying aspects of the model that represent the algorithm itself and decoupling them from implementation and experimental concerns, the design and discussion can be better focused around the algorithm. Chapter 4 continues this separation by defining a language that specifies the compositional aspects of the model; it focuses on the algorithm and encourages the use of a structured approach to present the required components. Chapter 5 defines the behaviour of the model to ensure that it can be interpreted consistently and unambiguously. Chapter 6 discusses the practical applications of a general description of algorithms, particularly in the areas of research, design and implementation that were discussed in Chapter 2.

To ensure that the model does not preclude existing algorithms, seven archetypal algorithms have been described. These descriptions are brief but include enough detail to provide an intuitive understanding of the algorithms. Chapters 4 and 6 expand these examples to the point where a complete, executable algorithm is shown.

Chapter 4

ESDL

In the previous chapter, a generalised model for EAs was presented in detail. This model defines a small set of components that may be combined relatively arbitrarily to describe specific algorithms. Using this model allows researchers to direct and focus their work and provides greater flexibility and code reuse for implementers of EA applications, libraries and frameworks. However, even with a convenient model, it is still necessary to specify all the details of an algorithm concisely and unambiguously. Most algorithms rely on behavioural assumptions and default values, which are areas where graphical or prose specifications can be clumsy and imprecise. Domain-Specific Languages (DSLs) are an approach to reducing complexity in models by abstracting specifications into the minimum representation required for communication with other domain experts. This chapter introduces Evolutionary System Definition Language (ESDL), which represents algorithms designed using the earlier model with plain text. ESDL allows practitioners to compose arbitrary algorithms without being restricted into fixed structures or sets of operators.

4.1 Reusable Software

For development, reusable software generally comes in one of three forms: a library, a framework or a language. Each of these provides functionality that a programmer can use for their own purposes in slightly different forms, each with associated benefits and drawbacks. Since most researchers working in EC are required to develop their own software, the task of selecting a library, framework or language to use (or choosing to not use any) is very likely to occur.

A library consists of executable routines that the developer can use for their own application. For EC, the library may provide an object model for an algorithm, allowing the developer to integrate the code into an existing user interface.

Frameworks also consist of executable routines, but are generally distinguished from libraries by also providing the main application structure and interface. For example, an EC framework may include a user interface and statistical analysis, but require the developer to provide the algorithm or some operators. This is sometimes referred to as inversion of control, since while a developers code would typically call into a library, in a framework their code would provide the responses.

Finally, languages provide reusable software in a very transparent manner. By integrating very common patterns into their syntax, such as objects or functions, once a developer is familiar with a language they can be very productive. The effectiveness of a language depends on the language model matching the intended development, as well as the mental model of the developer. Languages can be broadly classed as *general purpose*, which includes languages such as C++, Java and XML, or *domain specific*. Since this chapter introduces a Domain-Specific Language (DSL) for the model of Chapter 3, a deeper discussion of DSLs is worthwhile.

4.1.1 Domain-Specific Languages

In a software engineering context, a DSL is "a computer programming language of limited expressiveness focused on a particular domain" [35, p. 27]. Domain language, or jargon, has the same definition without the necessity of being a computer programming language. In both cases, "limited expressiveness" is the aspect that makes them useful. Restricting the range of applicability limits the potential meanings and interpretations of particular words such that one familiar with the language can efficiently discern the precise meaning from a small amount of text.

The converse of domain-specific language is general-purpose language, which allows individual words to have one or more meanings depending on context, inflection and pronunciation. English is a general-purpose communication language that is notorious for its wide range of synonyms, homophones and grammatical inconsistencies. By contrast, musical notation is a domain-specific language that can precisely communicate a large amount of information in a concise form. English is able to describe a specific musical composition, but because of its greater expressiveness ambiguity is likely and a greater number of words are required to convey equivalent information. Domain language does not preclude general language from describing a subject; rather, it reduces the scope of a general language to optimise for the characteristics of the subject.

Similar contrasts can be observed between general-purpose programming languages and DSLs. HTML (HyperText Markup Language) is a DSL for describing

the style, appearance and structure of documents, while XML (eXtensible Markup Language) is a general-purpose language for describing structured documents. Both share a very similar syntax, but despite the similarities, HTML is a subset of what may be expressed using XML. The limited expressiveness of HTML and its highly targeted applications results in a language that is more easily understood and more efficient for its purpose.

Other examples reinforce the distinction: regular expressions provide a very terse notation for describing complex text comparisons, though most parsing tasks (such as matching nested parentheses) are beyond their capabilities. SQL (Structured Query Language) provides a specification for data queries that is easy to read and modify while being completely inadequate for data processing, and the structure of a "makefile" [87] allows a build process to be specified without providing the full flexibility of the underlying platform and file system. These examples show the limited expressiveness of successful DSLs, and in each case it is the limitation that makes the language successful. General-purpose languages can be used to produce identical functionality, though more description of a higher complexity is required and there is an increased likelihood of error.

A second defining characteristic of DSLs is that they are "a thin facade over [a] model" [35, p. 18]. Each term in a domain language is associated with a much more specific meaning than used by general language. As an example, a musician understands the tempo markings *largo* and *lento* to indicate a similar speed but a significantly different way to interpret and perform the music, such that one cannot be substituted for the other. (In general language, the words have completely different meanings that barely relate to their musical application.) The closest equivalent word in English would be "slowly," which captures only the speed without the associated implications of note durations, attack strength and assertiveness. *Largo* and *lento* are the terms associated with different instantiations of a particular musical model that makes them sound different—a wide range of parameter values are encapsulated in a single word.

Fluency is the primary manner in which DSLs improve programmer productivity. For spoken languages, fluency is understood to mean an efficiency and effectiveness at communicating in a language and the same applies for developers. Low- and machine-level languages are designed for the ease of understanding and execution by the processor; as problems become larger and more difficult at the algorithm level, complexity of translation makes it harder for developers to write fluently at this level. Continuing the musical analogy, representing music as note names or letters provides an easier task for the mechanical process of pressing the correct keys, though at the expense of obscuring a wider view of the composition. Higher-level languages provide libraries of common machine code patterns, keywords or

domain language, with the syntax and semantics to compose them into a complete program; the gap between the problem language and the programming language is narrowed, and greater fluency for the developer is achieved.

However, as well as a 'distance' between problem and program language, there is also a direction. Some higher-level languages are more fluent for particular problems—a source of near-constant conflict between devotees of functional programming and object-oriented programming. A well-designed DSL targets a specific area with the aim of representing the problem so closely that non-programmers are able to use, read and comprehend the language, while programmers with an understanding of the domain can do so with greater ease.

A major benefit of fluency is rapid development and testing cycles. Software development is well known to consist of as much or more debugging than programming [63], which is a reflection of the range of incorrect programs that can be created by a language: a general-purpose development language has few limits on what can be produced, intentionally or otherwise, while remaining syntactically valid. Since a DSL is far more limited, invalid programs often cannot be expressed without using invalid syntax, which can be easily identified.

4.1.2 Code Reuse in Evolutionary Computation

EC has a poor history of code reuse. It can be safely estimated that of the hundreds of libraries written for one or more algorithms, as well as potentially thousands of single-use applications, there is little evidence of code being used more than once. Where there is evidence, which is rarely more than a brief reference in an application paper, it typically appears in work by the same research group that originally developed the code. It has been suggested that such a vast amount of code has been produced for EC because most practitioners are computer scientists who enjoy writing code or have had poor experiences using other people's code [53, 88]. Given researchers' perceptions that they are the "first to release code" for their topic [88, p. 19], the associated belief that existing code is not helpful is somewhat understandable. Another possibility is that some researchers may consider the use or adaptation of someone else's software for their own publications to be intellectually dishonest or inappropriate.

The list of widely used libraries is disappointingly short: ECJ [60] (while technically a framework, rather than a library) is a clear leader with many users outside of the research group that developed it. EO (Evolvable Objects) [42] and its extension to parallel computing, ParadisEO [23], also see significant use. However, there is little evidence that would support labelling other libraries as 'popular'—words such as 'unmaintained,' 'ignored' and 'dead' are far more accurate. This is unfortunate,

```
while (Pxover <= 0.9) do
    Pmutation := 0.1;
    while (Pmutation <= 0.2) do
        init; // initialize population
        call_EA; // run EA for one epoch
        writeresult;
        Pmutation := Pmutation + 0.01
    end;
    Pxover := Pxover + 0.05
end
```

Listing 4.1: Example PPCEA script for dynamically adjusting EA parameters (adapted from [58]).

since many libraries contain well-designed, well-developed and well-tested code that could accelerate development for many researchers.

Apart from libraries and frameworks, three other classes of relevant software are workbenches, templating engines and DSLs. HeuristicLab [95] is a workbench, a single application that provides the full algorithm development, design, test and analysis workflow. Extensibility is available through a plugin system and a graphical designer is provided for designing algorithms using a model similar to (though developed independently to) that in Chapter 3. However, as is common with workbenches, interoperability and sharing of designs is difficult [35].

EASEA (EAsy Specification of Evolutionary Algorithms) [14] is described as "a high-level language" but more closely resembles a template-based external DSL. Implementations are provided by a user in C++ code for an evaluator and crossover and mutation operators, while selection and replacement operators are chosen from a list. Recent features added to EASEA include distributed and GPU-based parallel processing [62]. The enforced algorithm structure and lack of flexibility in selectors make EASEA a convenience for those who can work within it, but not a substitute for those developing algorithms based on different structures.

PPCEA (Programmable Parameter Control for Evolutionary Algorithms) [58] is a scripting language for actively controlling algorithm parameters, though it provides little functionality for defining the algorithm itself. An example PPCEA script is shown in Listing 4.1. PPCEA appears to have very little use even within its original development group, where it has been rewritten [99] and later used as a test case for a web-service approach to DSL implementation [57]. Examples of publications using PPCEA for EC-related experiments could not be found.

EAML (Evolutionary Algorithm Modeling Language) [94] represented the main breeding process of an algorithm in a portable and interoperable form based on XML. However, it suffered from requiring a strict algorithm structure, as well as

```
<EAML standalone="true" project="bits">
  <Code name="objfunc"><![CDATA[
    double obj = ECPARAM(CBits, gene, size);
    for (int i = 0; i < ECPARAM(CBits, gene, size); i++)
      if (GETDATA(i) == 1) obj -= 1;
    return obj;
  ]]></Code>
  <Algorithm name="CBits" size="20" direction="minimize"
             generations="100" optimum="0.0"
             elitistRate="10%" operatorRate="90%">
    <objective><Use ref="objfunc"/></objective>
    <genome><BitString size="30" group="1"/></genome>
    <selection><RouletteWheel/></selection>
    <operator><Group><operators>
      <Binary rate="90%" succRate="10%">
        <OnePoint/>
        <succUnary>
          <PointIncrement min="0" max="1" step="1"/>
        </succUnary>
      </Binary>
      <Unary rate="10%">
        <PointIncrement min="0" max="1" step="1"/>
      </Unary>
    </operators></Group></operator>
    <initial><RandomInitial min="0" max="1"/></initial>
  </Algorithm>
</EAML>
```

Listing 4.2: Representation of an algorithm using EAML (adapted from [94]).

the common criticism of XML being better suited to machine parsing rather than human editing [35]. An example algorithm is shown in Listing 4.2.

More recently, ECML (Evolutionary Computation Modeling Language) has been proposed [6]. Based on UML, ECML generalises the representation of individuals in an algorithm, rather than modelling the entire algorithm. The authors emphasise that "ECML is not a domain-specific modeling language" in direct contradiction to its name, which specifies both the domain and its purpose as a modelling language. Meta-evolutionary algorithms (MEAs) are also proposed to provide a fixed algorithmic structure but use the operators specified by an ECML model. Although MEAs can "dynamically adapt to changes in the specification," this benefit is not unique to ECML and could be implemented in any algorithm. MEAs as described in [6] are limited to using one crossover operator and one mutation operator; combinations that are more complex do not appear to be supported.

In summary, while DSLs are known to provide significant benefits in accurate and fluent communication, particularly between non-programmer domain experts and software developers, there are very few DSLs for EC. The suggestion put here is

that this is partly due to the lack of a unifying model and the associated algorithmic segregation, even within libraries and frameworks, as well as a general inclination amongst researchers to write their own code. A useful DSL should remove this segmentation by allowing arbitrary composition of operators and supporting flexible structures rather than attempting to provide a predefined set. The risk of misidentification of such a language as too simple for practical use is a further challenge, particularly by experts who are already capable of, and used to, implementing their algorithms directly. The following section presents a design for an EC DSL based on the model in Chapter 3, and hence with inherent support for flexible algorithms.

4.2 Describing algorithms with ESDL

This section describes Evolutionary System Definition Language (ESDL), a DSL based on the unified model specified in Chapter 3. ESDL represents the structure of an EA without including unnecessary clutter; solution representation, operators and termination conditions are abstracted. This abstraction encourages greater code reuse, since self-contained operators can be shared more easily than code with tight internal coupling, and provides a structured form for presenting and publishing algorithms.

Section 4.2.1 provides details about ESDL that, while not directly related to the behaviour of any particular algorithm, are important for correct use and understanding of the language. They are given before syntactical details, sections 4.2.2–4.2.4, despite the potential for confusion due to lack of context, to provide a better understanding of the purpose and scope of ESDL.

A BNF-style grammar for ESDL is included in Appendix B. It may be used as an easy way to understand the syntactic construction of ESDL, though it does not describe any of the semantic behaviours and is ultimately not a substitute for the descriptions in this chapter.

4.2.1 Basic Conventions

These conventions specify some of the fundamental rules used by ESDL with regard to style and scope. Some of these deliberately limit the general applicability of ESDL. They have been considered and tested thoroughly, either specifically for ESDL or more generally, and are intended to keep ESDL focused on its primary use: describing the algorithmic structure of evolutionary algorithms. Readers familiar with other description or programming languages are likely to appreciate these conventions on first reading, though the relevance of some may not become apparent until later examples.

ESDL is **plain text**: no special characters or symbols are used and punctuation is kept to a minimum. This makes writing and reading ESDL simpler, makes it

portable through media such as email, alleviates the need to translate for inclusion in typesetting systems and reduces the potential for typing errors.

ESDL is **case-insensitive**: keywords, variables and group names consisting of the same letters in different cases are treated as the same. As a convention, keywords are written in uppercase and other terms in lowercase, though where alternative formatting can be applied, the importance of casing to readability is reduced.

ESDL **only has global scope**: regardless of where in the code a name is declared or referenced, it always resolves to the same element. This discourages the reuse of names for different purposes, removing the need to consider the entire description to determine behaviour and avoiding complicated scoping rules and syntax.

ESDL has **implicit variable declarations**: assigning a value is equivalent to defining the variable. There is no need to list every name at the beginning of ESDL code. Declaring the type of a variable is not necessary, since types can be inferred from usage and are effectively limited to real numeric values by the nature of the application.

ESDL has **two data types**: a variable is either a group or an opaque value reference (see Section 5.2.1). Both types may be passed as parameter values, but there is otherwise a clear delineation between where groups and non-groups may be used. Individuals within groups are inaccessible within ESDL and are not subject to this restriction, but rely on an assumption of compatible operators being available. Specifying provably type-safe algorithms is not the purpose of ESDL; a more pragmatic approach is intended.

ESDL **statements are end-of-line terminated**: no semicolon or other termination character is required. However, if the last character on the line (before any comment) is a backslash, the statement continues onto the following line.

ESDL has **no control flow**: conditional statements such as "if" and "while" do not exist. This keeps the code easily understandable by reducing the complexity and encourages designers to use abstractions for algorithmic variation. Subroutines cannot be specified in ESDL, but ESDL may link to external functions that are implemented in other languages.

ESDL has **single-line comments**: any text between a double-slash "`//`", number sign "`#`" or semicolon "`;`" and the end of the line is ignored. Comments are generally discouraged, with ESDL code intended to be readable without peripheral description. Text following a grave accent "`` ` ``" is also ignored by ESDL, but is intended as a directive or *pragma* to an interpreter, analyser or compiler.

4.2.2 Composing Algorithms

Individuals are the fundamental element of an EA; their creation, evaluation and use as sampling bias are central to an algorithm's optimisation ability. Direct access

```
FROM population SELECT ...
FROM group1, group2 SELECT ...
FROM generator() SELECT ...
```

Listing 4.3: Examples of the `FROM` clause.

and control of isolated individuals is rarely necessary at a structural level. Rather, it is the distribution of solution values and fitness within an entire group that is useful, and individual manipulation can be abstracted into operators. As a result, the operations with the greatest influence on the structure of an algorithm are stream use and group creation.

A syntactical construct for creating and manipulating groups must be capable of expressing the following six operations:

1. Create a group from a stream
2. Create a stream from a generator
3. Merge two or more streams
4. Partition a stream into one or more groups
5. Apply operators to streams
6. Join one or more streams

ESDL uses a `FROM-SELECT` statement to handle the first five of these operations, with the sixth handled by a `JOIN-INTO` statement. This division allows the majority of processing to be described using `FROM-SELECT` statements and `JOIN-INTO` is restricted to performing exactly one task. Syntactically, the two statements are identical, though combining them into a single set of keywords would obscure rather than clarify intent.

A `FROM-SELECT` statement consists of three clauses: `FROM`, `SELECT` and `USING`. The `FROM` clause is followed by the names of one or more groups or generators, separated by commas. If more than one is specified, they are merged as described in Section 3.3.2, resulting in a single stream. Listing 4.3 shows some example `FROM` clauses.

The `SELECT` clause is required and is followed by one or more group names. Group declarations are implicit; there is no need to declare a group before its name appears in a `SELECT` clause. Each name may be preceded by a size specifier—an integer, variable or expression (see Section 4.2.3)—which partitions the stream as described in Section 3.3.3; omitting the size implies that the remainder of the stream should be taken for that group. If more than one group is specified, all but the last must have a size specifier. The last group only needs a size specifier if the stream is infinite or if the partitioning is relevant to the algorithm. Listing 4.4 shows some example `SELECT` clauses. The `FROM` and `SELECT` clauses provide operations one through four.

The `USING` clause is optional. When omitted, groups are merged and partitioned without modifying or reordering any individuals; when specified, `USING` is followed by

```
... SELECT (size*0.4) groupA, (size*0.6) groupB
... SELECT 1 single, rest USING ...
... SELECT N population USING ...
```

Listing 4.4: Examples of the `SELECT` clause.

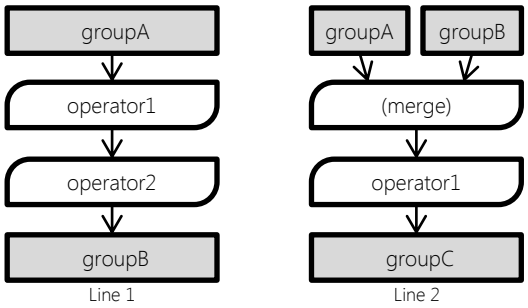

Figure 4.1: The `FROM-SELECT` statements from Listing 4.5 shown graphically.

```
1 FROM groupA SELECT groupB USING operator1, operator2
2 FROM groupA, groupB SELECT groupC USING operator1
```

Listing 4.5: Example `FROM-SELECT` statements with `USING` clauses.

a list of operators to apply to the stream created by the `FROM` clause. Operators are fully enclosed components with a well-defined interface, as discussed in Section 3.3.1, that produce an output stream from a merged input stream defined by the `FROM` clause.

Each operator is applied in the order specified; the final stream is the one used by the `SELECT` clause. Listing 4.5 shows some example `FROM-SELECT` statements with `USING` clauses, and Figure 4.1 shows the same statements graphically to illustrate the mapping to the model. The `USING` clause provides operation five.

The `JOIN-INTO` statement is fundamentally the same as `FROM-SELECT`: the clauses are `JOIN`, `INTO` and `USING` and multiple source groups are joined as described in Section 3.3.4 rather than merged. To simplify ESDL further, `JOIN-INTO` does not provide the same operator chaining functionality as `FROM-SELECT`; allowing multiple operators would require that the first is a joiner and subsequent operators are not, as well as creating the possibility for the result of a `JOIN-INTO` statement being something other than a group of joined individuals. One joiner may be specified after the `USING` clause, or `USING` may be omitted, in which case individuals are joined as index-associated pairs (as shown in the example in Figure 4.1). Listing 4.6 shows some example `JOIN-INTO` statements.

Specifying a group as both a source and a destination in a single `FROM-SELECT` or `JOIN-INTO` statement is permitted; the original group is used throughout the entire statement. In effect, the group name is merely a reference that is updated to refer

```
JOIN groupA, groupB INTO groupC USING joiner
JOIN groupA, generator() INTO 50 groupB
```

Listing 4.6: Example `JOIN-INTO` statements.

to a newly created group after the statement completes. However, specifying a single group as a destination multiple times in a single statement is not permitted, since the only benefit of such a construct (fewer groups that are never referenced) is outweighed by the ambiguity of multiple, equally valid, potential outcomes.

The contents of a group are never directly modified; replacing, rearranging or changing individuals require the creation of entirely new groups. Most software implementations of algorithms reuse allocated memory for performance, which is only a secondary concern for ESDL; the primary purpose of communication with humans is not well served by designing for performance, particularly where such an optimisation is not necessarily beneficial for all applications.

4.2.3 Operators and Parameters

While specifying chains of operators and groups is sufficient to define an algorithm's structure, most operators can be adjusted and tuned by varying parameters. These parameters are an essential part of a specification, as anyone who has attempted to reconstruct an experiment without them will attest. Operators are less reusable if tuning requires manual code changes, and parameter control requires operators that can modify their behaviour during execution. In order to support these applications, ESDL provides numeric variables, basic arithmetic expressions, external functions and support for parameterised operators.

The `USING` clause introduces a list of operators, specified by name with an optional parenthesised list of arguments. Unlike most common programming languages, argument lists are comma-separated lists of parameter names or "name=value" pairs. The set of valid parameter names is defined by the operator and using a name not in this set is an error, while names that are not specified take a default value also given by the operator definition. Order is not important, since names are always used and positional parameters are not supported. When no arguments are provided to an operator, the parentheses are optional. All the examples in Listing 4.7 are syntactically correct.

Parameter names specified without a value are given the value of the variable with the same name, that is, line 3 in Listing 4.7 implies `parameter=parameter`. If no such variable exists, the constant `true` is used. This allows the operator designer to provide a set of parameters for which the user can specify one to select the behaviour of the operator, for example, `USING crossover(one_child)` or `USING crossover(two_children)`.

```
... USING an_operator
... USING an_operator()
... USING an_operator(parameter)
... USING an_operator(parameter=value)
... USING an_operator(parameter1=value1, parameter2=value2)
```

Listing 4.7: Valid operator specifications with and without arguments.

```
def adapt(old_rm, SR):
    if SR < 0.2:
        return old_rm * 1.1
    else:
        return old_rm * 0.9
```

Listing 4.8: Equation (4.1) as a Python function.

Variables and functions operate independently of groups and operators. Naming collisions are not permitted but otherwise there are no implicit interactions or ordering dependencies between the two. This allows executable implementations to distribute or reorder calculations depending on how they intend to obtain the best performance. For example, an implementation may choose to execute group operators using a GPU and calculate variable values using a CPU. Provided the values are synchronised correctly, the order of execution is flexible.

Variables are not explicitly typed and may be assigned values of any type supported by the underlying implementation. At a minimum, variables must be able to store real numbers; any other types should be considered extensions that may not be supported by all implementations. Integer division as used by many systems programming languages[1] is not used—all division operations produce the correct mathematical result (within representational precision).

External functions allow calculations that are more complex than arithmetic to be abstracted from ESDL. These functions may be written in any language supported by the implementation and described or shared in any suitable form, such as a mathematical equation or a flowchart. Since ESDL does not include control flow, external functions are the only mechanism by which decisions based on value comparisons can be made. For example, some variants of ES use parameter adaptation based on whether a success rate is above or below a constant (typically $1/5$). Such a condition cannot be expressed as a simple expression, but is easily abstracted into a function as in Listing 4.8.

[1] Where dividing two integer values produces an integer result that is rounded, typically towards negative infinity or zero, such that $3/2$ results in 1, rather than 1.5. This obscure behaviour provides a performance improvement in many cases but is not appropriate for the communication objectives of ESDL.

$$r_m^{'} = \begin{cases} r_m \times 1.1 & \text{if } SR < 0.2 \\ r_m \times 0.9 & \text{if } SR \geq 0.2 \end{cases} \tag{4.1}$$

```
1 SR = 0.5
2 Rm = 100
3 Rm = adapt(SR, old_rm=Rm)
```

Listing 4.9: Example invocation of the `adapt` function in Listing 4.8.

Functions take an argument list with the same format as operators, but always require parentheses even if no arguments are provided. While operators must provide default values for all parameters, function parameters may be required or optional. As for operators, positional arguments are not supported. Listing 4.9 shows an invocation of the function in Listing 4.8, with one argument specified explicitly, one implicitly and in the opposite order to that specified in the definition. Note that `SR` on line 3 of Listing 4.9 is the parameter name with an implicit value, rather than the variable name with the parameter implied by position. Giving `Rm` on its own would be an error, since there is no parameter with that name, and specifying `old_rm` without a value would pass `true`, which is meaningless in this context.

Groups may be passed as arguments to a function, though they cannot be modified or returned. This is intended to allow specific intermediate statistics to be calculated, for example, line 1 of Listing 4.9 should be replaced with a call to a function calculating the success rate from the current `population` and `offspring` groups. Statistical observations that do not affect the algorithm's behaviour, such as mean and maximum fitness, are best handled using the `YIELD` statement described in Section 4.3.2.

4.2.4 Evaluating Individuals

As discussed in Section 2.3, the purpose of an EA is to optimise a set of values.[2] In most simple cases, there is one problem used throughout the algorithm; any time fitness needs to be evaluated this *default evaluator* can be used. Default evaluators are not specified as part of an ESDL algorithm, since this would reduce the generality of the specification by including information that does not define the algorithm. (For example, GA does not become a different algorithm when applied to a different benchmark problem.) Associating a problem with an algorithm description creates a specific experiment.

However, in some algorithms, the components of a problem are often necessary parts of the algorithm description. For example, co-evolutionary systems handle

[2] As was also discussed, optimisation is not the only application for EAs. However, most applications can conveniently be represented as optimisation problems, and those that cannot are not the focus of ESDL (nor are we aware of any current EA that applies to non-optimisable problems).

```
EVALUATE population USING evaluator
EVALUATE parents1, parents2 USING evaluator(parameter=value)
EVAL population USING dynamic_evaluator(time=t)
EVAL population
```

Listing 4.10: Example evaluator specification with the `EVALUATE-USING` statement.

```
EVAL population USING evaluator(time=t)
t = t + 1
FROM population SELECT N parents USING best
```

Listing 4.11: Potentially ambiguous use of a parameterised evaluator.

credit assignment by using multiple evaluators (as described in Section 3.3.4) and dynamic problems require the evaluator to change on each iteration. To support these cases, an ESDL statement is necessary to specify an evaluator as part of an algorithm.

A group's evaluator is changed using the `EVALUATE` statement. Conceptually, the statement may be interpreted as performing the evaluation immediately, as well as changing the association, though immediate evaluation is not strictly required. Evaluator associations propagate from the first source group in `FROM-SELECT` statements to each of the destination groups. There is no propagation in `JOIN-INTO` statements, as the resulting group contains individuals of a different type and the likelihood of the same evaluator being useful is low.

An `EVALUATE` clause (which may be abbreviated to `EVAL`) is followed by a list of group names and a `USING` clause, which is followed by a single evaluator, specified identically to an operator. Omitting the `USING` clause implies that the default evaluator should be associated with the groups. Listing 4.10 shows valid examples of `EVALUATE` statements.

Lazy evaluation is a convenient mechanism for reducing computational requirements. An `EVALUATE` statement creates the association with the evaluator, but can defer the computation until the group or individual is used in a context that requires the fitness. Whether the evaluation is lazy at the group or individual level depends on the implementation: lazy individual evaluation is likely to result in many individuals never being evaluated, while parallel implementations may compute fitnesses for an entire group more efficiently than selecting individual members. In either case, this is a merely a performance enhancement and implementations that evaluate fitness immediately at the `EVALUATE` statement fulfil the requirement that fitness is available when needed; any alternative approach must appear to have done the same.

```
t = t + 1
EVAL population USING evaluator(time=t)
FROM population SELECT N parents USING best
```

Listing 4.12: Unambiguous use of an evaluator that is parameterised over time.

```
1 JOIN groupA, groupB INTO groupC
2 EVAL groupC USING joined_evaluator
3 EVAL groupA, groupB USING assignment(source=groupC)
```

Listing 4.13: Using a credit assignment evaluator with joined individuals.

```
JOIN groupA, groupB INTO groupC
EVAL groupC USING joined_evaluator
EVAL groupA USING assignment(source=groupC)
EVAL groupB USING assignment(source=groupC)
```

Listing 4.14: Potentially incorrect use of credit assignment where `groupA` and `groupB` share individuals.

Dynamic evaluators are those that do not obviously follow the 'one fitness per individual per evaluator' constraint by allowing successive evaluations to produce different results for the same individual. In order to meet this requirement, the evaluator must be parameterised, such that changing the parameter changes the identity of the evaluator. Changing the time implies a potential change to the fitness of every individual; reusing a given time should reproduce the earlier results. 'Noisy' evaluators that include a purely random component are unlikely to adhere to this requirement, though since the pseudo-random number generators used for most simulations are deterministic for a given seed it is possible to produce a conforming noisy evaluator. Since the new evaluator is distinct from the previous one, the `EVALUATE` statement is necessary for associating it with groups.

Listing 4.11 shows an incorrect, or at least unclear, use of a dynamic evaluator. The argument `time` is passed the value of `t` at the `EVALUATE` statement; changing `t` afterwards does not affect the evaluator or the fitness of `population` until the next iteration. Listing 4.12 correctly reassociates `population` with an evaluator after updating the parameter. Reassociating evaluators before critical operations ensures correctness and removes the need for readers to trace propagation.

A joined individual can be evaluated and have a fitness that is independent from that of its component individuals. Evaluators do not propagate through `JOIN-INTO` statements, so evaluating a group of joined individuals requires an explicit `EVALUATE` statement. An evaluator for a joined individual cannot assign fitnesses directly to the component individuals; they need to be separately evaluated and have due credit assigned. Listing 4.13 shows an example of joining two groups, evaluating

```
JOIN groupA, groupB INTO groupC
EVAL groupC USING joined_evaluator
EVAL groupA USING assignment(source=groupC, component=1)
EVAL groupB USING assignment(source=groupC, component=2)
```

Listing 4.15: Assigning credit based on the position within the joined individual.

with `joined_evaluator` and assigning fitness values to the original individuals using `assignment`. The joined individuals are passed to the credit assignment evaluator as a parameter on line 3. Different evaluators may be used for `groupA` and `groupB` to assign credit independently.

One potential complication appears when an individual appears in multiple groups; in Listing 4.13, two identical individuals in `groupA` and `groupB` would be associated with a single evaluator. The 'one fitness per individual per evaluator' rule indicates that despite belonging to different groups and being used differently in the combined group, the fitness must be the same. Since the same value may have greater utility in one of the first or second positions, the fitness should be distinct. An intuitive correction is shown in Listing 4.14, though since evaluators are identified by name and parameters, this snippet is identical to Listing 4.13. To achieve the correct behaviour, an extra parameter can be added to the evaluator, as shown in Listing 4.15, or two evaluators with different names may be used.

4.3 Structuring ESDL systems

The statements described by Section 4.2 constitute the two steps of the general EA developed in Chapter 2. One or more `FROM-SELECT` statements with suitable operators can improve a group of potential solutions. However, without repeating this gradual improvement, they do not describe an EA and cannot represent existing algorithms.

This section continues to develop ESDL in order to support the iterative behaviour that is required for EAs (Section 4.3.1), and to allow observation and termination of a running algorithm (Section 4.3.2).

4.3.1 Algorithm Iteration

Section 3.3.8 introduced a clear delineation between the *initialisation* and *iteration* phases of an algorithm. Initialisation occurs once and typically uses generators to create the initial set of groups. Co-evolutionary algorithms may also require different approaches to joining in the absence of fitness values. By isolating initialisation, the iteration phase may then assume that all groups and variables are valid, which can also be checked by static analysis. There may be some code duplication between initialisation and iteration phases, particularly when evaluation involves joins and credit assignment. However, ESDL systems are naturally terse, and very few lines of

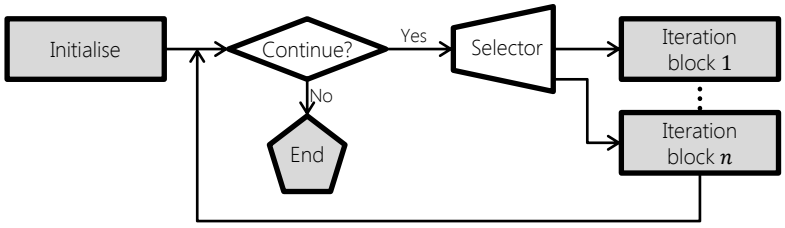

Figure 4.2: Structure of an ESDL system with multiple named iteration blocks.

```
FROM a_generator SELECT 100 population
EVAL population USING an_evaluator

BEGIN default
  FROM population SELECT 100 parents USING a_selector
  FROM parents SELECT offspring USING a_mutator
  FROM population, offspring SELECT 100 parents USING best
END

BEGIN scatter
  FROM population SELECT population USING major_mutation
END
```

Listing 4.16: ESDL system definition with an initialisation section and two iteration blocks.

code are likely to be reproduced. Necessary reproduction also encourages developers to consider whether subtle changes should be made for the initialisation section, for example, using a uniformly random selector rather than a fitness-based one.

Blocks are used to identify sequences of statements belonging to the initialisation section or an iteration section. The initialisation block is implicitly created from the leading statements of an ESDL description; iteration blocks are explicitly labelled with `BEGIN` and `END` statements. Multiple iteration blocks are permitted, with precisely one being used for each iteration. 'Calling' between blocks is not possible; they are per-iteration control flow rather than subroutines. Figure 4.2 shows the general structure of ESDL systems, including a *selector* for switching between multiple iteration blocks.

An iteration block is started by a `BEGIN` statement followed by the block name. Names follow the same rules as variables and groups and cannot match any variable or group name, primarily to avoid reader confusion. The first `BEGIN` statement indicates the end of initialisation block; subsequent `BEGIN` statements are only permitted immediately following the `END` statement that terminates the previous block. The `END` keyword may be followed by any text; it is effectively a comment marker for the remainder of the line. Listing 4.16 shows an example algorithm with two iteration blocks: `default` and `scatter`.

```
i = 0
rate = 0.9
REPEAT (length)
  FROM groupA SELECT groupA USING operator(rate, index=i)
  i = i + 1
  rate = rate - (0.5 / length)
END
```

Listing 4.17: Using a `REPEAT` block to perform a parameter sweep.

By default, each block is used in turn for each iteration in the order they are specified, such that `default` would alternate with `scatter` every iteration. In most cases, this does not lead to a useful algorithm, though it is convenient for testing purposes. For actual algorithms, an external function selects which block to execute, the only constraints being that the block cannot be aborted partway through or the iteration retried with a different block. By externalising this decision as a selector, the difficulty of providing a general syntax or interface to support the range of potential switching reasons is avoided. For example, blocks may be switched based on some global or local statistic, after a certain number of iterations or after a predetermined number of minutes or seconds. Further, authors can choose to present their switching strategy in any suitable form, such as a timeline or flowchart, which may significantly improve the ease of understanding compared to a plain-text notation. [26]

Many algorithms require repetition within each iteration. For example, SSGA [31] typically repeats its iteration phase n times per "generation-equivalent" (iteration), where n is the number of individuals in the main population. Rather than treating each sub-iteration as a major step, a `REPEAT` block may be used to execute a set of statements multiple times. The `REPEAT` clause is followed by an expression specifying the number of repetitions; conditional loops are not supported. A `REPEAT` block is closed with an `END` statement, with nesting levels determining whether the `END` relates to a `REPEAT` block or an iteration block. Using a `REPEAT` block is equivalent to reproducing the statements the number of times specified, with the advantages that the repetition count may vary from iteration to iteration and no actual code reproduction is necessary. Variables may be used to obtain a repetition index or sweep parameters, as shown in Listing 4.17.

4.3.2 Statistics Collection and Termination

As discussed in Section 3.3.8, termination is an experimental concern with a wide variety of options and combinations of criteria. Despite much work using iteration count or a single individual's fitness for termination, real-world applications may prefer criteria based on actual elapsed time or fitness distributions. The reaction to

```
1 FROM random_real SELECT 100 population
2 YIELD population
3
4 BEGIN iteration
5   FROM population SELECT 100 parents USING uniform_random
6   FROM parents SELECT offspring USING mutate
7
8   FROM offspring, population SELECT 100 population USING best
9   YIELD offspring, population
10 END
```

Listing 4.18: Using the `YIELD` statement to identify groups for statistics collection.

any of these criteria may vary, from a signal that the algorithm has succeeded to an automated restart or the execution of a different named block. With such a range of options, both known and yet to be designed, it would not be helpful for ESDL to specify a mechanism for termination or statistical analysis.

Instead, to support these scenarios, ESDL includes the `YIELD` statement. `YIELD` acts as a tagging mechanism, indicating to an underlying framework that the specified group or groups are intended to be fully enumerated, evaluated and collated for statistical analysis. This may be as simple as finding the maximum fitness value or may involve a full diversity and distribution analysis; the design, and resulting performance implications, is left to the user and the underlying framework. While such analysis is based on the individuals in the group when it is yielded, the actual calculations could be deferred or delegated to a separate system. For implementations running on distributed or heterogeneous hardware, this may remove the need to copy large amounts of data unnecessarily.

The `YIELD` statement is followed immediately by one or more group names, separated by commas. No size specifiers, generators or streams are permitted. Similar to "return" statements, which are common to many programming languages, the groups listed are passed to a higher-level handler. However, `YIELD` has more in common with "print" statements, in that information is passed out but execution is not interrupted.[3] Groups that are not yielded are, in general, not available for checking against termination conditions or statistical analysis. Listing 4.18 shows a simple algorithm that includes two `YIELD` statements, at lines 2 and 9, which yield the `population` and `offspring` groups. It is plausible that the underlying framework is configured to test `population` for termination criteria and collect fitness distribution information from both, though for a well-designed framework, this should be easily reconfigurable according to preference.

[3] `YIELD` is most similar to the equivalently named statements in C# and Python, which are used in iterators (coroutines) as continuable return statements.

Statistics collected by the underlying framework are not necessarily made available directly within the algorithm, though allowing external functions access to group statistics may be convenient in some cases. In particular, block selectors (see Section 4.3.1) could benefit from access to relevant metrics. While it may be convenient to have statistical values present directly as variables in a system definition, this would require a potentially restrictive interface across all ESDL implementations. Since externally specified operator and function implementations are already framework specific, limiting access to within functions enables algorithms to take full advantage of the features provided by a specific target framework without having to compromise for portability.

4.4 Example ESDL Systems

The algorithms presented in this section are the same as described in Section 3.5. Since the intent is to demonstrate complete descriptions of existing algorithms, a complete description is necessary for comparison. The code given for each of these algorithms in [12] is used as a reference.

Descriptions from the previous chapter have not been reproduced; rather, they have been substituted by a level of textual description considered appropriate for accompanying an ESDL system, as well as the pseudocode, equations or diagrams required to specify operator behaviour. It is expected that most of the operators used in this section would form part of a standard library and would not typically require definition. However, they are included here as examples and as demonstration of the sufficiency of describing and composing algorithm components. No attempt is made to support or justify the design of the algorithms themselves, and so the examples are shorter than the presentation of a new, novel algorithm.

4.4.1 Evolution Strategies

This ES instance is a $(30 + 20)$-ES with self-adaptive parameter control. Each individual has two real-valued vectors of length two: one representing the solution to the problem and the other being strategy parameters. The initial population of 30 individuals is created by selecting random values for solution vectors from the interval $[-5, 5)$ and strategy values from $[0, 0.5)$.

From the `population` group, `offspring` is created by selection and mutation, is then merged with `population` and reduced to become the next population. In the mutation step, the n strategy parameters in $\vec{\sigma}$ are varied according to

$$\sigma_i' = \sigma_i e^{\left(2\sqrt{1/n}\right)(N_{i1})+(1/2n)(N_{i2})}$$

and the n solution values in X are mutated according to

```
FROM random_indiv(length=2) SELECT 30 population
YIELD population

BEGIN iteration
  FROM population SELECT 20 offspring USING best, es_mutate

  FROM population, offspring SELECT 30 population USING best
  YIELD population
END
```

Listing 4.19: ESDL definition for $(30 + 20)$ -ES.

```
func random_indiv(length=10)
  repeat forever:
    solution and strategy = empty lists
    repeat length times:
      solution.append(random_uniform() × 10 − 5)
      strategy.append(random_uniform() × 0.5)
    yield (solution, strategy)
```

Listing 4.20: Pseudocode for generating ES individuals.

$$X_i' = X_i + \sigma_i N_{i3}$$

where i is each index of the vector and N_{ij} is a normally distributed random value that is reselected for each value of i or j. The `best` operator orders the source group by descending fitness, such that a partition operator will select the fittest individuals to survive to the next iteration.

The ESDL definition for this algorithm is given in Listing 4.19. Pseudocode definitions for `random_indiv` and `es_mutate` are given in listings 4.20 and 4.21, respectively.

4.4.2 Evolutionary Programming

This EP example represents each individual as a sequence of letters from A through Z, storing each component as an integer between 0 and 25 (inclusive). An initial population of 100 random individuals is created, and a variation is created every iteration for each member of the population by randomly adding or subtracting 1 to approximately half of the components. Listing 4.22 shows pseudocode for mutating each individual; components are permitted to move outside of the valid range when mutated [33]. Tournament selection is applied to both the variations and the original individuals by comparing each to seven randomly selected others and ranking every individual by how many individuals they are fitter than. A pseudocode function for

```
func es_mutate(source):
  for each indiv in source:
    n = length(indiv)
    new_strategy and new_solution = empty lists
    for each stddev and value in indiv's strategy and solution:
      n1 and n2 = different normally distributed random values
      new_stddev = stddev × exp(n1 × sqrt(4÷n) + n2 ÷ (2×n))
      new_value = value + new_stddev × random_normal()
      new_strategy.append(new_stddev)
      new_solution.append(new_value)
    yield (new_solution, new_strategy)
```

Listing 4.21: Pseudocode for ES mutation.

```
func ep_mutate(source, step_size [default 1]):
  for each indiv in source:
    new_indiv = empty list
    for each value in indiv:
      new_value = value + {one of [step_size, −step_size, 0, 0] with equal probability}
      new_indiv.append(new_value)
    yield new_indiv
```

Listing 4.22: Single-step mutation for EP as pseudocode.

this selection is given in Listing 4.23. The generator is given in Listing 4.24 and the ESDL definition is shown in Listing 4.25.

```
func tournament(source, k [default 5]):
  results = empty list of individuals ordered by score

  for each indiv in source:
    pool = {k random selections from source}
    score = 0
    for each competitor in pool:
      if fitness(indiv) > fitness(competitor):
        score = score + 1
    results.append(score, indiv)

  return results ordered by score
```

Listing 4.23: Tournament selection as pseudocode.

```
func random_int(length, lowest [default 0], highest [default 100]):
  repeat forever:
    new_indiv = empty list
    repeat length times:
      new_indiv.append(random_integer(lowest, highest+1))
    yield new_indiv
```

Listing 4.24: Integer-valued individual generation as pseudocode.

```
FROM random_int(length=8, lowest=0, highest=25) SELECT 100 population
YIELD population

BEGIN iteration
  FROM population SELECT offspring USING ep_mutate(step_size=1)
  FROM population, offspring SELECT 100 population USING tournament
  YIELD population
END
```

Listing 4.25: ESDL definition for EP.

```
func binary_tournament(source):
  repeat forever:
    indivs = { 2 different random selections from source }
    if fitness(indivs[0]) > fitness(indivs[1]):
      yield indivs[0]
    else:
      yield indivs[1]
```

Listing 4.26: Binary tournament selector (with replacement) as pseudocode.

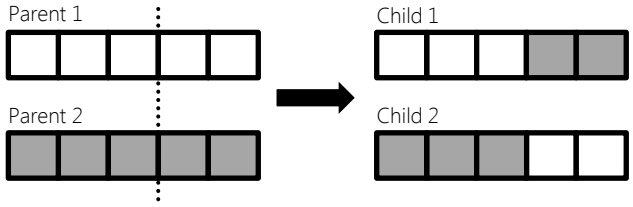

Figure 4.3: Diagrammatic representation of single-point crossover.

```
func crossover(source, per_pair_rate):
  for each parent1 and parent2 in source:
    cut = random_integer(1, length(parent1) - 2)

    if random_probability() < per_pair_rate:
      child1 = parent1[...cut] + parent2[cut...]
      child2 = parent2[...cut] + parent1[cut...]
    else:
      child1 = parent1
      child2 = parent2

    yield child1 and child2
```

Listing 4.27: Single-point crossover operator as pseudocode.

4.4.3 Genetic Algorithms

GA uses a population of n individuals represented as binary strings. This example uses strings of length 64, binary tournament selection as shown in Listing 4.26, single-point recombination as shown in Figure 4.3 and Listing 4.27, and a `mutate_bitflip` operator, shown in Listing 4.28, that inverts approximately one bit in each individual. The ESDL definition is shown in Listing 4.29.

4.4.4 Differential Evolution

DE uses a population of fixed-size vectors of real values. From this population, each vector, the "target," is matched with three randomly selected others; a base $\vec{B}$ and two mutation vectors $\vec{P_1}$ and $\vec{P_2}$. These are used to create a mutant $\vec{M}$ according to (4.2), where F is a constant scaling factor, normally between 0.0 and 1.0. A trial

```
func mutate_bitflip(source, per_gene_rate):
  for each indiv in source:
    new_indiv = empty list
    for each gene in indiv:
      if random_probability() < per_gene_rate:
        new_indiv.append(1 − gene)
      else:
        new_indiv.append(gene)
    yield new_indiv
```

Listing 4.28: Point mutation operator as pseudocode.

```
FROM random_binary(length=64) SELECT (100) population
YIELD population

BEGIN generation
  FROM population SELECT (100) parents USING binary_tournament
  FROM parents SELECT population \
      USING crossover(per_pair_rate=0.98, two_children), \
            mutate_bitflip(per_gene_rate=1/64)
  YIELD population
END
```

Listing 4.29: ESDL definition for GA.

vector is created by selecting components from the target and the mutant with equal probability. If the trial vector is as fit as or fitter than the target, it is retained; otherwise, it is discarded.

$$\vec{M}=\vec{B}+F\left(\vec{P_1}-\vec{P_2}\right) \tag{4.2}$$

The `random_tuples` joiner creates an unbounded stream of joined individuals by randomly selecting from each of the source groups; the `distinct` setting indicates that repetitions are not permitted within each tuple. A `mutate_de` operator based on (4.2) returns a stream of mutants. By joining rather than merging these with population, the `crossover_tuple` operator (shown in Figure 4.4) can combine each mutant with its target, rather than the individuals that are adjacent in each group. Finally, joining the original population with `trials` and applying `best_of_tuple` performs a comparison between the target and the trial, keeping whichever is more fit or the first (from `trials`) if they are equal. The ESDL definition is given in Listing 4.30.

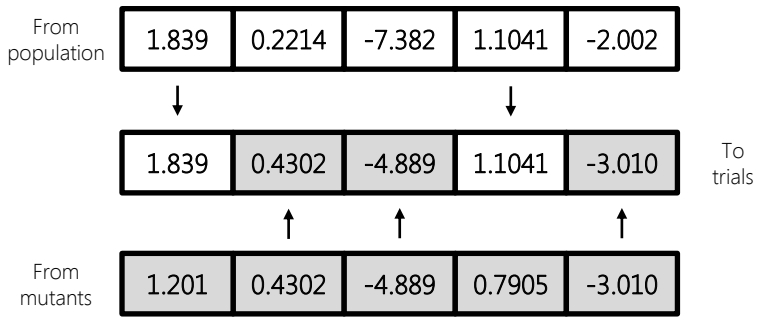

Figure 4.4: Diagrammatic representation of uniform crossover selecting components from each parent.

```
FROM random_real(length=3,lowest=-5,highest=5) SELECT 30 population
YIELD population

BEGIN generation
  JOIN population, population, population INTO bases USING random_tuples(distinct)
  FROM bases SELECT mutants USING mutate_de

  JOIN population, mutants INTO parents
  FROM parents SELECT trials USING crossover_tuple

  JOIN trials, population INTO trial_target_pairs
  FROM trial_target_pairs SELECT population USING best_of_tuple

  YIELD population
END
```

Listing 4.30: ESDL definition for DE.

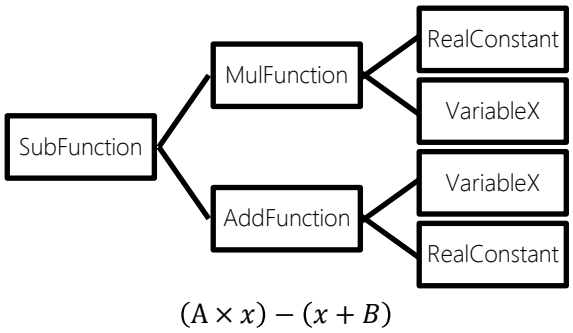

Figure 4.5: GP individual represented as a function tree.

4.4.5 Genetic Programming

Genetic Programming uses a tree-based structure to represent executable programs as functional expressions. The programs are typically based on Lisp syntax, though the available functions and terminals (either constants or externally provided variables) are language independent and problem specific. Here individuals are represented as a linked tree.

Each program contains a root node that is an instance of a function class, such as those shown in Listing 4.31. The population is initialised from the `random_program` generator, shown in Listing 4.33, which produces trees like that shown in Figure 4.5. (Note that `random_program` provides default values for each parameter, as required.) Each generation, parents are selected using the same selector as for GA (Section 4.4.3) and partitioned into three groups: one for crossover, one for mutation and one for reproduction. After variation, the groups are merged back together to form the next `population`.

Crossover selects a random function node in each tree of two individuals and exchanges the branches beginning from that node. In order to maintain the depth limit, nodes can be selected carefully or "pruned" by replacing nodes at the depth limit with terminals. Mutation is similar; a random function node is selected in a single individual and the entire branch beginning from that node is replaced. Rather than coming from another individual, the replacement branch is randomly generated as for the initial population.

Evaluation of GP individuals varies depending on the type of problem. Some 'robot control' problems use a set of commands as terminals and execute the program many times to obtain a sequence of instructions, while expression generators provide the values for one or more variables and evaluate the individual as a function tree. The functions in Listing 4.33 are intended for arithmetic expressions; Listing 4.32 defines an evaluator that compares individuals against x^3+x^2+x+1 and Listing 4.34 shows the ESDL definition.

```
class AddFunction:
  child_count = 2

  def __init__(self, A, B):
    self.A, self.B = A, B

  def evaluate(self, **data):
    return self.A.evaluate(**data) + self.B.evaluate(**data)

class VariableX:
  child_count = 0

  def evaluate(self, **data):
    return data['x']
```

Listing 4.31: Two example GP nodes specified as Python classes.

```
def eval_expression(individual):
  error_sum = 0.0
  for _ in range(20):
    x = random()
    expected = x**3 + x**2 + x + 1
    actual = individual.evaluate(x=x)
    error_sum += (actual - expected) ** 2

  # Scale the error sum so that fitter values are larger (closer to one)
  return 1.0 / (1.0 + error_sum)
```

Listing 4.32: Evaluator for comparing against $x^3 + x^2 + x + 1$, specified in Python.

```
from random import random, choose
FUNCTIONS = [AddFunction, SubFunction, MulFunction, DivFunction]
TERMINALS = [RealConstant, VariableX]

def one_random_program(max_depth, terminal_prob):
  '''Called recursively to create program trees.'''
  if max_depth == 0 or random() < terminal_prob:
    return choose(TERMINALS)()

  func = choose(FUNCTIONS)
  parameters = [one_random_program(max_depth-1, terminal_prob)
                for _ in range(func.child_count)]
  return func(*parameters)

def random_program(max_depth=10, terminal_prob=0.1):
  '''Infinitely returns random programs.'''
  while True:
    yield Individual(one_random_program(max_depth, terminal_prob))
```

Listing 4.33: `random_program` generator specified in Python.

```
FROM random_program(max_depth=15) SELECT (size) population
max_depth = 7
YIELD population

BEGIN generation
  FROM population SELECT (size) parents USING fitness_proportional
  FROM parents SELECT (size*0.9) p1, (size*0.02) p2, p3

  FROM p1 SELECT o1 USING crossover(max_depth)
  FROM p2 SELECT o2 USING mutate(max_depth)

  FROM o1, o2, p3 SELECT (size) population
  YIELD population
END
```

Listing 4.34: ESDL definition for GP.

```
FROM random_binary(length=8) SELECT (100) population
YIELD population

BEGIN generation_equivalent
  REPEAT 100
    FROM population SELECT (2) parents, rest \
        USING binary_tournament(without_replacement)
    FROM parents SELECT offspring \
        USING crossover(per_pair_rate=0.98, two_children), \
              mutate_bitflip(per_gene_rate=1/64)
    FROM offspring, rest SELECT population
  END
  YIELD population
END
```

Listing 4.35: ESDL definition for SSGA.

4.4.6 Steady-State Genetic Algorithms

SSGA uses operators that are nearly identical to GA (Section 4.4.3); only the ESDL definition needs to be changed. The `fitness_proportional` selector has `no_replacement` specified (implicitly; equivalent to passing `true` in this context), which selects two distinct parents and stores the rest of the population into `rest`. After variation, `rest` is merged with `offspring` to produce the next population. The iteration block has been renamed from `generation` to `generation_equivalent` to reflect standard SSGA terminology, though this has no effect on the behaviour.

The updated ESDL system for SSGA is shown in Listing 4.35, using the `fitness_proportional` operator from Listing 4.26.

4.4.7 Particle Swarm Optimisation

As discussed in Section 3.5.7, despite not being considered a traditional EA, PSO can be represented using the streams and operator model and ESDL. Each individual contains two real-valued vectors of equal length, where one represents the position of the particle and the other the velocity. A population of individuals, known as the `swarm`, is maintained. Alongside this swarm, a second group of individuals called `p_bests` contains the best locations seen by the swarm member at the matching index.

Every iteration, the velocity of each particle is updated according to the combination of some random values, the relative location of the particle's `p_bests` member and the best location currently known by any particle in a predefined neighbourhood. Here, the neighbourhood includes all particles, as in [12], and the velocity is updated according to:

$$v_{id}^{'} = v_{id} + c_1 r_1 \left(p_{id} - x_{id}\right) + c_2 r_2 \left(p_{gd} - x_{id}\right)$$

where x_{id} and v_{id} are the position and velocity of particle i in dimension d, p_{id} is the best position known by particle i and p_{gd} is the best position known by any neighbour. c_1 and c_2 are constant learning factors and are assumed here to both be 2. r_1 and r_2 are independent random values between 0.0 and 1.0, resampled for each dimension of each individual. If the updated velocities exceed the specified limit, they are replaced with that limit.

After updating the velocity, positions are updated assuming a time delta of one; in effect, the current velocity vector of each particle is added to the position vector:

$$x_{id}^{'} = x_{id} + v_{id}$$

When an updated position is beyond the limits of the search space, it is moved back into the search space and the velocity in the same dimension is reversed, as if the particle bounced off a wall.

Pseudocode for the `random_particle` generator is given in Listing 4.36 and the ESDL definition is shown in Listing 4.37.

4.5 Chapter Summary

This chapter has described Evolutionary System Definition Language (ESDL), a domain-specific language for describing EAs based on the model in Chapter 3. ESDL encourages a structured approach to EAs by requiring a clear separation between the implementation of each operator and the overall composition of the algorithm. The loosely-coupled designs promoted by this approach are more likely to result in

```
func random_particle(length, low_limit [default 0], high_limit [default 1]):
  repeat forever:
    new_velocity = list()
    new_position = list()
    repeat length times:
      new_velocity.append(0)
      new_position.append(random_uniform(low_limit, high_limit))
    yield create_particle(new_position, new_velocity)
```

Listing 4.36: Generator definition for PSO as pseudocode.

```
FROM random_particle(length=2, low_limit=-5, high_limit=5) SELECT 50 swarm
FROM swarm SELECT p_bests
YIELD swarm

BEGIN iteration
  FROM swarm SELECT 1 g_best USING best
  JOIN swarm, p_bests INTO particles_with_pbest
  FROM particles_with_pbest SELECT swarm USING \
       update_velocity(global_best=g_best, low_limit=-100, high_limit=100), \
       update_position(low_limit=-5, high_limit=5, bounce)
  YIELD swarm
END
```

Listing 4.37: ESDL definition for PSO.

easily reusable components that can be shared between different algorithms, which was noted as an issue in Chapter 2. Operator interfaces and their interactions with groups, streams and individuals are strictly defined but sufficiently general to not restrict those who are creating new algorithms.

DSLs are known to be beneficial for reducing the complexity of software models, often to the point where domain experts who are not programmers can develop using them. ESDL is intentionally designed as a communication tool, with an emphasis on clear and correct presentation and ease of use. A clear separation exists between the model of EAs and ESDL, allowing the abstraction and the specification language to be used independently if desired. Taken together, ESDL and the model provide a concise and eloquent approach to designing, implementing and sharing algorithms, offering the guidance that Chapter 2 noted as lacking.

Existing algorithms can be described with ESDL, often more succinctly than in other forms because many aspects are already defined. In conjunction with operators specified separately from the ESDL definition, a range of EAs and similar algorithms can be specified concisely and unambiguously. The examples in this chapter included Genetic Algorithms, Differential Evolution and Particle Swarm Optimisation.

Despite the focus on presentation, ESDL is suitable for compilation into an executable language. Chapter 5 discusses many of the aspects related to implementing an ESDL system or creating a compiler to do so automatically. Chapter 6 discusses ESDL's application as a tool for designing, sharing, evaluating and testing algorithms, as well as an actual implementation of a framework that uses ESDL to compose algorithms from an operator library.

Chapter 5

Execution

The previous chapter presented ESDL, a language for describing the composition of EAs. One of the significant benefits of ESDL is independence from conventional programming languages. However, without the ability to consistently interpret the language, it is impossible to guarantee the behaviour of any algorithm specified using it. ESDL is intended to disambiguate algorithm descriptions, and hence guaranteed behaviour is necessary. This chapter reviews prior examples of implementing EA software, describes the intended behavioural interpretation of ESDL and discusses specific implementation concerns, such as data types, memory model and sequencing. These models are necessary to ensure that implementations of ESDL produce equivalent results while not preventing frameworks from providing distinguishing functionality.

5.1 Evolutionary Algorithm Software

As discussed in Section 2.2.3, the majority of EA research appears to use custom-written software. Most researchers in the field have programming skills and the confidence to produce working code. Unfortunately, code that is written for a particular publication is normally not released to the public, making a survey of the software in use difficult to conduct. Software that is released typically consists of libraries and frameworks; that is, software deliberately developed to be shared and used by others. When intended for reuse, the structure and design of software can be very different to prototypes that are intended for only a single use. This section briefly reviews a selection of easily found[1] EA libraries and frameworks to identify common architectural features.

[1] Within the first 10 results when searching for "evolutionary algorithm software" or "evolutionary algorithm library" with popular search engines, as well as some that have been heavily promoted at various events during 2010–2012.

Table 5.1: Frameworks and libraries for implementing EAs.

	Name	Language	URL
a	**AForge.Genetic**	C# library	code.google.com/p/aforge/source/browse
b	**Algorithm::Evolutionary**	Perl framework	opeal.sourceforge.net
c	**Distributed Evolutionary Algorithms in Python**	Python library	code.google.com/p/deap
d	**EASEA**	C++ framework	lsiit.u-strasbg.fr/easea
e	**ECJ**	Java framework	cs.gmu.edu/~eclab/projects/ecj
f	**Evo**	C# library	evo.codeplex.com
g	**Evolving Objects (EO)**	C++ library	eodev.sourceforge.net
h	**Genetic Algorithm Utility Library (GAUL)**	C++ library	gaul.sourceforge.net
i	**HeuristicLab**	C# framework	dev.heuristiclab.com
j	**Java Genetic Algorithms Package (JGAP)**	Java library	jgap.sourceforge.net
k	**stupidalgorithm**	C++ framework	code.google.com/p/stupidalgorithm
l	**Watchmaker**	Java library	watchmaker.uncommons.org

Twelve software packages were reviewed, listed in Table 5.1, consisting of a mixture of libraries and frameworks in a range of implementation languages. Here, as in in Section 4.1, a library is defined as a collection of code (typically objects in object-oriented languages) that cannot be used as an entire application without writing more code, while a framework can be configured and executed without the user necessarily having to create a new program. All of the frameworks include or require libraries, though not all of them allow these libraries to be used independently.

All of the reviewed software packages constitute component frameworks, that is, they are each a library of components, a software architecture based around these components and some form of "glue" for connecting components and allowing data transfer [68]. Every package is written using object-oriented principles, typically, though not universally, representing algorithms and various operators as distinct classes. The type of glue and communication protocols also differ between implementations, encompassing code-only approaches, configuration files, drag-and-drop design, explicit data passing and implicit shared scopes. However, the general notion of components as "self-contained configurable entities which can be composed to build an application" exists throughout [61, p. 25].

From Table 5.1, packages *f*, *h* and *k* appear to be unmaintained, while *a* has not been updated recently despite the wider library it belongs to being under active

development. Apart from framework *d*, none of the implementations listed has any form of compatibility with the others; taking an experiment from one to another requires a complete re-implementation. Framework *d* uses a separate library, rather than including its own, and generates code for library *g*. However, the compilation step is necessary; source code is not directly interchangeable.

Each package uses an architecture that can be said to follow either a population-centric (*a*, *d*, *e*, *h*, *j*) or an algorithm-centric (*b*, *c*, *f*, *g*, *i*, *k*, *l*) model. Population-centric models embed knowledge of the evolutionary process (selection, variation operators and rates) into the population—the group of individuals 'knows' how to evolve itself. In code, this may look like `population.mutate_ga()`. Algorithm-centric models treat groups of individuals as data structures with no, or few, EA-specific behaviours; an algorithm class 'knows' how to evolve any population that is provided. In code, this would appear as `mutate_ga(population)`. The algorithm-centric approach appears to provide a better separation of concerns, since these operations are not strictly properties of the groups, though it could limit the range of data structures that are usable as populations. In practice, the difference between population-centric and algorithm-centric is largely conceptual and affects the location of code rather than its behaviour.

Apart from *h* and *k*, all of the libraries allow individuals of arbitrary type if variation operators are available (*h* and *k* require complete algorithm implementations for new types). The static language implementations provide abstract classes or use templates/generics, while the dynamic languages either test for particular object members (*b*, in Perl) or assume the user has provided suitable types (*c*, in Python). For the algorithm-centric implementations, operators are specified as part of the same configuration as the individual representation or generator, simplifying the task of ensuring the types are matched. None of the reviewed software appeared to provide methods to select compatible operators automatically, though framework *i* limits the list of operators shown in its user interface to those that support the selected representation.

Operator specifications differ significantly between the various implementations. Since the choice and combination of operators distinguishes each EA, it is worth investigating and comparing the approaches and interactions of the packages in detail.

Libraries *a*, *f* and *h* all provide fixed algorithms with one or more substitutable operators. In order to change an algorithm structure, such as the order in which operators are applied, the user must define a new population or algorithm class. All three of these libraries are inactive, probably indicating there is little interest in software that does not easily allow algorithms to differ from predefined structures.

Framework *d* uses a fixed structure but requires users to implement each variation operator, while a fixed list of selectors is provided for before and after variation.

Packages *b*, *g*, *j* and *l* use partially fixed algorithm classes, allowing users to define a wider variety of algorithms without resorting to library-level programming. One or two selection operators may be specified, which are applied before and/or after variation. Variation is performed by applying operators from a user-provided list to the selected individuals. The user can include any operators in this list in any order, for example, applying mutation prior to crossover. Libraries *g* and *l* allow considerable complexity in the operator list through aggregation operators that combine multiple variation operators and select one to apply based on relative probabilities. Packages *b* and *j* could also implement similar operators, though they do not appear to include any.

Library *c* provides a number of basic algorithms, but also allows arbitrary algorithms to be coded using the operators directly, rather than having to implement a new algorithm class. Library *k* is a collection of fixed algorithms; any changes beyond parameter values require the user create a new class.

Frameworks *e* and *i* support arbitrary operator networks, including selection and variation stages, as well as complete algorithms. Arbitrary connections allow algorithms to use multiple selection operators with separate variation operators for each, permitting the creation of practically any imaginable algorithm. However, the cost associated with arbitrary networks is the complexity of configuration. Framework *e*, the most mature package reviewed, uses plain-text configuration files based around the variation of "subpopulations" through "breeding pipelines"—effectively equivalent to the groups and operators as defined in Chapter 3. Listing 5.1 shows a partial parameter file for implementing a simple GA.

Such a description format allows for very precise control over the combination of operators in an algorithm, though the verbosity produces lengthy parameter files. There are mechanisms for including parameters from other files, although these often obscure the actual settings in use as much as simplifying configuration. Nonetheless, of all the software packages reviewed, it has been most cited by other researchers.

Framework *i* is completely GUI-based and all configuration is performed in a drag-drop environment. However, as with framework *e*, the flexibility leads to complicated operator graphs; Figure 5.1 shows the complete graph for a simple GA, including termination conditions. Rather than representing a simple flow, the graph is interpreted as a tree with a single (conditional) cycle. This representation is different from the typically linear algorithm styles, but can simplify parallel or partial execution.

```
pop.subpops  = 1
pop.subpop.0 = ec.Subpopulation

pop.subpop.0.size              = 10
pop.subpop.0.duplicate-retries = 0
pop.subpop.0.species           = ec.vector.VectorSpecies

pop.subpop.0.species.fitness = ec.simple.SimpleFitness
pop.subpop.0.species.ind     = ec.vector.BitVectorIndividual

pop.subpop.0.species.genome-size    = 20
pop.subpop.0.species.crossover-type = one
pop.subpop.0.species.crossover-prob = 1.0
pop.subpop.0.species.mutation-prob  = 0.01

pop.subpop.0.species.pipe = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.source.0 = ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.source.1 = ec.select.TournamentSelection

select.tournament.size = 2
```

Listing 5.1: Partial GA configuration for framework ***e*** (ECJ).

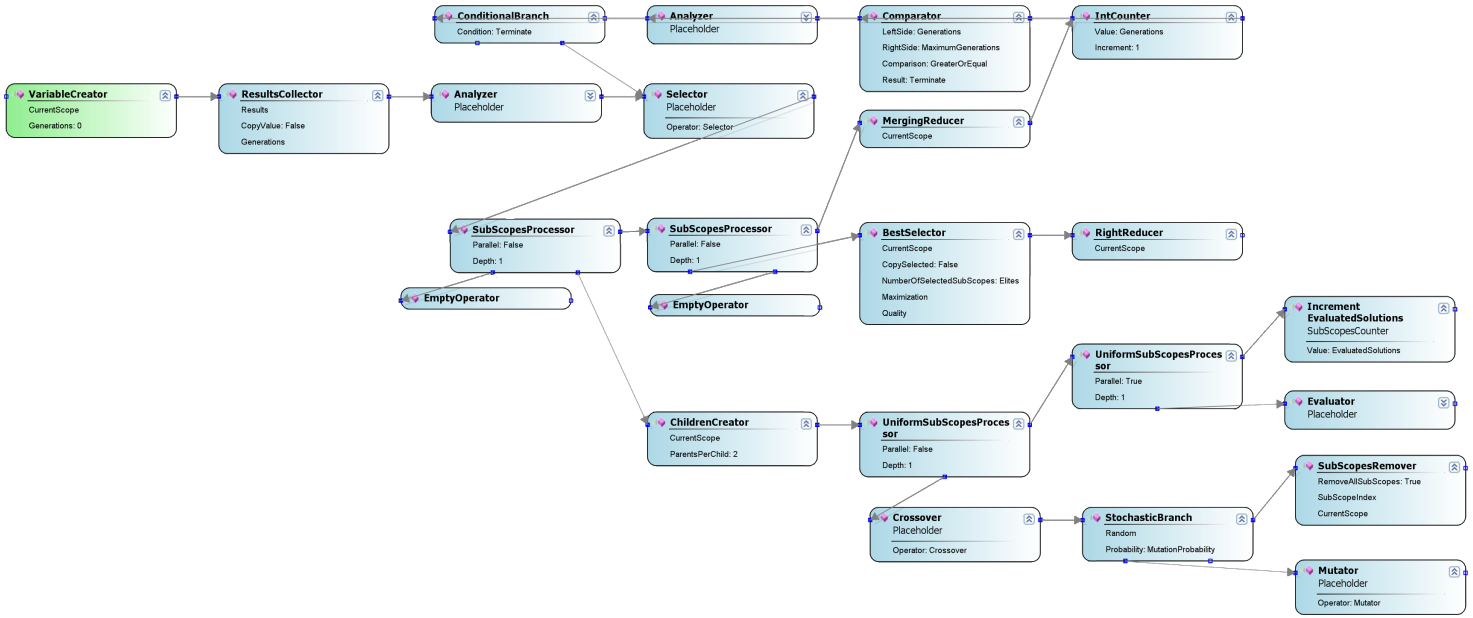

Figure 5.1: An operator graph from framework *i* (HeuristicLab) for a simple GA.

A second architectural concern is how data is shared within the algorithm, particularly with respect to collections of individuals. The primary distinction between data passing approaches is whether they are implicit or specified explicitly. Explicit approaches require the designer manually connect the output of one operator to the input of the next—such as storing a value returned from a function and later specifying it as an argument. Implicit approaches allow operators to access shared data storage, similar to a global variable though, in such a constrained environment, without the usual shortcomings. From the point of view of the operator, data may be provided as one individual (or enough to produce one result), a group or a reference to the group in shared storage. When an entire group is available, the operator may be able to modify the individuals in place rather than having to create copies.

All of the reviewed packages apart from *a*, *d* and *k* define operators as independent components. Packages *a* and *d* use member functions on the individual type while *k* uses methods on the algorithm type. Packages *a*, *b*, *d*, *g*, *h* and *i* explicitly provide individuals one (or more, for sexual operators) at a time to variation operators while the other six provide the entire group; selectors are always provided random access to the entire group. Implementations that provide the entire group use the same operator interface regardless of arity; those packages passing individuals separately use different method signatures or interfaces for asexual and sexual operators.

Frameworks *e* and *i* have shared scopes: operator connections are defined in the configuration and used at runtime to retrieve and store groups. Shared scopes allow connections to be specified using late bound names; however, issues of immutability, synchronisation and sequencing are exacerbated when separate components may have aliased access to the same data. Framework *e* mitigates these issues within the provided algorithm implementations, using global threading parameters to subdivide and schedule the required work, ensuring that operator instances use separate data. Framework *i* uses a unified hierarchical architecture to represent groups, individuals and their properties; a "sub-scope" is created for each individual. This allows the algorithm controller to manage the data each operator can access.

Individual and group immutability varies throughout, and within, the reviewed implementations. Packages *a*, *c*, *f*, *h*, *j*, *k* and *l* prevent individuals from modification by convention, either providing operators with mutable destinations or requiring them to create clones. The other packages (*b*, *d*, *e*, *g* and *i*) perform modifications directly on the individual and use explicit cloning to avoid corruption. Framework *e* clearly defines the points where cloning occurs—selection operators create new groups containing clones; variation operators directly modify this group—while libraries *b* and *g* require the algorithm implementer to define and follow their own convention. Framework *i* includes configuration options to specify whether par-

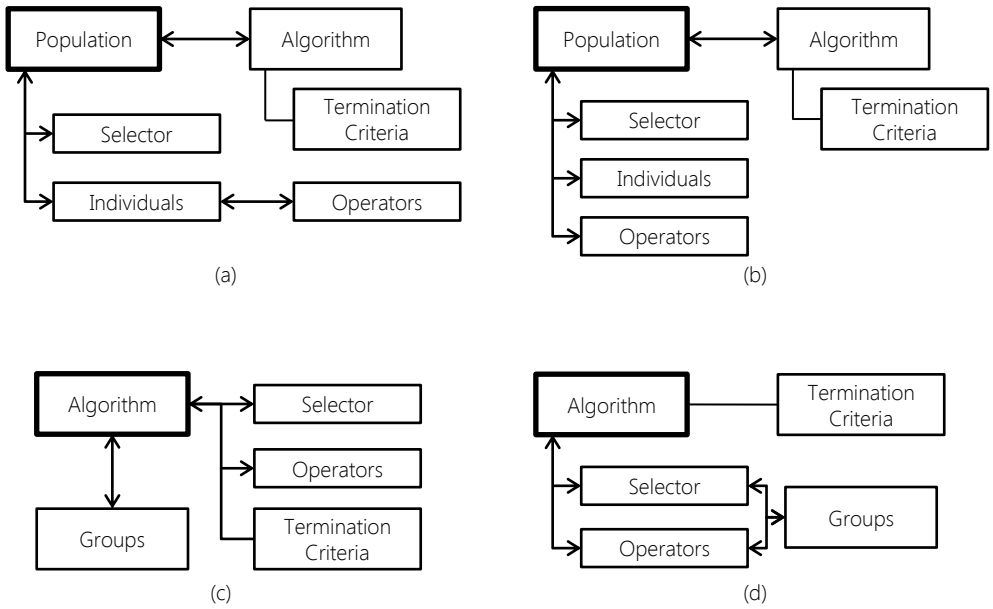

Figure 5.2: Overview of the reviewed software architectures.

ticular selection operators create clones, since asexual variation operators directly modify the source individuals.

Figure 5.2 shows a graphical summary of the observed architectures. Architecture *a* represents a population-centric algorithm similar to that used in library *a* and framework *d*: the population is created with a selection operator and specific individual representation. Individuals in this case are directly associated with their variation operators; the algorithm is no more than a loop that repeats until the termination criteria are met. Architecture *b* is similar but specifies variation operators separately to the individual type; this is similar to packages *e*, *h* and *j*.

Architecture *c* is algorithm-centric, with operators specified independently and access to groups of individuals is controlled by the algorithm. Packages *b*, *c*, *f*, *g*, *k* and *l* follow this architecture, which allows very flexible operator combination though normally requires the algorithm to be manually coded rather than specified though simpler configuration formats. Architecture *d* is similar with the use of a shared scope rather than passing groups through the algorithm. Framework *i* uses this architecture to reduce coupling between operators.

The wide range of approaches to implementing EAs reflects the range of developers and developer backgrounds but makes it difficult to identify the best approach objectively. Further, advances in hardware architectures change the suitability of certain software designs. Most of the reviewed packages use mutable groups or individuals, which used to be an essential optimisation but is unnecessarily complex with abundant memory or multiprocessing systems.

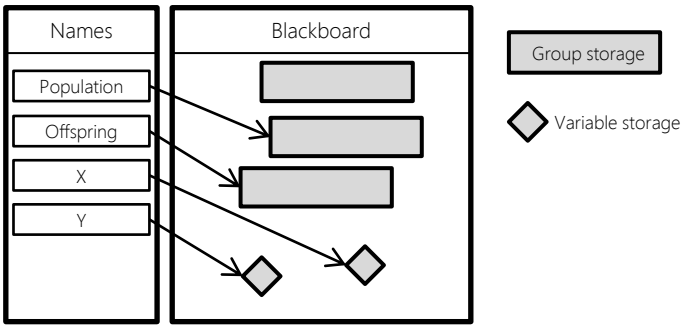

Figure 5.3: Blackboard with named references.

Section 5.2 describes the architecture intended for interpreting algorithms written in ESDL in detail. Section 5.3 compares the decisions made in Section 5.2 to those made by the software packages reviewed in this section.

5.2 Interpreting ESDL Systems

For proper and consistent interpretation of ESDL across multiple platforms, languages and frameworks, a number of areas relating to execution and architecture must be specified. These include a memory model, detailing where data is stored and how it may be accessed, a sequencing model, specifying the limits of overlapping and out-of-order execution, and an extensibility model, formalising the approach by which users can create and use their own components.

This section describes these models in general terms, avoiding references to specific programming languages, styles or patterns except to identify examples raised in Section 5.1. Actual implementations should have the freedom to use approaches that provide the best performance as long as the results are the same. Chapter 6 describes an actual implementation as an example of how these models influence implementation decisions.

5.2.1 Memory Model

Similar to framework *i* from Section 5.1, data storage for ESDL systems consists of a central repository or "blackboard" [41, p. 12], which is used for storage of groups and variables referenced from the system. Items on the blackboard are accessed by name using reference semantics; the names are independent of data and can be re-bound to refer to a different storage location, as shown in Figure 5.3. There is no nested or stack-based scoping mechanism as in general-purpose languages; rather, all names on the blackboard and the data they refer to are available throughout the ESDL definition. When external code is invoked with a group or variable reference, the name is resolved before being passed; the name itself does not leak out from the ESDL definition unless done deliberately, such as through a `YIELD` statement.

Conceptually, the blackboard is write-once, allowing all reads to assume that the data is immutable. In practice, the contents may be changed or deleted provided the sequencing guarantees described in Section 5.2.2 are met. With the restrictions on blackboard accessibility, reference counting is sufficient memory management for entire variables, groups or individuals, since reference cycles are impossible. Values such as iteration and evaluation counts are not stored on the blackboard, though implementations may choose to make them available within the ESDL system or operators. Implementations are free to select a format to store data on the blackboard that is optimal for their particular platform. However, in order to achieve correct and relatively efficient performance for the range of possible algorithms and operators, a number of constraints must be considered.

Individuals are streamed between operators one at a time on a private channel; only the receiver can read the individuals. Implementations can implement batch transfers by passing individuals in groups up to or larger than the required size, providing order is maintained. Operators may be connected using a publisher-subscriber model, where the earlier operator in the chain pushes each individual to the later, or a pull model, where the later operator requests the exact number of individuals it requires. Both are equally valid, though it is also possible for other models to be used. A pull model minimises processing but may be more difficult to distribute across processors. Push models require special handling to allow infinite streams. Selecting a performant streaming model is left to the implementation.

By design, most operators will assume that only sequential access to the source group is possible. Since operators may be chained arbitrarily, there is no way to guarantee or infer whether random access is available. Operators requiring random access, such as those that reorder based on fitness, have to be able to cache the incoming stream in order to handle all possible operator arrangements. The model implementation should optimise for sequential access, while allowing random access where necessary. Iterator (sometimes called enumerator or generator) and list or array constructs common to many languages support this approach and are likely to be familiar to operator developers.

Unlike streams, the contents of groups may be accessed more than once, which generally requires that the group be stored rather than only existing temporarily within a particular operation. When it can be determined that a group is used only once and does not persist between iterations, there is no need to retain its contents; otherwise, the source stream will need to be fully or partially enumerated and stored on the blackboard. Any data structure is suitable for this purpose provided the original order of the individuals is retained. The selected data structure may be of fixed size if the number of individuals it contains can be determined. It is not always possible to determine the final size of a stream before enumerating it, so some form

of dynamically expanding array is required, but since many algorithms depend on groups being an expected size the final size or an upper-bound may be known or can be calculated.

Non-group variables are not considered to have any type, or alternatively, to have an opaque type supporting only assignment. ESDL does not require any operations other than assignment and passing as an argument (nothing more than a special case of assignment), which allows implementations and individual developers the freedom to use any types for any purpose. For the sake of ESDL's human readers, once a name is used as a variable it may not be used for a group, and neither may a group name be used as a variable. While in practice it may be possible to use groups as variables or variable as groups, or even to treat them completely independently, any language is easiest to understand when each word means precisely one thing.

It is expected that most implementations will be able to handle numbers and basic arithmetic operators on variables despite the opaque types.[2] Beyond basic arithmetic, however, there are no defined operators or object models for non-group variables. This significantly limits the complexity of the systems that are written using ESDL, as well as simplifying the creation of compilers and interpreters—there is no need to support object member functions, array indexing, function overloading or type casting.

The purpose of the `YIELD` statement is to expose the contents of the blackboard to external code. Ignoring sequencing, yielding an immutable group is simply a case of providing a reference to the external handler. For implementations using mutable groups, the contents of the group yielded must be synchronised with the `YIELD` statement to ensure that if the group is modified, the changes are not observable until another `YIELD` occurs. Yielding is equivalent to copying the group into publicly accessible storage. Any analysis of the yielded group may occur immediately and synchronously, or be deferred.

5.2.2 Sequence Model

While a correct sequencing model could simply insist that all operations be performed atomically in the order specified in the ESDL definition, it is likely that implementations will seek to distinguish themselves though improved execution performance. To avoid limiting flexibility in this area, a higher complexity model is defined to enable implementations to reorder and parallelise execution without producing incorrect or inconsistent results. The remainder of this section considers the flexibility available to multi-processor execution of EAs while maintaining correct behaviour in the absence of strict sequential consistency.

[2] Omitting this as a strict requirement allows conforming implementations to be created for machines that cannot support it. While seemingly far-fetched, there are already distributed platforms that do not easily allow centralised calculations in a performant manner.

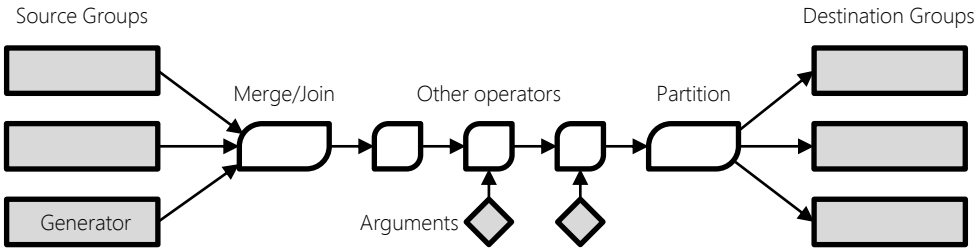

Figure 5.4: The general store operation.

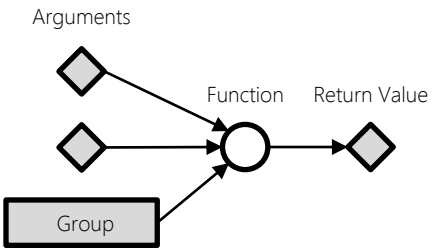

Figure 5.5: The general function operation.

Interactions between processors in a multi-processor system are considered to relate only to externally observable events, typically memory reads and writes. Here, all possible interactions are isolated to the shared blackboard. With the reference semantics described in Section 5.2.1 and the write-once nature of the blackboard, as long as no named reference to the data exists until after it is initialised, there is no possibility of accessing invalid values. In a practical implementation, synchronising the deallocation of memory is necessary, though simple reference counting is sufficient for unified memory architectures.

To simplify discussion, two executable operations are defined: *stores* and *functions*. A store is an operator chain from a list of source groups to a set of destination groups, as shown in Figure 5.4. A function represents a transformation from one or more variables or groups to one variable using externally defined code, as shown in Figure 5.5. Both of these operations produce results that are stored on the blackboard and hence have visible effects. Side effects are deliberately not accounted for, since no statement in ESDL is capable of producing side effects and any other varying state must therefore belong to external code. Allowing side effects is an implementation specific decision; an implementation allowing them will require stricter sequencing restrictions than those described here.

If destination groups and return values are stored on the blackboard before variable names are updated, the need for explicit synchronisation on variable initialisation is avoided. Further, recursive references are impossible, since an operation's

```
FROM population SELECT 100 parents USING tournament(k), unique
YIELD parents
p = p1 + p2

FROM parents SELECT offspring1 USING crossover
rate = calculate_rate(prob=p, size=100)
FROM parents SELECT offspring2 USING mutate(per_indiv_rate=rate)

bias = calculate_bias(source=offspring2)
FROM offspring1, offspring2 SELECT 100 population \
    USING biased_selection(bias)
YIELD population
```

Listing 5.2: ESDL system definition with dependencies between stores and functions.

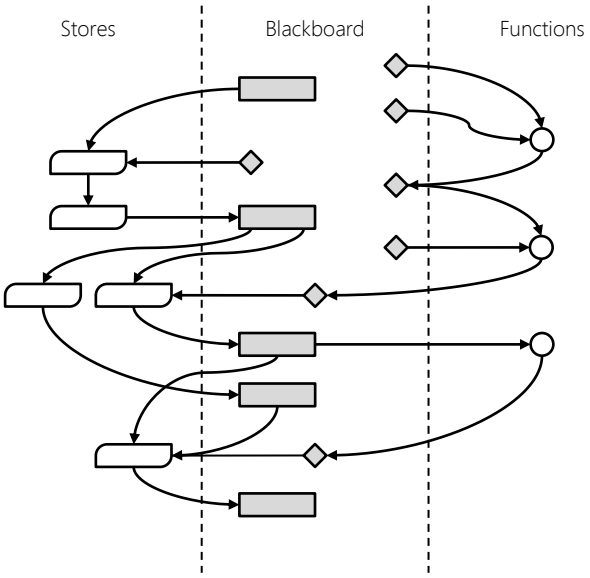

Figure 5.6: Store and function operations interacting through a shared blackboard.

output cannot be used as its own input in the absence of a named reference.[3] This behaviour is similar to normal function calls, and though renaming semantics are used rather than copy assignment, the result is identical.

Groups and variables may be accessed by both stores and functions, though complete isolation between the two types of operation may be achieved with a shared blackboard. This approach may be valuable where processors are dedicated or optimised for particular operations. The system in Listing 5.2 is shown in Figure 5.6 with a clear separation between stores and functions; implying the blackboard rather than explicitly showing it results in the equivalent but simplified Figure 5.7.

Any operation with all inputs available may begin processing, which is trivial to detect for functions but more complex for multi-operator stores. Operators that

[3] An operator or function implementation may use recursion, since they have full access to the functionality of the underlying language. However, nothing specified using ESDL can produce recursion directly.

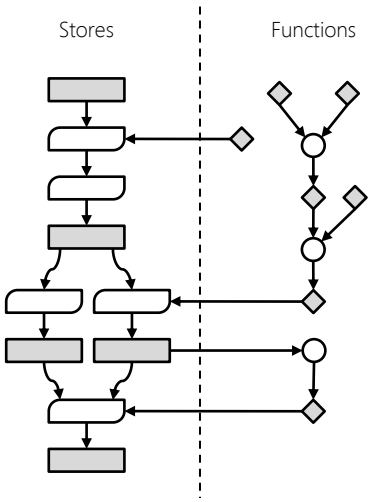

Figure 5.7: Simplified representation of Figure 5.6 that omits the blackboard.

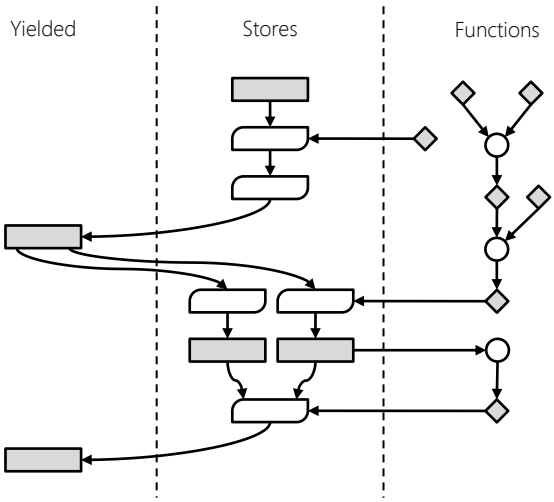

Figure 5.8: Yielding groups under an immutable memory model.

take parameters need to wait for their arguments before beginning, though previous operators in the chain may begin processing in some circumstances. Forcing a store to wait for all its dependencies before beginning is safest and likely to be efficient enough, since most arguments will be constants or arithmetic expressions.

Depending on the memory model used, the `YIELD` and `EVALUATE` statements may not require any sequencing consideration. For immutable groups and variables using reference semantics, both statements may be treated declaratively, in effect, `YIELD` as an access modifier and `EVALUATE` as a static property of the group, with notifications for an asynchronous analysis or evaluation tasks. This case, shown in Figure 5.8, effectively stores yielded groups in a separate memory location.

Alternatively, when not using immutable data structures, sequencing becomes a significant concern. Rather than adjusting the storage location of yielded groups, a copy must be made at some point after the group was last modified and before it is next modified. Omitting the copy may be possible for implementations that

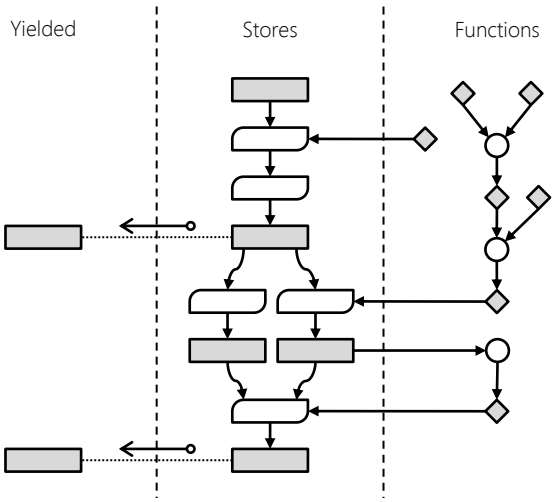

Figure 5.9: Yielding groups by copying to publicly accessible memory.

perform all required analysis synchronously and do not need to retain the individuals belonging to the group when yielded. Figure 5.9 shows the system from Listing 5.2 with explicit copies.

The `EVALUATE` statement changes the evaluator associated with one or more groups. Depending on how eagerly fitness is evaluated, some evaluations may occur immediately, though to support the propagation rules in Section 4.2.4, an association between the group and the evaluator must be made. It is possible for evaluator associations to be determined statically, though parameter values may not always be calculable at compile-time. Implementations are left to select their own strategy for evaluations, since representation and the software architecture will provide tighter constraints than portability requires; provided the associations are correct and the fitness values are obtained from the correct evaluator, there is no need for further sequencing restrictions.

5.2.3 Extensibility Model

Since it is impossible to predict the full range of components that may ever be invented, a central aspect of the EA approach described in this work is to allow users to introduce their own operators rather than being limited to a pre-defined list. Any ESDL implementation that does not support this is ignoring the fundamental purpose of ESDL and the associated model.

Given the current state of the field, where those who invent new operators are typically programmers, there is no need to disguise the fact that an implementation is based on a certain programming language. For example, a Java-based implementation can require that operators be written in Java (or use a Java-based adapter) and follow a Java style coding convention. Practically all the programmatic flexi-

```
FROM generator(parameter=value) SELECT n group
FROM group SELECT group USING operator(parameter=value)
EVAL group USING evaluator(parameter=value)
return_value = function(parameter=value)
```

Listing 5.3: Example generator, operator, evaluator and function invocations in ESDL.

bility comes from operator implementations, so while the external interface should be strictly defined, implementation guidelines are preferable to rules.

The definition of interfaces depends on the architecture of the implementation, but regardless of architecture, there are three aspects that must exist: parameter settings or updates, an input stream (or multiple streams for joiners) and an output stream. Apart from operators, the other extensible elements are generators, evaluators and functions. Each of these has parameters provided in the same manner (as shown in Listing 5.3), though generators have an output stream with no input, evaluators continue to exist beyond the statement where they are invoked, and functions have a return value but no streams.

As well as the interfaces to each of these extensible elements, some method of loading or including them in a definition is necessary. In particular, implementations that use operator metadata need to specify the format and provisioning method. The following sections describe each extensible type in detail, though most of the discussion is the same for each type and is covered in Section 5.2.3.1.

5.2.3.1 Operator Interface

Operators perform processing on streams of individuals. They have an implemented behaviour, zero or more parameters, one input stream and one output stream.

Parameter updates provide new values from the blackboard (Section 5.2.1) to the operator, potentially through a constructor or another method. Values may change unpredictably in subsequent iterations, requiring updates to be explicitly specified. Updates typically occur at the beginning of the store operation containing the operator, since there is no other way to reference an operator instance. Implementations that reconstruct operators each iteration are unaffected by these changes, provided the sequence model is respected, but those that use long-running operator instances need to explicitly support updates.

As the data type of parameters is opaque within ESDL, some form of conversion is likely to be necessary. At a minimum, users should not be concerned with multiple numeric types: there is no need for an operator interface to demand integers rather than accept real values, though the implementation is able to use any types that are supported by the language. Non-numeric variables are always considered implementation specific, and the interface specification must account for these

separately. Groups may be provided as read-only parameters, allowing access to individuals by index, though the intent is for use as statistics or comparisons rather than as a substitute for the input stream. Finally, parameters may be passed `null` (or the synonym `none`) where either a group or variable is expected, indicating that no value is intended. All parameters that expect groups must specify `null` as the default value.

Parameters cannot be passed by modifiable reference; groups and values are read-only on the blackboard and so should not be modified by external code.[4] For the most common case of numbers, pass by value is suitable, though groups may be more efficiently passed by reference (directly to the storage space on the blackboard, rather than to the updateable name). For languages that do not support parameter names or passing arguments by name, a separate mapping is also required: a list of the parameter names to expect in the ESDL definition and the position or register to pass values in. Similarly, all operator parameters require default values, and languages that do not intrinsically support defaults need metadata to specify the values and code generation to provide missing arguments.

Input streams provide individuals in the order they were provided by the preceding operator. Whether they are obtained one at a time or in greater numbers depends on the implementation; a push model is likely to provide one at a time, while a pull model could retrieve exactly as many individuals are required. Regardless of the model, an operator should be able to request all available individuals from a finite source stream, typically in order to use a data structure with random access semantics (an array) or to sort or aggregate all individuals. Implementations that provide an explicit mechanism for this action are better able to detect when an infinitely long stream is used improperly and provide helpful feedback rather than an out-of-memory error.

Output streams receive individuals sequentially from an operator. As with input streams, individuals may be provided in blocks of one or more, as long as the order is not modified. Output streams may be terminated, indicating a finite-length stream; termination is communicated to the consuming operator and no further processing should be performed by the producing operator. If the producer and consumer disagree about rate, the implementation is responsible for caching individuals. For example, if the consumer is requesting one individual at a time but the producer returns two, the consumer will receive each of the two individuals on subsequent requests. Dropping or duplicating individuals are not appropriate ways of handling this scenario.

[4] Implementations may provide opaque data types that reference mutable data at their own risk, both in terms of synchronisation and confusing developers and readers.

```
JOIN a, b    INTO pairs    USING tuples
JOIN a, b, c INTO triplets USING tuples
```

Listing 5.4: Use of a joiner that can handle a variable number of sources.

Streams are only conceptually realised as one-way pipes and implementations may choose alternate representations, provided order is preserved and operators can freely vary the resulting number of individuals. In a pull model, passing complete arrays between operators is efficient, though in a push model this is likely to result in some wasted computation. Depending on the architecture of the entire implementation, this wastage may allow faster execution overall. For example, working in powers of two has efficiency benefits for many systems and algorithms, often enough to outweigh the cost of padding.

5.2.3.2 Joiner Interface

Joiners are identical to operators but allow multiple input streams. Whereas a `FROM-SELECT` statement concatenates each source before providing it to the first operator, a `JOIN-INTO` statement passes each source stream independently. Further, while a normal operator may be limited to homogenous groups, a joiner is expected to handle groups containing different types of individuals, though the ability to join different types arbitrarily depends on the available joiner implementations.

Depending on the underlying architecture and language, it may be possible to write joiners that take an arbitrary number of source streams. However, many languages do not allow this or complicate it so much it becomes unimplementable. In these cases, implementations should use either a convention or metadata to allow the ESDL description to use the same name regardless of joiner arity. For example, each statement in Listing 5.4 may require two different `tuples` joiners—one to handle two sources and the other for three—but it is not the ESDL developer's responsibility to resolve this in the definition.

5.2.3.3 Generator Interface

In terms of extensibility, generators are identical to operators without an input stream. A generator continuously appends to its output stream, which is connected to either another operator or a merge operator (Section 3.3.2). Once the consumer indicates that it has finished producing its own output, the generator is no longer required, but must otherwise continue to produce individuals forever. Repetitions are permitted in the output of a generator.

Generators are only specified in the list of source groups in a `FROM-SELECT` or `JOIN-INTO` statement, as shown in Listing 5.5. In a `FROM-SELECT`, the generator must appear last, and because generators are always infinitely long, a size must be specified for

```
FROM generator(parameter=value)       SELECT n group
FROM a, b, generator(parameter=value) SELECT n group

JOIN a, generator(), generator() INTO   joined_group USING tuples
JOIN generator(), generator()    INTO n joined_group USING tuples
```

Listing 5.5: Generator invocations in ESDL.

```
EVAL a USING evaluator
FROM a, b, c SELECT d
# d is implicitly associated with evaluator
YIELD d
```

Listing 5.6: Evaluator propagation in ESDL.

each destination group. When using a `JOIN-INTO` statement, the joiner used will determine the restrictions on whether a group size is necessary. For example, the `tuples` joiner (defined in Section A.4.1) requires at least one finite group or a size specifier on the destination.

5.2.3.4 Evaluator Interface

Evaluators are invoked similarly to operators, though they do not have the same need to provide input and output streams. However, they do need to be persistent and associable with groups, since evaluator propagation may result in an evaluator being used for individuals that are not part of the groups originally specified in the `EVALUATE` statement. For example, group `a` in Listing 5.6 is associated with `evaluator`. The following `FROM-SELECT` means that group `d` is also evaluated with `evaluator`. If the evaluation had occurred immediately and no association formed, the individuals from groups `b` and `c` would not be evaluable at the `YIELD` statement, or may be evaluated incorrectly.

5.2.3.5 Function Interface

Functions include a return value, which is of the same opaque type that can be stored on the blackboard, but have no streams. The return value may be omitted, though in the absence of side effects this results in a useless function. As mentioned in Section 5.2.2, implementations that allow side effects need to specify a sequencing model that accounts for functions with no return value.

As for operators (see Section 5.2.3.1), ESDL requires all parameters must have names and be passable out of order. Functions are not required to have default values for all parameters. Most general-purpose programming languages support parameter names, many allow default parameters and some allow passing arguments by names rather than position. For languages lacking one or more of these, including

the missing information as compile-time metadata coupled with code generation can provide the functionality. Variable types are unnecessary in ESDL, but may be relevant enough for an implementation that metadata and code generation also need to include parameter types.

5.2.3.6 Loading

Most important to an extensibility model is the ease with which users can make their own extensions available within an ESDL system. This is referred to variously as including, importing or linking in general-purpose programming languages. In ESDL, there is no equivalent mechanism, since various implementations are better placed to choose an appropriate approach. The general guidance is away from namespaces and toward a fine-grained inclusion mechanism that adds individual operators as necessary.

Since the set of operators to be used is easily determined when parsing the ESDL definition, the more difficult problem is mapping names to a specific implementation. For a framework with a large collection of operators, a list of built-in operators is likely to be included, though appending to this list, either permanently or for a single experimental configuration, should be relatively straightforward. For an ESDL compiler that statically links to extension implementations, location information may need to be embedded in the source ESDL definition (in a pragma, Section 4.2.1) or provided in a separate command file. ESDL has no dynamic loading mechanism, so all names are fixed at compile time, even if they are not resolved until run-time.

5.2.4 Parsing and Compiling

As discussed in Chapter 4, ESDL is a language with restricted expressiveness and a consistent, context-free grammar, which makes it suitable for automatic parsing. The ability to use an ESDL description as configuration for a software framework allows the description to be verified without requiring a translation to or from another form.

The target framework or language for generated code does not significantly affect the process of parsing ESDL. System definitions may be parsed into a sequence of text-based tokens and converted into an abstract compilation model based on the description in Chapter 4, the model in this chapter and the grammar given in Appendix B. While different implementations will likely require their own compiler implementation, the general behaviour is as described in this section. `esdlc`, the Python based compiler used for the implementation in Chapter 6, follows the described sequence closely. Appendix C describes `esdlc` in detail.

Each ESDL statement is separated by line breaks, though a backslash may be used to ignore a break and continue the statement onto the following line. Whitespace acts as token separators but is ignored, while other punctuation characters

```
`include "Evaluator.h"
lowest = -100  # the lowest value
highest = 100  ; the highest value
FROM random_real(length=8, lowest, highest) SELECT n population
EVALUATE population USING evaluator()
YIELD population

BEGIN generation_equivalent
  REPEAT n
    FROM population SELECT 2 parents, rest \ // backslash before
         USING fitness_proportional(no_replacement)
    FROM parents SELECT offspring USING crossover, \
         mutate(per_gene_rate=0.1)
    FROM offspring, rest SELECT population
  END
  YIELD population
END
```

Listing 5.7: Example ESDL code.

(except for decimal points in numbers) are always one token each—there are no multi-character operators. Symbol names are consecutive sequences of letters, digits and underscores that begin with a letter or underscore, while numbers are consecutive sequence of digits and one (optional) period. Comments extend from a pound character (`#`), a semicolon (`;`) or double-slash (`//`) until the end of the line and do not become tokens; any continuation backslash must appear prior to a comment. Pragmas, on lines beginning with a backtick (grave accent, `` ` ``) are read as a single token until the end of the line; the actual text is retained unmodified.

Following these tokenisation rules, the ESDL system shown in Listing 5.7 becomes the stream of tokens shown in Listing 5.8. At this stage, each token is literal text except for `EOS`, which represents the end of each statement, and all text has been converted to lowercase. Tokenising each line break as `EOS` and backslashes as a literal token allows for simple filtering to handle line continuations.

Any form of parser may be used to produce a syntax tree from this token stream. In general, the first symbol of each statement identifies the structure of the rest, allowing a partial recursive-descent parser to provide efficient execution and more precise error messages than a full shift-reduce parser, though the existence of infix arithmetic equations in certain (known) locations may justify the use of a complex parser. The parser in `esdlc` branches based on the first token of each statement and parses the remainder of the statement linearly.

Regardless of the process used to derive the tree, the resulting structure should match, or be convertible to, a list of blocks containing lists of statements, where each statement is one tree. Listing 5.9 shows the parse tree for Listing 5.7 with

```
"`include \"Evaluator.h\"" EOS
"lowest" "=" "-" "100" EOS
"highest" "=" "100" EOS
"from" "random_real" "(" "length" "=" "8" "," "lowest" ","
"highest" ")" "select" "n" "population" EOS
"evaluate" "population" "using" "evaluator" "(" ")" EOS
"yield" "population" EOS
"begin" "generation_equivalent" EOS
"repeat" "n" EOS
"from" "population" "select" "2" "parents" "," "rest" "\" EOS
"using" "fitness_proportional" "(" "no_replacement" ")" EOS
"from" "parents" "select" "offspring" "using" "crossover"
"," "\" EOS "mutate" "(" "per_gene_rate" "=" "0.1" ")" EOS
"from" "offspring" "," "rest" "select" "population" EOS
"end" EOS
"yield" "population" EOS
"end" EOS
```

Listing 5.8: Token stream created for Listing 5.7.

Table 5.2: ESDL keywords.

Keyword	Statement tree location	Child node types
`BEGIN`	Root	Block name
`EVAL/EVALUATE`	Root	Groups, `USING`
`FROM`	Root	Groups, generators, `SELECT`, `USING`
`JOIN`	Root	Groups, `INTO`, `USING`
`REPEAT`	Root	Numeric expression
`YIELD`	Root	Groups
`END`	None	None
`INTO`	Child of `JOIN`	Sized groups
`SELECT`	Child of `FROM`	Sized groups
`USING`	Child of `FROM`, `JOIN` or `EVAL`	Operators or evaluators

each node's children enclosed in braces. Square brackets are used for simplicity when representing comma-separated lists, which may otherwise be represented as recursive structures; for example, `LIST{ 1, LIST{ 2, 3 } }` rather than `[1, 2, 3]`. Lists containing one element retain the bracket in this listing.

All of the keywords available in ESDL are shown in Table 5.2, along with an indication of where in a statement tree they may appear. Most are valid only as the root of a statement: the first keyword. Apart from `END`, which indicates the end of a `BEGIN` or `REPEAT` node's statement list (and has already been removed from the tree in Listing 5.9), the remainder indicate the beginning of another branch of their parent node.

Where a child type is a plural in Table 5.2, a comma-separated list of at least one item is required. `USING` nodes are optional, but all other children are mandatory.

```
BeginStmt{ (initialisation), [
  PragmaStmt{`include "Evaluator.h"},
  ={ lowest, -{ 100 } }
  ={ highest, 100 }
  FromStmt{ [random_real[={ length, 8 }, lowest, highest]],
    SelectStmt{ [population{ n }] }
  },
  EvalStmt{ [population],
    UsingStmt{ [evaluator] }
  },
  YieldStmt{ [population] }
]}

BeginStmt{ generation_equivalent, [
  RepeatStmt{ n, [
    FromStmt{ [population],
      SelectStmt{ [parents{ 2 }, rest] },
      UsingStmt{ [fitness_proportional[no_replacement] ] }
    },
    FromStmt{ [parents],
      SelectStmt{ [offspring] },
      UsingStmt{ [crossover, mutate[={ per_gene_rate, 0.1 }] ] }
    },
    FromStmt{ [offspring, rest],
      SelectStmt{ [population] }
    },
  ]},
  YieldStmt{ [population] }
]}
```

Listing 5.9: Parse tree created for Listing 5.7.

A numeric expression is a constant number, variable name, function call or an expression constructed from these and arithmetic operators `+`, `-`, `*`, `/`, `%` (modulus) and `^` (power). Groups are represented as a single symbol token, sized groups as a single symbol optionally preceded by a numeric expression, and generators as a symbol token followed by a list of parameters (in effect, a function call). For example, the token stream "`groupA , groupB`" could be either two groups or two sized groups, while the stream "`7 groupA , ( n - 7 ) groupB`" is two sized groups. The tokens "`1 groupA , genB ( )`" are invalid syntax, since sized groups and generators cannot appear in the same list. Group names cannot be parenthesised because of ambiguity with generators or functions in numeric expressions, and while parenthesising size specifications assists with readability it is not required.

Any statement that does not begin with a root keyword from Table 5.2 is an assignment or a function call. The parsing rules are identical for function calls as a statement or as part of an expression, while assignments are parsed as a node with the source expression and destination symbol as children. Implementations that do not allow side effects in functions can safely ignore statements containing only a function call with no assignment.

Function calls consist of a name followed by a parenthesised, comma-separated list of arguments. The function name in the simplest case is a symbol, though `esdlc` allows the symbol to be preceded by any expression and separated by a period (in effect, the dot notation common to accessing object members in object-oriented languages). The interpretation of this expression is implementation dependent, but should represent either a scope resolution behaviour, where the preceding expression disambiguates the last part of the name, or an implicit parameter behaviour, where the result of the preceding expression is passed as part of the invocation (a 'this' parameter).

Argument lists are based on parameter names rather than positions. Each argument is a single symbol matching one of the parameter names, optionally followed by an equals character and an expression for that argument. If the equals character and expression are omitted, the value passed is a variable with a name matching the parameter name or, if no matching variable exists, a Boolean true. Parsers may choose to add a variable or literal reference immediately or to defer the decision to the code generator, depending on whether the full set of available variables is known while parsing.

Error detection after parsing can detect most misplaced or omitted nodes, unmatched `END` statements, invalid arithmetic expressions and incompatible group specifications. Unused, uninitialised and misused variables cannot be detected without introducing a symbol table. Depending on the code generation to be performed, there may be opportunities to detect these and other violations later in the process.

```
BLOCK {name: (initialisation),
  PRAGMA {`include "Evaluator.h"},
  FUNCTION "=" {source: FUNCTION "-" { right: 100 }, destination: lowest},
  FUNCTION "=" {source: 100, destination: highest},
  STORE {
    source: OPERATOR "merge" {
      FUNCTION "random_real" {length: 8, lowest: *, highest: *}},
    destination: OPERATOR "partition" {GROUP {population, size: n}}},
  EVAL {population, evaluator: FUNCTION "evaluator"},
  YIELD {population}
}

BLOCK {name: generation_equivalent,
  BLOCK {count: n,
    STORE {
      source: OPERATOR "fitness_proportional" {
        no_replacement: *,
        source: OPERATOR "merge" {GROUP {population}}},
      destination: OPERATOR "partition" {
        GROUP {parents, size:2}, GROUP {rest}}},
    STORE {
      source: OPERATOR "mutate" {
        per_gene_rate: 0.1,
        source: OPERATOR "crossover" {
          source: OPERATOR "merge" {GROUP {parents}}}},
      destination: OPERATOR "partition" {GROUP {offspring}}},
    STORE {
      source: OPERATOR "merge" {GROUP {offspring}, GROUP {rest}},
      destination: OPERATOR "partition" {GROUP {population}}},
  },
  YIELD {population},
}
```

Listing 5.10: Textual representation of the execution model for Listing 5.7.

The final parsing step converts the parse tree to the execution model elements described in Section 5.2.2: stores, functions, yields, evaluations, pragmas and blocks. Maintaining instruction sequence at this stage allows code generators to use that information; a pure operator graph representation may be useful, but reconstructing the original statement order is then not reliable. Assuming the order is retained, the result of converting Listing 5.7 to the execution model is shown in Listing 5.10.

Code generation uses this model to produce executable code or to directly interpret the algorithm. Observation of a running algorithm can reveal an incorrect description, such as groups that increase in size each iteration or operators that do not produce the expected output streams. Chapter 6 describes a software framework that uses ESDL as configuration information and dynamically generates code using its library of operators.

5.2.5 Summary

Because ESDL is independent from any programming language, platform or framework, the interpretation of the language needs to be specified. This section has defined the behavioural constraints necessary to ensure that ESDL descriptions are interpreted consistently in a variety of contexts. Section 5.2.1 defined the conventions of named group references and immutability that simplify reasoning about variable values and group contents within an algorithm. Section 5.2.2 discussed the freedom available to interpretations of ESDL that use non-sequential execution. Section 5.2.3 covered the addition of new functionality into an execution configuration, and Section 5.2.4 discussed the approach to converting a textual description into an executable model.

The following section places this model of interpreting ESDL in the context of the features reviewed in Section 5.1. Since ESDL is not a software package, it cannot be compared with the reviewed software directly; however, a number of features and behavioural decisions can be compared to those made in existing work. Chapter 6 demonstrates that the models in this section are sufficient by implementing a software package that uses ESDL and reproducing the behaviours of existing algorithms.

5.3 Comparison with Existing Software

Section 5.1 reviewed and compared a number of EA software packages with respect to the architectural decisions that are evident in their designs. Since ESDL is not a piece of software, direct comparison is not possible; an implementation of ESDL has considerable freedom to use any convenient or performant architecture. However, Section 5.2 specified a number of architectural decisions that are necessary to ensure correct behaviour. This section identifies the aspects of the execution model that constitute an ESDL architecture and compares them to the decisions taken by other software packages.

In Section 5.1, a distinction between population-centric and algorithm-centric architectures was made, with the packages from Table 5.1 evenly split between the two. ESDL is algorithm-centric: groups are simple data structures with no application-specific behaviour. The primary impact of this approach is centralising the customisable aspects of the implementation—the entire configuration is contained in the ESDL definition and the operator implementations. However, ESDL is neither a library nor a framework, since it is not directly executable code. Software that is capable of using ESDL is likely to be a framework and include a library of operators; some of the reviewed frameworks are hypothetically capable of using ESDL in place of their existing configuration mechanisms. ESDL provides the “glue” that

is an essential part of component frameworks, rather than providing the framework itself.

Since groups have no behaviour, the way in which they are passed between the processing elements is important. However, the immutability constraint provides a significant simplification by removing the need for explicit cloning. From Table 5.1, none of the packages use true immutability, preferring conventions for cloning or simply permitting mutable data. ESDL forbids changing data, or at least requires the appearance of immutability, which means explicit cloning of individuals is not required at the algorithm level. Because groups are immutable, reordering or updating individuals requires the entire group to be recreated in a new storage location. This results in a larger number of memory allocations and copies than in other approaches, though on modern processors the benefit of linear memory access patterns often outweighs the expense of extraneous copies.

Section 5.2.1 described the streaming behaviour of groups, suggesting a similar approach to those reviewed packages (*a*, *b*, *d*, *g*, *h* and *i*) that pass enough individuals to an operator to produce a single result. However, because ESDL operators are known to apply to an entire stream, the distinction is unimportant for analysis. Operator chains can mostly be viewed or implemented as either per-individual or per-stream, with the caveat that a per-stream view must account for the ability of operators to modify the length of the stream—most of the reviewed software assumes or requires that the size of the result is known. ESDL does not necessarily embed this information, though it can be specified or inferred with knowledge of the operators that are involved.

As with most of the reviewed software, ESDL supports any representation or species that has operators available. Section 5.2.3 described the general architecture for integrating new operators with an ESDL definition. None of the reviewed software has a similarly defined approach, primarily because they are already restricted to a particular programming language and its extensibility features. An implementation of ESDL will be similarly restricted, though the model itself does not require any formal notion of extensibility—context and consistent naming is sufficient in written presentations, as shown in the examples in Section 4.4.

It was noted in Section 5.1 that none of the reviewed software would automatically select operators to suit the representation in use. ESDL does not explicitly require or forbid automatic selection, but assumes that the operators specified in a system definition will be usable with the individuals provided. Some form of operator overloading to select an operator based on name and the individual type simplifies system definitions, since it avoids the alternative mitigation of embedding the representation into every operator name—an overloaded `mutate_gaussian` oper-

ator is preferable to `mutate_gaussian_integer` and `mutate_gaussian_real`. However, not every platform can support such a feature, and so it is not mandated.

Compatibility with existing software is a slightly different issue than for the reviewed software. ESDL is primarily intended for humans to read as a substitute for extended textual descriptions; if a system description can be converted by hand into an algorithm with an existing framework, it can be said to be compatible, while an automatic conversion requires specific software support. The models in Section 5.2 are intended to ensure software that adds support for ESDL does so in a way that is compatible with other implementations. Appendix A suggests a number of components and their behaviour that would further ensure that algorithms can be shared between implementations without changes in behaviour.

5.4 Chapter Summary

This chapter has reviewed a selection of available EA software, discussed the execution model that underlies ESDL and located ESDL within existing software packages. Section 5.1 showed that component models with loosely coupled operators are popular among published libraries and frameworks. Separation of algorithm design and configuration from operator implementation was also common—an architectural approach that is implicit in the model of Chapter 3 and ESDL.

ESDL is not a replacement for existing software. Rather, it is a form of communication that can be used with existing or specially-designed software to ensure that written descriptions of EAs match the implementations used to perform experiments. Constraints such as immutability, linear streaming and global variable scope are intended to simplify understanding, in part by removing the need to redefine how operators interact in each new algorithm definition.

Defining the behaviour, limitations and freedoms in ESDL allows implementers to produce interoperable but distinct software. Central to the implementation is a write-once blackboard, providing all parts of an ESDL system read-only access to groups and variables. Rearranging the sequence of execution is simplified by immutability while allowing considerable freedom for implementations to differentiate themselves by providing parallel, distributed or otherwise enhanced execution. Operators and functions must be easily addable to an ESDL system or else the limitations of a fixed library will prevent the invention of novel algorithms. Chapter 6 presents an implementation of ESDL in terms of these models and demonstrates how it fits into research workflows.

Chapter 6

Application

Earlier chapters have described a component-based model for EAs, a description language, ESDL, for composing entire algorithms and an execution model that ensures consistent interpretation of the language. While primarily intended as a tool for written communication, ESDL can also be used to configure a software framework and produce executable code for an algorithm. ESDL can specify an EA with equal accuracy and less text, making it suitable for sharing where code written in a general-purpose programming language would be inappropriate or unclear. Because the description used for experimentation can be published without modification, the likelihood of introducing errors is reduced. When a reader wishes to implement a published algorithm, an ESDL description increases their prospects of doing so correctly. In this chapter, the application of ESDL to research, design, implementation and communication are discussed and demonstrated. A specific implementation is described and used to demonstrate that ESDL descriptions are sufficient to describe entire algorithms and are automatically compilable. Comparison to equivalent implementations using other approaches shows that ESDL descriptions are more concise.

6.1 A Hypothetical Workflow

This section illustrates the problems identified in Chapter 2 using hypothetical stories about a researcher and the work he performs in studying and reproducing an EA. The two stories are completely fictional and presented as illustrative examples of the contentions made in this work rather than as evidence. Presentation as fiction allows the use of specific references and examples when discussing the issues that occur; purely abstract discussion would be less tangible. In the absence of a

long-term study evaluating ESDL's effect on research and development, plausible and logically consistent discussion is necessary to recognise potential benefits.

Section 6.2 refers to the stories in this section as concrete examples of potential issues and the mitigations provided by the model in Chapter 3 and ESDL from Chapter 4. Section 6.3 describes a software framework, `esec`, that interprets and executes ESDL systems. Finally, the example algorithms from Chapter 4 are expanded to complete specifications for `esec` and compared to other equally complete descriptions in Section 6.4.

Section 6.1.1, the first story, describes a scenario where ESDL does not exist. The issues that arise in this story include ambiguous directions, conflicting specifications, omitted information, supplementary materials of limited use and poorly organised discussion. These issues have been noted earlier, and while it is extremely unlikely that they would all appear in a single work, almost any would be sufficient on their own to prevent an accurate reproduction of the algorithm.

Section 6.1.2 is the same story but based on the assumption that ESDL does exist and is used extensively by the original authors and the researcher. The initial textual description is structured around the sequence of an algorithm iteration rather than grouping similar types of behaviour. In a sense, it describes each ESDL statement rather than each operator. As a result, the background description is significantly clearer. Also observable in this story is that the full ESDL description includes all the parameters that were omitted from Story 1, and the operator definitions are precise and implementable.

Both stories illustrate extreme scenarios that are unlikely to occur (or to be published) in reality. The middle ground between these two scenarios includes many good descriptions created without using ESDL, as well as the potential for good and bad descriptions to be interpreted and implemented using ESDL. Section 6.2 discusses the application of ESDL to designing, implementing and sharing algorithms, and explicitly covers the middle ground between the two stories.

6.1.1 Story 1

A researcher has discovered a reference to an interesting sounding algorithm called Angry Mob Optimisation and wants to try it out. The reference includes a description of the algorithm as follows:

> Angry Mob Optimisation (AMO) is inspired by the flocking
> behaviour of large groups of people who have a single objective.
> Such behaviour has been observed in political protests,
> witch-hunts, revolutions and amongst sports fans.

> The useful characteristic of such groups is that, while the entire group (the "mob") is absolutely certain about its purpose, this purpose has been defined by a much smaller number of individuals (the "instigators"). Often the instigators do not even participate in the mob, preferring to move on and avoid responsibility.
>
> The AMO algorithm utilises this behaviour to perform function optimisation on deceptive expressions with multiple optima. We define both instigators and mob members as real vectors representing the function parameters.
>
> The number of instigators and mob members is fixed throughout, though mob members significantly outnumber the instigators. Mobs are always the same size, but members may move between mobs at any time, based on which mob they are closest to. All entities move throughout the landscape using directed mutation; mob members move towards mobs and instigators while instigators move away from mobs.
>
> Each mob member is mutated twice and the best variations are retained—no crossover is used—while instigators are mutated far more often and travel further with each mutation. Instigators avoid mobs more strongly than mob members are attracted, which results in instigators moving quickly around the landscape with mobs forming in their wake.

The reference also includes the flowchart shown in Figure 6.1 and the diagram in Figure 6.2. Table 6.1 shows the fitness of the best individual found after varying numbers of iterations, taken as an average across 100 independent runs for each.

Intrigued by the algorithm, the researcher decides to try it on some problems other than the benchmarks used by the original authors. Aware that a number of details are absent from the written description, the researcher finds that the authors have kindly made their source code available. It is written in C++ but the researcher is unable to compile the experiment, despite having an up-to-date compiler. The code is based around one function that implements the entire algorithm, with classes for storing the values of instigators, mobs and mob members. Comments indicate places where more debugging is required, variable names appear to be based on abbreviations that are never explained and none of the numbers appear to match those that were mentioned in the algorithm description.

Eventually, the researcher gives up and decides to create his own implementation based on the written information. While coding, he identifies that the following aspects are unspecified or ambiguous:

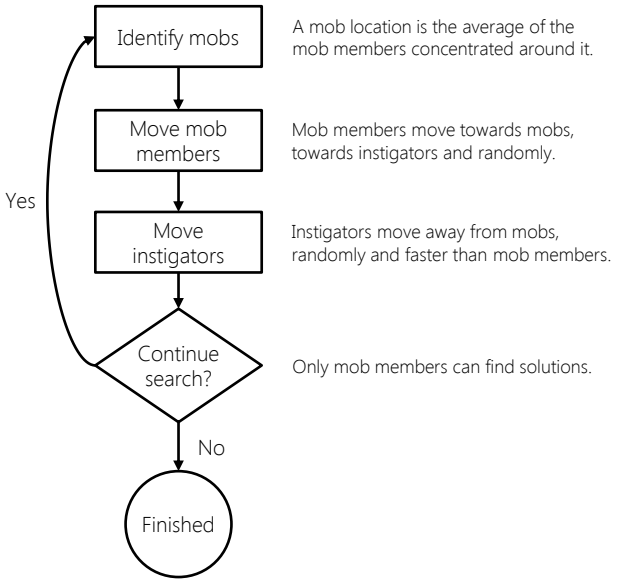

Figure 6.1: Flowchart of the hypothetical Angry Mob Optimisation algorithm.

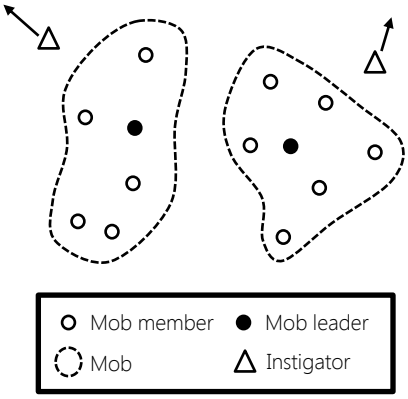

Figure 6.2: Elements of the hypothetical Angry Mob Optimisation algorithm.

Table 6.1: Best solutions found using Angry Mob Optimisation.

	Sphere ($N = 10$)	Rastrigin ($N = 2$)	Rastrigin ($N = 10$)
After 10 iterations	9069.	57.37	8705.
After 100 iterations	161.6	6.368	244.2
After 500 iterations	107.9	2.458	192.5

- The ratio of instigators to mob members and mobs is suggested by the diagram but not clearly identified.
- Precise numbers of instigators, mob members and mobs are not specified for the experiments conducted.
- Initial ranges of solution values are not given.
- Identification of mobs based on the locations of mob members is not specified and is only vaguely described.
- The selection of mob members and instigators for mutation is unspecified, except that each existing element may be selected more than once.
- The nature of mutation operation intended is unclear from the description.
- The relative importance of mob location, instigators and randomness to mob member mutations is not specified.
- "Mob leaders" are shown in the diagram, but are not mentioned elsewhere in the description.

The researcher fills the gaps by finding other publications of the same algorithm, decrypting the source code and making what appear to be reasonable guesses. After two days work, he has a full implementation, and by the end of the week he has fixed most of his own bugs. Unfortunately, the results he obtains are inconsistent with those that were published. Having spent a week working on AMO with no reward, he simply moves on to other research.

While this story has an overly poor description of an algorithm, any one of the problems listed above would be sufficient to prevent accurate implementation. Assuming that authors intend for their work to be reproducible and useful to others, accidental omissions need to be avoided. Executable code necessarily includes all details, however, programming languages are typically not suitable for publication or explanation, and the conversion to a more suitable form may introduce ambiguity. The next story makes use of ESDL as a human- and machine-readable language that is suitable for direct publication. Avoiding the conversion step reduces the potential for information loss, while the structural decomposition helps the textual description to focus better on each step rather than taking the entire algorithm as a whole. As a result, Story 2 has a happier ending than Story 1.

6.1.2 Story 2

A researcher has discovered a reference to an interesting sounding algorithm called Angry Mob Optimisation and wants to try it out. The reference includes a description of the algorithm as follows:

Angry Mob Optimisation (AMO) is inspired by the flocking behaviour of large groups of people who have a single objective. Such behaviour has been observed in political protests, witch-hunts, revolutions and amongst sports fans.

The useful characteristic of such groups is that, while the entire group (the "mob") is absolutely certain about its purpose, this purpose has been defined by a much smaller number of individuals (the "instigators"). Often the instigators do not even participate in the mob, preferring to move on and avoid responsibility.

The AMO algorithm utilises this behaviour to perform function optimisation on deceptive expressions with multiple optima. We define both instigators and mob members as real vectors representing the function parameters. The number of instigators and mob members is fixed throughout and mob members significantly outnumber instigators.

One mob exists for each instigator and all mob members are assigned to one mob each iteration. A mob leader is determined by finding the mob member with the highest fitness that has not yet been assigned. The remainder of the mob is filled by those mob members most similar to the leader; for real-valued domains, the Euclidean distance is suitable. The location of the mob is the average (mean) of each dimension of each member.

After determining mob locations, all mob members are moved towards each mob in inverse proportion to the distance between the member and the mob location in each dimension. Using the same calculation but a lower scale value, mob members are then moved towards each instigator. Finally, a normally distributed random value is added to each dimension. Each mob member is varied twice and the best half of the entire resulting group is retained.

Instigators are varied similarly to mob members, except that they move away from mobs with a higher speed and are not affected by instigators. The random value is taken from a distribution with higher variance, increasing the potential movement of instigators, and more variations are tried before reducing to only the best.

```
FROM random_real(length,lowest,highest) SELECT N_i instigators
FROM random_real(length,lowest,highest) SELECT N_m mob_members

YIELD mob_members

BEGIN iteration
  FROM mob_members SELECT mobs USING find_mobs(count=N_i)

  FROM mob_members SELECT N_m*2 new_mob_members \
    USING move_towards(targets=mobs, speed=0.9), \
          move_towards(targets=instigators, speed=0.8), \
          repeated, \
          mutate_gaussian(sigma=1)
  FROM new_mob_members SELECT N_m mob_members USING best

  FROM instigators SELECT N_i*50 new_instigators \
    USING move_away(targets=mobs, speed=2), \
          repeated, \
          mutate_gaussian(sigma=5)
  FROM new_instigators SELECT N_i instigators USING best

  YIELD mob_members
END
```

Listing 6.1: ESDL description of the hypothetical Angry Mob Optimisation algorithm.

Table 6.2: Variable values for Listing 6.1.

Variable	Value
length	2
lowest	10
highest	100
N_i	10
N_m	500

```
func find_mobs(source, count):
  mob_locations = empty list
  N = length(source) ÷ count
  while length(mobs) < count:
    leader = best individual in source

    mob = { leader and N−1 individuals in source closest to leader }
    source.remove_all(mob)

    mob_locations.append(average(mob))

  return mob_locations
```

Listing 6.2: Pseudocode for the hypothetical `find_mobs` operator.

The reference includes the ESDL definition shown in Listing 6.1, the values listed in Table 6.2 and a definition of the `move_towards` operator as:

$$x_i^{'} = x_i + \frac{S}{N} \sum_{j=1}^{N} \frac{1}{y_{ji} - x_i}$$

where x_i is the value of the source individual for dimension i, S is the speed parameter, N is the number of targets and y_{ji} is the value of target individual j for dimension i. It is noted in the explanatory text that where $x_i = y_{ji}$, the contribution in that dimension is zero rather than infinity. The `move_away` operator is identical with the value of S negated. Finally, a pseudocode listing of the `find_mobs` operator is included as Listing 6.2 and the results presented in the reference are summarised in Table 6.1. The flowchart and figure from the first story (figures 6.1 and 6.2) are also included, but the researcher finds that they help clarify his intuitive understanding of how the algorithm works rather than forming a specific description.

Using a software framework that supports ESDL, the researcher copies the system definition and writes three small functions for `move_towards`, `move_away` and `find_mobs`. The framework already includes implementations of the other operators and the two evaluators used in the original reference, and supports the design of a single experiment to perform the same configurations as in the original work. After producing results similar to those already published, the researcher is confident that he has a working implementation of AMO he can use for his own experiments.

6.2 Applying the ESDL approach

Section 6.1 showed two descriptions of the hypothetical Angry Mob Optimisation algorithm. The first description included a textual description, flowchart and a diagram; however, a number of important details were omitted, making the algorithm unimplementable. The second description used a more organised textual descrip-

tion and an ESDL description, two equations and a short pseudocode listing, which provided a complete description of the algorithm.

The following list of issues was identified in Story 1:

- The ratio of instigators to mob members and mobs is suggested by the diagram but not clearly identified.
- Precise numbers of instigators, mob members and mobs are not specified for the experiments conducted.
- Initial ranges of solution values are not given.
- Identification of mobs based on the locations of mob members is not specified and is only vaguely described.
- The selection of mob members and instigators for mutation is unspecified, except that each existing element may be selected more than once.
- The nature of mutation operation intended is unclear from the description.
- The relative importance of mob location, instigators and randomness to mob member mutations is not specified.
- "Mob leaders" are shown in the diagram, but are not mentioned elsewhere in the description.

As noted, while the hypothetical presentation has an unreasonable number of issues, any one of these is sufficient to make reproducing the original experiment impossible. A very careful author can make sure that none of these issues arise, while a formalised approach can significantly reduce the effort required.

In Story 1, the authors of AMO implemented their algorithm in C++. While not specified, it is fair to assume that the implementation evolved over time as bugs were fixed, different ideas were tried and the target problem was changed, resulting in code that the researcher was unable to understand or use. Recognising this, the authors translated their algorithm into text and diagrams, aiming to describe its implementation in enough detail to be reproducible. However, without a well-structured approach to design or translation, important information was inadvertently omitted. For Story 2, the ESDL that was used to design and implement the algorithm is suitable for direct publication, removing the translation step that caused the information loss in Story 1, as well as providing the structure for a clearer textual description.

Chapter 2 identified a number of issues that result from the lack of a single, fundamental algorithm class for EAs, each of which can be observed in the hypothetical stories:

- Difficulty in sharing advances between distinct algorithms
- Limited guidance in designing (rather than tuning) an algorithm to suit the targeted problem

- Adherence to conventions for historical reasons rather than reassessing their suitability for a new context
- Replication of software development because of perceived or actual unsuitability for reuse

Story 1 did not specify the details of how individuals are varied randomly, while Listing 6.1 in Story 2 makes clear that a random value is taken from a normal distribution, as is commonly used in EP. The textual description avoids specifying that the mutation is 'like EP,' since such a statement has more implications than are applicable. In the ESDL description, however, the `mutate_gaussian` operator clearly specifies the desired behaviour without inviting direct comparisons to another algorithm. When the desired behaviour is cleanly encapsulated into an algorithm-independent operator, new algorithms can share similar behaviour without implying a more significant relationship.

Lack of design guidance was noted in Chapter 2 and is demonstrated in both stories. Algorithms are inferred from an inspiration or metaphor, rather than being designed for application to a specific problem. While it is possible that the metaphor will result in a useful algorithm, significant tuning is still likely to be required for each new application. The stochastic nature of most EAs means that tuning almost always improves the performance. However, because tuning typically does not involve structural changes, redesigning the entire algorithm for specific problem characteristics may be necessary to improve the ability of the algorithm to be tuned.

The third point, unnecessary use of conventions from earlier algorithms, appears briefly in Story 1: the authors state that "no crossover is used," despite there being no reason to consider that crossover would be beneficial or suit the metaphor. In a similar manner, the changes made to individuals are referred to as "mutations" even though the values being 'mutated' are not representative of genetics. This terminology is inspired by GA and used throughout the field, though the implications are often unclear when discussing other algorithms. In Story 2, the hypothetical authors avoid all references to GA-inspired terminology, using the less-specific "operator" as defined in Chapter 3. The ability to use these terms, and others such as "selector," "individual" and "group," in context but without ambiguity is one of the main benefits of the clear definitions made in Chapter 3.

Finally, the researcher in Story 1 deliberately set out to rewrite software that already existed because it was of too low quality to be useful to him. Developing software for reuse is difficult and does not often occur in experimental research, as discussed in Section 2.2.3. The most significant issue caused by unusable software is difficulty in reproducing experiments. This section discusses the use of ESDL for designing and sharing algorithms in ways that support, rather than hinder, independent reproduction.

6.2.1 Designing Algorithms

ESDL and the model described in Chapter 3 provide an alternative conceptualisation to that provided by biomimicry and other metaphors. This is most obvious in the complete absence of biological or genetic terms such as "genome" and "offspring" throughout the definitions in Chapter 3, on the basis that while they support the design of an algorithm, the metaphor is not necessary for specification or description. The proposed alternative model represents the algorithm as instances of potential solutions being modified by a network of operators.

Because the interactions between operators within the network are well defined, reasoning about the effects of adding, changing or removing an operator is simplified. For example, if greediness or premature convergence is a concern, all the operators in a network that have an effect on selection pressure can be identified. When balancing an algorithm for either exploration or exploitation, particular variation operators can be evaluated independently, taking into account any interactions that may exist, thereby allowing informed decisions to be made.

As discussed in Section 2.2.2, algorithm designs often include concepts from existing algorithms without apparent reason, though potentially because of the difficulty in constructing and implementing an entirely new algorithm without a template. However, composing a network from known operators is a less complex task than producing a complete design, with a result that is more flexible than an executable program. When combined with a software framework that can interpret ESDL, the write-test-revise process is more rapid than when implementing an algorithm directly.

Both stories in Section 6.1 describe extremes, either extremely poor presentation or extremely convenient description, and a researcher's actual experience is likely to incorporate aspects of both scenarios. However, even if ESDL is not used by the original author, a reader can use it as an aid to interpreting and implementing the algorithm. For example, suppose the researcher in Story 1 decided to reimplement the algorithm using ESDL rather than a general-purpose programming language. In this case, the first step is a functional decomposition to identify the groups and operators that make up the description. Even with a poor explanation, it is possible to create an ESDL description. Listing 6.3 shows an annotated example of the ESDL system that may be obtained from the description given in Story 1. Comparison with Listing 6.1 shows significant similarity, despite the missing and ambiguous information. In Story 2, the textual description has clearly been structured based on the decomposition in the later system definition, and converting this description to ESDL should result in a near identical system.

```
# Somehow need to create a group of instigators
FROM new_instigator SELECT m instigators
# Also need a group of mob members
FROM new_mob_member SELECT n mob_members

BEGIN iteration
  # Not sure how to find the mobs
  FROM mob_members SELECT mobs USING find_mobs

  # Varying mob members
  FROM mob_members SELECT n*2 offspring \
      USING repeated, \ # "each mob member is mutated twice"
            move_towards(target=mobs, strength=0.5), \
            move_towards(target=instigators, strength=1), \
            move_random

  # Keep the best mob members
  FROM offspring SELECT n mob_members USING best

  # Varying instigators
  FROM instigators SELECT k offspring \
      USING repeated, \ # "each instigator is mutated more often"
            move_away(target=mobs, strength=1), \ # "avoiding mobs"
            move_random

  # Keep the best instigators
  FROM offspring SELECT m instigators USING best
END
```

Listing 6.3: ESDL description derived from the text in Story 1.

Though at first glance Listing 6.3 may be daunting, especially compared to the flowchart in Figure 6.1, it contains considerably more detail than the diagram. Communicating an algorithm using ESDL is covered in more detail in Section 6.2.2, but it is clear that ESDL is not suited to the sort of high-level overview that can be obtained from a simple diagram.

From the system in Listing 6.3, the elements that need to be defined are obvious: the two generators `new_instigator` and `new_mob_member`, group sizes `m`, `n` and `k`, and the `find_mobs`, `move_towards`, `move_away` and `move_random` operators. While there are too many details missing from the textual description to produce a matching result, the use of ESDL here provides a convenient work breakdown. This design guidance directly addresses the issue of designing an algorithm—regardless of whether or not the system description is used for anything more than planning—and supports work in implementing algorithms.

Another possible situation that was not covered in the hypotheticals is where the algorithm is published using ESDL but the researcher does not want to use an ESDL-based software framework. He still wishes to reproduce the experiments, but the implementation will be based on either a separate library or written from scratch. The precise mapping—or whether a mapping is even possible—from EDSL to a particular framework depends on the architecture of the target. Most of the software reviewed in Chapter 5 can only represent a subset of possible ESDL systems without significant redevelopment. In particular, the population-centric approaches are effectively restricted to algorithms that do not merge, join or partition groups. However, those frameworks that support arbitrary or flexible sequences of operators can support a significant range of systems.

Writing an implementation without the use of a library, despite being identified as contributing to some of the issues in Chapter 2, is sometimes a useful approach, since frameworks that interpret arbitrary ESDL systems are unlikely to obtain better execution performance than a specific implementation of a single algorithm. As with manually converting ESDL to use a library, this transformation uses the system definition as a precise specification of the intended behaviour, providing the implementer with a structure and validation criteria that may not exist with a less formal textual description.

A direct translation of the system—in effect, recreating the behaviour of an ESDL interpreter—is better performed by an automatic compiler; a complete, handcrafted implementation should optimise in ways that simple operator composition cannot. For example, many algorithms begin with a selection operator that requires a group to be sorted by fitness immediately after the group has been sorted in order to calculate statistics. Interpreters are unlikely to recognise that the group does not need to be reordered, while a human developer can easily identify this optimisation.

High performance algorithm implementations are necessary for large datasets or extended parameter sweeps. Assessing and comparing the algorithmic performance[1] of an EA in any meaningful sense requires many repetitions, the total number of which increases exponentially with the number of parameter configurations. Reducing the actual time taken to evaluate hundreds or thousands of configurations allows the researcher greater flexibility in adjusting either the algorithm construction or to use a wider range of test problems. Reproducing the results in Table 6.1 in Story 1 requires 900 separate experiments, totalling over 500 000 iterations of the algorithm; saving 10 milliseconds each iteration would reduce the total time by almost 90 minutes. Such an improvement can be achieved by optimising (or obtaining an optimised version of) a single operator, performing the evaluation on parallel hardware or distributing the entire algorithm across multiple processors, any of which are easily available when the algorithm design is abstracted from operator implementations but would require complete reimplementation of monolithic code.

The benefits of the ESDL approach to designing and implementing algorithms come primarily from the decomposition and greater structure encouraged by the use of loosely coupled operators. Because each operator is isolated from those around, modifications or substitutions are straightforward and operators created and shared by others can be integrated without modification.

6.2.2 Sharing Algorithms

Despite the obvious application of ESDL as a programming language, its intended purpose is as a description language. This implies that ESDL is not meant to be created and then hidden, as code typically is, but to be included directly in written communication as a clear, unambiguous specification of the algorithm being discussed. While Section 2.2.3 saw that some researchers may be uncomfortable about publishing code as an intrinsic part of their work, it should be noted that there is little difference between using ESDL and any other equally precise algorithm description.

ESDL provides a number of advantages over alternative notations that may be used in a publication. As plain text, it is very simple to include directly into a paper without translating into markup for a symbolic, mathematical or pseudocode notation and little or no formatting is required to assist with readability. ESDL reduces the effort required to understand and apply an algorithm, and is more easily recognisable than pseudocode, which automatically provides context for a familiar reader. Unfamiliar readers can see that each line is an English sentence describing a definite action, and reading an entire system is more like reading a

[1] Based on measures such as convergence speed, best fitness or diversity, rather than memory usage, CPU load or time per iteration.

textual description than source code or mathematics. Even if a reader is uncertain of exactly how the algorithm should work, the use of a software interpreter means they are able to use the ESDL code directly to produce an implementation that matches the author's intent.

Presentation of an ESDL system is expected to follow a regular pattern, with variations depending on the emphasis desired by the author, similar to how Story 2 presented the hypothetical Angry Mob Optimisation. A textual overview or discussion of the algorithm is given to provide background, inspiration and metaphor, but not to act as a complete specification. This overview may include a diagram to further illustrate the algorithm, potentially in the style of Section 3.5, which matches the decomposition provided by ESDL. Evaluators and operators that are specific to the algorithm are discussed and specified in a form the author considers most appropriate. For example, a crossover operation may be shown with a diagram, while vector operations may be better shown as an equation. Finally, the ESDL system definition is provided as a figure or listing, depending on style. Actual code for operators should be deferred to an appendix or accompanying material, unless it has been presented as the operator specification.

As a more demonstrative example than Story 2, the remainder of this section describes the second algorithm from [2] in full, following the guidelines given earlier. Details that were omitted from the original paper have been found or invented and so the original results may not be reproducible from this description, though the nature of the algorithm is retained. Notably, both variation operators used were cited but not described, necessitating reference to other works and risking the introduction of errors in transfer. Finally, while the original description draws heavily on both ES and GA in describing this algorithm, the main focus here is on its behaviour and not its inspirations.

6.2.2.1 Affenzeller 2 (Untitled)

This algorithm represents the diversity of biological evolution by applying different variation operators to parts of a population, rather than using the same operators on the entire population. Sub- or "island" populations are sometimes used for this purpose, though migration rates tend to be low, resulting in little improvement over using a single population with a fixed variation scheme. This algorithm creates new "islands" from a single population each iteration to maximise information sharing while dynamically adapting operator usage by varying the size of each subpopulation.

The two crossover operators used are Edge Recombination Crossover (ERX) [96] and Order Crossover (OX) [69]. Both operators use the information represented by two parent individuals to create a single child. ERX treats each individual as

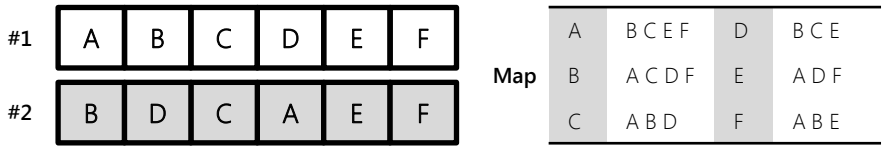

Figure 6.3: Adjacency map created for Edge Recombination Crossover.

```
from random import randrange
A = ['a', 'b', 'c', 'd', 'e', 'f']
B = ['b', 'd', 'c', 'a', 'e', 'f']
C = [None] * len(A)

i = randrange(len(A) - 2)
j = k = randrange(i + 1, len(A))
C[i:j] = A[i:j]

while j != i:
    while B[k] in C: k = (k + 1) % len(B)
    C[j] = B[k]
    j = (j + 1) % len(C)
```

Listing 6.4: Python example of Order Crossover.

circular and uses an adjacency map to select a new sequence from two parents. Figure 6.3 shows two example individuals and their adjacency map.

The child individual is created by selecting the first component of either parent (either A or B in Figure 6.3) with the least edges in the map, randomly selecting one if they are equal. For this example, B is selected at random. The next component is selected from B's edges—A, C, D and F—preferring the one with the fewest edges. Since B has been used, A has three remaining edges while C, D and F each have two; one of C, D or F is selected at random. This is repeated until all components have been used exactly once.

OX moves a randomly selected substring from the first individual directly to the offspring and fills out the rest of the components in the same order as they appear in the second individual. Listing 6.4 shows a short snippet of Python code that performs OX on two lists.

An initial population of 500 individuals is created by randomly shuffling the set of all nodes. No repetitions of components within an individual are permitted, though identical individuals are not prevented. In the first iteration, 100 offspring are created using ERX and 900 using OX; this proportion changes linearly over 400 iterations so that by the last iteration 900 are created by ERX and 100 by OX (a change of two individuals each iteration).

```
FROM shuffled_integer(maximum=6) SELECT 500 population
YIELD population

erx_count = 100

BEGIN generation
  FROM population SELECT (erx_count) offspring1 USING tournament, erx
  FROM population SELECT (1000-erx_count) offspring2 USING tournament, ox

  FROM offspring1, offspring2 SELECT 500 population USING best
  YIELD population

  erx_count = erx_count + 2
END
```

Listing 6.5: ESDL description of Affenzeller 2.

Parents are selected using a simple tournament, returning the fitter of two randomly selected individuals or the first of the two if equally fit. The 1000 offspring are reduced to 500 each iteration by keeping the fittest. Listing 6.5 shows the ESDL description of this algorithm.

6.2.3 Summary

This section has described how ESDL can be used to support work with EAs by providing design guidance and simplifying and ensuring the accuracy of presentation. Separating operator implementations or descriptions from the overall structure of the algorithm is central to improving the clarity of both parts. These approaches to designing, implementing and sharing EAs, combined with the execution model from Chapter 5, are the main proposal of this thesis.

The following section describes an actual implementation of a software framework supporting ESDL, which is then used in Section 6.4 to demonstrate that complete and unambiguous ESDL descriptions are capable of implementing a range of existing algorithms and are more concise than alternative approaches.

6.3 Executing ESDL systems

EC is predominantly a research field, with a higher demand for publishable results rather than commercial quality software components. Given a target audience with a greater interest in innovation than performance, a flexible ESDL-based software framework allows fast evolution of algorithm structures and implementations and the ability to easily try ideas that are not already widely used; a feature that was found to be absent from the EA software packages reviewed in Section 5.1. Typically, the cost of greater flexibility is reduced execution performance. ESDL's advantage

in this area is portability, since systems can be developed in a flexible environment and transferred to a high-performance framework without modification.

This section describes `esec`, which is a flexible framework that uses ESDL as its main configuration mechanism. While the ability to directly execute ESDL systems is not essential, it is helpful as a validation tool when applying the approach to design and research. While a human reader can take advantage of implied context, an ambiguous or incomplete description cannot be compiled or executed automatically. Development of `esec` was necessary since no existing software uses the model described in this thesis, hence the minor hypocrisy of creating yet more code. While we do not expect `esec` itself to be widely reused (though it certainly could be), the provided model and structure should significantly reduce the implementation overhead faced by future developers.

This description of `esec` is based on the execution models described in Chapter 5 and acts as a specific example of instantiating those models for a particular platform; Appendix F describes the use of these models for a different platform. These examples are intended to demonstrate that the models in Chapter 5 are not contradictory and are able to be implemented.

`esec` is used in Section 6.4 to implement a range of EAs using ESDL and ensure the descriptions are complete and unambiguous—`esec` cannot execute an ambiguous ESDL system. The volume of code required for these descriptions is then compared to alternative, equally precise implementation approaches.

6.3.1 Major Components

The `esec` framework is based around *experiments*, representing algorithms with complete sets of parameters. Each experiment is initialised with a dictionary of configuration information. The ESDL system is the majority of the definition. Starting the experiment compiles the ESDL definition into a `System` object, which includes references to a *monitor*, a *landscape* and a block *selector*, as well as all the operators required for running the algorithm. Figure 6.4 shows the architecture of an initialised experiment.

Monitors provide statistical analysis, termination and output functionality by responding to various events, such as the start and end of each iteration, `YIELD` statements and exceptions. For example, the `CSVMonitor` class writes details from each iteration to a file in CSV format. `esec` can be adapted to run using a GUI by implementing a custom monitor that acts as an intermediary between the experiment and the user's view. Monitors are responsible for termination conditions, whether based on elapsed iterations, evaluations or individuals' fitnesses.

Landscapes, in `esec`, represent the problems being solved. The term comes from the geometric view of benchmark problems as a two-dimensional surface with a high-

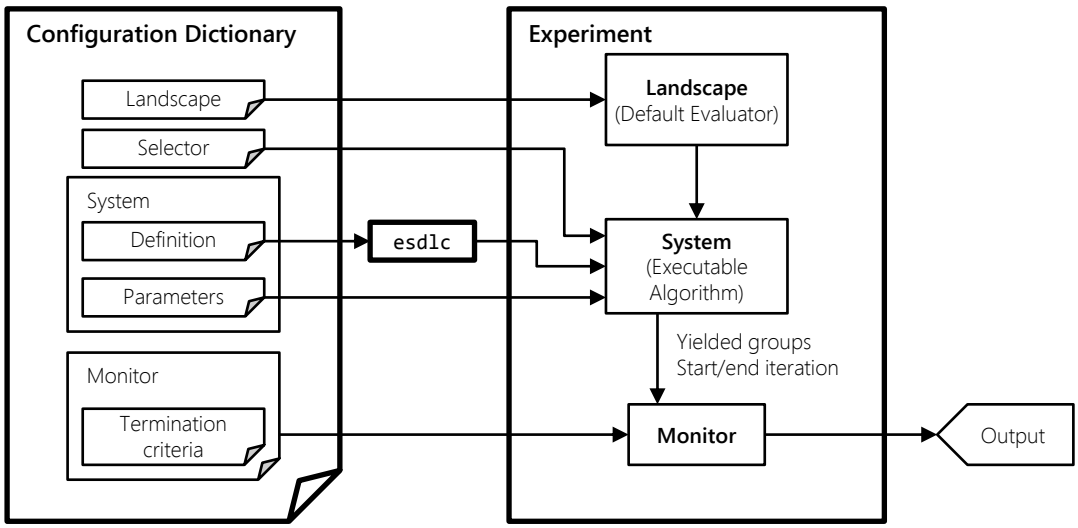

Figure 6.4: Architecture of an `esec` experiment.

est (or lowest) point, though there are no restrictions on the number of dimensions a landscape may use. When specified in the configuration dictionary, the landscape acts as the default evaluator (see Section 4.2.4, page 71).

The block selector is an iterable object that returns the name of the block to execute next (as described in Section 4.3.1, page 74). By default, the selector is a list containing the names of each block available but more complicated selectors may include a reference to the monitor, allowing statistical information to be used to determine the block to select.

Python's support for dynamic compilation and execution is used to transform the ESDL definition into Python code, with operators provided as callable objects in the configuration dictionary. There is no explicit distinction made between any of the external elements: operators, variables, groups, functions, generators and evaluators are all provided in the same way and accessed directly through generated Python code. Generators and operators provided with `esec` are automatically available, including all those defined in Appendix A.

Listing 6.6 shows an example operator that uses Python iterators to read from the input stream and write to the output stream. The for loop retrieves individuals from the source stream one at a time and correctly handles propagating the end of stream signal, while the `yield` statement makes the result available to the next operator in the chain and waits for it to be consumed before continuing. In this way, each operator only processes as many individuals as required by the subsequent operator.

The following sections detail how `esec` implements each of the three models described in Section 5.2.

```
def mutation(_source):
  for indiv in _source:       # read from source stream
    ...                       # perform mutation
    yield new_indiv           # append to output stream
```

Listing 6.6: `esec` operator implemented in Python.

6.3.2 Memory Model

Central to the memory model is the blackboard, which is essentially a mapping from names to the group or variable associated with the name at that time. Mappings in Python, including variable scopes, are implemented with dictionaries. Using the same data structure allows the blackboard to be initialised directly from the configuration dictionary and also to be used as the scope for generated code. Listing 6.7 shows an example of initialising the blackboard from a configuration dictionary. Details relating to external operators, monitors and selectors are omitted for brevity; these are covered in later sections.

Using `exec` with an explicit scope parameter ensures that all variables created, modified or accessed by the generated code are stored in `blackboard` and separated from the outer scope. Due to Python's scoping behaviour, operator and function implementations cannot access the blackboard, as specified by Section 5.2.1.

Operators are implemented as Python iterators with groups represented by lists. This allows language features and libraries to be used to provide merge and partition operators. Lists may be used as iterators implicitly, which allows interchangeable use of operators and groups as sources to other operators. Further discussion of operator implementations is deferred to Section 6.3.4; however, the partitioning operator is relevant to memory allocations and requires coverage here.

Partitioning is performed by the `_part` function which returns the resulting group. Assigning the result to the group variable updates the blackboard reference to a fully constructed list, rather than creating an empty list and appending individuals. This ensures that the partially initialised groups cannot be accessed. Further, since dictionary contents are covered by Python's built-in reference counting, the memory is recovered automatically once a value has no references remaining. Private channels are established for transferring individuals since the iterator reference is only provided to the next operator in the chain.

Fitness values are stored as a member of the individual but are only evaluated (and cached) when the fitness attribute is accessed. This apparent breach of the immutability constraint actually has no impact in a single-threaded environment. Users can provide alternate representations that evaluate fitness eagerly if required.

```
config = {
  'system': {
    'size': 100,
    'definition': '''
FROM random_indiv SELECT (size) population
YIELD population

BEGIN generation
  FROM population SELECT (size) population \
      USING fitness_proportional, crossover_one, mutate_random
  YIELD population
END'''
  },
}

blackboard = dict(config['system'])         # shallow-copy original dict
esdl_source = blackboard.pop('definition')  # remove ESDL system from blackboard
py_source = esdlc.compileESDL(esdl_source)

exec(py_source, blackboard)
# blackboard now contains _block_init() and _block_generation() functions,
# as well as a size variable

exec('_block_init()', blackboard)   # run initialisation block
# blackboard now contains a population variable, which is a list of individuals

while shouldContinue():             # actually belongs to the monitor
  block_name = 'generation'         # actually comes from the selector

  exec('_block_' + block_name + '()', blackboard)
  # blackboard now contains the updated population
```

Listing 6.7: Initialisation of the `esec` blackboard in Python.

For non-group variables, the opaque data type (Section 5.2.1) is fulfilled directly by Python's dynamic type system; since any value may be assigned to any variable, regardless of its previous type, there is no need to declare or restrict variables. Any Python object may be provided in the configuration dictionary, returned by an external function or specified as an argument. Support is included in the ESDL compiler to allow any variable to be called (as if a function) rather than only explicitly declared functions, which allows Python objects to provide methods within system definitions.

Finally, `YIELD` is implemented as a call to the monitor with a reference to the yielded group. This formalises the access to contents of the blackboard, and since neither the monitor not the generated code modify the contents of the group, there is no need to make a copy; Python's reference counting ensures that the group is not released as long as the monitor holds a reference. Depending on the implementation of the monitor, analysis may be performed immediately or deferred to the end of the iteration—deferring even later is possible if the group is not used for termination conditions. The included `ConsoleMonitor` and `CSVMonitor` classes collect basic statistics such as fitness range and distribution immediately, which has the added effect of evaluating all individuals in the group that do not already have cached fitness values.

In summary, ESDL's memory model (Section 5.2.1) is similar to Python's execution model and can be implemented easily. Dynamic execution is restricted to the scope of the blackboard and streaming is provided by Python's iterator pattern. Opaque variable types match the behaviour of Python's dynamic typing and the memory model can be fully implemented in Python without modification or relaxation.

6.3.3 Sequence Model

As stated in Section 5.2.2, a model can behave correctly by atomically executing each statement in the specified order. Since `esec` favours flexibility over execution performance, overlapping and out-of-order execution are not used. Further, preserving statement order allows users to include non-pure functions (with side effects) without requiring explicit synchronisation.

Each store operation is converted into multiple lines of Python code: one to construct the operator chain and one line for each destination group. When only one destination group is specified the entire operation can be collapsed to one line of code; otherwise, a reference to the operator chain needs to be stored temporarily to allow each destination group to use it in turn. The `itertools.chain` function performs merging and `itertools.islice` handles partitioning, though `esec` uses `_merge` and

```
# FROM a, b SELECT 100 c, d USING crossover, mutation

_gen = mutation(_source=crossover(_source=_merge(a, b)))
c = _group(_part(_gen, 100))
d = _group(_gen)
```

Listing 6.8: Python code generated for a `FROM-SELECT` statement.

```
# a = 10
# result = function_call(parameter=value, a, b)

result = function_call(parameter=value, a=a, b=True)
```

Listing 6.9: Python code generated for a function call.

`_part` wrapper functions that also handle parameterless generators[2] and update the blackboard correctly. Listing 6.8 shows an example `FROM-SELECT` statement and the equivalent Python statement. `JOIN-INTO` statements are constructed in an identical manner, except that `_join` is used instead of `_merge`.

Function operations are written almost identically to the ESDL definition, as Python supports similar named parameter syntax to ESDL. Implicit parameters are handled by reproducing the name if a variable exists or specifying `True` otherwise. Listing 6.9 shows the code generated for a function call with one explicit parameter and two implicit parameters, only one of which has a value. Variables specified only in the configuration dictionary but not the ESDL definition are made available to the compiler, ensuring that code generation is correct.

Due to the sequential nature of the code, until a function returns no other processing takes place. The same applies to `YIELD` statements, so if the monitor performs statistical analysis synchronously, the algorithm will halt until it is complete. With the lack of statement reordering and freedom in variable types, developers can implement systems with complex side effects but known behaviour. These are unlikely to be portable to other implementations, but may provide useful functionality for innovative research.

In summary, `esec` avoids the need for a complicated scheduling model by using a purely sequential model. This may affect execution performance, though it does allow the use of functionality such as non-pure functions and user feedback or interactivity. The store and function operations (Section 5.2.2) are used as models for code generation, ensuring that valid code is created for all arrangements of source and destination groups and operator chains.

[2] Essentially by detecting whether the object can be iterated over or called. Generators with parameters are called and their result (an iterator) is passed to `_merge`.

6.3.4 Extensibility Model

Each extensible object in `esec` is a callable Python object—typically a function. Since the configuration dictionary is capable of containing references to these objects, users can include operators and functions as key-value pairs. The `system` section is already used as the initial blackboard for the algorithm (see Listing 6.7), and `esec` allows variables to be called as functions or operators without an explicit declaration, making the inclusion of external functions or operators a matter of adding an entry to the configuration dictionary. This has the added benefit of clearly documenting which externals are in use for a given experiment.

Since Python supports late binding, named parameters and default values, there is no need to specify an operator or function prototype as in a language like C++. The only constraint on operators is the inclusion of a parameter named `_source` to receive the source stream. This parameter would usually be positioned first, though since it is referred to by name in generated code the position is irrelevant. Names beginning with an underscore are reserved for internal use and hence no other parameter names should conflict with `_source`.

Implementing operators using Python's iterator pattern provides the full interface specified in Section 5.2.3: parameters are set when the function is called, the source stream is provided as an iterator and the output stream is the iterator that is returned.[3] For joiners, which in `esec` are only distinguished by their appearance in `JOIN-INTO` statements, the `_source` parameter contains a list of the source groups. Generators are defined similarly to operators but have no `_source` parameter and iterate forever. Output streams in all cases may be provided as any iterable object, such as returning a list; using the `yield` statement is not essential.

In summary, Python's callable object and iterator patterns fully support the extensibility model (Section 5.2.3) with very few extra requirements. Python supports named parameters and default arguments, and provided a `_source` parameter is included, operators can be implemented as filters using the iterator pattern. The fallback procedure to instance methods where operators are not specified (Section 6.3.1) allows different implementations to be provided depending on the representation of the individuals. A simple implementation of the extensibility model allows users to easily define and use novel operators to create and evaluate algorithms.

6.3.5 Configuration Files

`esec` includes a `run.py` script intended for running reproducible experiments, making use of externally defined configuration files. These files use Python syntax and include a configuration dictionary named `config`, any operator or evaluator definitions,

[3] Use of the `yield` statement transforms the function into an iterable object that is returned immediately.

and an optional configuration generator function called `batch()`. Listing 6.10 shows a complete configuration file for a GA, including an evaluator implementation, a configuration dictionary for the system and the monitor, the random seed value to use and the `batch()` function. If `run.py` is invoked in batch mode, this function is called to produce a list of configurations; in Listing 6.10 this includes all combinations of the values for `length` and `p_m` in the loops. The `CSVMonitor` class is used to write output to files.

6.3.6 Summary

This section has shown that `esec` implements the execution model described in Chapter 5. Appendix E has a more detailed overview of the design and capabilities of `esec`, and Appendix C specifies the behaviour of the ESDL compiler.

Section 6.2 examined how ESDL and `esec` can support research with algorithms, and the following section evaluates this guidance by applying it to describe the EAs from chapters 3 and 4.

6.4 Code Comparison

This chapter has discussed how ESDL can provide significant benefits to those working with EAs. Through two hypothetical situations in Section 6.1, the potential for ambiguity has been highlighted and a workflow using ESDL was shown that can aid with avoiding these problems. Section 6.2 discussed ESDL's contribution to approaching the design, implementation and sharing of EAs, and Section 6.3 presented `esec`, a software framework that can interpret ESDL systems directly without a manual translation.

Comparing systems described in ESDL to other software implementations is not straightforward. Differences in programming languages, styles and libraries make objective comparisons difficult. An algorithm specification for one framework may be trivial due to existing support, while other algorithms may be impossible to create without effectively implementing all of the required components.

One of the software packages reviewed in Chapter 5 was ECJ. A framework written in Java, ECJ specifies algorithms using parameter files that define the structure and configuration of operator pipelines. Programming in Java is required to provide operators and representations that are not included in the library of classes. ECJ compares similarly to `esec` in that both forms of parameter/configuration files can fully describe an experiment, including the algorithm, problem, termination conditions and level of output. Both can stand alone as descriptions of an experiment, with reference to lower-level code not necessary to obtain an overview of the algorithm.

```
from esec import esdl_eval
from esec.monitors import CSVMonitor

@esdl_eval
def one_max(indiv):
    return sum(1 if gene else 0 for gene in indiv)

config = {
  'random_seed': 177388292,
  'system': {
    'size': 20,
    'length': 10,
    'p_m': 0.01,
    'definition': '''
FROM random_binary(length) SELECT (size) population
EVAL population USING one_max
YIELD population

BEGIN GENERATION
  FROM population SELECT (size) parents USING fitness_proportional
  FROM parents SELECT offspring USING crossover_one, \
                                      mutate_bitflip(per_gene_rate=p_m)

  FROM offspring SELECT population
  YIELD population
END''' },
  'monitor': {
    'class': CSVMonitor,
    'limits': { 'iterations': 100 }
  }
}

def batch():
  for p_m in [0.0, 0.01, 0.05, 0.10]:
    config['system']['p_m'] = p_m
    for length in [20, 50, 100, 200]:
      yield { 'config': config, 'settings': 'system.length=%d' % length }
```

Listing 6.10: `esec` configuration file for GA.

However, because of the differences between ECJ's and `esec`'s libraries, implementing identical algorithms are not comparable volumes of work. Some algorithms can be constructed from available operators, while others require an entirely new pipeline implementation that is equivalent to using ECJ as a library rather than a configurable framework. A similar issue occurs with comparing any EA libraries—a full algorithm implementation may be part of the library, which reduces the effort required to define that particular algorithm, but has little or no effect on the ease of describing others.

In this section, six of the seven algorithms[4] used in chapters 3 and 4 are converted to experiment descriptions in `esec` configuration files and compared to three other descriptions. The experiment is the same in each description and the comparison uses volumetric measures, specifically, lines, words and characters of code, as a weak proxy for effort in creating, reading and understanding the description.

Comparisons are made between `esec` configurations (Section 6.4.1), ECJ parameter files (Section 6.4.2), C# programs using the FakeEALib library (Section 6.4.3) and C# without any library (Section 6.4.4) are used. All of the code, including FakeEALib, was created specifically for this comparison in order to control variations in library support and coding styles; the comparison is between the efficiency of each approach to describe these algorithms and not the skill level of the creator of particular existing descriptions. All of the code created was executable and verified to ensure equivalent behaviour and solution quality. Section 6.4.5 summarises the comparison results and Appendix F contains the full text or code of each configuration.

6.4.1 `esec` Configurations

The format of `esec`'s configuration files is described in Section 6.3.5. Converting the algorithm descriptions from Section 4.4 is achieved by using the ESDL descriptions without modification and translating the pseudocode operator descriptions into Python. Where an operator is provided by the library its implementation is not counted towards the volume of the description.

In all the configuration files, the ESDL definition is separated from the configuration dictionary by storing it in a constant string variable. This string describes the intended algorithm in a way that could be separated completely from the configuration file and presented alone. The other approaches have no equivalent segment that is separable without significant loss of context. However, the entire configuration file is counted for this comparison, since the algorithm cannot be executed without the configuration.

[4] GP is omitted because it depends so heavily on the style of library used that a comparison is effectively comparing libraries rather than the description.

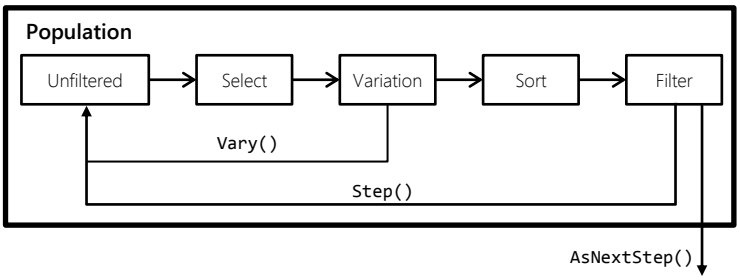

Figure 6.5: The Population class from FakeEALib.

6.4.2 ECJ Parameter Files

Parameter files for ECJ consist of name-value pairs that define an operator pipeline by specifying the source of each operator. A full description of ECJ and its configuration file format can be found in [60]. ECJ is used as a comparison because it is the most widely used framework identified in Chapter 5 and hence is likely to be representative of actual experience.

For comparison purposes, the entire parameter file is counted including implementations of classes or functions that were specified in the `esec` configuration files. While this includes some code that is part of ECJ's library, it omits code that was written to provide equivalence with `esec`, such as real-valued individuals with strategy vectors (as used for EP). Inevitably, because ECJ and `esec` use different approaches to achieve the same quality of description, some concession must be made in order to provide comparability. Appendix F includes all the code that was counted toward the volume measurements shown in Section 6.4.5.

6.4.3 FakeEALib Programs

FakeEALib is a hypothetical library created solely for the purpose of comparing the use of a library to ESDL. Based on the common architectural traits found in the review of EA software in Chapter 5, FakeEALib uses a population-centric model with a sequence description as a separate function. Individuals are cloned explicitly as part of the algorithm but are modified directly by operators. Each population object performs a fixed sequence of steps—select, vary, sort and filter, as shown in Figure 6.5—where each step may consist of one or more operators to apply. For example, the population for EP uses clone selection, expands the population using mutation, and applies a fitness-based sort and a truncation filter to reduce it back to its original size. Each time the `Step()` function is called, these steps are executed and the population is updated.

As an alternative to updating the population directly, the `AsNextStep()` function returns an enumerable sequence of individuals, which performs the same steps but does not update the population directly. Instead, the selected or varied individuals

can be added into another population. ES and SSGA use this approach to transfer the main population into a second one that performs variation. The `Vary()` function bypasses sorting and filtering, but like `Step()` it updates the contents of the population with the new individuals. A separate `VariationMode` property determines whether these functions expand the population, replace the population or compete with their parents. Expanding causes individuals to be cloned prior to variation, while competition automatically evaluates individuals against their parent and keeps the best.

The algorithm functions that initialise, associate and control `Population` instances are used for the volume measurements and are included in Appendix F. They are written using elegant C# using modern language constructs (such as extension methods from `System.Linq`) rather than attempting to optimise for readability, compatibility or minimal code volume, and are intended to be representative of the quantity of code that is written when using the libraries reviewed in Chapter 5.

6.4.4 C# Programs

As a baseline comparison, full implementations of all the algorithms are created using C#. These programs do not require any EA libraries and have the same behaviour as the `esec` descriptions. They are intended as examples of elegant C# code that avoid using functional-programming inspired features, preferring imperative loops. Appendix F includes the full code of these implementations.

6.4.5 Results

Three volumetric measures were taken for each algorithm description. Lines of code were defined as any line containing letters or numbers; those with purely punctuation or comments were not counted. Table 6.3 shows the values for each description and Figure 6.6 shows each count relative to `esec`. Words of code were each consecutive run of either letters or digits and decimal points (in regular expressions: "`([a-z]+|[0-9.]+)`"). Each word was counted once, and all code comments were ignored. Table 6.4 contains the counts for each description and Figure 6.7 shows them relative to `esec`. Finally, all non-whitespace characters were counted for each description, excluding comments. Table 6.5 shows the number of characters in each description and Figure 6.8 shows these relative to `esec`.

Words of code is the fairest comparison of the three, since it eliminates the bias due to Python's typically short variable names and Java's typically long names, as well as the balance of operations per line, which varies between languages. Notably, DE and PSO are very similar in code size across all descriptions, which is largely due to having most of the algorithm contained within either one or two operators. (ES is also similar, but has a much simpler implementation and is therefore less

noteworthy.) The other algorithms are able to make much better use of composition of existing operators, resulting in a significantly reduced amount of text.

Across all measures, `esec` and ESDL consistently require less code than the other approaches. Taken as a weak proxy for effort, this implies that ESDL systems require less effort to create, read and understand than other, equally precise descriptions. The implication is limited in that a person's experience with a form of description is likely to have a greater effect on comprehension or design than size alone, while also failing to account for the ESDL description being separable from the surrounding Python code. Presented alone, ESDL suffers less from loss of context than the equivalent parts, if any, of the other descriptions.

6.5 Chapter Summary

This chapter has discussed the application of ESDL to designing, sharing and implementing EAs. Through two hypothetical stories in Section 6.1, the issues identified in Chapter 2 were demonstrated and mitigated through the approach presented in this work.

ESDL can be used to direct the design process when creating an EA, or to provide a framework for understanding and implementing another description, as discussed in Section 6.2.1. Sharing operators between algorithms is aided by avoiding the nomination of a particular metaphor underlying ESDL. Software frameworks can be based around ESDL, such as `esec`, described in Section 6.3. Parsing and compiling or interpreting ESDL allows for a rapid write-test-revise workflow, as well as verifying that an algorithm presented as discussed in Section 6.2.2 is a correct and complete description.

Section 6.4 showed that EA experiments using ESDL as a description generally require less code compared to current libraries and frameworks. The ESDL descriptions were executed with `esec`, proving that ESDL can correctly and concisely describe a range of existing algorithms and also that it can be automatically compiled in order to compose a working algorithm from a library of operators. The full code files used for this comparison are included in Appendix F.

The discussion and examples in this chapter demonstrate the second contention of the original hypothesis: that "a unified model and approach will aid the understanding, development, implementation and presentation" of EAs.

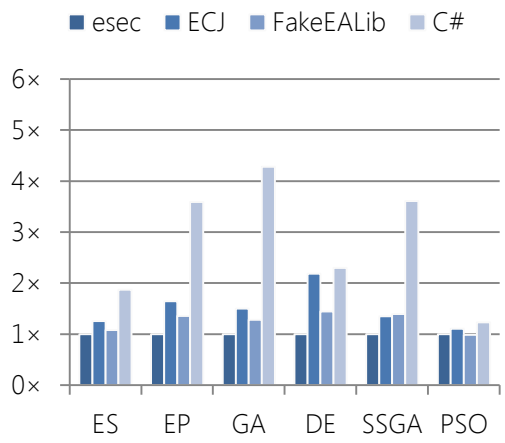

Figure 6.6: Lines of code relative to esec.

Table 6.3: Lines of code for each algorithm.

	esec	ECJ	Lib	C#
ES	39	49	42	73
EP	17	28	23	61
GA	18	27	23	77
DE	27	59	39	62
SSGA	23	31	32	83
PSO	66	73	65	81

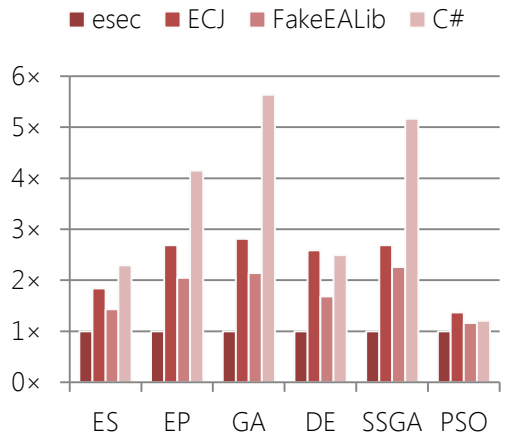

Figure 6.7: Words of code relative to esec.

Table 6.4: Words of code for each algorithm.

	esec	ECJ	Lib	C#
ES	196	360	280	449
EP	82	220	168	340
GA	79	222	169	445
DE	157	406	264	391
SSGA	93	250	210	480
PSO	375	512	435	453

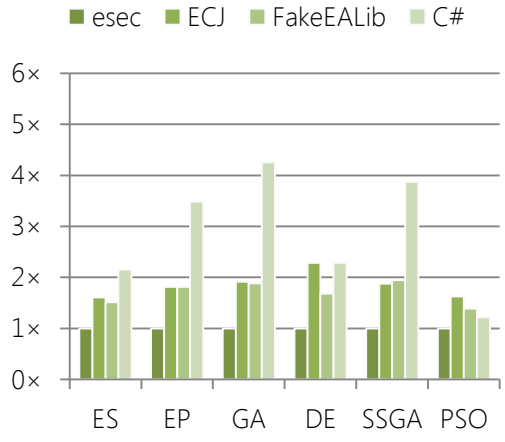

Figure 6.8: Characters of code relative to esec.

Table 6.5: Characters of code for each algorithm.

	esec	ECJ	Lib	C#
ES	1090	1753	1646	2343
EP	529	958	961	1841
GA	526	1006	991	2237
DE	864	1976	1451	1974
SSGA	620	1164	1207	2399
PSO	1848	3002	2562	2253

Chapter 7

Conclusions

This work has described a unified model of Evolutionary Algorithms that supports algorithms from the three principal antecedents, ES, EP and GA, as well as other algorithms with similar iterative structures. The model has been developed into a description language, ESDL, with an execution model that supports accurate specification of algorithms with less code than other approaches and in a form that can easily be shared.

7.1 Research Goals

This work set out to test the hypothesis that

> Evolutionary Algorithms are a single class of algorithm that no longer needs the separation resulting from their distinct origins, and a unified model and approach will aid the understanding, development, implementation and presentation of these algorithms.

This section reviews the contributions of this work towards the six questions posed in Section 1.2.

1. **Are there problems with how EAs are designed, implemented and communicated, and if so, how do these affect practitioners and researchers?** Chapter 2 showed that the current approach is negatively affected by a significant adherence to algorithmic distinctions that are only relevant historically. This adherence results in assumptions of incompatibility between algorithms, limited guidance on constructing targeted algorithms, use of inefficient algorithm elements and overly specific simulation software. These problems result in longer research cycles as new software must often be developed and potentially limit the conception of novel algorithms.
2. **What, fundamentally, is an EA?** Chapter 2 identified that the separation between algorithms is purely theoretical and presented a new definition of

EAs based on the general approach and applications of prior work. Chapter 3 developed this model into a full specification suitable for using as a reference when designing, implementing or presenting an algorithm. Modelling EAs as an iterative process of variation and reduction on groups of potential solutions distinguishes EAs from other classes of algorithms while providing a useful understanding of how the algorithms are applied. Describing EAs as arbitrary networks of operators provides a model that can easily be described, allows for reliable interpretation and supports design and implementation.

3. **How should the design, implementation and sharing of EAs be approached?** Chapter 3 presented a model of EAs that allows reliable interpretation and understanding, which Chapter 4 extended with a text-based description language, ESDL, for concisely expressing specific algorithms. Chapter 5 detailed the behavioural interpretation of ESDL to ensure that ESDL descriptions provide consistent results between human readers and software compilers. Chapter 6 combined the guidance of chapters 3–5, illustrating the problems found in Chapter 2 and demonstrating the mitigations provided by the model and ESDL.
4. **Does this model prevent existing algorithms from being described?** Ensuring that existing work is not invalidated is a critical aim, since the definition of an EA is well understood if not easily expressible. As the model and ESDL were developed through chapters 3 and 4, descriptions of seven representative algorithms (ES, EP, GA, GP, DE, SSGA, PSO) were simultaneously developed to ensure that existing work was included. In Chapter 6 and Appendix F, these algorithms were fully implemented using ESDL and shown to be more concise than alternative forms of description. Chapter 2 noted that algorithms not always considered to be EAs can also be understood using this model, which the description of PSO and the hypothetical AMO (Chapter 6) supports.
5. **Does this model solve the problems found by Question 2?** The issues identified in Chapter 2 are the central motivation of this work, and addressing or mitigating these problems is the aim. Chapter 6 showed that separating algorithm structure and behaviour provides guidance for constructing a targeted algorithm, for decomposing an algorithm description and for sharing operators between algorithms previously considered to be incompatible. Using ESDL with a structured development process or a supporting framework reduces the development and testing effort, allowing greater flexibility for researchers. Software reuse is supported by clearly defined interactions and reduced coupling between components, while decoupling operators and solution representations clarifies that neither are intrinsic to particular algorithms.

7.2 Contributions

The key contributions of this research work are:

- A conceptual model (Chapter 3), representing EAs as compositions of *individuals*, *groups*, *operators* and *streams*. Rather than treating algorithms as indivisible, the model extracts operators and abstracts the commonalities between algorithms into individuals, groups and streams. Individual operators can be defined, shared, exchanged and assessed independently from an algorithm. Entire algorithms can be constructed from operators known to be suitable for the problem, rather than selecting an algorithm that may or may not be suited.
- Evolutionary System Definition Language (ESDL, Chapter 4) for describing the composition and parameterisation of an algorithm. Part programming language and part description language, ESDL allows EAs to be described concisely and precisely, reducing the chance of a reader misinterpreting a written description when reading or implementing the algorithm. Interpreters and compilers can be created for ESDL, removing the manual translation step and allowing simple reuse of published algorithms.
- An execution model (Chapter 5), precisely defining the behaviour and interpretation of ESDL systems in the context of both sequential and parallel machines. It is designed to guide compiler and interpreter developers while allowing freedom to differentiate their own work but retain compatibility with other ESDL implementations.

Software contributions include:

- `esec` (Chapter 6 and Appendix E), a Python-based EC framework that uses ESDL as its configuration mechanism. `esec` includes a large library of operators and benchmark problems, and allows fast prototyping of novel algorithms. It also supports reproducible batches for automating a set of experiments.
- `esdlc` (Appendix C), a Python-based compiler for ESDL that can be extended to target various platforms. The current version is embedded in `esec` to produce Python code, and also supports compilation for GPU using C++ AMP (Appendix D).
- A proposed standard library (Appendix A) of operators for an ESDL implementation. Having a shared minimum set allows users of ESDL to use basic operators with confidence that they always behave the same, regardless of the implementation used.

7.3 Limitations

This section discusses some limitations of the approaches used in this work, identifying alternative approaches to achieve more detailed or robust results and the benefits and restrictions of each. Three specific limitations are covered: the qualitative evaluation of the current approach to EAs, the lack of mathematical formalism in the description of the model and ESDL, and the purely volumetric comparison of algorithm descriptions. The potential mitigations described here are presented in greater detail in Section 7.4.

7.3.1 Qualitative Assessment

Chapter 2 presented an evaluation of issues with designing, implementing and communicating EAs. This assessment was structured and based on a wide reading of recent and historical literature, but did not include any quantitative evaluation. Such an evaluation would be possible by classifying a large number of publications by description formats, completeness of algorithm definitions, use of existing software and tracing the ancestry of the algorithm defined or used. Such a survey would provide an objective measure of the scope of issues with the current approach, and efforts could then be better targeted to the most problematic areas.

However, a fair and reliable survey would require an immense amount of effort, not least because obtaining a sufficient and objective sample of publications requires the evaluation of entire conference proceedings and journals. A random sample from recent conference proceedings ($N \sim 280$) was used to test the value of a wider survey, and found that while "technical reproducibility" is often one of the publication criteria, work that is not sufficiently detailed to allow accurate reproduction is still published—independent verification would be necessary to determine the full scope of this issue. The sample also identified that in a majority of publications simulation software is not mentioned ($\sim 65\%$); some specified that their software was written by the researchers themselves ($\sim 20\%$) and the rest identified a publicly available framework.

Under these circumstances, determining the effect of algorithmic segregation is infeasible. Surveying researchers is also difficult due to the potentially controversial nature of the questions raised and the need for a large and fair sample. Assuming the opposite hypothesis, that algorithm segregation is not an issue, allowed the qualitative approach with counter-examples that was used in Chapter 2 where a quantitative assessment would have been impractical.

7.3.2 Informal Language Model

As a field, computer science has well-established approaches to presenting programming language designs, which were deliberately avoided in this work. The use of

computational algebra and formal notation is typically used to demonstrate the expressiveness of a language and to prove the constraints and limits of a system. Showing that a language construct can be represented by formalisms such as the lambda calculus is a useful proof of correctness.

The approach used in this work relies on discussion and demonstrative examples rather than algebraic reduction and proof. Because the scope of ESDL is deliberately restricted, deriving the language from a general-purpose algebra (or retrospectively applying one) does not demonstrate its ability to describe EAs any more than the mapping to diagrams and executable code in a full-featured programming language that was used. As a software engineering approach, clearly defined coupling between components is sufficient to guarantee the interactions, but the specific behaviours within operators or algorithms as a whole are not within the scope of this work.

However, there are benefits to theoretical analysis of EAs that may be better supported by a formalised set of interactions. The definitions presented in this work are suitable for implementation, experimental use and conceptual understanding. Theoretical analysis should also benefit from the decomposition and separation of operators, but may need a more formal composition approach than is provided.

7.3.3 Volumetric Analysis

Chapter 6 used a volumetric analysis of a range of algorithm descriptions to demonstrate the conciseness of unambiguous ESDL descriptions. Such an analysis is straightforward and easily reproducible and verifiable, but provides limited information on the complexity, readability or usability of ESDL. Where text volume is significantly larger it is fair to infer a slight negative effect on readability, due simply to the extended reading time required. However, other variables are far more significant in a usability assessment than size alone.

A better approach to evaluating the benefits of ESDL over other descriptions would require experiments with human subjects, such as testing for comprehension. Variables to be considered in such an assessment include the subject's understanding of EAs, prior experience with developing or implementing EAs and other algorithms, pre-existing programming skills, computer science aptitude and learning ability. A useful usability assessment would require multiple versions of ESDL to be assessed, which then necessitates accounting for cross-contamination between versions. Finding and controlling a population in order to achieve a statistically relevant result is the most significant difficulty; the benefits were not considered to be critical to this thesis, although a formal usability survey remains important future work.

7.4 Future Work

This work is the first exposition of ESDL and its associated conceptual and execution models. At present, there is one stable implementation (`esec`) and one prototype. Ideally, frameworks using ESDL will become available for a range of popular programming languages, particularly C++, Java and C#, which are all well-represented in recent EA research. Providing users with the ability to select a compiler to match either their preferred language or the interface of existing code is important to achieving wider use of ESDL, without which the full potential benefit cannot be realised.

Much work is ongoing in areas where ESDL can provide support. Theoretical analysis, experimental analysis, parameter tuning and control, network topologies and applications research can all benefit from a common description language and frameworks that execute it. Implementing a parser for ESDL is not complicated and it allows software to support a significant range of algorithms. With large amounts of research not resulting in reusable software, ESDL can reduce the effort required from both the researcher and those who later make use of that research.

This section discusses some specific areas for future research, either because they would benefit from greater coverage than was afforded in this work, or because they are topics for which ESDL may be able to provide significant benefit.

7.4.1 Tool Support

As with any language, supporting tools can significantly increase writer or developer productivity. Editors that provide spelling and grammar suggestions are common for written languages, while programming languages receive assistance with keyword and name completion, error reporting and parameter information. Debuggers, profilers and static analysis tools also provide insight into the execution and allow developers to perfect their work. ESDL can support such tools at two levels: creating ESDL descriptions and implementing operators.

Researchers working with ESDL to implement their algorithms would benefit from editor support to provide assistance with syntax, group names, available operators and parameters. Debugger-like tools could provide introspection into the algorithm itself, for example, allowing a developer to interactively pause execution and inspect the contents of groups, modify parameters and retry statements. Users could identify bottlenecks, poor performing or incorrectly specified operators, validate assumptions and make improvements to an algorithm that may otherwise require many experiments to discover. With consistent operator implementations, debugging an ESDL system is platform independent, retaining the benefit of transferring a description from a debuggable platform to a high-performance one.

Debuggers and profilers for those implementing operators can assist with producing stable and performant components. Depending on the implementation language, such tools may already be available, though are not likely to have any awareness of ESDL and as a result cannot yet use system definitions. Test-benches for simplifying verification of operator behaviours could assist with the guarantees of correctness that those using operators need for their own work. These tools also apply to evaluators, generators and individual representations, all of which require development work outside of that written in ESDL.

Another set of tools could directly support those who are publishing algorithms developed with ESDL. The LaTeX styling used in this work is one example, while other tools might produce diagrams, tables of results and graphs directly from a system definition. Automating these publication preparation tasks would allow researchers to focus on tasks that cannot be automated and reduce the time required for each publication.

7.4.2 Distributed Implementations

An obvious way to improve the execution performance of an algorithm is to distribute processing across multiple computers or processors. Parallelism is inherent in EAs and various approaches have been used to distribute the processing. ESDL is modelled as a network of communicating operators, which matches many existing distributed processing models and makes it safe to execute parts of operator chains or independent operators in parallel.

There is a lot of potential for designing a framework that intelligently distributes operators from an ESDL system. For example, if an algorithm needs to use a particular processing node for a variation operator, but only for one individual, rather than transferring the entire group across a low-bandwidth connection, the selection may be performed on the same processor as the group and only transmit the selected individual (as shown in Figure 7.1). With sufficient metadata, such optimisations could be performed automatically, though a distributed framework may also allow manual resource allocations.

Currently, distributing EAs across separate processors usually requires manual intervention. ESDL's compositional approach embeds enough information about the algorithm in the description to enable automatic parallelisation of EAs. With most consumer hardware containing multiple processors and specialised supercomputers providing far greater parallelism, support for ESDL would allow greater utilisation for a larger group of practitioners.

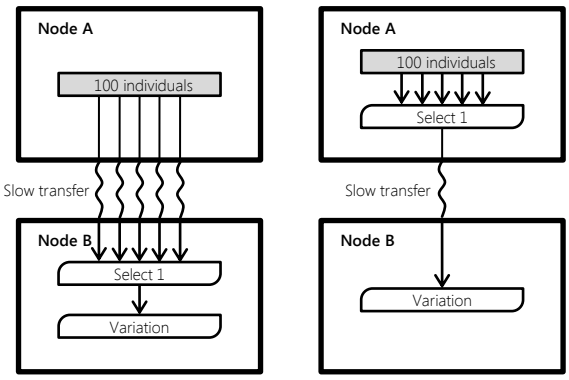

Figure 7.1: Splitting operators across distributed processing nodes.

7.4.3 Language Extensions

While ESDL naturally supports multiple populations, creating large networks of coarse-grained EAs requires significant duplication within the definition, in effect, writing each deme separately. Part of this limitation occurs because of the nature of the fixed network. A potential solution to this could be an implementation that executes multiple blocks in parallel—a literal interpretation of the island populations concept. Inter-algorithm communication would be necessary to transfer selected individuals between the otherwise isolated systems, ideally in a form that allowed systems to be replicated, for example, providing a single ESDL block and letting the framework manage hundreds or thousands of instances in various topological structures. Pragmas are deliberately included in ESDL to allow this form of extension without having to resort to non-standard language modifications.

A second pattern that may benefit from new syntax is similar to 'for each' loops in various programming languages. The model currently supports a simple approach: "for each individual in the group, apply this operator." However, a potential variation that is not supported is: "for each individual in the group, apply these ESDL statements." Such a construct would enable operators to be composed from other operators within the definition itself, allowing operations such as "for each individual in the group, create three variations and retain the best." While the ESDL statements "`FROM a SELECT b USING repeat_each(count=3), vary`" and "`FROM b SELECT (n) c USING best`" describe a similar idea, it does not guarantee that exactly one variation of each individual in group `a` will exist in group `c`. A 'for each' style construct would be able to guarantee this, as well as supporting more complex variation than a single operator.

7.4.4 Theoretical Formalism

Section 7.3.2 discussed the limitations of the informal language model with respect to defining ESDL. While the restricted scope and behaviours of ESDL reduce the

need for a formal definition in order to describe a correct and useful language, theoretical analysis is likely to depend on formally defining the composition of operators. Producing a formal composition model of ESDL may be essential to support pure theoretical analysis of EAs, particularly with respect to deriving global algorithm properties from the known properties of individual contributing components. An understanding of the overall effect a particular operator has on a system is beneficial to the ability to construct algorithms that are targeted to specific problems.

7.4.5 Usability Study

As discussed in Section 7.3.3, the analysis of ESDL performed in this work is limited and does not fully investigate the usability of ESDL. While the model and language are sufficient, a proper usability study could identify behaviours of the model or keywords and syntax in ESDL that are unintuitive or convoluted. The results of such a study may be used to improve ESDL or identify areas in need of extra explanation or documentation.

A usability study would likely require a significant group of users in order to control for varying backgrounds, especially with respect to prior experience with EAs and general programming. Qualitative assessments on a broad scale require accounting for individual biases, with preliminary investigations suggesting the potential for resistance to a new approach due to strong familiarity with another. A potentially beneficial approach may involve using ESDL and the model in teaching EAs, where existing syllabi are available as a control and the prior experience of student participants is better controlled than among active researchers. Existing assessments and surveys or interviews could provide useful qualitative and quantitative feedback under these conditions.

7.5 Final Words

While the most significant contribution of this work is the EA model, the most useful is ESDL. The ability to model, describe and implement EAs in an efficient, unambiguous, portable form could transform the nature of sharing in the EC field. At present, we are not aware of any algorithms generally considered to be EAs that cannot be described with ESDL, and there are many algorithms that are not EAs that can be described. This work describes ESDL in its current form, but is in no way intended to prevent future changes to the language that may be necessary to meet the needs of an actively evolving field. It is hoped that ESDL can help transfer the time spent implementing, debugging and testing software into time spent on algorithm design and validation, where it can help the field continue to evolve and develop.

Bibliography

[1] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*. MIT Press, 1986. 31, 32

[2] M. Affenzeller, "New variants of genetic algorithms applied to problems of combinatorial optimization," in *Proceedings of the EMCSR 2002*, vol. 1, 2002, pp. 75–80. 133

[3] E. Alba and J. M. Troya, "A survey of parallel distributed genetic algorithms," *Complexity*, vol. 4, pp. 31–52, 1999. 16

[4] J. Alcalá-Fernández, L. Sánchez, S. García, M. J. del Jesus, S. Ventura, J. M. Garrell, J. Otero, C. Romero, J. Bacardit, V. M. Rivas, J. C. Fernández, and F. Herrera, "KEEL: a software tool to assess evolutionary algorithms for data mining problems," *Soft Computing*, vol. 13, pp. 307–318, February 2009. 21

[5] M. G. Arenas, P. Collet, A. E. Eiben, M. Jelasity, J. J. Merelo, B. Paechter, M. Preuß, and M. Schoenauer, "A framework for distributed evolutionary algorithms," in *Parallel Problem Solving from Nature – PPSN VII, 7th International Conference*, vol. 2439. Springer, 2002, pp. 665–675. 21

[6] H. Aydt, S. J. Turner, C. Wentong, M. Y. H. Low, O. Yew-Soon, and R. Ayani, "Toward an evolutionary computing modeling language," *IEEE Transactions on Evolutionary Computation*, vol. 15, pp. 230–247, April 2011. 18, 64

[7] T. Bäck, *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996. 11, 14, 15, 16, 19, 41, 45, 48, 50

[8] T. Bäck, F. Hoffmeister, and H.-P. Schwefel, "A survey of evolution strategies," in *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1991, pp. 2–9. xv, 9, 11

[9] D. Berlinski, *Infinite Ascent*. Modern Library, 2005. 33

[10] M. Brameier and W. Banzhaf, *Linear genetic programming*. Springer, 2007. 16

[11] H.-J. Bremermann, "Optimization through evolution and recombination," *Self-Organizing Systems*, pp. 93–106, 1962. 13

[12] J. Brownlee, *Clever Algorithms: Nature-Inspired Programming Recipes*. Lulu, 2011. 45, 48, 50, 78, 88

[13] E. Cantú-Paz, "A survey of parallel genetic algorithms," *Calculateurs paralleles, reseaux et systems repartis*, pp. 141–171, 1998. 250

[14] P. Collet, E. Lutton, M. Schoenauer, and J. Louchet, "Take it EASEA," in *Parallel Problem Solving from Nature – PPSN VI*. Springer, 2000, pp. 891–901. 21, 63

[15] D. Dagum, "Introducing C++ accelerated massive parallelism (C++ AMP)," June 2011, http://blogs.msdn.com/b/vcblog/archive/2011/06/15/introducing-amp.aspx. 252

[16] K. A. De Jong, "An analysis of the behavior of a class of genetic adaptive systems," Ph.D. dissertation, University of Michigan, 1975. 13, 14, 15

[17] ——, "Genetic algorithms are NOT function optimizers," *Foundations of genetic algorithms*, vol. 2, pp. 5–17, 1993. 23

[18] ——, "Two grand challenges for EC," *Frontiers of Evolutionary Computation*, pp. 37–51, 2004. 17, 21

[19] ——, *Evolutionary Computation: A Unified Approach*. MIT Press, 2006. 17, 18, 45, 48

[20] ——, "Evolutionary computation: a unified approach," in *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*. ACM, New York, United States, 2010, pp. 2289–2302. 17, 18

[21] K. A. De Jong and W. M. Spears, "On the state of evolutionary computation," in *Proceedings of the Fifth International Conference on Genetic Algorithms*, 1993, pp. 442–459. 17

[22] S. Debattisti, N. Marlat, L. Mussi, and S. Cagnoni, "Implementation of a simple genetic algorithm within the CUDA architecture," in *Poster session presented at: GECCO '09: Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference*. ACM, New York, United States, 2009. 251

[23] DOLPHIN Project Team, "Paradiseo," 2012, http://paradiseo.gforge.inria.fr. 62

[24] M. Dorigo, M. Birattari, and T. Stützle, "Ant colony optimization," IRIDIA, Tech. Rep. TR/IRIDIA/2006-023, 2006. 16

[25] M. Dorigo, V. Maniezzo, and A. Colorni, "Positive feedback as a search strategy," Polytechnic University of Milano, Tech. Rep., 1991. 283

[26] S. Dower, "ESDL multiblock extension proposal," Swinburne University of Technology, Tech. Rep. TR/CIS/2010/6, 2010. 76

[27] ——, "Bitonic sort for C++ AMP," Swinburne University of Technology, Tech. Rep. TR/CIS/2012/1, 2012. 251

[28] S. Dower and C. J. Woodward, "ESDL: a simple description language for population-based evolutionary computation," in *Proceedings of the 13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011*. ACM, New York, United States, 2011, pp. 1045–1052. 21

[29] A. E. Eiben, "Principled approaches to tuning EA parameters," in *Tutorials—IEEE Congress on Evolutionary Computation (CEC 2009)*, 2009. 18, 21

[30] A. E. Eiben and M. Jelasity, "A critical note on experimental research methodology in EC," in *Proceedings of the 2002 Congress on Evolutionary Computation*. IEEE Press, 2002, pp. 582–587. 20, 21

[31] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Springer, 2003. 18, 41, 45, 55, 76

[32] D. B. Fogel, *Evolutionary Computation: The Fossil Record*. Wiley-IEEE Press, 1998. 17

[33] ——, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, 2007. xv, 7, 8, 12, 17, 48, 49, 79

[34] L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial intelligence through simulated evolution*. John Wiley & Sons, Inc., 1966. 11

[35] M. Fowler, *Domain-Specific Languages*. Addison Wesley, 2010. 60, 61, 63, 64, 234

[36] A. S. Fraser, "Simulation of genetic systems by automatic digital computers. I. Introduction," *Australian Journal of Biological Sciences*, vol. 10, pp. 484–491, 1957. 13

[37] ——, "Simulation of genetic systems by automatic digital computers. II. Effects of linkage rates of advance under selection," *Australian Journal of Biological Sciences*, vol. 10, pp. 492–499, 1957. 13

[38] C. Gagné and M. Parizeau, "Genericity in evolutionary computation software tools: Principles and case-study," *International Journal on Artificial Intelligence Tools*, vol. 15, pp. 173–194, 2006. 21

[39] R. Galar, "Evolutionary search with soft selection," *Biological Cybernetics*, vol. 60, pp. 357–364, 1989. 13

[40] M. Gallagher and B. Yuan, "A general-purpose tunable landscape generator," *IEEE Transactions on Evolutionary Computation*, vol. 10, pp. 590–603, 2006. 283

[41] D. Garlan and M. Shaw, "An introduction to software architectures," *Advances in software engineering and knowledge engineering*, vol. 1, pp. 1–40, 1993. 99

[42] Geneura Team, "Evolvable objects," http://eodev.sourceforge.net/. 20, 62

[43] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 16, pp. 122–128, 1986. 15

[44] K. Gregory, "Overview and C++ AMP approach," September 2011, http://www.gregcons.com/CppAmp/OverviewAndCppAMPApproach.pdf. 250, 252

[45] S. Harding and W. Banzhaf, "Fast genetic programming on GPUs," in *Genetic Programming*. Springer, 2007, pp. 90–101. 250

[46] J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan, Ann Arbor, Michigan, United States, 1975. 13, 14, 15

[47] ——, *Adaptation in Natural and Artificial Systems*. MIT Press, 1992. 13, 51

[48] J. N. Hooker, "Testing heuristics: We have it all wrong," *Journal of Heuristics*, vol. 1, pp. 33–42, 1995. 20, 21, 43

[49] L. Howes and D. Thomas, "Efficient random number generation and application using CUDA," in *GPU Gems 3*. Addison Wesley, 2007. 261

[50] J. Hughes, "Why functional programming matters," *The computer journal*, vol. 32, pp. 98–107, 1989. 45

[51] ISO/IEC14882:2011, "Information technology – programming languages – C++," 2011. 263

[52] J. Kennedy and R. C. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4. IEEE, 1995, pp. 1942–1948. 16, 19

[53] T. Kovacs and R. Egginton, "On the analysis and design of software for reinforcement learning, with a survey of existing systems," *Machine Learning*, vol. 84, pp. 7–49, July 2011. 18, 20, 45, 62

[54] J. R. Koza, *Genetic Programming: On The Programming of Computer Programs by Natural Selection*. MIT Press, 1992. 16, 53

[55] ——, *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994. 276

[56] H. J. Lichtfuß, "Evolution eines rohrkrümmers," Master's thesis, Technische Universität Berlin, 1965. 8

[57] S.-H. Liu, A. Cardenas, X. Xiong, M. Mernik, B. R. Bryant, and J. Gray, "Can domain-specific languages be implemented by service-oriented architecture?" in *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, New York, United States, 2010, pp. 2491–2492. 63

[58] S.-H. Liu, M. Mernik, and B. R. Bryant, "Parameter control in evolutionary algorithms by domain-specific scripting language PPCEA," in *Proceedings of the 1st International Conference on Bioinspired Optimization Methods and their Applications*, 2004, pp. 41–50. xix, 63

[59] A. J. Lockett and R. Miikkulainen, "Real-space evolutionary annealing," in *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, New York, United States, 2011, pp. 1179–1186. 19

[60] S. Luke, "ECJ," http://cs.gmu.edu/~eclab/projects/ecj/. 20, 30, 62, 146

[61] M. Lumpe, "A pi-calculus based approach for software composition," Ph.D. dissertation, University of Bern, 1999. 92

[62] O. Maitre, L. Baumes, N. Lachiche, A. Corma, and P. Collet, "Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA," in *GECCO '09: Proceedings of the 11th annual conference on Genetic and evolutionary Computation*. ACM, New York, United States, 2009, pp. 1403–1410. 21, 63, 250

[63] S. McConnell, *Code Complete*. Microsoft Press, 2004. 62

[64] M. Might, "The CRAPL: an academic-strength open source license." 45

[65] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Genetic Programming, European Conference, Edinburgh, Scotland, UK, April 15-16, 2000, Proceedings*, vol. 1802. Springer, 2000, pp. 121–132. 16

[66] A. Moraglio, "Towards a geometric unification of evolutionary algorithms," Ph.D. dissertation, University of Essex, November 2007. 18, 19, 20

[67] F. Nadi and A. T. Khader, "A parameter-less genetic algorithm with customized crossover and mutation operators," in *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, New York, United States, 2011, pp. 901–908. 19

[68] O. Nierstrasz and D. Tsichritzis, "Component-oriented software technology," in *Object-Oriented Software Composition*. Prentice Hall, 1995, pp. 3–28. 33, 35, 92

[69] I. M. Oliver, D. J. Smith, and J. R. C. Holland, "A study of permutation crossover operators on the travelling salesman problem," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, 1987, pp. 224–230. 133

[70] M. O'Neill and C. Ryan, *Grammatical Evolution*. Kluwer Academic Publishers, 2003. 16

[71] M. O'Neill, C. Ryan, M. Keijzer, and M. Cattolico, "Crossover in Grammatical Evolution," *Genetic Programming and Evolvable Machines*, vol. 4, no. 1, March 2003. 18

[72] Oxford Dictionaries, "evolution," 2012, http://oxforddictionaries.com/definition/evolution. 25

[73] T. Painter, "Grammatical Evolution in Python," 2006. 45

[74] P. Pospichal, J. Jaros, and J. Schwarz, "Parallel genetic algorithm on the CUDA architecture," in *Applications of Evolutionary Computation*. Springer, 2010, pp. 442–451. 251

[75] M. A. Potter and K. A. De Jong, "A cooperative coevolutionary approach to function optimization," in *Proceedings of the The Third Conference on Parallel Problem Solving from Nature*, 1994, pp. 249–257. 16, 39

[76] K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution*. Springer, 2005. 16, 39, 45, 51

[77] I. Rechenberg, "Cybernetic solution path of an experimental problem," Technische Universität Berlin, Tech. Rep., 1965. 8

[78] ——, *Evolutionsstrategie, Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Friedrich Frommann Verlag, 1973. 8, 9, 11

[79] J. D. Schaffer, "Multiple objective optimization with vector evaluated genetic algorithms," in *Proceedings of the 1st International Conference on Genetic Algorithms*. L. Erlbaum Associates Inc., 1985, pp. 93–100. 19

[80] H.-P. Schwefel, "Kybernetische evolution als strategie der experimentellen forschung in der strömungstechnik," Master's thesis, Technische Universität Berlin, 1965. 9

[81] ——, "Projekt MHD-Staustrahlrohr: Experimentelle optimierung einer zweiphasendüse, teil I," Technischer Bericht 11.034/68, 35, AEG Forschungsinstitut, Berlin, Germany, Tech. Rep., 1968. 8

[82] ——, "Evolutionsstrategie und numerishe optimierung," Ph.D. dissertation, Technische Universität Berlin, 1975. 48

[83] ——, *Numerical Optimization of Computer Models*. Wiley, 1981. 9, 11

[84] S. K. Smit and A. E. Eiben, "Comparing parameter tuning methods for evolutionary algorithms," in *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*. IEEE Press, 2009, pp. 399–406. 18

[85] I. Sommerville, *Software Engineering*. Addison Wesley, 2010. 20

[86] W. M. Spears and V. Anand, "A study of crossover operators in genetic programming," in *ISMIS '91: Proceedings of the 6th International Symposium on Methodologies for Intelligent Systems*. Springer, 1991, pp. 409–418. 18

[87] R. Stallman, "GNU make: A program for directing recompilation," Free Software Foundation, Tech. Rep., 1994. 61

[88] V. Stodden, "The scientific method in practice: Reproducibility in the computational sciences," MIT Sloan School of Management, Tech. Rep., February 2010. 20, 62

[89] R. M. Storn and K. V. Price, "Differential evolution: A simple and efficient adaptive scheme for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, pp. 341–359, 1997. 16

[90] H. Sutter, "The free lunch is over," *Dr. Dobb's Journal*, vol. 30, March 2005. 249

[91] ——, "Welcome to the jungle," 2011, http://herbsutter.com/welcome-to-the-jungle/. 249

[92] C. Szyperski, D. Gruntz, and S. Murer, *Component software: beyond object-oriented programming*. Addison Wesley, 2002. 169

[93] R. Vasa, "Growth and change dynamics in open source software systems," Ph.D. dissertation, Swinburne University of Technology, 2010. 33

[94] C. B. Veenhuis, K. Franke, and M. Köppen, "A semantic model for evolutionary computation," in *6th International Conference on Soft Computing*, 2000. xix, 63, 64

[95] S. Wagner and M. Affenzeller, "Heuristiclab: A generic and extensible optimization environment," in *Adaptive and Natural Computing Algorithms*. Springer, 2005, pp. 538–541. 21, 63

[96] D. Whitley, T. Starkweather, and D. Shaner, "The traveling salesman and sequence scheduling: Quality solutions using genetic edge recombination," in *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991, pp. 350–372. 133

[97] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, pp. 67–82, 1997. 9, 19

[98] C. J. Woodward, "Ecosystems, complexity, topology and evolutionary computation," Ph.D. dissertation, Swinburne University of Technology. Faculty of Information and Communication Technologies, 2010. 21, 271

[99] M. Zubair, "Design patterns for programmable parameter control for evolutionary algorithms," Master's thesis, Department of Computer Science, California State University, December 2009. 63

Appendix A

Standard Library

A.1 Overview

This section presents a suggested library of operators. It is expected that implementations supporting ESDL will provide a collection of operators, partly for users' convenience but also to enable operator-specific optimisations. The library here is not mandated, and implementations of ESDL are not required to provide any or all of these components. However, for the full benefits of ESDL to be apparent, the effort to produce an executable algorithm from a description must be minimised. A common set of available operators goes a significant way towards achieving the critical mass of components required for a successful framework [92].

A number of individual representations are also suggested, primarily to allow representation-specific operators to be specified with context. A perhaps notable exception from this section is the tree-based representation used in GP. This representation is highly algorithm-specific: implementations that wish to provide it are welcome to, but it is not suitable for inclusion in a standard library.

All probabilities are in the interval $[0, 1]$ and are tested against a random number selected in the interval $[0, 1)$. If the random number is less than the probability, it is considered to be met.

Reference implementations are provided using Python as a form of pseudocode; actual implementations should exhibit the same behaviour but are unlikely to use the code without modification. The `yield` statement indicates that the value specified is next in the result stream. This code assumes the objects representing individuals can be compared directly, with fitter individuals comparing as less-than.

A.2 Selectors

A.2.1 Repeated

Creates an infinite stream from a source stream by concatenating itself. The order of individuals is unchanged.

Repeated signature.

```
repeated
```

Repeated implementation in Python.

```
def repeated(source):
  group = []
  for indiv in source:
    yield indiv
    group.append(indiv)
  while True:
    for indiv in group:
      yield indiv
```

A.2.2 Repeat Each

Creates a stream from a source stream by repeating each individual the specified number of times. Repetitions are adjacent.

Repeat Each signature.

```
repeat_each(count=2)
```

Repeat Each parameters.

Name	Default	Range	Description
count	2	$[1, \infty)$	The number of times to repeat each individual.

Repeat Each implementation in Python.

```
def repeat_each(source, count):
  for indiv in source:
    for _ in range(count):
      yield indiv
```

A.2.3 Best

Creates a stream from a finite source stream by ordering the individuals by fitness, descending. The first individual is the fittest and the last is the least fit.

Best signature.

```
best
```

Best implementation in Python.

```
def best(source):
  return sorted(source)
```

A.2.4 Worst

Creates a stream from a finite source stream by ordering the individuals by fitness, ascending. The first individual is the least fit and the last is the fittest.

Worst signature.

```
worst
```

Worst implementation in Python.

```
def best(source):
  return sorted(source, reverse=True)
```

A.2.5 Uniform Random

Creates an infinite stream from a finite source stream by ordering the individuals randomly. Individuals may be returned more than once by this selector.

Uniform Random signature.

```
uniform_random
```

Uniform Random implementation in Python.

```
from random import randrange

def uniform_random(source):
  group = list(source)
  while True:
    yield group[randrange(len(group))]
```

A.2.6 Uniform Shuffle

Creates a stream from a finite source stream by ordering the individuals randomly. Individuals are only returned once by this selector.

Uniform Shuffle signature.

```
uniform_shuffle
```

Uniform Shuffle implementation in Python.

```
from random import randrange

def uniform_shuffle(source):
  group = list(source)
  while group:
    yield group.pop(randrange(len(group)))
```

A.2.7 Rank Proportional

Creates a stream from a finite source stream by selecting individuals with a probability proportional to their rank within the source. If the `with_replacement` parameter is true *or* `without_replacement` is false, the resulting stream will be infinite.

Rank Proportional signature.

```
rank_proportional(expectation=1.1, invert=false,
                  with_replacement=false, without_replacement=true)
```

Rank Proportional parameters.

Name	Default	Range	Description
`expectation`	1.1	$[1.0, 2.0]$	The relative probability of selecting the fittest individual.
`invert`	False	$\{\text{True}, \text{False}\}$	If true, the rankings are reversed and less fit individuals are more likely to be selected.
`with_replacement`	True	$\{\text{True}, \text{False}\}$	If true, individuals may be selected multiple times. Setting `without_replacement` to true implies that this is false.
`without_replacement`	False	$\{\text{True}, \text{False}\}$	If true, individuals may only be selected once. Setting `with_replacement` to false implies that this is true.

Rank Proportional implementation in Python.

```
from random import random

def rank_proportional(source, expectation, invert,
                      with_replacement, without_replacement):
  group = sorted(source, reverse=invert)
  size = len(group) # assume size > 1
  wheel = [(expectation - 2.0 * rank * (expectation-1.0) / (size-1.0),
            indiv) for rank, indiv in enumerate(group)]
  total = sum(i[0] for i in wheel)

  while wheel:
    p = random() * total
    i = 0
    while i < len(wheel) and p > wheel[i][0]:
      p -= wheel[i][0]
      i += 1
    if i >= len(group): i = len(group) - 1

    if with_replacement and not without_replacement:
      yield wheel[i][1]
    else:
      prob, indiv = wheel.pop(i)
      total -= prob
      yield indiv
```

A.2.8 Rank-based Stochastic Uniform Sampling

Creates an infinite stream from a finite source stream by selecting individuals proportionally to their rank within the source. The number of intended selections is provided to determine the distribution; taking more than this amount of individuals selects from the group repeatedly.

Rank SUS signature.

```
rank_sus(expectation=1.1, mu=0)
```

Rank SUS parameters.

Name	Default	Range	Description
`expectation`	1.1	$[1.0, 2.0]$	The relative probability of selecting the fittest individual.
`mu`	0	$[0, \infty)$	The number of selections to be made. If zero, use the size of the source stream.

Rank SUS implementation in Python.

```
from random import random

def rank_sus(source, expectation, mu):
  group = sorted(source)
  size = len(group) # assume size > 1
  wheel = [(expectation - 2.0 * rank * (expectation-1.0) / (size-1.0),
            indiv) for rank, indiv in enumerate(group)]
  total = sum(i[0] for i in wheel)
  if mu <= 0: mu = size

  p_delta = total / mu
  p_next = wheel[i][0]
  p = random() * p_delta
  i = 0
  while True:
    while p > p_next:
      i = (i + 1) % size
      p_next += wheel[i][0]
    yield wheel[i][1]
    p += p_delta
```

A.2.9 Tournament

Creates a stream from a finite source stream by selecting the best of a pool of `k` random individuals repeatedly. A greediness factor adds a probability of not selecting the fitter individual.

Tournament signature.

```
tournament(k=2, greediness=1.0, with_replacement=true, without_replacement=false)
```

Tournament parameters.

Name	Default	Range	Description
`k`	2	$[2, \infty)$	The size of pool to use.
`greediness`	1.0	$[0.0, 1.0]$	The probability of selecting the best individual in the pool. If not met, the first individual chosen for the pool is selected.
`with_replacement`	True	$\{\text{True}, \text{False}\}$	If true, individuals may be selected multiple times. Setting `without_replacement` to true implies that this is false.
`without_replacement`	False	$\{\text{True}, \text{False}\}$	If true, individuals may only be selected once. Setting `with_replacement` to false implies that this is true.

Tournament implementation in Python.

```
from random import random, randrange

def tournament(source, k, greediness, with_replacement, without_replacement):
  group = list(source)

  while group:
    pool = [randrange(len(group)) for _ in range(k)]
    if greediness >= 1.0 or random() < greediness:
      winner = max(pool, key=lambda i: group[i].fitness)
    else:
      winner = pool[0]

    if with_replacement and not without_replacement:
      yield group[winner]
    else:
      yield group.pop(winner)
```

A.2.10 Fitness Proportional

Creates a stream from a finite source stream by selecting individuals with a probability proportional to their normalised fitness. If the `with_replacement` parameter is true *or* `without_replacement` is false, the resulting stream will be infinite. This operator is only useful where fitnesses are simple numeric values and more-positive represents fitter.

Fitness Proportional signature.

```
fitness_proportional(offset=null, with_replacement=true, without_replacement=false)
```

Fitness Proportional parameters.

Name	Default	Range	Description
`offset`	Null	Group	A group where the first individual's fitness should be subtracted from all fitnesses as part of normalisation. If omitted, the least-fit individual from the source stream is used; this individual will then have a selection probability of zero.
`with_replacement`	True	{True, False}	If true, individuals may be selected multiple times. Setting `without_replacement` to true implies that this is false.
`without_replacement`	False	{True, False}	If true, individuals may only be selected once. Setting `with_replacement` to false implies that this is true.

Fitness Proportional implementation in Python.

```
from random import random

def fitness_proportional(source, offset, with_replacement, without_replacement):
  group = sorted(source)
  min_fitness = (offset[0] if offset else group[-1]).fitness
  wheel = [(indiv.fitness - min_fitness, indiv) for indiv in group]
  total = sum(i[0] for i in wheel)

  while wheel:
    p = random() * total
    i = 0
    while i < len(wheel) and p > wheel[i][0]:
      p -= wheel[i][0]
      i += 1
    if i >= len(group): i = len(group) - 1

    if with_replacement and not without_replacement:
      yield wheel[i][1]
    else:
      prob, indiv = wheel.pop(i)
      total -= prob
      yield indiv
```

A.2.11 Fitness-based Stochastic Uniform Sampling

Creates an infinite stream from a finite source stream by selecting individuals proportionally to their normalised fitness. The number of intended selections is provided to determine the distribution; taking more than this amount of individuals selects from the group repeatedly.

Fitness SUS signature.

```
fitness_sus(offset=null, mu=0)
```

Fitness SUS parameters.

Name	Default	Range	Description
`offset`	Null	Group	A group where the first individual's fitness should be subtracted from all fitnesses as part of normalisation. If omitted, the least-fit individual from the source stream is used; this individual will then have a selection probability of zero.
`mu`	0	$[0, \infty)$	The number of selections to be made. If zero, use the size of the source stream.

Fitness SUS implementation in Python.

```
from random import random

def fitness_sus(source, offset, mu):
  group = sorted(source)
  size = len(group)
  min_fitness = (offset[0] if offset else group[-1]).fitness
  wheel = [(indiv.fitness - min_fitness, indiv) for indiv in group]
  total = sum(i[0] for i in wheel)
  if mu <= 0: mu = size

  p_delta = total / mu
  p_next = wheel[i][0]
  p = random() * p_delta
  i = 0
  while True:
    while p > p_next:
      i = (i + 1) % size
      p_next += wheel[i][0]
    yield wheel[i][1]
    p += p_delta
```

A.3 Filters

A.3.1 Unique

Removes duplicate individuals from a stream. The `duplicates` filter is the inverse of this filter.

Unique signature.

```
unique
```

Unique implementation in Python.

```
def unique(source):
  seen = set()
  for indiv in source:
    if indiv not in seen:
      yield indiv
      seen.add(indiv)
```

A.3.2 Duplicates

Removes the first instance of each individual from a stream; only duplicates are included in the result. The `unique` filter is the inverse of this filter.

Duplicates signature.

```
duplicates
```

Duplicates implementation in Python.

```
def duplicates(source):
  seen = set()
  for indiv in source:
    if indiv in seen:
      yield indiv
    else:
      seen.add(indiv)
```

A.3.3 Legal

Removes individuals that are not suitable based on constraints built into either the representation or the evaluator. Specification of these constraints is implementation defined. The `illegal` filter is the inverse of this filter.

Legal signature.

```
legal
```

Legal implementation in Python.

```
def legal(source):
  for indiv in source:
    if indiv.legal():
      yield indiv
```

A.3.4 Illegal

Removes individuals that meet constraints built into either the representation or the evaluator. Specification of these constraints is implementation defined. The `legal` filter is the inverse of this filter.

Illegal signature.

```
illegal
```

Illegal implementation in Python.

```
def illegal(source):
  for indiv in source:
    if not indiv.legal():
      yield indiv
```

A.4 Joiners

A.4.1 Tuples

Associates individuals by index within each source stream. The length of the resulting group will be the same as the length of the shortest source.

Tuples signature.

```
tuples
```

Tuples implementation in Python.

```
def tuples(sources):
  iters = [iter(s) for s in sources]
  while True:
    yield JoinedIndividual([next(i) for i in iters])
```

A.4.2 Random Tuples

Associates each individual in the first source stream with a randomly selected individual from each other stream. If the `distinct` parameter is specified, individuals are not associated with another at a matching index. In cases where this is impossible (more indexes are required than are available), implementations may either fail completely or use non-unique indexes. All streams must be finite length.

Random Tuples signature.

```
random_tuples(distinct=false)
```

Random Tuples parameters.

Name	Default	Range	Description
`distinct`	False	{True, False}	True to attempt to use unique indexes for each component individual from each source stream.

Random Tuples implementation in Python.

```
from random import randrange

def random_tuples(sources, distinct):
  groups = [list(s) for s in sources]
  limits = [len(g) for g in groups]
  for i in range(len(groups[0])):
    indexes = [i]
    for g in groups[1:]:
      j = randrange(len(g))
      while distinct and j in indexes:
        j = randrange(len(g))
      indexes.append(j)

    yield JoinedIndividual([g[i] for g, i in zip(groups, indexes)])
```

A.5 Variation Operators

A.5.1 Mutate Random

Creates a stream of individuals by replacing zero or more adjacent elements in each individual in the source stream with a random value. This operator requires overloading for different representations but the interface should be the same or a superset of that shown here.

Mutate Random signature.

```
mutate_random(per_indiv_rate=1.0, per_gene_rate=0.1, genes=0)
```

Mutate Random parameters.

Name	Default	Range	Description
`per_indiv_rate`	1.0	$[0.0, 1.0]$	The probability of each individual being varied. If not met, the individual is included in the result stream unmodified.
`per_gene_rate`	1.0	$[0.0, 1.0]$	The probability of each element being modified.
`genes`	0	$[0, \infty)$	The exact number of elements in each individual to be modified. If zero, `per_gene_rate` is used instead.

Mutate Random implementation in Python.

```
from random import randrange, random

# Assuming binary-valued individuals
def mutate_random(source, per_indiv_rate, per_gene_rate, genes):
  if per_indiv_rate <= 0 or (per_gene_rate <= 0 and genes <= 0):
    return source

  for indiv in source:
    if per_indiv_rate >= 1.0 or random() < per_indiv_rate:
      new_indiv = indiv.clone()
      if genes > 0:
        indices = range(len(indiv))
        for _ in range(genes):
          i = indices.pop(randrange(len(indices)))
          new_indiv[i] = (random() < 0.5)
      else:
        for i in range(len(indiv)):
          if random() < per_gene_rate:
            new_indiv[i] = (random() < 0.5)
      yield new_indiv
    else:
      yield indiv
```

A.5.2 Mutate Insert

Creates a stream of individuals by inserting a sequence of zero or more adjacent random elements in each individual from the source stream. This operator requires overloading for different representations but the interface should be the same or a superset of that shown here.

Mutate Insert signature.

```
mutate_insert(per_indiv_rate=1.0, length=0, shortest=1, longest=10,
              longest_result=0)
```

Mutate Insert parameters.

Name	Default	Range	Description
`per_indiv_rate`	1.0	$[0.0, 1.0]$	The probability of each individual being varied. If not met, the individual is included in the result stream unmodified.
`length`	0	$[0\infty)$	The exact number of adjacent elements to insert. If zero, values are randomly selected from between `shortest` and `longest` inclusively.
`shortest`	1	$[1, \infty)$	The minimum number of elements to insert.
`longest`	10	$[1, \infty)$	The maximum number of elements to insert. If equal to or less than `shortest`, the value of `shortest` is used for each individual.
`longest_result`	0	$[0, \infty)$	The maximum length of an individual after inserting elements. If zero, no limits are applied.

Mutate Insert implementation in Python.

```
from random import randrange, random

# Assuming binary-valued individuals
def mutate_insert(source, per_indiv_rate, length, shortest, longest,
                  longest_result):
  if per_indiv_rate <= 0:
    return source
  if length > 0:
    shortest = longest = length

  if longest_result > 0:
    for indiv in source:
      if (longest_result - len(indiv) < shortest and
          (per_indiv_rate >= 1.0 or random() < per_indiv_rate)):
        if longest > shortest:
          max_len = min(longest, longest_result - len(indiv))
          new_len = randrange(shortest, max_len+1)
        else:
          new_len = shortest
        new_genes = [(random() < 0.5) for _ in range(new_len)]
        cut = randrange(len(indiv))
        new_indiv = indiv.clone_with(indiv[:cut] + new_genes + indiv[cut:])
        yield new_indiv
      else:
        yield indiv
  else:
    for indiv in source:
      if per_indiv_rate >= 1.0 or random() < per_indiv_rate:
        if longest > shortest:
          new_len = randrange(shortest, longest+1)
        else:
          new_len = shortest
        new_genes = [(random() < 0.5) for _ in range(new_len)]
        cut = randrange(len(indiv))
        new_indiv = indiv.clone_with(indiv[:cut] + new_genes + indiv[cut:])
        yield new_indiv
      else:
        yield indiv
```

A.5.3 Mutate Delete

Creates a stream of individuals by removing a sequence of zero or more random elements from each individual in the source stream.

Mutate Delete signature.

```
mutate_delete(per_indiv_rate=1.0, length=0, shortest=1, longest=10,
              shortest_result=1)
```

Mutate Delete parameters.

Name	Default	Range	Description
`per_indiv_rate`	1.0	$[0.0, 1.0]$	The probability of each individual being varied. If not met, the individual is included in the result stream unmodified.
`length`	0	$[0, \infty)$	The exact number of adjacent elements to delete. If zero, values are randomly selected from between `shortest` and `longest` inclusively.
`shortest`	1	$[1, \infty)$	The minimum number of elements to delete.
`longest`	10	$[1, \infty)$	The maximum number of elements to delete. If equal to or less than `shortest`, the value of `shortest` is used for each individual.
`shortest_result`	1	$[1, \infty)$	The minimum length of an individual after inserting elements.

Mutate Delete implementation in Python.

```
from random import randrange, random

def mutate_delete(source, per_indiv_rate, length, shortest, longest,
                  shortest_result):
  if per_indiv_rate <= 0:
    return source
  if length > 0:
    shortest = longest = length

  for indiv in source:
    if (len(indiv) - shortest >= shortest_result and
        (per_indiv_rate >= 1.0 or random() < per_indiv_rate)):
      max_cut = min(longest, len(indiv) - shortest_result)
      if max_cut > shortest:
        cut_len = randrange(shortest, max_cut)
        cut1 = randrange(len(indiv) - cut_len)
        cut2 = cut1 + cut_len
      else:
        cut1 = shortest_result
        cut2 = len(indiv)
      new_indiv = indiv.clone()
      del new_indiv[cut1:cut2]
      yield new_indiv
    else:
      yield indiv
```

A.5.4 Crossover

Creates a stream of individuals from a stream of adjacent pairs of individuals using aligned subsequences from either.

Crossover signatures.

```
crossover(points=1, per_pair_rate=1.0, one_child=true, two_children=false)
crossover_one(per_pair_rate=1.0, one_child=true, two_children=false)
```

Crossover parameters.

Name	Default	Range	Description
`points`	1	$[1, \infty)$	The number of crossover points.
`per_pair_rate`	1.0	$[0.0, 1.0]$	The probability of each pair being combined. If not met and `two_children` is false, only the first individual of the pair is included in the result stream.
`one_child`	True	{True, False}	Produces one child from every two parents. Setting `two_children` to true implies that this is false.
`two_children`	False	{True, False}	Produces two children by exchanging elements. Setting `one_child` to false implies that this is true.

Crossover implementation in Python.

```
from random import random, randrange

def crossover(source, points, per_pair_rate, one_child, two_children):
  if per_pair_rate <= 0 and two_children:
    return source

  while True:
    parent = next(source)
    mate = next(source)
    if per_pair_rate >= 1.0 or random() < per_pair_rate:
      new_indiv1 = parent.clone()
      new_indiv2 = mate.clone()
      max_i = min(len(parent), len(mate))
      i_list = []
      indices = range(max_i)
      for _ in range(points):
        i_list.append(indices.pop(randrange(len(indices))))

      exchanging = False
      for i in range(max_i):
        if i in i_list:
          exchanging = not exchanging
        if exchanging:
          new_indiv1[i], new_indiv2[i] = new_indiv2[i], new_indiv1[i]
    else:
      new_indiv1 = parent
      new_indiv2 = mate

    if one_child and not two_children:
      yield new_indiv1 if random() < 0.5 else new_indiv2
    else:
      yield new_indiv1
      yield new_indiv2
```

A.5.5 Crossover Different

Creates a stream of individuals from a stream of adjacent pairs of individuals using unaligned subsequences from either. If the length of either child exceeds `longest_result`, both are discarded and the originals returned.

Crossover Different signatures.

```
crossover_different(points=1, per_pair_rate=1.0, longest_result=0,
                    one_child=true, two_children=false)
crossover_one_different(per_pair_rate=1.0, longest_result=0,
                        one_child=true, two_children=false)
```

Crossover Different parameters.

Name	Default	Range	Description
`points`	1	$[1, \infty)$	The number of crossover points.
`per_pair_rate`	1.0	$[0.0, 1.0]$	The probability of each pair being combined. If not met and `one_child` is true, one individual from the pair is returned at random.
`longest_result`	0	$[0, \infty)$	The maximum length of both individuals after exchanging elements. If zero, no limits are applied.
`one_child`	True	{True, False}	Produces one child from every two parents. Setting `two_children` to true implies that this is false.
`two_children`	False	{True, False}	Produces two children by exchanging elements. Setting `one_child` to false implies that this is true.

Crossover Different implementation in Python.

```
from random import random, randrange

def crossover_different(source, points, per_pair_rate, longest_result,
                        one_child, two_children):
  if per_pair_rate <= 0 and two_children:
    return source

  while True:
    parent = next(source)
    mate = next(source)
    if per_pair_rate >= 1.0 or random() < per_pair_rate:
      new_genes1 = []
      new_genes2 = []
      i_list1 = [len(parent)]
      i_list2 = [len(mate)]
      indices1 = range(len(parent))
      indices2 = range(len(mate))
      for _ in range(points):
        i_list1.append(indices1.pop(randrange(len(indices1))))
        i_list2.append(indices2.pop(randrange(len(indices2))))
      i_list1.sort()
      i_list2.sort()

      exchanging = False
      i1 = 0
      i2 = 0
      while len(indices1) > 0 and len(indices2) > 0:
        if exchanging:
          new_genes2.extend(parent[i1:indices1[0]])
          new_genes1.extend(mate[i2:indices2[0]])
        else:
          new_genes1.extend(parent[i1:indices1[0]])
          new_genes2.extend(mate[i2:indices2[0]])
        exchanging = not exchanging
        i1 = indices1.pop(0)
        i2 = indices2.pop(0)

      if (longest_result > 0 and
        (len(new_indiv1) > longest_result or len(new_indiv2) > longest_result)):
        new_indiv1 = parent
        new_indiv2 = mate
    else:
      new_indiv1 = parent
      new_indiv2 = mate

    if one_child and not two_children:
      yield new_indiv1 if random() < 0.5 else new_indiv2
    else:
      yield new_indiv1
      yield new_indiv2
```

A.5.6 Crossover Uniform

Creates a stream of individuals from a stream of adjacent pairs of individuals using elements from either.

Crossover Uniform signature.

```
crossover_uniform(per_pair_rate=1.0, per_gene_rate=0.5, genes=0,
                  one_child=true, two_children=false)
```

Crossover Uniform parameters.

Name	Default	Range	Description
`per_pair_rate`	1.0	$[0.0, 1.0]$	The probability of each pair being combined. If not met and `two_children` is false, only the first individual of the pair is included in the result stream.
`per_gene_rate`	0.5	$[0.0, 1.0]$	The probability of each element being exchanged.
`genes`	0	$[0, \infty)$	The exact number of elements in each pair to be exchanged. If zero, `per_gene_rate` is used instead.
`one_child`	True	$\{\text{True}, \text{False}\}$	Produces one child from every two parents. Setting `two_children` to true implies that this is false.
`two_children`	False	$\{\text{True}, \text{False}\}$	Produces two children by exchanging elements. Setting `one_child` to false implies that this is true.

Crossover Uniform implementation in Python.

```
from random import random, randrange

def crossover_uniform(source, per_pair_rate, per_gene_rate, genes,
                      one_child, two_children):
  if ((per_pair_rate <= 0 or (per_gene_rate <= 0 and genes <= 0)) and
      two_children):
    return source

  while True:
    parent = next(source)
    mate = next(source)
    if per_pair_rate >= 1.0 or random() < per_pair_rate:
      new_indiv1 = parent.clone()
      new_indiv2 = mate.clone()
      max_i = min(len(parent), len(mate))
      if genes > 0:
        indices = range(max_i)
        for _ in range(genes):
          i = indices.pop(randrange(len(indices)))
          new_indiv1[i], new_indiv2[i] = new_indiv2[i], new_indiv1[i]
      else:
        for i in range(max_i):
          if random() < per_gene_rate:
            new_indiv1[i], new_indiv2[i] = new_indiv2[i], new_indiv1[i]
      yield new_indiv1
      if two_children:
        yield new_indiv2
    else:
      yield parent
      if two_children:
        yield mate
```

A.5.7 From Tuple

Creates a stream from a source stream of joined individuals by selecting the component individual with the specified index. Despite appearing to be a selector, this is a variation operator because the individuals returned are a different type to the source.

From Tuple signature.

```
from_tuple(index=1)
```

From Tuple parameters.

Name	Default	Range	Description
index	1	$[1, n]$	The one-based index to select individuals from. n is the number of component individuals in the joined individuals.

From Tuple implementation in Python.

```
def from_tuple(source, index):
  for joined_indiv in source:
    yield joined_indiv[index]
```

A.5.8 Best of Tuple

Creates a stream from a source stream of joined individuals by selecting the component individual with the highest fitness. If more than one individual have equally high fitnesses, the first in the tuple is returned. Despite appearing to be a selector, this is a variation operator because the individuals returned are a different type to the source.

Best of Tuple signature.

```
best_of_tuple(greediness=1.0)
```

Best of Tuple parameters.

Name	Default	Range	Description
greediness	1.0	$[0.0, 1.0]$	The probability of selecting the best individual in the joined individual. If not met, a random individual is selected.

Best of Tuple implementation in Python.

```
def best_of_tuple(source, greediness):
  for joined_indiv in source:
    if greediness >= 1.0 or random() < greediness:
      yield max(joined_indiv, key=lambda indiv: indiv.fitness)
    else:
      yield joined_indiv[randrange(len(joined_indiv))]
```

A.5.9 Crossover Tuple

Creates a stream of individuals from a stream of joined individuals using elements from any of the components.

Crossover Tuple signature.

```
crossover_tuple(per_indiv_rate=1.0, greediness=0.0)
```

Crossover Tuple parameters.

Name	Default	Range	Description
per_indiv_rate	1.0	$[0.0, 1.0]$	The probability of each individual being combined. If not met, the first component individual of the joined individual is included in the result stream.
greediness	0.0	$[0.0, 1.0]$	The probability of guaranteeing an element is selected from the first individual. If not met, every component individual (including the first) has an equal probability.

Crossover Tuple implementation in Python.

```
from random import random, randrange

def crossover_tuple(source, per_pair_rate, greediness):
  if per_indiv_rate <= 0 or greediness >= 1.0:
    for indiv in source:
      yield source[0]

  for indiv in source:
    if per_pair_rate >= 1.0 or random() < per_pair_rate:
      new_indiv = indiv[0].clone()
      for i in range(len(new_indiv)):
        if greediness <= 0.0 or random() >= greediness:
          new_indiv[i] = indiv[randrange(len(indiv))][i]
      yield new_indiv
    else:
      yield indiv[0]
```

A.6 Binary-valued Operators

A.6.1 Representation

A variable-length array of rank one containing Boolean values. This is equivalent to a fixed-length array if no length-varying operators are used. Implementations may choose to use a different internal representation depending on the parameters provided to the generator.

A.6.2 Random Binary Generator

Creates an infinite stream of binary-valued individuals that are initialised from a uniform random distribution.

Random Binary signature.

```
random_binary(length=0, shortest=10, longest=10, true_rate=0.5)
```

Random Binary parameters.

Name	Default	Range	Description
`length`	0	$[0, \infty)$	The number of elements in each individual. If zero, the range between `shortest` and `longest` is used instead.
`shortest`	10	$[1, \infty)$	The minimum number of elements in each individual.
`longest`	10	$[1, \infty)$	The maximum number of elements in each individual. If equal to or less than `shortest`, the value of `shortest` is used for every individual.
`true_rate`	0.5	$[0.0, 1.0]$	The probability of each element of each individual being true.

Random Binary implementation in Python.

```
from random import random, randrange

def random_binary(length, shortest, longest, true_rate):
  if length > 0:
    shortest = longest = length

  while True:
    if longest <= shortest:
      indiv_len = shortest
    else:
      indiv_len = randrange(shortest, longest+1)

    yield BinaryIndividual([random() < true_rate for _ in range(indiv_len)])
```

A.6.3 Binary True and False Generators

Creates an infinite stream of binary-valued individuals that are all initialised to either true or false. These are equivalent to using `random_binary` with `true_rate` set to 1.0 or 0.0.

Binary True and False signatures.

```
binary_false(length=0, shortest=10, longest=10)
binary_true(length=0, shortest=10, longest=10)
```

Binary True and False parameters.

Name	Default	Range	Description
`length`	0	$[0, \infty)$	The number of elements in each individual. If zero, the range between `shortest` and `longest` is used instead.
`shortest`	10	$[1, \infty)$	The minimum number of elements in each individual.
`longest`	10	$[1, \infty)$	The maximum number of elements in each individual. If equal to or less than `shortest`, the value of `shortest` is used for every individual.

Binary False implementation in Python.

```
from random import randrange

def binary_false(length, shortest, longest):
  if length > 0:
    shortest = longest = length

  while True:
    if longest <= shortest:
      indiv_len = shortest
    else:
      indiv_len = randrange(shortest, longest+1)

    yield BinaryIndividual([False] * indiv_len)
```

Binary True implementation in Python.

```
from random import randrange

def binary_true(length, shortest, longest):
  if length > 0:
    shortest = longest = length

  while True:
    if longest <= shortest:
      indiv_len = shortest
    else:
      indiv_len = randrange(shortest, longest+1)

    yield BinaryIndividual([True] * indiv_len)
```

A.6.4 Mutate Bit Flip

Creates a stream of binary-valued individuals by inverting zero or more elements in each individual in the source stream.

Mutate Bit Flip signature.

```
mutate_bitflip(per_indiv_rate=1.0, per_gene_rate=0.1, genes=0)
```

Mutate Bit Flip parameters.

Name	Default	Range	Description
`per_indiv_rate`	1.0	$[0.0, 1.0]$	The probability of each individual being varied. If not met, the individual is included in the result stream unmodified.
`per_gene_rate`	0.1	$[0.0, 1.0]$	The probability of each element being modified.
`genes`	0	$[0, \infty)$	The exact number of elements in each individual to be modified. If zero, `per_gene_rate` is used instead.

Mutate Bit Flip implementation in Python.

```
from random import random, randrange

def mutate_bitflip(source, per_indiv_rate, per_gene_rate, genes):
  if per_indiv_rate <= 0 or (per_gene_rate <= 0 and genes <= 0):
    return source

  for indiv in source:
    if per_indiv_rate >= 1.0 or random() < per_indiv_rate:
      if genes > 0:
        new_indiv = indiv.clone()
        indices = range(len(indiv))
        for _ in range(genes):
          i = indices.pop(randrange(len(indices)))
          new_indiv[i] = not new_indiv[i]
      else:
        new_indiv = [not i if random() < per_gene_rate else i
                     for i in indiv]
      yield new_indiv
    else:
      yield indiv
```

A.6.5 Mutate Inversion

Creates a stream of individuals from a stream of binary-valued individuals by inverting entire individuals.

Mutate Inversion signature.

```
mutate_inversion(per_indiv_rate=1.0)
```

Mutate Inversion parameters.

Name	Default	Range	Description
per_indiv_rate	1.0	[0.0, 1.0]	The probability of each individual being varied. If not met, the individual is included in the result stream unmodified.

Mutate Inversion implementation in Python.

```
from random import random

def mutate_inversion(source, per_indiv_rate):
  if per_indiv_rate <= 0:
    return source

  for indiv in source:
    if per_indiv_rate >= 1.0 or random() < per_indiv_rate:
      new_indiv = BinaryIndividual([not i for i in indiv])
      yield new_indiv
    else:
      yield indiv
```

A.6.6 Mutate Gap Inversion

Creates a stream of individuals from a stream of binary-valued individuals by inverting zero or more adjacent elements in each.

Mutate Gap Inversion signature.

```
mutate_gap_inversion(per_indiv_rate=1.0, length=0, shortest=1, longest=10)
```

Mutate Gap Inversion parameters.

Name	Default	Range	Description
`per_indiv_rate`	1.0	$[0.0, 1.0]$	The probability of each individual being varied. If not met, the individual is included in the result stream unmodified.
`length`	0	$[0, \infty)$	The exact number of adjacent elements to insert. If zero, values are randomly selected from between `shortest` and `longest` inclusively.
`shortest`	1	$[1, \infty)$	The minimum number of elements to invert.
`longest`	10	$[1, \infty)$	The maximum number of elements to invert. If equal to or less than `shortest`, the value of `shortest` is used for each individual.

Mutate Gap Inversion implementation in Python.

```
from random import random, randrange

def mutate_gap_inversion(source, per_indiv_rate, length, shortest, longest):
  if length > 0:
    shortest = longest = length
  if per_indiv_rate <= 0 or longest <= shortest <= 0:
    return source

  for indiv in source:
    if per_indiv_rate >= 1.0 or random() < per_indiv_rate:
      if longest <= shortest:
        inv_len = shortest
      else:
        inv_len = randrange(shortest, longest+1)

      new_indiv = indiv.clone()
      start = randrange(0, len(indiv) - inv_len)
      for i in range(start, start + inv_len):
        new_indiv[i] = not new_indiv[i]
      yield new_indiv
    else:
      yield indiv
```

A.7 Real-valued Operators

A.7.1 Representation

A variable-length array of rank one containing real values. The initial range of values is remembered to allow their use later as constraints. Implementations may choose to use a different internal representation depending on the parameters provided to the generator.

A.7.2 Random Real Generator

Creates an infinite stream of real-valued individuals that are initialised from a uniform random distribution.

Random Real signature.

```
random_real(length=0, shortest=10, longest=10, lowest=0.0, highest=1.0)
```

Random Real parameters.

Name	Default	Range	Description
`length`	0	$[0, \infty)$	The number of elements in each individual. If zero, the range between `shortest` and `longest` is used instead.
`shortest`	10	$[1, \infty)$	The minimum number of elements in each individual.
`longest`	10	$[1, \infty)$	The maximum number of elements in each individual. If equal to or less than `shortest`, the value of `shortest` is used for every individual.
`lowest`	0.0	$(-\infty, \infty)$	The lowest possible value of each element.
`highest`	1.0	$(-\infty, \infty)$	The highest possible value of each element.

Random Real implementation in Python.

```
from random import randrange
from sys import maxsize

def random_real(length, shortest, longest, lowest, highest):
  if length > 0:
    shortest = longest = length

  scale = (highest - lowest) / float(maxsize - 1)
  while True:
    if longest <= shortest:
      indiv_len = shortest
    else:
      indiv_len = randrange(shortest, longest+1)

    yield RealIndividual(
      [randrange(maxsize) * scale + lowest for _ in range(indiv_len)],
      lowest, highest)
```

Note that `randrange` is used on the last line to ensure that element values are selected in $[0.0, 1.0]$ rather than $[0.0, 1.0)$.

A.7.3 Real Value, Low, Mid and High Generators

Creates an infinite stream of real-valued individuals that are initialised at a specific value, the lowest value, the highest value or the midpoint of the range.

Real Value, Low, Mid and High signatures.

```
real_value(length=0, shortest=10, longest=10, lowest=0.0, highest=1.0, value=0.0)
real_low(length=0, shortest=10, longest=10, lowest=0.0, highest=1.0)
real_mid(length=0, shortest=10, longest=10, lowest=0.0, highest=1.0)
real_high(length=0, shortest=10, longest=10, lowest=0.0, highest=1.0)
```

Real Value, Low, Mid and High parameters.

Name	Default	Range	Description
`length`	0	$[0, \infty)$	The number of elements in each individual. If zero, the range between `shortest` and `longest` is used instead.
`shortest`	10	$[1, \infty)$	The minimum number of elements in each individual.
`longest`	10	$[1, \infty)$	The maximum number of elements in each individual. If equal to or less than `shortest`, the value of `shortest` is used for every individual.
`lowest`	0.0	$(-\infty, \infty)$	The lowest possible value of each element.
`highest`	1.0	$(-\infty, \infty)$	The highest possible value of each element.
`value`	0.0	$(-\infty, \infty)$	The value to initialise to (`real_value` only).

Real Value implementation in Python.

```
from random import randrange

def real_value(length, shortest, longest, lowest, highest, value):
  if length > 0:
    shortest = longest = length

  while True:
    if longest <= shortest:
      indiv_len = shortest
    else:
      indiv_len = randrange(shortest, longest+1)

    yield RealIndividual([value] * indiv_len, lowest, highest)
```

Real Low, Mid and High implementations in Python.

```
def real_low(length, shortest, longest, lowest, highest):
  return real_value(length, shortest, longest, lowest, highest, lowest)

def real_mid(length, shortest, longest, lowest, highest):
  value = lowest + (highest - lowest) * 0.5
  return real_value(length, shortest, longest, lowest, highest, value)

def real_high(length, shortest, longest, lowest, highest):
  return real_value(length, shortest, longest, lowest, highest, highest)
```

A.7.4 Clamp

Creates a stream of individuals from a stream of real-valued individuals by changing values to be within either the specified range or the range used to create the individuals.

Real Clamp signature.

```
clamp(lowest=0.0, highest=0.0)
```

Real Clamp parameters.

Name	Default	Range	Description
`lowest`	0.0	$(-\infty, \infty)$	The lowest value of each element.
`highest`	0.0	$(-\infty, \infty)$	The highest value of each element. If less than or equal to `lowest`, the initialisation range is used.

Real Clamp implementation in Python.

```
def clamp(source, lowest, highest):
  for indiv in source:
    if highest <= lowest:
      low, high = indiv.lowest, indiv.highest
    else:
      low, high = lowest, highest

    new_indiv = indiv.clone()
    changed_any = False
    for i in range(len(new_indiv)):
      v = new_indiv[i]
      if v < low:
        changed_any = True
        new_indiv[i] = low
      elif v > high:
        changed_any = True
        new_indiv[i] = high

    yield new_indiv if changed_any else indiv
```

A.7.5 Mutate Delta

Creates a stream of individuals from a stream of real-valued individuals by adding a fixed-size step value to zero or more elements.

Real Mutate Delta signature.

```
mutate_delta(per_indiv_rate=1.0, per_gene_rate=1.0, genes=0,
             step_size=1.0, positive_rate=0.5)
```

Real Mutate Delta parameters.

Name	Default	Range	Description
`per_indiv_rate`	1.0	$[0.0, 1.0]$	The probability of each individual being varied. If not met, the individual is included in the result stream unmodified.
`per_gene_rate`	1.0	$[0.0, 1.0]$	The probability of each element being modified.
`genes`	0	$[0, \infty)$	The exact number of elements in each individual to be modified. If zero, `per_gene_rate` is used instead.
`step_size`	1.0	$[0, \infty)$	The value to add to or subtract from selected elements.
`positive_rate`	0.5	$[0.0, 1.0]$	The probability of adding `step_size` rather than subtracting it.

Real Mutate Delta implementation in Python.

```
from random import random, randrange

def mutate_inversion(source, per_indiv_rate, per_gene_rate, genes,
                     step_size, positive_rate):
  if per_indiv_rate <= 0 or (per_gene_rate <= 0 and genes <= 0):
    return source

  for indiv in source:
    if per_indiv_rate >= 1.0 or random() < per_indiv_rate:
      new_indiv = indiv.clone()
      if genes > 0:
        indices = range(len(indiv))
        for _ in range(genes):
          i = indices.pop(randrange(len(indices)))
          if random() < positive_rate:
            new_indiv[i] += step_size
          else:
            new_indiv[i] -= step_size
      else:
        for i in range(len(indiv)):
          if random() < per_gene_rate:
            if random() < positive_rate:
              new_indiv[i] += step_size
            else:
              new_indiv[i] -= step_size
      yield new_indiv
    else:
      yield indiv
```

A.7.6 Mutate Gaussian

Creates a stream of individuals from a stream of real-valued individuals by adding random values selected from a Gaussian distribution to zero or more elements.

Real Mutate Gaussian signature.

```
mutate_gaussian(per_indiv_rate=1.0, per_gene_rate=1.0, genes=0,
                step_size=1.0, sigma=1.0)
```

Real Mutate Gaussian parameters.

Name	Default	Range	Description
`per_indiv_rate`	1.0	$[0.0, 1.0]$	The probability of each individual being varied. If not met, the individual is included in the result stream unmodified.
`per_gene_rate`	1.0	$[0.0, 1.0]$	The probability of each element being modified.
`genes`	0	$[0, \infty)$	The exact number of elements in each individual to be modified. If zero, `per_gene_rate` is used instead.
`step_size`	1.0	$(0.0, \infty)$	A scale factor to apply to each step.
`sigma`	1.0	$(0.0, \infty)$	The standard deviation of the distribution.

Real Mutate Gaussian implementation in Python.

```
from random import random, randrange, gauss

def mutate_gaussian(source, per_indiv_rate, per_gene_rate, genes,
                    step_size, sigma):
  if per_indiv_rate <= 0 or (per_gene_rate <= 0 and genes <= 0):
    return source

  for indiv in source:
    if per_indiv_rate >= 1.0 or random() < per_indiv_rate:
      new_indiv = indiv.clone()
      if genes > 0:
        indices = range(len(indiv))
        for _ in range(genes):
          i = indices.pop(randrange(len(indices)))
          new_indiv[i] += step_size * gauss(0, sigma)
      else:
        for i in range(len(indiv)):
          if random() < per_gene_rate:
            new_indiv[i] += step_size * gauss(0, sigma)
      yield new_indiv
    else:
      yield indiv
```

A.7.7 Crossover Average

Creates a stream of individuals from a stream of adjacent pairs of real-valued individuals using the mean values of matching elements.

Real Crossover Average signature.

```
crossover_average(per_pair_rate=1.0, per_gene_rate=1.0, genes=0)
```

Real Crossover Average parameters.

Name	Default	Range	Description
`per_pair_rate`	1.0	$[0.0, 1.0]$	The probability of each pair of individuals being combined. If not met, the first individual of the pair is included in the result stream unmodified.
`per_gene_rate`	1.0	$[0.0, 1.0]$	The probability of each element being modified. If not met, the value from the first individual of the pair is retained.
`genes`	0	$[0, \infty)$	The exact number of elements in each individual to be modified. If zero, `per_gene_rate` is used instead.

Real Crossover Average implementation in Python.

```
from random import random, randrange

def crossover_average(source, per_pair_rate, per_gene_rate, genes):
  if per_pair_rate <= 0 or (per_gene_rate <= 0 and genes <= 0):
    return source

  while True:
    parent = next(source)
    mate = next(source)
    if per_pair_rate >= 1.0 or random() < per_pair_rate:
      new_indiv = parent.clone()
      max_i = min(len(parent), len(mate))
      if genes > 0:
        indices = range(max_i)
        for _ in range(genes):
          i = indices.pop(randrange(len(indices)))
          new_indiv[i] = (parent[i] + mate[i]) * 0.5
      else:
        for i in range(max_i):
          if random() < per_gene_rate:
            new_indiv[i] = (parent[i] + mate[i]) * 0.5
      yield new_indiv
    else:
      yield parent
```

A.8 Integer-valued Operators

A.8.1 Representation

A fixed-length array of rank one containing integer values. The initial range of values is remembered to allow its use later as constraints. Implementations may choose to use a different internal representation depending on the parameters provided to the generator.

A.8.2 Random Integer Generator

Creates an infinite stream of integer-valued individuals that are initialised from a uniform random distribution.

Random Integer signature.

```
random_integer(length=0, shortest=10, longest=10, lowest=0, highest=100)
```

Random Integer parameters.

Name	Default	Range	Description
`length`	0	$[0, \infty)$	The number of elements in each individual. If zero, the range between `shortest` and `longest` is used instead.
`shortest`	10	$[1, \infty)$	The minimum number of elements in each individual.
`longest`	10	$[1, \infty)$	The maximum number of elements in each individual. If equal to or less than `shortest`, the value of `shortest` is used for every individual.
`lowest`	0	$(-\infty, \infty)$	The lowest possible value of each element.
`highest`	100	$(-\infty, \infty)$	The highest possible value of each element.

Random Integer implementation in Python.

```
from random import randrange

def random_integer(length, shortest, longest, lowest, highest):
  if length > 0:
    shortest = longest = length

  while True:
    if longest <= shortest:
      indiv_len = shortest
    else:
      indiv_len = randrange(shortest, longest+1)

    yield IntIndividual([randrange(lowest, highest+1) for _ in range(indiv_len)],
                        lowest, highest)
```

A.8.3 Integer Value, Low, Mid and High Generators

Creates an infinite stream of integer-valued individuals that are initialised at a specific value, the lowest value, the highest value or the midpoint of the range.

Integer Value, Low, Mid and High signatures.

```
integer_value(length=0, shortest=10, longest=10, lowest=0, highest=100, value=0)
integer_low(length=0, shortest=10, longest=10, lowest=0, highest=100)
integer_mid(length=0, shortest=10, longest=10, lowest=0, highest=100)
integer_high(length=0, shortest=10, longest=10, lowest=0, highest=100)
```

Integer Value, Low, Mid and High parameters.

Name	Default	Range	Description
`length`	0	$[0, \infty)$	The number of elements in each individual. If zero, the range between `shortest` and `longest` is used instead.
`shortest`	10	$[1, \infty)$	The minimum number of elements in each individual.
`longest`	10	$[1, \infty)$	The maximum number of elements in each individual. If equal to or less than `shortest`, the value of `shortest` is used for every individual.
`lowest`	0	$(-\infty, \infty)$	The lowest possible value of each element.
`highest`	100	$(-\infty, \infty)$	The highest possible value of each element.
`value`	0	$(-\infty, \infty)$	The value to initialise to (`integer_value` only).

Integer Value implementation in Python.

```
from random import randrange

def integer_value(length, shortest, longest, lowest, highest, value):
  if length > 0:
    shortest = longest = length

  while True:
    if longest <= shortest:
      indiv_len = shortest
    else:
      indiv_len = randrange(shortest, longest+1)

    yield IntIndividual([value] * indiv_len, lowest, highest)
```

Integer Low, Mid and High implementations in Python.

```
def integer_low(length, shortest, longest, lowest, highest):
  return integer_value(length, shortest, longest, lowest, highest, lowest)

def integer_mid(length, shortest, longest, lowest, highest):
  value = lowest + (highest - lowest) / 2
  return integer_value(length, shortest, longest, lowest, highest, value)

def integer_high(length, shortest, longest, lowest, highest):
  return integer_value(length, shortest, longest, lowest, highest, highest)
```

A.8.4 Clamp

Creates a stream of individuals from a stream of integer-valued individuals by changing values to be within either the specified range or the range used to create the individuals.

Integer Clamp signature.

```
clamp(lowest=0, highest=0)
```

Integer Clamp parameters.

Name	Default	Range	Description
`lowest`	0	$(-\infty, \infty)$	The lowest value of each element.
`highest`	0	$(-\infty, \infty)$	The highest value of each element. If less than or equal to `lowest`, the initialisation range is used.

Integer Clamp implementation in Python.

```
def clamp(source, lowest, highest):
  for indiv in source:
    if highest <= lowest:
      low, high = indiv.lowest, indiv.highest
    else:
      low, high = lowest, highest

    new_indiv = indiv.clone()
    changed_any = False
    for i in range(len(new_indiv)):
      v = new_indiv[i]
      if v < low:
        changed_any = True
        new_indiv[i] = low
      elif v > high:
        changed_any = True
        new_indiv[i] = high

    yield new_indiv if changed_any else indiv
```

A.8.5 Mutate Delta

Creates a stream of individuals from a stream of integer-valued individuals by adding a fixed-size step value to zero or more elements.

Integer Mutate Delta signature.

```
mutate_delta(per_indiv_rate=1.0, per_gene_rate=1.0, genes=0,
             step_size=1, positive_rate=0.5)
```

Integer Mutate Delta parameters.

Name	Default	Range	Description
`per_indiv_rate`	1.0	$[0.0, 1.0]$	The probability of each individual being varied. If not met, the individual is included in the result stream unmodified.
`per_gene_rate`	1.0	$[0.0, 1.0]$	The probability of each element being modified.
`genes`	0	$[0, \infty)$	The exact number of elements in each individual to be modified. If zero, `per_gene_rate` is used instead.
`step_size`	1	$[0, \infty)$	The value to add to or subtract from selected elements.
`positive_rate`	0.5	$[0.0, 1.0]$	The probability of adding `step_size` rather than subtracting it.

Integer Mutate Delta implementation in Python.

```
from random import random, randrange

def mutate_delta(source, per_indiv_rate, per_gene_rate, genes,
                 step_size, positive_rate):
  if per_indiv_rate <= 0 or (per_gene_rate <= 0 and genes <= 0):
    return source

  step_size = int(step_size)
  for indiv in source:
    if per_indiv_rate >= 1.0 or random() < per_indiv_rate:
      new_indiv = indiv.clone()
      if genes > 0:
        indices = range(len(indiv))
        for _ in range(genes):
          i = indices.pop(randrange(len(indices)))
          if random() < positive_rate:
            new_indiv[i] += step_size
          else:
            new_indiv[i] -= step_size
      else:
        for i in range(len(indiv)):
          if random() < per_gene_rate:
            if random() < positive_rate:
              new_indiv[i] += step_size
            else:
              new_indiv[i] -= step_size
      yield new_indiv
    else:
      yield indiv
```

A.8.6 Mutate Gaussian

Creates a stream of individuals from a stream of integer-valued individuals by adding random values selected from a Gaussian distribution to zero or more elements. Values are scaled and then truncated (rounded towards zero) before being added.

Integer Mutate Gaussian signature.

```
mutate_gaussian(per_indiv_rate=1.0, per_gene_rate=1.0, genes=0,
                step_size=1.0, sigma=1.0)
```

Integer Mutate Gaussian parameters.

Name	Default	Range	Description
`per_indiv_rate`	1.0	$[0.0, 1.0]$	The probability of each individual being varied. If not met, the individual is included in the result stream unmodified.
`per_gene_rate`	1.0	$[0.0, 1.0]$	The probability of each element being modified.
`genes`	0	$[0, \infty)$	The exact number of elements in each individual to be modified. If zero, `per_gene_rate` is used instead.
`step_size`	1.0	$(0.0, \infty)$	A scale factor to apply to each step.
`sigma`	1.0	$(0.0, \infty)$	The standard deviation of the distribution.

Integer Mutate Gaussian implementation in Python.

```
from random import random, randrange, gauss

def mutate_gaussian(source, per_indiv_rate, per_gene_rate, genes,
                    step_size, sigma):
  if per_indiv_rate <= 0 or (per_gene_rate <= 0 and genes <= 0):
    return source

  for indiv in source:
    if per_indiv_rate >= 1.0 or random() < per_indiv_rate:
      new_indiv = indiv.clone()
      if genes > 0:
        indices = range(len(indiv))
        for _ in range(genes):
          i = indices.pop(randrange(len(indices)))
          new_indiv[i] += int(step_size * gauss(0, sigma))
      else:
        for i in range(len(indiv)):
          if random() < per_gene_rate:
            new_indiv[i] += int(step_size * gauss(0, sigma))
      yield new_indiv
    else:
      yield indiv
```

A.8.7 Crossover Average

Creates a stream of individuals from a stream of adjacent pairs of integer-valued individuals using the mean values of matching elements. Mean values are truncated (rounded towards zero).

Integer Crossover Average signature.

```
crossover_average(per_pair_rate=1.0, per_gene_rate=1.0, genes=0)
```

Integer Crossover Average parameters.

Name	Default	Range	Description
`per_pair_rate`	1.0	$[0.0, 1.0]$	The probability of each pair of individuals being combined. If not met, the first individual of the pair is included in the result stream unmodified.
`per_gene_rate`	1.0	$[0.0, 1.0]$	The probability of each element being modified. If not met, the value from the first individual of the pair is retained.
`genes`	0	$[0, \infty)$	The exact number of elements in each individual to be modified. If zero, `per_gene_rate` is used instead.

Integer Crossover Average implementation in Python.

```
from random import random, randrange

def crossover_average(source, per_pair_rate, per_gene_rate, genes):
  if per_pair_rate <= 0 or (per_gene_rate <= 0 and genes <= 0):
    return source

  while True:
    parent = next(source)
    mate = next(source)
    if per_pair_rate >= 1.0 or random() < per_pair_rate:
      new_indiv = parent.clone()
      max_i = min(len(parent), len(mate))
      if genes > 0:
        indices = range(max_i)
        for _ in range(genes):
          i = indices.pop(randrange(len(indices)))
          new_indiv[i] = (parent[i] + mate[i]) / 2
      else:
        for i in range(max_i):
          if random() < per_gene_rate:
            new_indiv[i] = (parent[i] + mate[i]) / 2
      yield new_indiv
    else:
      yield parent
```

Appendix B

ESDL Grammar

The notation used for this grammar specification is a variant of Backus-Naur Form, simplified for human consumption rather than as input to a parser generator. Some production rules (such as EOS) are described informally despite having a formal construction, and some rules allow for expressions that are not described in the specification. The specification in Chapter 4 gives the baseline requirements for an ESDL implementation; extensions or restrictions are permitted (though restrictions should provide more value than ease of compiler implementation) but the grammar given here is only a guide and not a specification.

Items contained in double quotes are literal, case-insensitive text; all other names refer to other production rules. Suffixes of `?`, `*` and `+` indicate "zero or one," "zero or more" and "one or more" of the preceding item or parenthesised expression. The `<anything>` markers represent that any text, generally up to the end of the line, should be accepted. Whether this text is retained or discarded depends on the context.

Operator precedence and associativity rules are omitted for simplicity; the standard mathematical order of operations is intended. Many invalid constructs are apparently valid under the production rules; for example, `ABC.123` is a binary expression consisting of name `ABC`, number `123` and the dot operator. While such an operation may be valid, it is not mentioned in the specification, nor is it common to contemporary programming languages and could reasonably be considered a syntax error.

ESDL Grammar.

```
Name       : <matches regular expression [a-z_][a-z0-9_]*>
Number     : <matches regular expression [0-9]+(\.[0-9]*)?>
EOS        : <a new line, except when immediately after a backslash>

System     : Statement* (BlockStmt EOS)*

Statement  : RepeatStmt EOS
           | FromStmt EOS
           | JoinStmt EOS
           | YieldStmt EOS
           | EvalStmt EOS
           | PragmaStmt EOS
           | AssignStmt EOS
           | CallFunc EOS

BlockStmt  : "BEGIN"  Name       EOS Statement* "END" <anything>
RepeatStmt : "REPEAT" Expression EOS Statement* "END" <anything>

FromStmt   : "FROM" GroupOrGens "SELECT" SizedGroups ("USING" Operators)?
JoinStmt   : "JOIN" Groups      "INTO"   Groups      ("USING" Operators)?

YieldStmt  : "YIELD" Groups

EvalStmt   : "EVAL" Groups ("USING" Operators)?

PragmaStmt : "`" <anything> EOS

AssignStmt : Name "=" Expression

Expression : Operand (BinaryOp Operand)*
Operand    : Name
           | Number
           | CallFunc
           | "(" Expression ")"
           | UnaryOp Operand
           | "true" | "false"
           | "null" | "none"

UnaryOp    : "-"
BinaryOp   : "+" | "-" | "*" | "/" | "^" | "."

Parameter  : Name ("=" Expression)?

CallFunc   : Name "(" Parameter ("," Parameter)* ")"
           | Name "()"

Groups     : Name (',' Groups)*
SizedGroups: Expression? Name (',' SizedGroups)*

GroupOrGens: Name     ("," GroupOrGens)*
           | CallFunc ("," GroupOrGens)*

Operators  : Name     ("," Operators)*
           | CallFunc ("," Operators)*
```

Appendix C

`esdlc` Architecture

C.1 Overview

`esdlc` is a compiler for ESDL systems that produces code for multiple targets. It is written in Python and consists of three main components: the lexer, the parser and a collection of code generators. Figure C.1 shows the general flow used by `esdlc` when compiling an ESDL description. The source ESDL is provided as a file or a string to the lexer (Section C.2), which produces a stream of tokens. These tokens are passed to the parser (Section C.3), which develops a syntax tree and then a model based on the one described in Section 5.2 (page 99). This model is used by one or more code generators, called *emitters* (Section C.4), to produce compilable code or an executable program.

The full source code for `esdlc` is available online at `http://esdlc.googlecode.com/` and includes two emitters, one for `esec` (Section C.4.2) and a prototype for generating C++ AMP code (Appendix D).

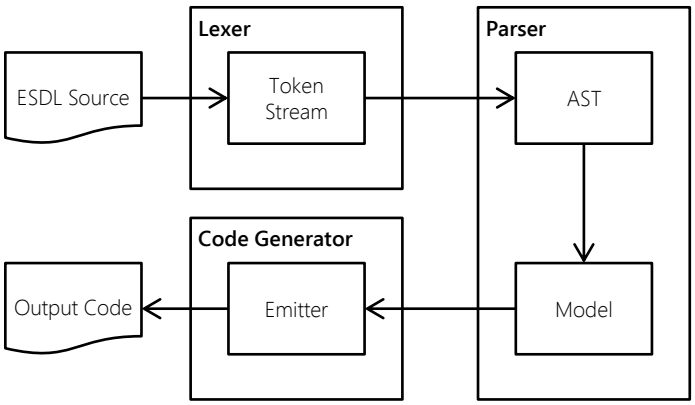

Figure C.1: The compilation flow of `esdlc`.

Table C.1: Regular Expressions used to identify ESDL tokens.

Token Type	Regex
comment	(#|;|//).*$
operator	\+ | - | * | \/ | \% | \^ | \.
assign	\=
comma	\,
name	(?!\d)\w+
number	(\d+\.\d*|\d+|\.\d+)(e[-+]?\d+)?
open	\(| \[| \{
close	\) | \] | \}
pragma	`.*$
skip_eos	\\\s*(($\|;\|//).*)?$

C.2 Lexer

The lexer in `esdlc` is based on an ordered list of regular expression match strings. Each of the regexes in Table C.1 is used to try and find a token at the current location. If a match is found, the matching text is appended to the token stream with its type and the current location is advanced. Tokens of type `name` have their text lowercased. If no regex matches, the position is advanced by one and the character is appended to an `error` token, which is appended to the token stream at the next successful match.

At the end of each line a special `eos` token is added, unless the preceding token is `skip_eos` in which case it is removed and no `eos` token is added. Each line is read separately from a file or a source string and location information (line and column number) is added to each token to allow informative error messages at later stages. The lexer never produces an error; instead, it creates `error` tokens that result in errors later.

Extending the lexer is generally not necessary, except where a language extension is being created. Where a new type of token is required, inserting a new regex into the list of tokens is all that is required. Tokens of that type will then appear in the stream handled by the parser.

C.3 Parser

The model produced by the parser is an instance of the `esdlc.model.System` class, typically one of the subclasses `FluentSystem` or `AstSystem`. `FluentSystem` is an object construction builder [35] that allows a system to be described using Python code, while `AstSystem` takes a token stream from the lexer. `AstSystem` is the more commonly used class, though `FluentSystem` is convenient for testing code generation independently from lexing and parsing source.

C.3.1 `System` class

`System` is the base class representing an ESDL system. It contains a dictionary `blocks` that maps block names to lists of statements. The `block_names` attribute contains the names of all blocks in the order they were originally specified. `variables` and `externals` contain mappings from names to objects representing variables; functions and operators, as in Python, are treated as variables that may be called.

C.3.2 `FluentSystem` class

The `FluentSystem` class is used to define a system with an ESDL-like syntax based on chaining function calls. Listing C.1 shows an example system that has the equivalent ESDL statements included as comments. Providing a `definitions` function is required to specify blocks, variables and groups, since `FluentSystem` does not automatically infer them from use. Other blocks are provided as functions called `block_<`*name*`>`, and specified with the `Block()` function.

Listing C.1: Defining a `FluentSystem` in Python.

```
class GAwithTournament(FluentSystem):
  def definitions(self):
    self.External("random_binary")
    self.External("onemax_evaluator")
    self.External("tournament")
    self.External("mutate_random")
    self.External("best")

    self.Group("population")
    self.Group("parents")
    self.Group("offspring")

    self.Variable("size")

    self.Block("generation")

  def block_init(self):
    # size = 100
    self.Assign("size", 100.0)

    # FROM random_binary(length=10) SELECT size group
    self.From(self.Generator("random_binary", length=10.0)) \
        .Select(self.Group("population", "size"))

    # EVAL population USING onemax_evaluator
    self.Eval("population", "onemax_evaluator")
    # YIELD population
    self.Yield("population")

  def block_generation(self):
    # FROM population SELECT size parents USING tournament(k=2)
    self.From("population") \
        .Select(self.Group("parents", limit="size")) \
        .Using(self.Function("tournament", k=2.0))
    # FROM parents SELECT offspring USING mutate_random(per_gene_rate=0.1)
```

```
        self.From("parents") \
            .Select("offspring") \
            .Using(self.Function("mutate_random", per_gene_rate=0.1))

        # FROM population, offspring SELECT size population USING best
        self.From("population", "offspring") \
            .Select(self.Group("population", limit="size")) \
            .Using("best")

        # YIELD population
        self.Yield("population")
```

The primary use of `FluentSystem` is to simplify testing of `System`; it is not intended to be the usual way to define systems. Most of the provided functions—`From`, `Join`, `Yield`, `Eval`, `Variable`, `Group` and `Function`—are trivial and simply construct an object with the parameters and append it to the relevant list. `FluentSystem` does not use the lexer, though it may be used with emitters in the same way as `AstSystem`.

C.3.3 `AstSystem` class

Creating a `System` instance from ESDL source involves two classes: `AST` and `AstSystem`. `AST` uses a token stream from the lexer to construct the syntax tree that `AstSystem` parses to instantiate the model. Separating the two steps allows users of `esdlc` to intercept the syntax tree and make modifications before converting it to a model; syntax trees can usually be converted back to the original source, while a `System` may have lost enough information to make this impossible.

`AST` is instantiated with a `TokenReader` object to simplify iteration and look-ahead on the list of tokens. The central function is `AST.parse_statement()`, which follows the structure shown in Figure C.2 based on the grammar given in Appendix B. The parse functions called from `parse_statement()` use shared functions for the `Groups`, `SizedGroups`, `GroupsOrGens` and `Operators` elements from the grammar, all of which perform basic validation according to the permissible child nodes. For example, the `parse_groups()` method will produce errors if group sizes or function calls are specified. The `parse_groups_or_generators()` and `parse_operators()` functions handle identical syntax, but the latter produces `CallFunc` nodes for all elements rather than only those with a parameter list.

Expressions are parsed using two functions: `parse_expression()` and `parse_operand()`, in keeping with the grammar. `parse_operand()` handles operator-less expressions and converts parenthesised expressions by calling `parse_expression()`. Since `parse_expression()` uses `parse_operand()`, there is mutual recursion into sub-expressions. However, the implementation of `parse_expression()` does not recurse for operators, as a typical recursive-descent parser would. Instead, operator and operand nodes are stored in a list and each binary operator is reduced in successive passes. This allows operator precedence to be clearly specified and reduce recursion,

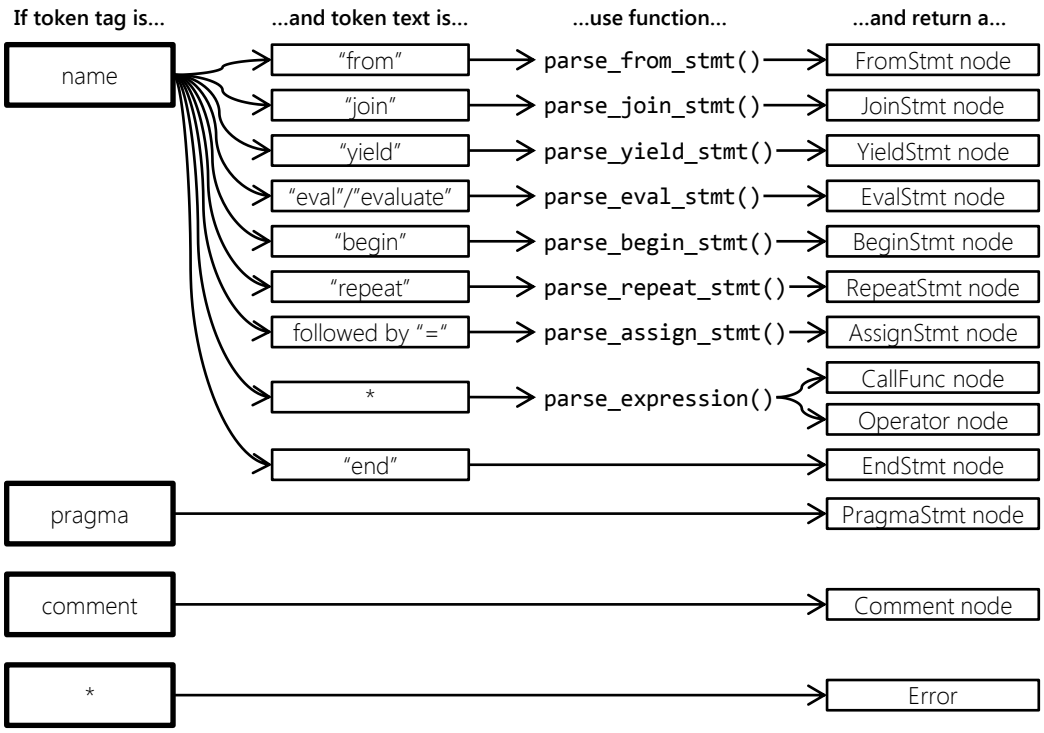

Figure C.2: Top-level `esdlc` parser structure.

though since most expressions in ESDL are unlikely to be highly complex, this is not a significant optimisation.

An example of the resulting syntax tree was shown in Listing 5.9 (page 113) and is reproduced here as Listing C.2.

`AstSystem` uses the syntax tree to fill the inherited members of `System`: `blocks`, `block_names`, `variables` and `_errors` (accessible through `Validator`'s `errors` and `warnings` properties). In effect, a similar parsing process is used to that in `AST`, though in this case the output is an executable model—each element includes an `execute()` function that provides the functionality. Directly executing the model is very inefficient compared to generating compilable code, but it is suitable for testing and validation purposes.

C.3.4 `Validator` class

The `Validator` class is used to produce error messages and warnings for invalid or potentially invalid constructs. When using `AstSystem`, many warnings are impossible to generate without causing an error at an earlier stage; however, since other `System` subclasses may not have similar safeguards, the checks are retained. Examples of issues that are detected include unused and uninitialised variables, naming collisions between groups, variables and blocks, missing operands in expressions, incorrect or unspecified group sizes and restricted or invalid variable names.

```
BeginStmt{ initialisation, [
  PragmaStmt{`include "Evaluator.h"},
  ={ lowest, -{ 100 } }
  ={ highest, 100 }
  FromStmt{ [random_real[={ length, 8 }, lowest, highest]],
    SelectStmt{ [population{ n }] }
  },
  EvalStmt{ [population],
    UsingStmt{ [evaluator] }
  },
  YieldStmt{ [population] }
]}

BeginStmt{ generation_equivalent, [
  RepeatStmt{ n, [
    FromStmt{ [population],
      SelectStmt{ [parents{ 2 }, rest] },
      UsingStmt{ [fitness_proportional[no_replacement] ] }
    },
    FromStmt{ [parents],
      SelectStmt{ [offspring] },
      UsingStmt{ [crossover, mutate[={ per_gene_rate, 0.1 }] ] }
    },
    FromStmt{ [offspring, rest],
      SelectStmt{ [population] }
    },
  ]},
  YieldStmt{ [population] }
]}
```

Listing C.2: Example syntax tree generated by the `AST` class.

```
def emit(model, out=sys.stdout, optimise_level=0, profile=False):
  # Generate output and write to out

  return code_string, context_dict
```

Listing C.3: `emit` function signature.

`Validator` is structured similarly to an emitter (Section C.4) in its traversal of the model, though instead of producing code it extends the list of errors associated with the `System`. These error objects are exposed through `Validator.errors` and `Validator.warnings`, which filter based on severity. In general, a system with errors should not be passed to the emitter, while one with warnings may.

C.4 Code Generation

C.4.1 Emitters

Emitters provide code generation by converting an instance of `System` into another form. This form is not necessarily executable code, but may be an image, diagram or other format that takes advantage of the semantic analysis. Basic syntax colouring, for example, does not require compilation of the source ESDL, and would only require an emitter if different colours were to be used for groups, variables and operators.

Emitters are implemented as Python functions with the signature shown in Listing C.3. The `out` parameter is used to write the output, which may also be returned as a string in `code_string` if appropriate. Where the output is not a string—an image, for example—it should be written to `out` and `None` returned in place of `code_string`. For the `optimise_level` and `profile` parameters, the actual implementation will vary depending on the target. `optimise_level` is intended to be zero for no optimisations with the level of optimisation increasing with larger values. `profile` is set to `True` to include hooks to allow the generated code to have its execution time measured, rather than profiling the performance of the emitter itself.

`context_dict` is an optional Python dictionary that includes extra definitions that may be required for actually using the code. If a `__compile()` function is returned in `context_dict` it is called with the original output path. For example, the `esec` emitter adds functions for merging, joining and partitioning groups, which `esec` includes in the execution context, while the C++ AMP emitter provides a `__compile()` function that invokes a C++ compiler if one is available.

New emitters may be added by creating a new module in the `esdlc.emitters` package; emitters are loaded at run-time based on the command line arguments to

```
# Original ESDL:
#   BEGIN generation
#     REPEAT 100
#       FROM a SELECT b USING c
#     END
#     YIELD b
#   END

# Generated code:
def _block_generation():
  for _ in xrange(100):
    b = _group(c(_source=_merge(a)))
  _yield("b", b)
```

Listing C.4: Python code generated for named and repeated blocks.

the `esdlc.py` script. The emitter for `esec` is discussed in the following section, while the C++ AMP emitter is described in Appendix D.

C.4.2 esec emitter

`esdlc` is embedded directly in `esec` and invoked automatically. Code generation for `esec` may also be achieved by passing `/e:esec` on the command line to `esdlc.py`.

The `esdlc.emitters.esec.emit()` function uses a private class `_emitter` to simplify handling state and output while generating the code. `_emitter` contains helper methods for appending code to the current output line with correct indentation. Each element type has a separate function on `_emitter` for generating the executable code.

`_emit_block()` is the main emitter function. It is called once for each named block to generate a function containing all the statements to execute in that block. Functions are named using the block name (or `_init` for the initialisation block) prefixed with `_block_`; for example, a `GENERATION` block would have a function `_block_generation()`. Within the block, each statement is passed to `_emit()`, which then invokes a more specific function. `REPEAT` blocks are written by `_emit_repeat()`, which creates a for-loop for the number of iterations specified. As with named blocks, each statement is passed to `_emit()`. Listing C.4 shows the code that is generated for named and repeat blocks.

Store operations are the most involved element, typically resulting in three or more lines of generated Python code. When `optimise_level` is zero, `_emit_store()` is called from `_emit()`; otherwise, the `_emit_store_optimised()` function is used, which generates less code though it is unlikely to be significantly faster. In both cases, the operator chain is created by invoking (calling or constructing, depending upon the type) each operator in turn, passing the previous as the `_source` parameter. The first operator is either `_merge` (for `FROM-SELECT`) or `_join` (for `JOIN-INTO`), both of which

```
# Original ESDL:
#   FROM a, b SELECT 100 c USING crossover, mutation
# Model elements:
#   Store([a, b], [(100, c)], [_merge, crossover, mutation])

# Unoptimised code:
_gen = _merge(a, b)
_gen = crossover(_source=_gen)
_gen = mutation(_source=_gen)
c = _group(_part(_gen, 100))

# Optimised code:
c = _group(_part(mutation(_source=crossover(_source=_merge(a, b))), 100))
```

Listing C.5: Python code generated for unoptimised and optimised stores.

take the source groups as a parameter list. At the end of the chain, calls to `_group()` and, if required, `_part()`, produce the group instances that are assigned to group variables. Listing C.5 shows examples of the code generated for unoptimised and optimised versions of the same `FROM-SELECT` statement. `JOIN-INTO` statements differ only by using `_join` in place of `_merge`.

Function calls are handled by the `_emit_function()` function, which delegates based on the function type. Assignments, indicated by a call to the pseudo-function `_assign`, are passed to the `_emit_assign()` function. Since Python has dynamic typing that matches the ESDL model, code generation is straightforward—apart from renaming accidental collisions with Python keywords, the assignment is written as in ESDL.

Attribute accesses ("`a.b`") and indexers ("`a[i]`") are parsed as `_getattrib` and `_getindex` pseudo-functions; the `_emit_getattribute()` and `_emit_getindex()` functions produce the equivalent Python code directly. External function calls are written almost identically to the ESDL definitions: Python supports named parameters and default values that make most function calls interchangeable with ESDL. The exceptions are parameter names that conflict with Python keywords but not ESDL keywords, such as `lambda` or `class`. Errors are avoided by appending underscores to these parameters—`lambda_` and `class_`—which is the Python convention for using keywords as names. Variables and groups are similarly protected.

Yields are implemented by invoking a `_yield()` function, which is typically provided by the `esec` framework. Both the group and a string with the group's name are passed to this function. The default implementation performs all analysis immediately before returning.

`EVAL` statements create an instance of the specified evaluator and assign it to each individual. Any cached fitness value is invalidated and no evaluation takes

```
# Original ESDL:
#   FROM a SELECT b USING unique
#   `py if len(b) > 100:
#   `py   print("Over 100 unique individuals")

# Generated code:
b = _group(unique(_source=_merge(a)))
if len(b) > 100:
  print("Over 100 unique individuals")
```

Listing C.6: Example of expanding a `py pragma into Python code.

place until it is next accessed. When no evaluator is specified, the individual's _eval member is set to None—on its next evaluation it is reset to the individual's _eval_default member, which contains the default evaluator for that species.[1]

Only one pragma is recognised by this emitter: lines beginning with `py are assumed to be literal Python code to include in the generated code. The command is removed and the remaining text is then written at the current indent level. One space exists between the command and the Python code, which is removed, but other whitespace is left untouched to allow blocks to be specified. Listing C.6 shows an example of a `py pragma that prints a message when a group contains more than a certain number of unique individuals.

The esec emitter includes support for profiling hooks. If the profiling parameter to emit() is True, an object _profiler with start and stop methods is assumed. Calls to this object are emitted around each ESDL statement with the full statement text, as shown in Listing C.7. The profiler object needs to be provided by the user—an example class (esdlc.emitters.esec.Profiler) exists, but is not used by default and does not record event timings.

Listing C.8 shows an example ESDL system that contains most of the available constructs. Listings C.9 and C.10 show the code generated by the esec emitter without and with optimisation, respectively.

[1] "Species" do not exist in ESDL; however, esec continues to include them. They have no impact on the code that is generated.

```
# Original ESDL:
#   FROM a, b SELECT 100 c USING crossover, mutation

# Example profiler implementation
class Profiler:
  def __init__(self):
    self.data = []

  def start(self, statement):
    self.data.append(None)
    self.data[-1] = (statement, 'Start')

  def end(self, statement):
    self.data.append((statement, 'End'))
_profiler = Profiler()

# Generated code:
_profiler.start("FROM a, b SELECT 100 c USING crossover, mutation")
_gen = _merge(a, b)
_gen = crossover(_source=_gen)
_gen = mutation(_source=_gen)
c = _group(_part(_gen, 100))
_profiler.stop("FROM a, b SELECT 100 c USING crossover, mutation")
```

Listing C.7: Python code generated with profiling enabled.

```
FROM random_binary(length=config.length.max) SELECT 100 population
t = 0
delta_t = -0.1
lambda = 100
EVAL population USING evaluators.population(t)
YIELD population

BEGIN generation
    t = t + (delta_t * 1.4)
    `py print(t)
    `cpp printf("%f\n", t);

    REPEAT 10
        FROM population SELECT 100 parents USING tournament(k=2, greediness=0.7)

        FROM parents SELECT mutated USING mutate_delta(stepsize)
        FROM parents SELECT crossed USING uniform_crossover
        EVAL mutated, crossed USING evaluator(t=t, lambda)

        JOIN mutated, crossed INTO merged USING tuples
        FROM merged SELECT offspring USING best_of_tuple

        FROM population, offspring SELECT 99 population, rest USING best
        FROM rest SELECT 1 extras USING uniform_random
        FROM population, rest, extras SELECT (((100))) population
    END REPEAT

    EVAL population USING evaluators.config(t)
    YIELD population
END generation
```

Listing C.8: Example ESDL system that includes most compilable constructs.

Listing C.9: Unoptimised Python code generated by `esdlc` for Listing C.8.

```
_global = globals()
def _block__init():
    # FROM random_binary(length=config.length.max) SELECT (100) population
    _gen = _merge(random_binary(length=config.length.max))
    _global["population"] = _group(_part(_gen, 100))

    # t = 0.0
    _global["t"] = 0.0

    # delta_t = (-0.1)
    _global["delta_t"] = (-0.1)

    # lambda = 100.0
    _global["lambda"] = 100.0

    # EVAL population USING evaluators.population(t)
    _eval = _evaluator(evaluators.population(t=t))
    for _indiv in _merge(population):
        _indiv._eval = _eval
        del _indiv.fitness

    # YIELD population
    _yield("population", population)


def _block_generation():
    # t = (t+(delta_t*1.4))
    _global["t"] = (t+(delta_t*1.4))

    # `py print(t)
    print(t)

    # `cpp printf("%f\n", t);

    # REPEAT 10.0
    for _ in _range(10.0):
        # FROM population SELECT (100) parents USING tournament(k=2.0, greediness=0.7)
        _gen = _merge(population)
        _gen = tournament(k=2.0, greediness=0.7, _source=_gen)
        _global["parents"] = _group(_part(_gen, 100))

        # FROM parents SELECT mutated USING mutate_delta(stepsize)
        _gen = _merge(parents)
        _gen = mutate_delta(stepsize=True, _source=_gen)
        _global["mutated"] = _group(_gen)

        # FROM parents SELECT crossed USING uniform_crossover()
        _gen = _merge(parents)
        _gen = uniform_crossover(_source=_gen)
        _global["crossed"] = _group(_gen)

        # EVAL mutated, crossed USING evaluator(t=t, lambda)
        _eval = _evaluator(evaluator(t=t, lambda_=_global["lambda"]))
        for _indiv in _merge(mutated, crossed):
            _indiv._eval = _eval
            del _indiv.fitness
```

```
        # JOIN mutated, crossed INTO merged USING tuples()
        _gen = _join(mutated, crossed)
        _gen = tuples(_source=_gen)
        _global["merged"] = _group(_gen)

        # FROM merged SELECT offspring USING best_of_tuple()
        _gen = _merge(merged)
        _gen = best_of_tuple(_source=_gen)
        _global["offspring"] = _group(_gen)

        # FROM population, offspring SELECT (99) population, rest USING best()
        _gen = _merge(population, offspring)
        _gen = best(_source=_gen)
        _global["population"] = _group(_part(_gen, 99))
        _global["rest"] = _group(_gen)

        # FROM rest SELECT (1) extras USING uniform_random()
        _gen = _merge(rest)
        _gen = uniform_random(_source=_gen)
        _global["extras"] = _group(_part(_gen, 1))

        # FROM population, rest, extras SELECT (100) population
        _gen = _merge(population, rest, extras)
        _global["population"] = _group(_part(_gen, 100))


    # EVAL population USING evaluators.config(t)
    _eval = _evaluator(evaluators.config(t=t))
    for _indiv in _merge(population):
        _indiv._eval = _eval
        del _indiv.fitness

    # YIELD population
    _yield("population", population)


_block__init()
```

Listing C.10: Optimised Python code generated by `esdlc` for Listing C.8.

```
_global = globals()
def _block__init():
    _global["population"] = _group(_part(_merge(random_binary(length=config.length.max)), »
    «100.0))
    _global["t"] = 0.0
    _global["delta_t"] = (-0.1)
    _global["lambda"] = 100.0
    _eval = _evaluator(evaluators.population(t=t))
    for _indiv in _merge(population):
        _indiv._eval = _eval
        del _indiv.fitness
    _yield("population", population)

def _block_generation():
    _global["t"] = (t+(delta_t*1.4))
    print(t)
    for _ in _range(10.0):
        _global["parents"] = _group(_part(tournament(k=2.0, greediness=0.7, _source=_merge(p»
    «opulation)), 100.0))
        _global["mutated"] = _group(mutate_delta(stepsize=True, _source=_merge(parents)))
        _global["crossed"] = _group(uniform_crossover(_source=_merge(parents)))
        _eval = _evaluator(evaluator(t=t, lambda_=_global["lambda"]))
        for _indiv in _merge(mutated, crossed):
            _indiv._eval = _eval
            del _indiv.fitness
        _global["merged"] = _group(tuples(_source=_join(mutated, crossed)))
        _global["offspring"] = _group(best_of_tuple(_source=_merge(merged)))
        _gen = best(_source=_merge(population, offspring))
        _global["population"] = _group(_part(_gen, 99.0))
        _global["rest"] = _group(_gen)
        _global["extras"] = _group(_part(uniform_random(_source=_merge(rest)), 1.0))
        _global["population"] = _group(_part(_merge(population, rest, extras), 100.0))
    _eval = _evaluator(evaluators.config(t=t))
    for _indiv in _merge(population):
        _indiv._eval = _eval
        del _indiv.fitness
    _yield("population", population)

_block__init()
```

C.5 Summary

`esdlc` is an extensible compiler for ESDL that produces a model usable for generating code for a variety of targets. The current implementation generates Python code for use with `esec`, and also supports generating C++ AMP code, as discussed in Appendix D. A lexer and parser convert ESDL code to a model that is validated and used for platform-specific code generation. `esdlc` can be partially or completely embedded in any program supporting Python evaluation, allowing ESDL compilation to be provided to users directly.

The full source code for `esdlc` is available online at `http://esdlc.googlecode.com/`.

Appendix D

Parallel Execution

D.1 Background

Beyond the need for rapid development, proving the capabilities of many algorithms requires significant computing power in order to complete a large number of experiments within a reasonable timeframe. For many years, Moore's Law[1] allowed CPUs to increase their operating speed on a regular basis, reducing the need for programmers to focus on optimisation. However, the effect of increasing transistor count on processor speed stalled, resulting in speeds levelling out during the mid-2000s. To continue the increases in processing speed, CPU developers increased throughput by including multiple processors on single chips—dual-core, quad-core and more—as well as combining different types of processors and connecting physically separate hardware. However, in order to utilise the full processing power of multicore, heterogeneous and distributed ("cloud") processing, new programming styles have become necessary. [90, 91]

Multithreading and parallel processing are not new concepts, having been investigated for decades and used on the earliest supercomputers, but with the general availability of truly parallel machines the need for these techniques is now mainstream. The two general approaches are broadly recognised as task-parallel and data-parallel. Tasks are independent sections of code that occasionally synchronise but generally have separate responsibilities. An algorithm can be converted to a task-parallel form by identifying subsequences of steps that have few dependencies on each other.

Data-parallel algorithms, by contrast, run exactly the same sequence of steps on separate portions of a large data set. In SIMD (Single-Instruction Multiple-Data) machines, these steps are locked, such that a group of processors perform the same instruction simultaneously using separate data; MIMD (Multiple-Instruction

[1] The prediction, made by Intel co-founder Gordon Moore, that the number of transistors that can be placed on an integrated circuit doubles every two years.

Multiple-Data) machines are not locked. SIMD machines have advantages in price, size and power consumption over MIMD, since the electronics required for instruction fetch and decode are shared, but generally cannot be used for task-parallelism. The cost to the developer is that data-parallel algorithms are significantly different from single-threaded algorithms and usually require redesign and reimplementation. Further, in order to observe any speed improvement over a single-threaded algorithm, the amount of data usually needs to be large ($N > 10^5$) [44].

Modern GPU cards are massively parallel SIMD machines, typically allowing hundreds of threads to be run in parallel. By comparison, current high-end desktop CPUs are capable of a degree of parallelism up to thirty-two through mixed MIMD and SIMD components. The difference between the two is the complexity of operations that may be performed on each element: CPUs handle extended, non-linear sequences of complicated instructions on linear sequences of data well; GPUs are best for simple, typically arithmetic and rarely branching operations with large arrays of data. While both GPUs and CPUs are theoretically capable of performing any task, selecting the right processor to use is necessary to achieve optimal performance.

EAs have a significant history of implementation on parallel hardware, typically using one of a few approaches. The earliest was the use of island populations—"demes"—which are separate instances of the algorithm that exchange individuals or use breeding neighbourhoods [13]. Since the entire algorithm is replicated, the ideal hardware configuration is one CPU for each instance. However, the maximum speedup obtainable from island populations is limited, and the nature of the algorithm changes such that any previous analysis of its behaviour or suitable problems may become irrelevant.

For algorithms with expensive evaluation routines, copying the entire population to separate parallel hardware just for evaluation may provide an overall performance improvement. This approach is sometimes known as "master-slave" parallelisation [13]. Parallel evaluation is an attractive prospect, since fitnesses are normally calculated independently for each individual. However, the cost of transferring a population between sequential and parallel processors, which rarely use a unified address space, may outweigh the benefits [45, 62].

Parallelising the parts of an EA that are independent, or nearly independent, for each individual is another option. In typical algorithms, mutation is independent and recombination involves only two individuals; parallelising these as well as evaluation moves more work to the parallel hardware, reducing the proportion of transfer overheads [13]. This is still considered master-slave parallelisation, though since recombination and mutation are normally very simple operations, there is little benefit in moving them to specialised hardware.

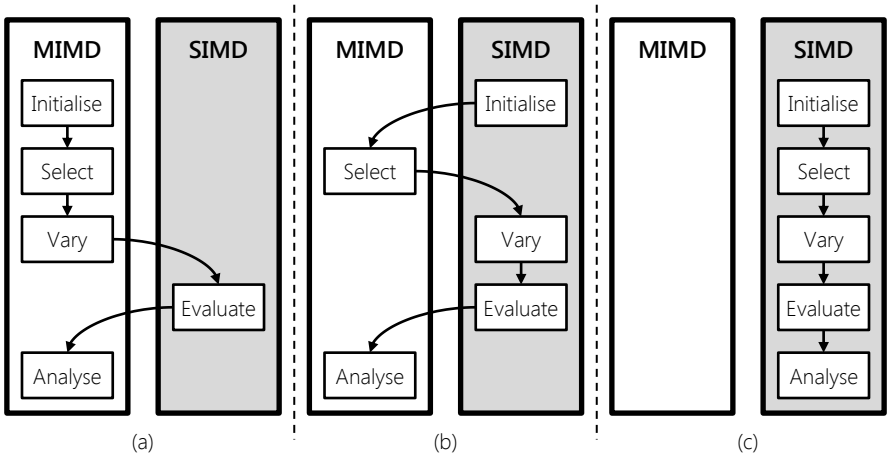

Figure D.1: Potential arrangements for an EA on heterogeneous hardware.

The final approach is to execute all aspects of the algorithm on parallel hardware, thereby removing all overheads relating to sharing data. However, algorithms that require random access to data are difficult to parallelise, the most important here being sort algorithms. The time taken to sort a population by fitness on a GPU compared to a CPU can easily eliminate the benefits of having avoided copying; as a mitigation, many implementations use tournament style selection that does not require full sorting [22, 74].

Figure D.1 shows three general approaches to parallelising EAs (excluding island populations) with MIMD and SIMD processors under a non-unified memory architecture (for example, a CPU and a GPU). (a) runs evaluations in parallel, copying the entire population to and from the SIMD processor as required. (b) performs initialisation, variation and evaluation, all inherently parallel operations, on the SIMD processor and uses the MIMD processor for sort operations. Finally, (c) runs the entire algorithm on the MIMD processor, avoiding the overhead of copying completely.

The difference in performance between (b) and (c) depends largely on the efficiency of the sort algorithm used. With a performant way to order the individuals in a group, selection and analysis can show equivalent or better performance compared to copying between devices. Sort algorithms for sequential architectures, particularly those supporting recursion, are mature and very efficient, while those for parallel architectures are not as well established. For this implementation, the arrangement of Figure D.1c is used with the sort algorithm described in [27].

D.2 C++ AMP

C++ AMP (C++ Accelerated Massive Parallelism) is a library and a language extension to C++ that supports algorithm development on heterogeneous platforms such as GPUs. The library provides a common interface for allocating and trans-

```
#include <amp.h>
using namespace concurrency;

void add_arrays(int N, float* A, float* B, float* C) {
    array_view<float, 1> vA(N, A), vB(N, B), vC(N, C);

    parallel_for_each(vC.extent, [=](index<1> i) restrict(amp) {
        vC[i] = vA[i] + vB[i];
    });
}
```

Listing D.1: Array addition in C++ AMP.

ferring memory between platforms, iterating over multi-dimensional collections of data, and distributing execution across heterogeneous hardware. A single language extension—the `restrict` modifier—allows code for other platforms to intermingle with CPU code, rather than requiring separate files and syntax. [15, 44]

C++ is used to write the code to execute in parallel, though due to the limited instruction sets on SIMD processors only a subset of the language is available. Notable limitations are the lack of virtual methods and hence useful polymorphism, variable references and interaction between code on different platforms. All functions used in a kernel must be marked with suitable `restrict` qualifiers and be able to be compiled inline. Polymorphic class hierarchies are not possible, but templates and generic programming can be used to provide compile-time type dispatch. Simple user-defined types can be used as array elements and accessed within kernels using normal C++ syntax, including constructors and overloaded operators.

C++ AMP does not automatically provide superior performance to alternative interfaces such as CUDA or OpenCL; the value proposition is developer productivity.[2] C++ AMP allows the developer to design for their data, rather than the hardware; iteration is the central concept, whereas CUDA and OpenCL place threads centrally. This abstraction allows a C++ developer to implement an algorithm using C++ AMP quickly and address optimisations for thread and memory layout later if necessary. In contrast, a CUDA or OpenCL developer must deal with these complexities up-front in order to create an executable algorithm.

Listing D.1 shows a trivial example of adding two arrays using C++ AMP. The use of lambda functions[3] is a convenience but not mandatory. Apart from the `restrict(amp)` modifier, there are no modifications to the C++ language; `array_view`, `index` and `parallel_for_each` are defined in the `amp.h` header file.

[2] By analogy, C++ does not provide better performance than assembly language, though the productivity benefits are so widely accepted as to be unworthy of comment.

[3] Lambda functions were introduced into C++ with the new standard in 2011 and are already supported by the major compilers.

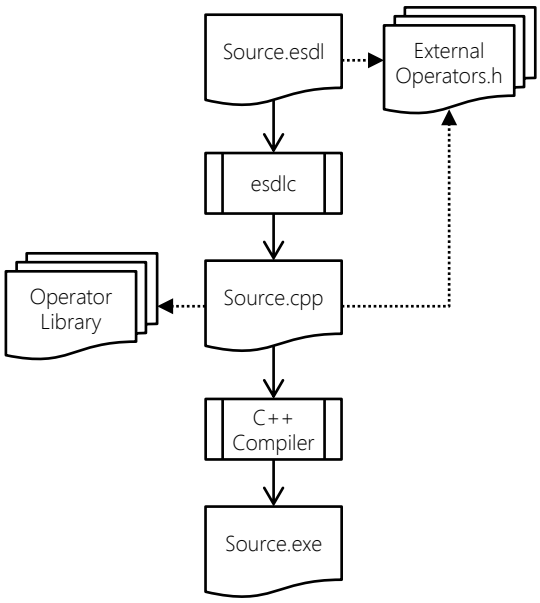

Figure D.2: Compilation workflow of the `cppamp` emitter.

C++ AMP has one implementation based on DirectCompute, which limits its use to computers running Microsoft Windows with recent hardware, though other compiler developers are working on implementations for their own platforms. The implementation is freely available as part of Microsoft Visual Studio 11 Beta.[4]

D.3 Execution Model

`esdlc` is a compiler for ESDL written in Python but supporting code generation for multiple targets. One supported target is C++ AMP: `esdlc` can produce an executable that runs the algorithm entirely using a DirectCompute-capable GPU. When invoking `esdlc` directly, the `/e:cppamp` command-line option uses the `cppamp` emitter that generates C++ AMP output.

A `.esdl` source file is used to provide the system definition, while external operators may be included using `` `include `` pragmas. `esdlc` creates a `.cpp` file with references to the EA library implemented for C++ AMP, which is based on that described in Appendix A. This `.cpp` file is compiled with a supporting compiler to produce the final executable. Section D.3.4 describes the command-line options for this program. Figure D.2 shows the compilation process graphically.

The ESDL algorithms that may be compiled to C++ AMP are restricted in order to achieve high execution performance. Features such as heterogeneous groups and non-linear individual representations cannot be implemented for parallel platforms without severely affecting performance. Some parameters need to be provided as compile-time constants. For example, the number of elements in an individual is an aspect of the group type (a template parameter) and so cannot be modified after the

[4]Details and downloads are available at `http://go.microsoft.com/fwlink/?LinkId=190957`

```
auto x = 1;
// identical to: int x = 1;

auto name = L"ESDL";
// identical to: const wchar_t name[5] = {'E', 'S', 'D', 'L', 0};

auto z = 1.0f * x;
// identical to: float z = 1.0f * x;
```

Listing D.2: Examples of the C++ `auto` type declaration.

C++ compilation step. Neither of these restrictions is defined as part of ESDL—they are features provided by `esec` but not the C++ AMP code generator. Variable-length groups, immutable individuals and reassignable evaluators are supported.

Individual representations are limited to arrays of 32-bit integers or floating-point values.[5] While other representations are possible, C++ AMP is optimised for arrays of these primitive types; representations that are more complex are unlikely to see any performance improvement over a single-threaded CPU implementation.

D.3.1 Memory Model

The blackboard in generated C++ AMP code is the scope of the `main` function: groups and variables are stored against either the names used in the original ESDL or 'mangled' versions of the names. C++ is statically typed, and so values must be converted to the type of the destination variable or else stored in a different variable. Operators and functions have no access to members of the outer scope.

ESDL variables are stored in C++ locals that are typed using the `auto` qualifier, which infers the type from the value being assigned. Listing D.2 shows some examples of using `auto` and the equivalent type declarations.[6] Here, `auto` saves `esdlc` from having to infer the type of the expression, though integer constants are converted to `float` to avoid later calculations being truncated. If the destination of an assignment already exists, the `decltype` operator is used to cast the new value to the type of the variable. This may result in errors when compiling the C++ code if the type of a variable changes, although basic numeric types should always work correctly.

Groups are represented by instances of the `esdlc::group` class, which is templated over the individual type. Listing D.3 shows the interface of group objects. The individuals are always stored on the GPU and referred to using the `concurrency::array`

[5] 64-bit floating-point values are also available on some GPU hardware and may be used to increase precision at the expense of performance and memory consumption. The fact that most EAs are inherently stochastic suggests that 32-bit is sufficient and the rounding errors (at approximately the seventh most significant decimal digit) can be ignored.

[6] As a very recent addition to the C++ standard, it is worth noting that `auto` provides no functionality beyond inferring the type of the expression being used to initialise the variable. After declaration, the variable is identical to one declared explicitly.

type. Groups can be manually copied to the CPU using the `as_list` and `as_vector` methods, but otherwise no transfers from the GPU occur. Instances of `shared_ptr` from the C++ standard library are used to avoid unnecessary copying of data on the GPU (the default behaviour for `concurrency::array`), allowing instances of `group` to be passed and returned to and from operators freely. A set of `make_group()` functions are provided to create new groups, either based on a size value or by cloning an existing group.

Since individuals are embedded in the group type, they may take any type provided it contains only the C++ AMP supported primitive types: `int`, `unsigned int` and `float`. Listing D.4 shows the basic set of members for fixed-length and variable-length individuals. The `genome` and `fitness` members are required to allow operators to access the relevant data, though since individuals are only ever accessed through a group, there is no need to derive from these structures—providing equivalently named members is sufficient.

Evaluators are associated with a group through a shared pointer. Type erasure[7] is used to store a pointer to any evaluator without changing the identity of the group. The `evaluate_using()` method performs the erasure and updates `evalptr` to point to the abstracted base class. Operators that require fitnesses call the `evaluate()` method, which performs an evaluation if necessary.

Dynamic-length arrays cannot be used in C++ AMP, which is why the variable-length individual reserves enough elements for the longest possible genome. (The `Shortest` parameter is used by the generator rather than the type definition, but may also be accessed later.) Individuals are allocated directly in the array stored with the group, as is typical for C++ programs.[8] This locality of data is critical for efficient GPU implementation and also simplifies the issue of aliasing, since two groups may only reference the same individual instance if they are actually the same group.

All operators return new groups, which necessarily involves cloning the individuals. Since the reference to the GPU array is created within the operator, no other operators can access it until after returning, at which point the group should be fully initialised. A pull model is used: each operator requests precisely the number of individuals it requires from the preceding operator until reaching a merge operator that can fulfil the request. Section D.3.3 contains more details about operator specification.

[7] Type erasure is a technique used to hide or 'forget' template type parameters, typically by using inheritance (effectively a private abstract base class) and templated functions.

[8] More precisely, a two-dimensional array is created, where one dimension contains each individual and the other contains the elements of the `fixed_individual` or `variable_individual` structure. C++ AMP hides this second dimension by exposing the array as a reference to the structure.

```
template<typename IndividualType>
class group
{
  shared_ptr<concurrency::array<IndividualType, 1>> ptr;
  shared_ptr<_evaluator_erased_t<group<IndividualType>>> evalptr;
public:
  bool evaluated;

  void evaluate();
  template<typename EvaluatorType>
  void evaluate_using(shared_ptr<EvaluatorType> newEval);

  operator bool() const;
  concurrency::array<IndividualType, 1>& operator*();
  concurrency::array<IndividualType, 1>* operator->();

  list<IndividualType> as_list() const;
  vector<IndividualType> as_vector() const;

  int size() const;
  void reset();
};
```

Listing D.3: Interface of the `group` class for `cppamp`.

```
template<typename ElementType, int Length>
struct fixed_individual {
  ElementType genome[Length];
  float fitness;
};

template<typename ElementType, int Shortest, int Longest>
struct variable_individual {
  ElementType genome[Longest];
  float fitness;
  int length;
};
```

Listing D.4: Interfaces for fixed- and variable-length individuals.

```
// FROM a, b SELECT 100 c, d USING crossover, mutation

decltype(a) _merge_1[] = { a, b };
auto _src_1 = esdl::merge(_merge_1);
auto _gen_1 = mutation(crossover(_src_1));
c = _gen_1(100);
d = _gen_1();
```

Listing D.5: C++ code generated for a `FROM-SELECT` statement.

`YIELD` is implemented by sorting the yielded group on the GPU and then copying to the CPU, where it is scanned for minimum, maximum, mean and median fitness. Depending on the output mode selected (see Section D.3.4), the entire group may be written to the console or a file.

D.3.2 Sequence Model

ESDL statements are executed in the order specified, since the main performance optimisation is obtained by parallelising the operators themselves rather than re-ordering statements. C++ AMP executes lazily by default, which reduces overhead by executing GPU kernels asynchronously only when the result is required, though operator parameters are copied when scheduled and cannot be modified.

Code is generated for each store operation in almost identical fashion to `esec`. Use of the `auto` type declaration, template-based operator definitions and mangled variable names allows simple code to be generated. C++ lacks robust variable-length parameter lists, and so merging multiple groups uses a locally declared array.[9]

Function calls are written to the generated code file in the same sequence as specified in the ESDL definition. This ensures that functions will be executed in the order specified, though due to C++ AMP's lazy scheduling, potentially out of order from store operations. Functions that access groups (by copying to the CPU) will block until all preceding stores have completed. Side effects in functions are permitted, though not recommended. The C++ code can be inspected and modified before compilation, allowing developers to inject any required synchronisation or shared state manually.

D.3.3 Extensibility Model

Operators and functions provided by users must be written in C++ and require the use of templates. Each operator fully contains its own GPU code; the only interactions between operators are the contents of groups. Since templates require method implementations to be available where instantiated, the only linking mechanism

[9] Variadic templates could be used to support arbitrary numbers of parameters. However, the only compiler supporting C++ AMP does not yet support variadics, so the array approach is required.

```
// ESDL generator: random_real(length=<int>10, lowest=(float)0, highest=(float)1)
// ESDL operator: fitness_sus(mu=(int)0)
// ESDL joiner: random_tuples(distinct=<bool>false)
// ESDL evaluator: sphere(origin=(float)0)
// ESDL function: es_adapt(source, current_step=(float)1)
```

Listing D.6: Specification comments for `cppamp` extensions.

required is including header files: `` `include `` pragmas in ESDL definitions are translated to C++ `#include` statements. To handle named parameters correctly, specially marked comments are added to `` `include ``d headers to indicate the names, positions, types and default values of each parameter, as well as to distinguish between functions, operators, joiners, generators and evaluators. Only the file specified in the `` `include `` pragma is scanned for specifications, even if it includes other files. Listing D.6 shows some examples of specification comments.

Specification comments consist of three parts: the marker, the name and the parameter list. The marker is the "ESDL …" text preceding the colon that identifies the type of the extension element. These types are enforced by `esdlc` to ensure that elements are used correctly. Names appear immediately after the colon and must match both the name of the C++ function and the name to be used in ESDL. Since C++ is case-sensitive, capitalisation must match the C++ function.

Four sets of information are embedded in a parameter list: the names of the valid parameters, the order in which the C++ function expects them, the types to use and the default values. Names are required for referencing each parameter in the ESDL definition, but for code generation named parameters are converted to positional. If a named parameter is omitted from the ESDL definition, the default value is used in its position instead. The casts are optional, but when they exist are applied to the value specified, whether it is a constant, an expression or a variable. ESDL functions may omit default values, in which case an error occurs if those parameters are not specified.

Parameter casts may be indicated with angle brackets for types of `bool`, `int` and `unsigned int`. These parameters are not passed as normal values, but are provided as compile-time constants using type parameters (one of `true_type`, `false_type` or `integral_constant`). Templated parameters must be specified with a constant value before code generation occurs. In some cases, such as specifying the length of an individual, type parameters are necessary, while in others they are convenient for producing efficient code. Listing D.7 includes an example of receiving a type parameter.

Apart from two exceptions, the source of every operator is another operator. The built-in merge operator takes one or more groups and optionally one generator, while

```
// ESDL operator: Operator(param1=(float)10, param2=<bool>false)

template<typename SourceType, bool param2>
class Operator_t {
  typedef <...> IndividualType;
public:
  Operator_t(SourceType source, float param1);
  group<IndividualType> operator()(int count);
  group<IndividualType> operator()();
};

template<typename SourceType, typename param2_t>
Operator_t<SourceType, param2_t::value>
Operator(SourceType _source, float param1, param2_t) {
  return Operator_t<SourceType, param2_t::value>(_source, param1);
}
```

Listing D.7: Interface for `cppamp` operators.

joiners take multiple operators, typically instances of the merge operator. Operators provide the `operator()` method to produce the result group. A `size` parameter allows the caller to request as many individuals as required to fulfil its own request. An overload without parameters allows operators to return all individuals; operators that produce infinite streams should omit this overload to produce a compilation error rather than a runtime out-of-memory error. Listing D.7 shows the general interface for operators, though since type variance is handled using templates rather than inheritance, operators only need to provide a matching interface, rather than deriving from this class.

Implementing `Operator` as a factory function rather than naming the class `Operator` is primarily for the convenience of code generation, since C++ supports template type inference on functions but not classes. Listing D.8 shows an example of the code that would have to be generated if Listing D.5 had no template type inference. Note that each operator now requires all of its parameters reproduced in order to specify the types; without `decltype()` and `auto`, the code generator would be required to perform C++ overload resolution, duplicating the most complicated part of a C++ compiler. Indirection through a function is an essential compromise.

Evaluators are provided as classes with a templated `operator()` overload taking a reference to a group, which is evaluated in-place. The general outline is shown in Listing D.9; note that there is no factory function, since the `sphere` class is not templated (though its `operator()` method is). The group's `evaluated` member is tested and updated automatically and the evaluator does not need to handle it.

`EVALUATE-USING` statements instantiate the evaluator using the C++ standard library function `make_shared`. The `evaluate_using()` method handles the type erasure,

```
// FROM a, b SELECT 100 c, d USING crossover, mutation

decltype(a) _merge_1[] = { a, b };
auto _src_1 = esdl::merge_t<decltype(a), 2>(_merge_1);
auto _gen_1 = mutation_t<decltype(crossover_t<decltype(_src_1)>(_src_1))>(
                                    crossover_t<decltype(_src_1)>(_src_1));
c = _gen_1(100);
d = _gen_1();
for (int _i_1 = 0; _i_1 < 2; ++_i_1) _merge_1[_i_1].reset();
```

Listing D.8: C++ code that would be required for Listing D.5 without template type inference.

```
// ESDL evaluator: sphere(origin=(float)0)

struct sphere {
  float origin;
  sphere(float origin) : origin(origin) { }

  template<typename IndividualType>
  void operator()(esdl::group<IndividualType, sphere>& source) {
    // Perform evaluation
    ...
  }
};
```

Listing D.9: Outline of a `sphere` evaluator implementation.

```
// ESDL evaluator: c(d=(float)0)
// EVALUATE a, b USING c(d=100)

auto _eval_1 = std::make_shared<c>((float)100);
a.evaluate_using(_eval_1);
b.evaluate_using(_eval_1);
_eval_1.reset();
```

Listing D.10: C++ code generated for `EVALUATE-USING` statements.

Table D.1: Command-line options for executables created with `cppamp`.

Command	Meaning
`/iteration:#` (`/iter`, `/i`)	Stop after the specified number of iterations
`/fitness:#` (`/fit`, `/f`)	Stop when the best fitness is better or equal to the value specified
`/evaluations:#` (`/evals`, `/e`)	Stop after the specified number of fitness evaluations
`/csv`	Write CSV output the console
`/verbose` (`/v`)	Write detailed output
`/quiet` (`/q`)	Do not display any output
`/seed:#` (`/s`)	Specify the seed value
`<name>=#`	Set variable `<name>` to the value specified

resets the `evaluated` field and updates the group's evaluator reference. After updating references, the local evaluator reference is reset to avoid keeping an unnecessary reference; `evaluate_using()` increments the reference count for as long as each group is associated with the evaluator. Listing D.10 shows the code generated for an `EVALUATE` statement with a `USING`.

D.3.4 Command-line Options

Executable files created by `esdlc`, while following a fixed algorithm, provide a number of settings that may be configured from the command line. These options are listed in Table D.1. Termination conditions can be specified at the command line. If none is set, the executable will pause after each iteration until a key is pressed; pressing "x" will terminate the algorithm.[10] The `/quiet` option disables all output, regardless of other settings. Output is always sent to the console, allowing users to redirect to files if desired.

Output formatting is controlled by `_output<T>` classes, which are specialised for each individual and fitness type that may be displayed. For example, the `_output<float>` class is used to format `float`s. Normal output produces a formatted table as shown in Listing D.11, while using the `/csv` option produces higher precision output as shown in Listing D.12.

Adding the `/verbose` option displays the contents of each group as it is yielded. Listings D.13 and D.14 show examples of verbose output with and without the `/csv` option.

Seeding the random number generator is essential for running reproducible experiments. By default, the seed is randomly selected; the `/seed` option may be used to specify a known value. The random number generator used is based on a hybrid Linear Congruential Generator and Combined Tausworth Generator [49]; the com-

[10] This also allows another process to control the algorithm by capturing its standard input and output streams.

```
 Iter  Evals  |  Best  |   Min   Mean    Med    Max
--------------+--------+---------------------------
    1     50  |   -0.5 |   -33   -2.6   -2.1   -0.5
    2    100  |   -0.5 |   -28   -2.3   -1.4   -0.9
    3    150  |   -0.1 |   -21   -1.9   -1.1   -0.1
--------------+--------+--------------------
Fitness limit reached

 Iter  Evals  |  Fitness  | Genome
--------------+-----------+------------------------
    3    150  |      -0.1 | [0.02, 0.04, -0.06]
```

Listing D.11: Normal output from a `cppamp` executable.

```
Iter, Evals, Best, Min, Mean, Med, Max
1, 50, -0.5013773, -33.29441, -2.572889, -2.098114, -0.5013773
2, 100, -0.5013773, -28.42933, -2.332981, -1.412944, -0.9134221
3, 150, -0.09982470, -21.02488, -1.910223, -1.145831, -0.09982470
Fitness limit reached
Iter, Evals, Fitness, Genome
3, 150, -0.09982470, 0.02093813, 0.03984101, -0.05832117
```

Listing D.12: CSV output from a `cppamp` executable.

```
Iteration 1
Block GENERATION
Yielded 5 offspring
Fitness | Genome
--------+--------
   10.4 | 1011101
   14.2 | 1101101
   48.9 | 0001111
   48.9 | 0001111
   57.1 | 0100111

Yielded 2 population
Fitness | Genome
--------+--------
   10.4 | 1011101
   14.2 | 1101101

Iter  Evals  |  Best  |    Min   Mean    Med    Max
-------------+--------+------------------------
   1      5  |  10.4  |   14.2   12.3   12.3   10.4
```

Listing D.13: Verbose output from a `cppamp` executable.

```
Iteration, Block, Group Size, Group Name
1, GENERATION, 5, offspring

Fitness, Genome
10.44, 1, 0, 1, 1, 1, 0, 1
14.18491, 1, 1, 0, 1, 1, 0, 1
48.97, 0, 0, 0, 1, 1, 1, 1
48.97, 0, 0, 0, 1, 1, 1, 1
57.07, 0, 1, 0, 0, 1, 1, 1

Iteration, Block, Group Size, Group Name
1, GENERATION, 2, population

Fitness, Genome
10.44, 1, 0, 1, 1, 1, 0, 1
14.18, 1, 1, 0, 1, 1, 0, 1

Iter, Evals, Best, Min, Mean, Med, Max
1, 5, 10.44, 14.18, 12.32, 12.32, 10.44
```

Listing D.14: Verbose CSV output from a `cppamp` executable.

monly used Mersenne Twister algorithm requires a large amount of state and is less efficient for parallel applications.

Finally, any uninitialised variables from the ESDL definition require specification on the command-line. Variables that are initialised cannot be overridden from the command line. Attempting to run an executable without specifying all uninitialised variables will fail and display a list of the names of the required variables. Names are not case-sensitive and values may only be numeric. Implicit parameters are not covered by this mechanism; any implicit parameter that has no other references in the ESDL definition is compiled as `true` and cannot be respecified later.

D.4 `cppamp` emitter

`esdlc` includes a library and emitter for C++ AMP code. The library contains implementations of the operators in Appendix A that run on a GPU. Operator implementations are discussed in Section D.3—this section focuses on the code generation. The code generated by the `cppamp` emitter could also be used with a compatible CPU-based library, though none has been implemented.

C++ does not support the dynamic typing used by ESDL, nor does C++ AMP have an iterator pattern similar to Python. However, the most recent C++ standard provides useful features such as `auto`, `decltype` and improved templates that simplify code generation for dynamic types [51]. The general approach used by the emitter is to assume that ESDL variables do not change type—a variable initialised with a

```
// x = 123
auto x = 123.000000f;

// x = 17 * 2
x = (decltype(x))(17.000000f * 2.000000f);
```

Listing D.15: Examples of C++ code generated for variables.

```
// FROM source SELECT group USING operator
auto _src_1 = esdl::merge(source);
auto _gen_1 = operator(_src_1);
auto group = _gen_1();              // first use of group

// FROM group SELECT 10 group USING operator
auto _src_2 = esdl::merge(group);
auto _gen_2 = operator(_src_2);
group = _gen_2((int)10.000000f);    // second use of group
```

Listing D.16: Examples of C++ code generated for groups.

number will always contain a number. The first use of each variable is generated as an `auto` initialiser, as shown in Listing D.15, while subsequent assignments force a cast (if necessary) using `decltype`. All numbers are treated as `float` to avoid issues due to integer division.

Groups are also declared with `auto` on first use and later uses are assumed to be the same type, as shown in Listing D.16.

The generated C++ file is based on six sections, of which some are fixed templates and others are created by the emitter:

1. *Includes*, which is created by adding an `#include` statement for each `` `include `` pragma in the source.
2. *Opening*, which is mostly predefined. It includes the `main()` definition and most command-line parsing, as well definitions for variables that are uninitialised in the ESDL definition.
3. *Variables*, which is completely generated and specifies the command-line parsing required to read values for externally provided variables.
4. *Initialisation*, which contains the code for the initialisation block.
5. *Termination*, which is fixed and tests the termination conditions including iteration count and fitness.
6. *Iteration*, which is generated and includes the code for each named block as well as invoking the selector for each iteration.

Apart from the initialisation block, every named block is written to the iteration section in an if–else if structure that selects based on the string returned by the selector. If no selector is provided with the `` `selector `` pragma, the modulus of the

```
// Original ESDL:
//  FROM population, offspring SELECT (99) population, rest USING best()

// Generated Code:
decltype(population) _merge_1[] = { population, offspring };
auto _src_1 = esdl::merge(_merge_1);
auto _gen_1 = best(_src_1);
population = _gen_1((int)99.000000f);
auto rest = _gen_1();
for (int _i_1 = 0; _i_1 < 2; ++_i_1) _merge_1[_i_1].reset();
```

Listing D.17: Generated C++ code for a store operation.

iteration count is used to select between multiple blocks. When only one named block exists no if statements are needed, though a selector will still be invoked each iteration if specified.

As with the `esec` emitter, a single function (`_write_block`, in this case) iterates through the ESDL statements and delegates code generation to more specific functions. Most of the processing is attached to the `EmitterScope` class, which maintains both global and local declaration information. Each block uses a different `EmitterScope`, though the initialisation block's instance is available to all others. This ensures that C++ variables declared in the initialisation block, which are available to other blocks, are not redeclared. However, variables declared within a block are not shared and must be declared again if another block uses it. ESDL specifies that blocks must be able to be executed in any order—the implication of this is that groups and variables that are shared between blocks must always be initialised in the initialisation section.

The `_store_to_lines()` function generates the code for store operations, returning lines of code as a list. Source groups are passed to the `esdl::merge` operator directly (when only one group or a group and a generator are passed) or using a temporary array. The array is typed using the first group, implicitly requiring all merged groups to be of the same type. After use, each element in the array has its `reset()` member called to reduce its internal reference count.

Each operator in the chain is combined into a single statement, passing the previous operator as the first parameter, with the result assigned to variable declared with `auto`. Destination groups are assigned the result of taking the required number of individuals from the operator; either a specified amount or all that are available. Listing D.17 shows an example of the code generated for a store operation—the suffix `_1` increments for each store operation to avoid naming collisions.

Code generation for function calls, operator invocations and evaluator creation was described in Section D.3.3 and does not require further discussion here.

`EVALUATE` statements generate code that instantiates the new evaluator and updates the groups. Type erasure is used internally to avoid having to encode the evaluator type into the group type, which allows a group to have its evaluator replaced without needing to generate a new variable. If no evaluator is specified and a default evaluator was given, it is assigned instead. The default evaluator is only instantiated once and reused for all groups.

Yielded groups are evaluated, sorted and copied to the CPU. The statistics collection is not as easily customisable as for `esec`; an included `esdl::make_stats()` function collects the minimum, maximum, mean and median fitnesses and stores the group in a local variable, primarily for viewing with a debugger. Depending on the output settings, the contents of the group may be printed to the console. At the end of the iteration the statistics for the *default group* (defined below) are printed.

Pragmas supported by the `cppamp` emitter are `` `cpp ``, `` `include ``, `` `evaluator ``, `` `selector `` and `` `yield ``. `` `cpp `` pragmas are treated as literal C++ and included in the generated code. `` `include `` commands specify external header files to include and parse the file for specification comments (see Section D.3.3). `` `evaluator `` is followed by an evaluator specification, as for `EVALUATE-USING` statements, that will be used as the default evaluator. The `` `selector `` pragma specifies a function name or call that will return a `std::wstring` containing the name of the block to execute. Finally, the `` `yield `` pragma indicates the default group, which is used for fitness-based termination conditions. It is only necessary when no groups are yielded in the initialisation block; otherwise, the first yielded group is used. All other pragmas are ignored silently.

Listing D.18 shows the code generated for the system in Listing C.8 (page 244) by the `cppamp` emitter.

Listing D.18: C++ code generated by `cppamp` for Listing C.8.

```
#include <esdl.h>
#include "test.h"

int wmain(int _argc, wchar_t** _argv) {
  int _esdl_iteration_limit = 0;
  int _esdl_evaluation_limit = 0;

  bool _esdl_has_iteration_limit = false;
  bool _esdl_has_evaluation_limit = false;
  bool _esdl_has_fitness_limit = false;
  std::wistringstream _esdl_fitness_limit_string;

  bool _esdl_quiet_output = false;
  bool _esdl_verbose_output = false;
  bool _esdl_csv_output = false;

  int _esdl_seed = 0;

  std::list<std::wstring> _esdl_uninitialised;
```

```
float config = 0; _esdl_uninitialised.push_back(L"config");

int _argi = 0;
for (auto _arg = esdl::get_command(_argc, _argv, _argi); _arg; _arg = esdl::get_command(»
 «_argc, _argv, _argi)) {

  if (!_arg->command.empty()) {
    if (esdl::equals(_arg->command, L"i", L"iter", L"iterations")) {
      _arg->value >> _esdl_iteration_limit;
      _esdl_has_iteration_limit = true;
    } else if (esdl::equals(_arg->command, L"e", L"eval", L"evaluations")) {
      _arg->value >> _esdl_evaluation_limit;
      _esdl_has_evaluation_limit = true;
    } else if (esdl::equals(_arg->command, L"f", L"fit", L"fitness")) {
      _esdl_fitness_limit_string.str(_arg->value.str());
      _esdl_has_fitness_limit = true;
    } else if (esdl::equals(_arg->command, L"s", L"seed")) {
      _arg->value >> _esdl_seed;
    } else if (esdl::equals(_arg->command, L"csv")) {
      _esdl_csv_output = true;
    } else if (esdl::equals(_arg->command, L"v", L"verbose")) {
      _esdl_verbose_output = true;
    } else if (esdl::equals(_arg->command, L"q", L"quiet")) {
      _esdl_quiet_output = true;
    } else {
      std::wcout << L"Unrecognised option: /"  << _arg->command << std::endl;
    }
  } else if (esdl::equals(_arg->variable, L"config")) {
    _arg->value >> config;
    _esdl_uninitialised.remove(L"config");
  }
}

if (!_esdl_uninitialised.empty()) {
  std::wcerr << L"Uninitialised variables:" << std::endl;
  std::for_each(std::begin(_esdl_uninitialised), std::end(_esdl_uninitialised), [](std::»
 «wstring _var) { std::wcerr << L"  " << _var << std::endl; });
  return 1;
}

esdl::seed(_esdl_seed);

// `include "test.h"
// FROM random_binary(length=config.length.max) SELECT (100) population
auto _src_0 = random_binary(std::integral_constant<int, 10>(), (float)0.5f);
auto _gen_0 = _src_0;
auto population = _gen_0((int)100.000000f);

// t = 0.0
auto t = 0.000000f;
// delta_t = (-0.1)
auto delta_t = (-0.100000f);
// lambda = 100.0
auto lambda = 100.000000f;
// EVAL population USING evaluators.population(t)
auto _eval_1 = std::make_shared<evaluators::population>((float)t);
population.evaluate_using(_eval_1);
_eval_1.reset();
```

```
// YIELD population
esdl::_default_statistics_name = L"population";
esdl::tt::fitness_type<decltype(population)>::type _esdl_fitness_limit;
if (_esdl_has_fitness_limit) {
  _esdl_fitness_limit_string >> _esdl_fitness_limit;
}
if (!_esdl_quiet_output) {
  esdl::write_statistics_header<typename esdl::tt::individual_type<decltype(population)>::»
 «type>();
}
population.evaluate();
auto _esdl_population_group = esdl::make_stats(L"population", population);
if (!_esdl_quiet_output && _esdl_verbose_output) {
  if (_esdl_csv_output) esdl::write_group_raw(L"population", _esdl_population_group);
  else esdl::write_group(L"population", _esdl_population_group);
}
esdl::EndReason _reason;
for (;; _iteration += 1) {
  if (!_esdl_quiet_output) {
    esdl::write_statistics(esdl::default_statistics, _esdl_population_group.front());
  }

  if (!(_esdl_has_iteration_limit || _esdl_has_evaluation_limit || _esdl_has_fitness_limit)»
 «) {
    int c = esdl::getch();
    if (c == 'x') break;
  }
  else if (_esdl_has_iteration_limit && _iteration > _esdl_iteration_limit) {
    _reason = esdl::IterLimit;
    break;
  } else if (_esdl_has_evaluation_limit && _evaluations > _esdl_evaluation_limit) {
    _reason = esdl::EvalLimit;
    break;
  } else if (_esdl_has_fitness_limit && _esdl_population_group.front().fitness > »
 «_esdl_fitness_limit) {
    _reason = esdl::FitLimit;
    break;
  }
  // t = (t+(delta_t*1.4))
  t = (decltype(t))(t+(delta_t*1.400000f));
  // `py print(t)
  // `cpp printf("%f\n", t);
  printf("%f\n", t);
  // REPEAT 10.0
  int _repeat_limit_1 = (int)10.000000f;
  for(int _repeat_1 = 0; _repeat_1 < _repeat_limit_1; ++_repeat_1) {
    // FROM population SELECT (100) parents USING tournament(k=2.0, greediness=0.7)
    auto _src_2 = esdl::merge(population);
    auto _gen_2 = tournament(_src_2, (int)2.000000f, (float)0.700000f, std::true_type(), »
 «std::false_type());
    auto parents = _gen_2((int)100.000000f);

    // FROM parents SELECT mutated USING mutate_delta(stepsize)
    auto _src_3 = esdl::merge(parents);
    auto _gen_3 = mutate_delta(_src_3, (float)1, (float)0.5, (float)1, (float)1, (int)0);
    auto mutated = _gen_3();
```

```
      // FROM parents SELECT crossed USING uniform_crossover()
      auto _src_4 = esdl::merge(parents);
      auto _gen_4 = uniform_crossover(_src_4, (float)1);
      auto crossed = _gen_4();

      // EVAL mutated, crossed USING evaluator(t=t, lambda)
      auto _eval_5 = std::make_shared<evaluator>((float)t, (int)lambda);
      mutated.evaluate_using(_eval_5);
      crossed.evaluate_using(_eval_5);
      _eval_5.reset();

      // JOIN mutated, crossed INTO merged USING tuples()
      auto _src_6 = tuples(esdl::merge(mutated), esdl::merge(crossed));
      auto _gen_6 = _src_6;
      auto merged = _gen_6();

      // FROM merged SELECT offspring USING best_of_tuple()
      auto _src_7 = esdl::merge(merged);
      auto _gen_7 = best_of_tuple(_src_7);
      auto offspring = _gen_7();

      // FROM population, offspring SELECT (99) population, rest USING best()
      decltype(population) _merge_8[] = { population, offspring };
      auto _src_8 = esdl::merge(_merge_8);
      auto _gen_8 = best(_src_8);
      population = _gen_8((int)99.000000f);
      auto rest = _gen_8();
      for (int _i_8 = 0; _i_8 < 2; ++_i_8) _merge_8[_i_8].reset();

      // FROM rest SELECT (1) extras USING uniform_random()
      auto _src_9 = esdl::merge(rest);
      auto _gen_9 = uniform_random(_src_9);
      auto extras = _gen_9((int)1.000000f);

      // FROM population, rest, extras SELECT (100) population
      decltype(population) _merge_10[] = { population, rest, extras };
      auto _src_10 = esdl::merge(_merge_10);
      auto _gen_10 = _src_10;
      population = _gen_10((int)100.000000f);
      for (int _i_10 = 0; _i_10 < 3; ++_i_10) _merge_10[_i_10].reset();

    }
    // EVAL population USING evaluators.config(t)
    auto _eval_11 = std::make_shared<evaluators::config>((float)t);
    population.evaluate_using(_eval_11);
    _eval_11.reset();

    // YIELD population
    population.evaluate();
    _esdl_population_group = esdl::make_stats(L"population", population);
    if (!_esdl_quiet_output && _esdl_verbose_output) {
      if (_esdl_csv_output) esdl::write_group_raw(L"population", _esdl_population_group);
      else esdl::write_group(L"population", _esdl_population_group);
    }
  }
}
```

Appendix E

esec Architecture

E.1 Overview

esec is a Python framework for EC that uses ESDL as its main configuration input. Originally designed by Clinton Woodward to support the ecosystem model work in [98], it was released for use by other EC researchers. esec has been used as a base for experimentation relating to this work, with the updated version representing one potential instantiation of ESDL and the associated EA model. The latest version is available from http://esec.googlecode.com/.

esec includes the following features:

- Fully customisable algorithms using ESDL specification.
- A range of individual representations (esec.species) including binary, real-valued, GP and GE.
- Standard benchmark problems (esec.landscape).
- Full unit test suite[1] and documentation.[2]
- Flexible Python-based configuration files.
- Extensible reporting and data logging (esec.monitors).
- Supported by Python 2.6 and later.[3]
- Independent of operating system and platform.

The packages included in esec are shown in Figure E.1. Some of the names reflect the original design rather than the names used in this work for equivalent concepts, while others are artefacts of the software's evolution. For example, the landscape and species packages existed in the original design, while the generators package was named to reflect the Python definition of the term (a programmable iterator roughly equivalent to a map function) rather than the ESDL definition (a

[1] Using the nose package, which is available online at http://readthedocs.org/docs/nose/en/latest/.

[2] Using epydoc, which is available online at http://epydoc.sourceforge.net/.

[3] The 2to3 tool must be used before esec can be used with Python 3.0 or later.

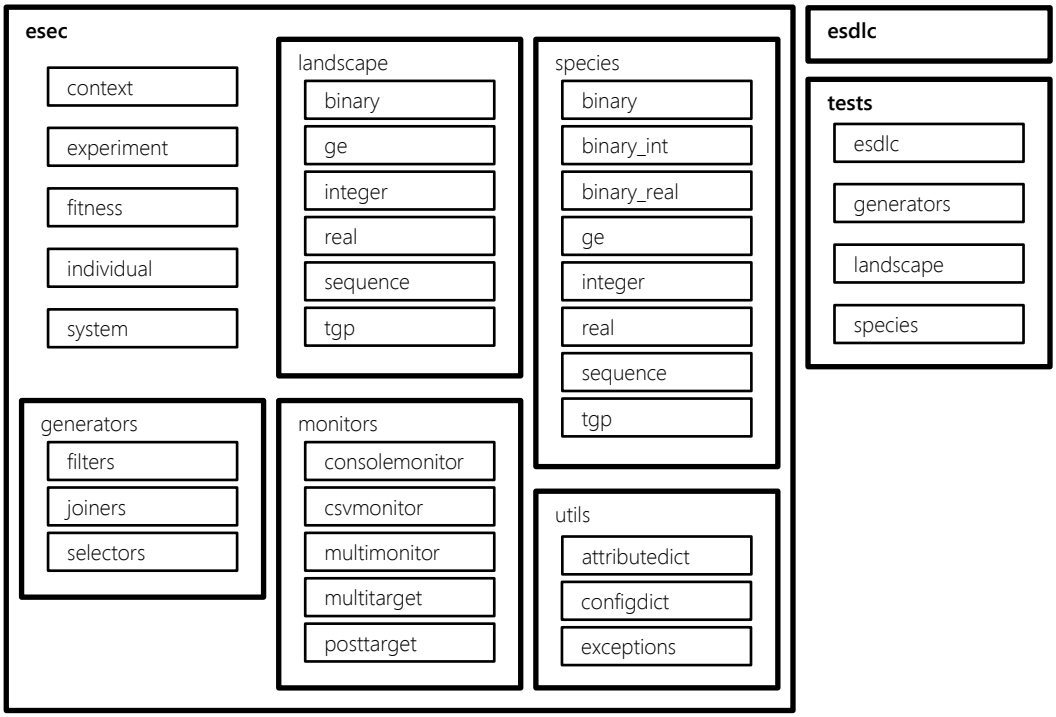

Figure E.1: `esec` package hierarchy.

pseudo-group containing an infinite number of individuals). Renaming the packages is a significant breaking change with too many unpredictable effects, particularly considering the software has been publicly available throughout its development.

The following sections describe the architecture, extensibility and use of `esec`. It is not complete documentation, but is intended to provide an overview for those new to the framework. Thorough documentation is included in the code and can be extracted using `epydoc`.

E.2 Python

The Python programming language was used for implementing `esec`. Python is a strong, dynamically typed language that has interpreters and compilers for many platforms. It does not strictly adhere to any programming paradigm, but can support imperative, object-oriented and functional programming styles among others. Importantly for this application, Python has intrinsic support for lists and iterators, including functions such as `filter` and `map`. Listing E.1 shows some examples of Python list and tuple syntax; lines beginning with `>>>` are code and others are the value of the preceding expression.

Python's dynamic typing means that the contents of collections may be non-homogeneous; any data type may be stored in a list that is used by code with no prior knowledge of its contents. Further, Python's dynamic name binding allows objects to fulfil a required interface by providing a member with a matching name. Listing E.2

```
>>> x = [1, 2, 3]    # list (mutable)
>>> y = (4, 5, 6)    # tuple (immutable)
>>> z = x + y        # concatenation
>>> z
[1, 2, 3, 4, 5, 6]
>>> z[1:4]           # slice
[2, 3, 4]
>>> map(lambda i: i*10, x)
[10, 20, 30]
>>> filter(lambda i: i>=5, y)
[5, 6]
```

Listing E.1: Lists and iterators in Python.

```
class Rectangle:
    def draw(self):
        ...
class Circle:
    def draw(self):
        ...

shapes = [Rectangle(), Circle()] # each class is instantiated here
for s in shapes:
    s.draw()
```

Listing E.2: Dynamic typing and name binding in Python.

shows an example of a list containing two different objects, a loop iterating over this list and invocation of two different (unwritten, in this example) functions.

Python classes and functions are instances of regular objects, which allows them to be treated identically to any other object. Lists or dictionaries may contain them (classes in Python are little more than a dictionary of member functions) and they can be passed as parameters to other functions. Listing E.3 shows an example of calling a function `func` with either a function or a class as the parameter. Since both types are callable, `func` can be used generically.

Modules, the equivalent of namespaces, are also based on regular objects and can be loaded from source files at runtime. Python provides run-time compilation of code from strings that returns an object that either contains callables (like a module) or *is* callable (like a function or class). Listing E.4 shows some examples of using Python's `exec` function to compile and execute code at runtime.

With these features, an interactive EA framework using ESDL can be created that allows researchers to easily create novel algorithms, integrate externally provided evaluators and produce results suitable for publication.

```
def func(t):
    return t(100)

def double(v):
    return v + v

class ValueStore:
    def __init__(self, v):
        self.value = v

>>> func(double)
200
>>> func(ValueStore)
<ValueStore object>
```

Listing E.3: Dictionaries and first-class types and functions in Python.

```
>>> statement = 'print("Hello, from exec.")'
>>> exec(statement)
Hello, from exec.

>>> function = '''
... def doubler(x):
...   print(2 * x)'''
>>> doubler(10)          # have not compiled the function yet
NameError: name 'doubler' is not defined

>>> exec(function)       # creates doubler globally
>>> doubler(10)
20

>>> scope = { }
>>> exec(function, scope) # creates doubler in scope
>>> scope['doubler']
<function doubler>
>>> scope['doubler'](10)
20
```

Listing E.4: Dynamic compilation and invocation in Python.

```
experiment.begin()

while experiment.step(always_step=False):
  pass

experiment.close()
```

Listing E.5: Python code to run an entire experiment.

E.3 Architecture

E.3.1 Experiments

The central class is `esec.experiment.Experiment`. It is instantiated with a configuration dictionary (Section E.4.1) that specifies the algorithm and all required settings. The public interface includes `begin()`, `step()` and `end()` functions: most algorithms can be run with the code in Listing E.5, which is provided as the `Experiment.run()` function.

`Experiment` defers most of the processing work to an instance of `esec.system.System`, which compiles the ESDL definition, calls the selector and invokes the required block (as described in Section 4.3, page 74). Compilation is performed by `esdlc`, which is included as a package.

The `esec.context` module provides access to the active blackboard dictionary (`esec.context.context`), the original configuration dictionary (`esec.context.config`), random number generator (`esec.context.rand`; an instance of `random.Random`) and notification method (`esec.context.notify()`). Each of these are thread-local, which prevents multiple experiments from being run on the same thread but allows independent experiments to be constructed and executed on separate threads. `esec` typically runs only a single experiment.

Accessing the blackboard outside ESDL code is discouraged, but permitted in `esec` for flexibility. A similar rationale applies to the configuration dictionary, which may not be identical to the original and is effectively read-only. However, the shared random number generator is essential to ensuring experiments are reproducible: a seed may have been provided in the configuration. The `notify(sender, name, value)` function allows indirect communication with the monitor (Section E.3.3) from operators, evaluators or external functions.

E.3.2 Species

Species classes define individual representations and sets of applicable operators. A species defines the ESDL generators that produce its individuals. When the individuals are created, they are given a reference to their species, which is treated as

a base class (through an override of `__getattr__`). The abstract `esec.species.Species` class provides support for obtaining parameters through the configuration dictionary, exposing functions in a species class's `public_context` dictionary to ESDL code and a range of representation-independent operators, such as discrete and point-based crossover.

Creating a new species typically requires defining an individual class, derived from `esec.individual.Individual`, as well as one derived from `Species`. By convention, individual classes contain the breeding representation in a property `genome`, the solution representation in a property `phenome`, user-readable versions of each of these in `genome_string` and `phenome_string` and a string describing the length of the individual in `length_string`. Separation into multiple properties allows representations such as the `BinaryIntegerSpecies`, which has a `genome` containing bits but a `phenome` of integer values generated from these bits based on a flexible mapping process. Operators should normally only access `genome`, while evaluators use `phenome`. The `length_string` property allows, for example, GP individuals to display both node count and tree depth; `len(genome)` and `len(phenome)` return the number of automatically-defined functions [55].

Species classes typically include implementations of operators that are specific to that representation. Python does not (easily) support overloading based on type, so unknown operators are looked up on the individual type and then the species. This allows both real and integer-valued individuals, as well as other user-defined species, to define operators such as `mutate_delta` without suffering naming collisions or invalid behaviour.

E.3.3 Monitors

Monitors provide the interface between the user and the experiment. They use an event-based model, where `System` invokes specific functions in response to a range of events. Table E.1 shows the function signatures required to exist on a monitor. In all cases, `sender` is the `System` instance raising the event. Deriving a monitor from `esec.monitors.MonitorBase` provides no-operation implementations of these functions, allowing only those required to be overridden.

The `should_terminate()` function almost always needs to be specified: the default in `MonitorBase` is to return `True`, which prevents algorithms from running any blocks after initialisation. For most research experiments, `ConsoleMonitor` and `CSVMonitor` are sufficient, since they provide basic statistics collection (in `on_yield()`), customisable termination conditions and flexible reporting formats. However, interactive applications may want to create a monitor that can communicate with a user interface.

Table E.1: Event handlers required on monitors.

Function	Event
`on_yield(sender, group_name, group)`	On `YIELD` statements
`on_notify(sender, name, value)`	On `notify()` calls
`on_pre_reset(sender)`	Before calling initialisation block
`on_post_reset(sender)`	After calling initialisation block
`on_pre_breed(sender)`	Before calling selector and named block
`on_post_breed(sender)`	After calling named block
`on_run_start(sender)`	At start of a run, before `on_pre_reset`
`on_run_end(sender)`	At end of a run
`on_exception(sender, exc_type, exc_value, trace)`	When an exception is raised in a block
`should_terminate(sender) -> bool`	Before each block, returns `True` to finish

A monitor is provided to `Experiment` as part of the configuration dictionary under the `'monitor'` key.[4] An instance may be specified directly, or a dictionary with a `'class'` value and other parameters can be given. The advantage of specifying a class and parameters is the ability to override settings directly from the command-line as demonstrated in Section E.4.2.

E.3.4 Landscapes

Although evaluators can be relatively simple functions, `esec` supports rich problem landscapes that may be parameterised through the configuration dictionary, provide feedback on individual legality, display textual information at varying degrees of verbosity and trivially provide maximising or minimising fitnesses. Landscape classes that derive from `esec.landscape.Landscape` or one of its typed subclasses such as `esec.landscape.real.Real` or `esec.landscape.binary.Binary` only need add a function `_eval(individual)` that returns a fitness (either a number or an instance of `esec.fitness.Fitness`). Class variables for `syntax` and `default` allow configuration dictionary entries to be validated, while the `maximise` property ensures the correct sense is used for numeric fitnesses.

General evaluator objects provide an `eval(individual)` function returning an instance of `Fitness` or a subclass specifying the sense (such as `FitnessMaximise` or `FitnessMinimise`); the rich landscapes provide an `eval()` implementation that invokes the `_eval()` function and adds the correct `Fitness` class. Unless overridden incorrectly (by not passing the `**other_cfg` parameter through), all the provided classes derived from `Landscape` can be used directly in `EVALUATE-USING` statements with all settings given as named parameters (dotted names are specified with underscores rather than periods).

[4] Multiple monitors may be provided using the `MultiMonitor` class, which forwards events to a list of monitors.

```
Experiment.syntax = {
  'random_seed': [int, None],
  'monitor': '*',    # pre-initialised MonitorBase instance, class or dict
  'landscape': '*',  # validated by Landscape
  'system': '*',     # validated by System
  'selector?': '*',  # validated by System
  'verbose': int,
}
```

Listing E.6: Configuration syntax from the `Experiment` class.

E.4 Use

`esec` provides two primary modes of use. For those wanting to include full ESDL support in their own project, the `Experiment` class may be used directly to construct and execute algorithms. Those who are interested in statistical analysis and performance of algorithms can use the provided `run.py` script. Both approaches use configuration dictionaries. The following sections describe how these dictionaries work and the two approaches to using `esec`.

E.4.1 Configuration Dictionaries

Python provides a built-in dictionary type that maps from a key to a value. The key and value may be of any type and there is no requirement for all the keys or values in a dictionary to be the same type. This flexibility, combined with a minimal syntax, makes dictionaries a simple way to provide arbitrary sets of parameters.

The configuration dictionaries used in `esec` are an extension of Python dictionaries to provide a consistent approach to specifying and validating syntax, including default values and segregating data. A single configuration dictionary contains all the settings for an experiment, including the system definition, external operators and functions, the evaluator and parameters, a monitor and its parameters, random seed and debug settings. The `ConfigDict.validate()` function allows a dictionary containing type information to be compared against the configuration dictionary to ensure all required values have been provided and are the correct types.

For example, Listing E.6 shows the syntax used in the `Experiment` class. Any configuration dictionary provided must include these keys (except `'selector'`, which is optional, as denoted by the `?` suffix). The value for `'random_seed'` must be an integer or `None`, `'verbose'` must be an integer and the rest are unrestricted.

This is a relatively basic use. Consider the `esec.landscape.real.Stabilising` class, which has an inheritance hierarchy leading back to `esec.landscape.Landscape`. The syntax dictionaries through these classes (Listing E.7) are merged automatically to

```
Landscape.syntax = {
  'class?': type,
  'instance?': '*',
  'random_seed': [int, None],
  'invert?': bool,
  'offset?': float,
  'parameters': [None, int],
  'size': {
    'min': int,
    'max': int,
    'exact': int
  },
}

Real.syntax = {
  'lower_bounds?': [tuple, list, int, float, str, None],
  'upper_bounds?': [tuple, list, int, float, str, None],
}

Stabilising.syntax = {
  'mean': float
}
```

Listing E.7: Configuration syntax for the `Landscape`, `Real` and `Stabilising` classes.

produce the syntax used (Listing E.8) to validate the value for `'landscape'` in the configuration dictionary.

This approach to merged syntax allows similar classes to automatically provide common parameters while only specifying those that are new or changed. Default values are similarly specified and merged. Including default values simplifies the specification of an experiment, since only some values need to be specified.

E.4.2 `run.py` Script

The `run.py` script provides a command-line and file interface for constructing configuration dictionaries. Its main option is `--config` (abbreviated to `-c`) which takes a list of *configuration names* separated by plus symbols (`+`). Each name refers to a predefined dictionary containing part of a complete configuration. As each name is specified, the associated dictionary is overlaid onto the configuration dictionary by adding or replacing values recursively. Names are defined in `run.py`, loaded from `dialects.py`, automatically generated for all known landscapes or loaded dynamically from files or other names.

As an example, passing "`-c RVP.Sphere+n2+GA+csv`" sets the landscape to the real-valued sphere (`RVP.Sphere`, generated from a list of landscapes) with two dimensions (`n2`, specified in `run.py`) running a standard GA (`GA`, loaded from `dialects.py`) and

```
merge_cls_dicts(self, 'syntax') = {
  'class?': type,
  'instance?': '*',
  'random_seed': [int, None],
  'invert?': bool,
  'offset?': float,
  'parameters': [None, int],
  'size': {
    'min': int,
    'max': int,
    'exact': int
  },
  'lower_bounds?': [tuple, list, int, float, str, None],
  'upper_bounds?': [tuple, list, int, float, str, None],
  'mean': float
}
```

Listing E.8: Merged configuration syntax for the `Stabilising` landscape.

writing the results to CSV files (`csv`, specified in `run.py`). Names are case-sensitive and always applied in the order specified.

New configuration names are provided by creating a Python file in the `esec/cfgs` directory. If a configuration name is not found among the predefined options, this directory is searched for a file with the same name (dotted names, like `RVP.Sphere`, recurse into subdirectories if they exist). The file is executed and should define a dictionary `config` with the configuration dictionary elements to be set. A dictionary `configs` may also be provided that contains new configuration names to allow later in the same command. For example, a configuration with three common parameter sets could specify each set as a configuration name rather than three separate configuration files.

As well as using configuration names, the `--settings` command-line option (abbreviated to `-s`) can be used to override individual values. For example, passing "`-s monitor.limits.iterations=100;random_seed=1`" will set the iteration limit to one hundred and the seed to one regardless of the underlying configuration. This option is a convenience for quickly testing with alternate settings—everything that can be specified using `--settings` could also be included in a configuration file.

The third important command-line option is `--batch` (abbreviated to `-b`). Configuration files, along with the `config` variable, can declare a function called `batch()` that returns a sequence of batch descriptors, each of which defines an entire experiment to be run. A batch descriptor is a dictionary mapping any or all of `'names'`, `'config'`, `'settings'` and `'tags'` to appropriate values. The `'names'` value contains a string using the same format as for the `--config` command line option: each configuration name is appended to the configuration dictionary. After names are applied,

Table E.2: `run.py` batch settings.

Setting Name	Description
`batch.dry_run`	If `True`, does not run any experiments, but still creates and saves the configuration dictionaries.
`batch.start_at`	The index of the descriptor to run first; all previous ones are skipped.
`batch.stop_at`	The index of the descriptor to run last; all later ones are skipped.
`batch.include_tags`	A list of tags, where at least one must be in a descriptor for it to be run.
`batch.exclude_tags`	A list of tags, where none must be in a descriptor for it to be run.
`batch.summary`	If `True`, writes a summary file containing the final results line from each experiment.
`batch.csv`	If `True`, writes output using `CSVMonitor` rather than `ConsoleMonitor`.
`batch.low_priority`	If `True`, attempts to reduce the priority of the Python process.
`batch.quiet`	If `True`, minimises the output displayed during each run.
`batch.pathbase`	An optional string specifying the path where output files are written. By default, it is "`results/`" followed by the configuration file name.

the dictionary in `'config'` is overlaid, followed by the list of overrides in `'settings'`, which matches the format for `--settings`.

Tags can be applied to each descriptor to simplify executing only a subset of the configurations. The value in the descriptor for `'tags'` is a list of strings. When starting a batch, tags may be marked as 'included' or 'excluded.' If no tags are marked, all configurations will run. If any tags are included, *only* batch descriptors containing at least one of those tags will run. When any tags are excluded, *any* batch descriptor containing any of those tags will *not* run.

Batch runs can be configured using a `settings` variable in the configuration file that contains a string with any of the settings in Table E.2. The default settings run all available batch descriptors with as much output and as high a priority as possible. Specifying a settings string such as "`batch.low_priority=True;batch.quiet=True`" is convenient for running experiments in the background. Included and excluded tags may be specified in this string, however it is simpler to specify them on the command line. The `--batch` option is followed by the name of the configuration file, which must appear first, then tag names joined with plus symbols (`+`). Tags with a leading exclamation mark (`!`) are excluded, while others are included. For example, "`-b DERosenbrock+F1.2+!CR0.0`" will use the `DERosenbrock.py` configuration file, running all experiments tagged with `F1.2` except for those with `CR0.0`.

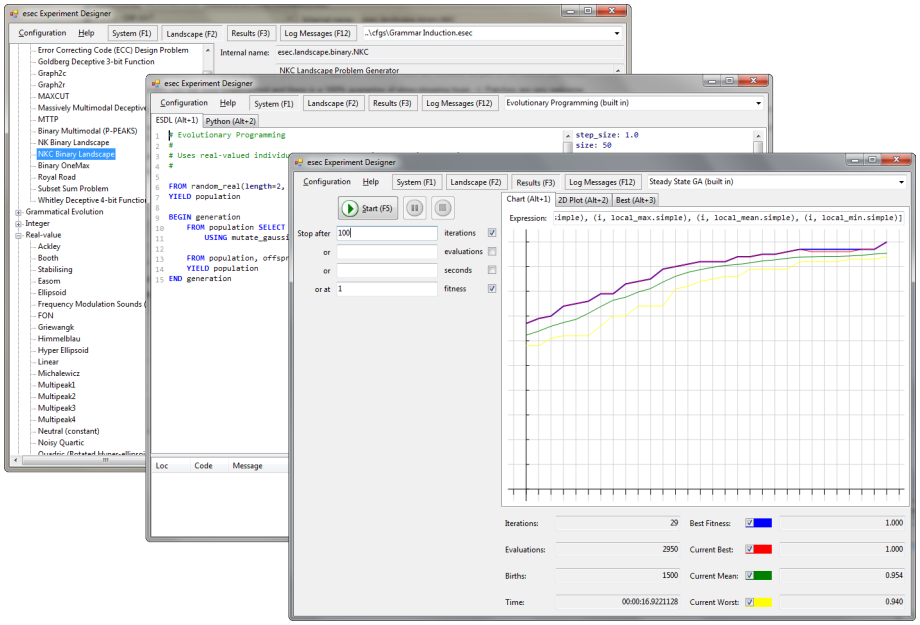

Figure E.2: Screenshots of `esecui`.

E.4.3 Embedding esec

For Python-based projects, embedding `esec` is a simple matter of importing and instantiating the `Experiment` class. Providing a complete configuration dictionary remains the easiest way to configure the experiment, though a Python project could also use operators directly if desired.

Projects that are not written in Python can still integrate `esec` by either hosting a Python interpreter or creating configuration files while executing. Hosted Python normally allows the larger application to interact directly with the experiment, typically through a custom monitor. CPython supports being embedded in most languages using a C-style interface, while IronPython[5] can be hosted in applications that use the Microsoft .NET Framework.

For example, `esecui` is written in C# and hosts IronPython to run `esec`. Configuration dictionaries are created as C# `Dictionary<string, object>` classes and used directly by `esec`, while a custom monitor class is implemented in C# to handle events. Figure E.2 shows some screenshots of `esecui`, which has source code and executable available from `http://esecui.googlecode.com/`.

E.5 Extensibility

While `esec` can be extended by adding new classes to the packages discussed, the plugin mechanism is generally simpler when using the `run.py` command-line interface (Section E.4). Python code files may be added to the `esec/plugins` directory as either

[5] An implementation of Python for the .NET Framework. Details are available online at `http://ironpython.net/`.

a module or a package with the plugin name. These files can define new species, landscapes and operators, which are made available whenever the plugin is specified.

A new species can derive from an existing species or the `esec.species.Species` class. Calling the `esec.species.include()` function with the new species class makes its generators available to the ESDL system. New configuration names may be added by defining a `configs` dictionary, and a `defaults` dictionary may be provided to add operators to the system or include a landscape.

For example, `esec` comes with a plugin providing a Max Set of Gaussians landscape generator [40]. The landscape depends upon the `numpy` package[6] and including it in the `esec.landscape` package would extend this dependency to all of `esec`. When `RVP.MSG` is specified as a configuration name, the plugin is loaded. In the `RVP.MSG.py` file is the `MSG` landscape class, a helper class and a `defaults` dictionary. The `defaults` dictionary sets the landscape to be the generator, which makes the behaviour of specifying `RVP.MSG` as a configuration identical to specifying any of the built-in landscapes.

A second plugin is for ACO. This plugin is provided as a package, in the `esec/plugins/ACO` directory. In this case, the `__init__.py` file is imported when `ACO` is given as a configuration name. Here, a new species (`plugins.ACO.tsp.TourSpecies`) and two configuration names (`configs['ACO.TSP']` and `configs['berlin52']`) are provided. The `ACO.TSP` configuration name specifies a system definition, adds a `create_pheromone_map` function to the ESDL system, specifies the built-in `esec.landscape.sequence.TSP` landscape and defines a more suitable default monitor format. The `berlin52` configuration provides values for the landscape and adds the optimum fitness for the landscape as a termination condition. Specifying "`-c ACO+ACO.TSP+berlin52`" will load the plugin, apply the two configurations and run ACO[7] until the best fitness is found.

For configuration files, rather than command-line configuration, plugin classes can be imported as members of the `plugins` package. This will typically include the species, though default values and configuration names will generally not be merged. For example, "`import plugins.ACO`" is sufficient to gain access to the `build_tours` generator, but `plugins.ACO.pheromone.PheromoneMap` must be added to the system manually, where the plugin normally includes it as part of the `ACO.TSP` configuration name.

Simpler extensions, such as a single operator or evaluator, can be specified directly in configuration files. `esec.esdl_func` and `esec.esdl_eval` are function decorators that may be applied to make Python functions available in the ESDL definition;

[6] Available online from `http://numpy.scipy.org/`.

[7] The default system definition actually specifies Ant System [25]. The plugin, however, supports a range of stigmergy or pheromone-based algorithms.

```
from esec import esdl_func, esdl_eval

def func(param):    # functions don't need default values
    ...
    return result

@esdl_func
def operator(_source, param=0): # _source receives the input stream
    for indiv in _source:
        ...
        yield new_indiv

@esdl_eval
def sphere(indiv):
    return sum(x**2 for x in indiv)

# @esdl_eval implicitly creates the following class
#class sphere_class:
#    def eval(self, indiv):
#        return sum(x**2 for x in indiv)

config = {
  'system': {
    'function': func,        # names may differ, if desired
    #'operator': operator,   # this is implied by @esdl_func
    #'sphere': sphere_class, # this is implied by @esdl_eval
    'definition': '''
v = function(param=100)      # use the name given in the dictionary

FROM random_real SELECT 100 population USING operator
EVALUATE population USING sphere
''' }
}
```

Listing E.9: Example of providing external Python functions in a configuration dictionary.

otherwise, a direct reference may be made in the `'system'` member of the configuration dictionary. Listing E.9 shows an example of including Python functions in a configuration file.

E.6 Summary

`esec` is the implementation of ESDL that was described in Chapter 6. It takes advantage of the dynamic features of Python to provide a framework capable of running experiments, investigating extensions to ESDL, rapidly prototyping and designing algorithms, and embedding in other applications.

Source code to `esec` is available online at `http://esec.googlecode.com/`.

Appendix F

Comparison Code

Chapter 6 discussed how ESDL can provide significant benefits to those working with EAs. One of these benefits is that precise algorithm descriptions can be specified with less code. This appendix contains the sections of code that were compared. Full code files that are compilable or executable with the associated library are available to download from http://stevedower.id.au/thesis.

F.1 Evolution Strategies

esec configuration file for ES

```
import math
import esec.landscape.real

def random_indiv(length=10):
    from esec.context import rand, context
    for indiv in context['random_real'](length=length, lowest=-5, highest=5):
        indiv.strategy = [rand.random() * 0.5 for _ in xrange(int(length))]
        yield indiv

def es_mutate(_source):
    from esec.context import rand
    for indiv in _source:
        n = len(indiv)
        new_strategy = []
        new_solution = []
        for (value, stddev) in zip(indiv, indiv.strategy):
            new_strategy.append(stddev * math.exp(
                rand.gauss(0, 1) * math.sqrt(4/n) +
                rand.gauss(0, 1) / (2*n)))
            new_solution.append(value + new_strategy[-1] * rand.gauss(0, 1))
        yield type(indiv)(new_solution, indiv, strategy=new_strategy)

SYSTEM_DEFINITION = r'''
FROM random_indiv(length=2) SELECT 20 population
YIELD population

BEGIN iteration
  FROM population SELECT 40 offspring USING repeated, es_mutate
```

```
  FROM population, offspring SELECT 20 population USING best
  YIELD population
END
'''

config = {
    'system': {
        'definition': SYSTEM_DEFINITION,
        'random_indiv': random_indiv,
        'es_mutate': es_mutate,
    },
    'landscape': { 'class': esec.landscape.real.Sphere, 'parameters': 2 },
    'monitor': {
        'report': 'brief+local+time_delta',
        'summary': 'status+brief+best_genome',
        'limits': { 'iterations': 50 }
    },
}
```

ECJ parameter file for ES

```
parent.0 = ECJ/ec/ec.params

eval.problem = ec.app.ecsuite.ECSuite
eval.problem.type = sphere

state = ec.simple.SimpleEvolutionState
init = ec.simple.SimpleInitializer
finish = ec.simple.SimpleFinisher
exch = ec.simple.SimpleExchanger
eval = ec.simple.SimpleEvaluator
stat = ec.simple.SimpleStatistics
generations = 50
quit-on-run-complete = true
pop = ec.Population
pop.subpops = 1
pop.subpop.0 = ec.Subpopulation
pop.subpop.0.duplicate-retries = 0
pop.subpop.0.species = ec.vector.FloatVectorSpecies
pop.subpop.0.species.pipe = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0 = ec.es.ESSelection

pop.subpop.0.species.fitness = ec.simple.SimpleFitness
pop.subpop.0.species.ind = dower.es.ESIndividual
pop.subpop.0.species.mutation-bounded = true

pop.subpop.0.species.min-gene = -5
pop.subpop.0.species.max-gene = 5
pop.subpop.0.species.genome-size = 2

es.mu.0 = 20
es.lambda.0 = 40
breed = ec.es.MuPlusLambdaBreeder

pop.subpop.0.species.mutation-prob = 1.0
pop.subpop.0.species.mutation-type = gauss
pop.subpop.0.species.mutation-stdev = 1.0
```

```
pop.subpop.0.size = 30

public void defaultCrossover(EvolutionState state, int thread,
    VectorIndividual ind)
    { }

public void defaultMutate(EvolutionState state, int thread)
    {
    FloatVectorSpecies s = (FloatVectorSpecies) species;
    boolean mutationIsBounded = s.mutationIsBounded;
    MersenneTwisterFast rng = state.random[thread];
    for (int x = 0; x < genome.length && x < strategy.length; x++)
        {
        double min = s.minGene(x);
        double max = s.maxGene(x);
        double stdev = strategy[x];

        stdev = stdev * Math.exp(rng.nextGaussian() * Math.sqrt(4.0 / genome.length) +
                          rng.nextGaussian() * (2.0 * genome.length));

        genome[x] = rng.nextGaussian() * stdev + genome[x];
        }
    }

public void reset(EvolutionState state, int thread)
    {
    super.reset(state, thread);

    FloatVectorSpecies s = (FloatVectorSpecies) species;
    for (int x = 0; x < strategy.length; x++)
        strategy[x] = (state.random[thread].nextDouble() * 0.5);
    }
```

FakeEALib program for ES

```
using System;
using System.Collections.Generic;
using FakeEALib;

namespace FakeEALib.Experiments {
    class EvolutionStrategy {
        public static void Run() {
            var population = new Populations.RealWithStrategy(10, -5.0, 5.0, 0.0, 0.5);
            var offspring = new Populations.RealWithStrategy(10, -5.0, 5.0, 0.0, 0.5);
            population.Evaluator = new Evaluators.Real.Sphere();

            offspring.Select.Add(new Selectors.Clone());
            offspring.Variation.Add(new MutateES());
            offspring.VariationMode = Operators.VariationMode.Replace;
            offspring.Sort.Add(new Sorters.Best());
            offspring.Filter.Add(new Filters.Truncate(20));
            offspring.Evaluator = population.Evaluator;

            population.CreateRandom(20);

            population.Evaluate();
            Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
```

```
            Console.WriteLine("{0} | {1} | {2} | {3}",
                0, population.Best.Fitness, population.MeanFitness, population.Worst.Fitness);

            for (int iteration = 1; iteration <= 50; ++iteration) {
                population.Evaluate();
                offspring.Clear();
                offspring.AddClonedRange(population);
                offspring.AddClonedRange(population);
                offspring.Vary();
                offspring.AddRange(population);
                offspring.Evaluate();
                offspring.SortFilter();

                population.Clear();
                population.AddRange(offspring);

                Console.WriteLine("{0} | {1} | {2} | {3}",
                    iteration, population.Best.Fitness, population.MeanFitness, population.»
    «Worst.Fitness);
            }

            Console.WriteLine("Best: {0} [{1}]", population.Best.Fitness, population.Best.»
    «ToString());
        }
    }

    class MutateES : Operators.IOperator {
        public void Vary(IList<Populations.IIndividual> source, int index) {
            var indiv = (Populations.IRealIndividualWithStrategy)source[index];

            for (int i = 0; i < indiv.Count; ++i) {
                indiv.Strategy[i] *= Math.Exp(
                    Random.Normal() * Math.Sqrt(4.0 / indiv.Count) +
                    Random.Normal() / (2.0 * indiv.Count));

                indiv[i] += indiv.Strategy[i] * Random.Normal();
            }
        }
    }
}
```

Full C# program for ES

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FullAlgorithms {
    static class EvolutionStrategy {
        private class Individual : IComparable<Individual> {
            public readonly double[] Genome;
            public readonly double[] Strategy;
            public double Fitness;

            public Individual(int length) {
                Genome = new double[length];
                Strategy = new double[length];
```

```
    }

    public int CompareTo(Individual other) {
        return other.Fitness.CompareTo(Fitness);
    }

    public override string ToString() {
        return String.Join(", ", Genome);
    }
}

private static double? _NextGauss = null;
private static double Normal(this Random rnd) {
    if (_NextGauss.HasValue) {
        var value = _NextGauss.Value;
        _NextGauss = null;
        return value;
    } else {
        double x2pi = rnd.NextDouble() * Math.PI * 2;
        double g2rad = Math.Sqrt(-2.0 * Math.Log(1.0 - rnd.NextDouble()));
        _NextGauss = Math.Sin(x2pi) * g2rad;
        return Math.Cos(x2pi) * g2rad;
    }
}

private static void PrintSummary(int iteration, List<Individual> indivs) {
    indivs.Sort();
    double total = 0;
    foreach (var indiv in indivs) {
        total += indiv.Fitness;
    }
    double average = total / indivs.Count;

    Console.WriteLine("{0} | {1} | {2} | {3}",
        iteration, indivs[0].Fitness, average, indivs[indivs.Count - 1].Fitness);
}

public static void Run() {
    var rnd = new Random();
    var population = new List<Individual>();

    for (int i = 0; i < 20; ++i) {
        var indiv = new Individual(10);
        for (int j = 0; j < indiv.Genome.Length; ++j) {
            indiv.Genome[j] = rnd.NextDouble() * 10 - 5;
            indiv.Strategy[j] = rnd.NextDouble() * 0.5;
        }
        population.Add(indiv);
    }

    foreach (var indiv in population) {
        double total = 0;
        for (int j = 0; j < indiv.Genome.Length; ++j) {
            total += indiv.Genome[j] * indiv.Genome[j];
        }
        indiv.Fitness = -total;
    }
```

```
            Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
            PrintSummary(0, population);

            for (int iteration = 1; iteration <= 50; ++iteration) {
                var offspring = new List<Individual>();
                for (int i = 0; i < 2; ++i) {
                    foreach (var indiv in population) {
                        var child = new Individual(10);
                        for (int j = 0; j < indiv.Genome.Length; ++j) {
                            child.Strategy[j] = indiv.Strategy[j] * Math.Exp(
                                rnd.Normal() * Math.Sqrt(4.0 / indiv.Genome.Length) +
                                rnd.Normal() / (2.0 * indiv.Genome.Length));

                            child.Genome[j] = indiv.Genome[j] + child.Strategy[j] * rnd.»
    «Normal();
                        }
                        offspring.Add(child);
                    }
                }

                foreach (var indiv in offspring) {
                    double total = 0;
                    for (int j = 0; j < indiv.Genome.Length; ++j) {
                        total += indiv.Genome[j] * indiv.Genome[j];
                    }
                    indiv.Fitness = -total;
                }

                population.AddRange(offspring);

                population.Sort();

                population.RemoveRange(20, population.Count - 20);

                PrintSummary(iteration, population);
            }

            Console.WriteLine("Best: {0} [{1}]", population[0].Fitness, population[0].»
    «ToString());
        }
    }
}
```

F.2 Evolutionary Programming

esec configuration file for EP

```
import esec.landscape.integer

SYSTEM_DEFINITION = r'''
FROM random_int(length=8, lowest=0, highest=25) SELECT 100 population
YIELD population

BEGIN iteration
  FROM population SELECT offspring USING mutate_delta(step_size=1, per_gene_rate=0.5)
  FROM population, offspring SELECT 100 population USING tournament(k=5)
```

```
  YIELD population
END
'''

config = {
    'system': { 'definition': SYSTEM_DEFINITION },
    'landscape': { 'class': esec.landscape.integer.Nsum, 'parameters': 8 },
    'monitor': {
        'report': 'brief+local+time_delta',
        'summary': 'status+brief+best_genome',
        'limits': { 'iterations': 50 }
    },
}
```

ECJ parameter file for EP

```
parent.0 = ec/ec.params

eval.problem = dower.evaluators.Nsum

state = ec.simple.SimpleEvolutionState
init = ec.simple.SimpleInitializer
finish = ec.simple.SimpleFinisher
exch = ec.simple.SimpleExchanger
eval = ec.simple.SimpleEvaluator
stat = ec.simple.SimpleStatistics
generations = 50
quit-on-run-complete = true
pop = ec.Population
pop.subpops = 1
pop.subpop.0 = ec.Subpopulation
pop.subpop.0.duplicate-retries = 0
pop.subpop.0.species = ec.vector.IntegerVectorSpecies
pop.subpop.0.species.pipe = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0 = ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.size = 5

pop.subpop.0.species.fitness = ec.simple.SimpleFitness
pop.subpop.0.species.ind = dower.ep.EPIndividual
pop.subpop.0.species.mutation-bounded = true

pop.subpop.0.species.min-gene = 0
pop.subpop.0.species.max-gene = 25
pop.subpop.0.species.genome-size = 8

breed = ec.simple.SimpleBreeder

pop.subpop.0.species.mutation-step = 1.0
pop.subpop.0.species.mutation-prob = 0.5

pop.subpop.0.size = 100
```

FakeEALib program for EP

```
using System;
using FakeEALib;
```

```
namespace FakeEALib.Experiments {
    class EvolutionaryProgramming {
        public static void Run() {
            var population = new Populations.Integer(8, 0, 25);
            population.Select.Add(new Selectors.Clone());
            population.Variation.Add(new Operators.MutateDelta(1, 0.5, 1.0));
            population.VariationMode = Operators.VariationMode.Expand;
            population.Sort.Add(new Sorters.Best());
            population.Filter.Add(new Filters.Truncate(100));
            population.Evaluator = new Evaluators.Integer.Nsum();

            population.CreateRandom(100);

            population.Evaluate();
            Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
            Console.WriteLine("{0} | {1} | {2} | {3}",
                0, population.Best.Fitness, population.MeanFitness, population.Worst.Fitness);

            for (int iteration = 1; iteration <= 50; ++iteration) {
                population.Step();

                population.Evaluate();
                Console.WriteLine("{0} | {1} | {2} | {3}",
                    iteration, population.Best.Fitness, population.MeanFitness, population.»
    «Worst.Fitness);
            }

            Console.WriteLine("Best: {0} [{1}]", population.Best.Fitness, population.Best.»
    «ToString());
        }
    }
}
```

Full C# program for EP

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FullAlgorithms {
    static class EvolutionaryProgramming {
        private class Individual : IComparable<Individual> {
            public readonly int[] Genome;
            public double Fitness;

            public Individual(int length) {
                Genome = new int[length];
            }

            public int CompareTo(Individual other) {
                return other.Fitness.CompareTo(Fitness);
            }

            public override string ToString() {
                return String.Join(", ", Genome);
            }
        }
```

```
private static void PrintSummary(int iteration, List<Individual> indivs) {
    indivs.Sort();
    double total = 0;
    foreach (var indiv in indivs) {
        total += indiv.Fitness;
    }
    double average = total / indivs.Count;

    Console.WriteLine("{0} | {1} | {2} | {3}",
        iteration, indivs[0].Fitness, average, indivs[indivs.Count - 1].Fitness);
}

public static void Run() {
    var rnd = new Random();
    var population = new List<Individual>();

    for (int i = 0; i < 100; ++i) {
        var indiv = new Individual(8);
        for (int j = 0; j < indiv.Genome.Length; ++j) {
            indiv.Genome[j] = rnd.Next(0, 26);
        }
        population.Add(indiv);
    }

    foreach (var indiv in population) {
        int total = 0;
        for (int j = 0; j < indiv.Genome.Length; ++j) {
            total += indiv.Genome[j];
        }
        indiv.Fitness = (double)total;
    }

    Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
    PrintSummary(0, population);

    for (int iteration = 1; iteration <= 50; ++iteration) {
        var offspring = new List<Individual>();
        foreach (var indiv in population) {
            var child = new Individual(8);
            for (int j = 0; j < indiv.Genome.Length; ++j) {
                double prob = rnd.NextDouble();
                if (prob < 0.25)
                    child.Genome[j] = indiv.Genome[j] + 1;
                else if (prob < 0.5)
                    child.Genome[j] = indiv.Genome[j] - 1;
                else
                    child.Genome[j] = indiv.Genome[j];
            }
            offspring.Add(child);
        }

        foreach (var indiv in offspring) {
            int total = 0;
            for (int j = 0; j < indiv.Genome.Length; ++j) {
                total += indiv.Genome[j];
            }
            indiv.Fitness = (double)total;
```

```
                }

                population.AddRange(offspring);

                population.Sort();

                population.RemoveRange(100, population.Count - 100);

                PrintSummary(iteration, population);
            }

            Console.WriteLine("Best: {0} [{1}]", population[0].Fitness, population[0].»
    «ToString());
        }
    }
}
```

F.3 Genetic Algorithms

esec configuration file for GA

```
import esec.landscape.real

SYSTEM_DEFINITION = r'''
FROM random_binary(length=64) SELECT 100 population
YIELD population

BEGIN generation
  FROM population SELECT 100 parents USING fitness_proportional
  FROM parents SELECT population USING crossover(per_pair_rate=0.98), \
                                       mutate_bitflip(per_gene_rate=1/64)
  YIELD population
END
'''

config = {
    'system': { 'definition': SYSTEM_DEFINITION },
    'landscape': { 'class': esec.landscape.binary.OneMax, 'parameters': 64 },
    'monitor': {
        'report': 'brief+local+time_delta',
        'summary': 'status+brief+best_genome',
        'limits': { 'iterations': 100 }
    },
}
```

ECJ parameter file for GA

```
parent.0 = ec/ec.params

eval.problem = dower.evaluators.OneMax

state = ec.simple.SimpleEvolutionState
init = ec.simple.SimpleInitializer
finish = ec.simple.SimpleFinisher
exch = ec.simple.SimpleExchanger
eval = ec.simple.SimpleEvaluator
```

```
stat = ec.simple.SimpleStatistics
generations = 100
quit-on-run-complete = true
pop = ec.Population
pop.subpops = 1
pop.subpop.0 = ec.Subpopulation
pop.subpop.0.duplicate-retries = 0
pop.subpop.0.species = ec.vector.VectorSpecies
pop.subpop.0.species.pipe = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.source.0 = ec.select.FitProportionateSelection
pop.subpop.0.species.pipe.source.0.source.1 = same

pop.subpop.0.species.fitness = ec.simple.SimpleFitness
pop.subpop.0.species.ind = ec.vector.BitVectorIndividual

pop.subpop.0.species.genome-size = 64
pop.subpop.0.species.crossover-type = one
pop.subpop.0.species.crossover-likelihood = 0.98
pop.subpop.0.species.mutation-prob = 0.015625

breed = ec.simple.SimpleBreeder

pop.subpop.0.size = 100
```

FakeEALib program for GA

```
using System;
using FakeEALib;

namespace FakeEALib.Experiments {
    class GeneticAlgorithm {
        public static void Run() {
            var population = new Populations.Binary(64);
            population.Select.Add(new Selectors.FitnessProportional(true, 100));
            population.Select.Add(new Selectors.Clone());
            population.Evaluator = new Evaluators.Binary.OneMax();
            population.Variation.Add(new Operators.Crossover(1, 0.98));
            population.Variation.Add(new Operators.MutateBitflip(1.0, 1.0 / 64));
            population.VariationMode = Operators.VariationMode.Replace;

            population.CreateRandom(100);

            population.Evaluate();
            Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
            Console.WriteLine("{0} | {1} | {2} | {3}",
                0, population.Best.Fitness, population.MeanFitness, population.Worst.Fitness);

            for (int iteration = 1; iteration <= 100; ++iteration) {
                population.Step();

                population.Evaluate();
                Console.WriteLine("{0} | {1} | {2} | {3}",
                    iteration, population.Best.Fitness, population.MeanFitness, population.»
    «Worst.Fitness);
            }
```

```
            Console.WriteLine("Best: {0} [{1}]", population.Best.Fitness, population.Best.»
    «ToString());
        }
    }
}
```

Full C# program for GA

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FullAlgorithms {
    static class GeneticAlgorithm {
        private class Individual : IComparable<Individual> {
            public readonly bool[] Genome;
            public double Fitness;

            public Individual(int length) {
                Genome = new bool[length];
            }

            public int CompareTo(Individual other) {
                return other.Fitness.CompareTo(Fitness);
            }

            public override string ToString() {
                return String.Join(", ", Genome);
            }
        }

        private static void PrintSummary(int iteration, List<Individual> indivs) {
            indivs.Sort();
            double total = 0;
            foreach (var indiv in indivs) {
                total += indiv.Fitness;
            }
            double average = total / indivs.Count;

            Console.WriteLine("{0} | {1} | {2} | {3}",
                iteration, indivs[0].Fitness, average, indivs[indivs.Count - 1].Fitness);
        }

        public static void Run() {
            var rnd = new Random();
            var population = new List<Individual>();

            for (int i = 0; i < 100; ++i) {
                var indiv = new Individual(64);
                for (int j = 0; j < indiv.Genome.Length; ++j) {
                    indiv.Genome[j] = rnd.NextDouble() < 0.5;
                }
                population.Add(indiv);
            }

            foreach (var indiv in population) {
                double total = 0;
```

```
    for (int j = 0; j < indiv.Genome.Length; ++j) {
        total += indiv.Genome[j] ? 1.0 : 0.0;
    }
    indiv.Fitness = total;
}

Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
PrintSummary(0, population);

for (int iteration = 1; iteration <= 100; ++iteration) {
    double fitnessTotal = 0;
    foreach (var indiv in population) {
        fitnessTotal += indiv.Fitness;
    }

    var parents = new List<Individual>();

    while (parents.Count < population.Count) {
        var prob = rnd.NextDouble() * fitnessTotal;

        foreach (var indiv in population) {
            prob -= indiv.Fitness;
            if (prob <= 0) {
                parents.Add(indiv);
                break;
            }
        }
    }

    population.Clear();
    for (int i = 0; i < parents.Count; i += 2) {
        var p1 = parents[i];
        var p2 = parents[i + 1];

        int cut = rnd.Next(1, p1.Genome.Length - 1);
        var c1 = new Individual(p1.Genome.Length);
        var c2 = new Individual(p1.Genome.Length);

        for (int j = 0; j < p1.Genome.Length; ++j) {
            if (j < cut) {
                c1.Genome[j] = p1.Genome[j];
                c2.Genome[j] = p2.Genome[j];
            } else {
                c2.Genome[j] = p1.Genome[j];
                c1.Genome[j] = p2.Genome[j];
            }

            if (rnd.NextDouble() < (1.0 / 64.0)) {
                c1.Genome[j] = !c1.Genome[j];
            }
            if (rnd.NextDouble() < (1.0 / 64.0)) {
                c2.Genome[j] = !c2.Genome[j];
            }
        }

        population.Add(c1);
        population.Add(c2);
    }
```

```
                foreach (var indiv in population) {
                    double total = 0;
                    for (int j = 0; j < indiv.Genome.Length; ++j) {
                        total += indiv.Genome[j] ? 1.0 : 0.0;
                    }
                    indiv.Fitness = total;
                }

                PrintSummary(iteration, population);
            }

            Console.WriteLine("Best: {0} [{1}]", population[0].Fitness, population[0].»
    «ToString());
        }
    }
}
```

F.4 Differential Evolution

esec configuration file for DE

```
import esec.landscape.real

def mutate_DE(_source, F=0.8):
    for base, p1, p2 in _source:
        new_genes = []
        for b, x1, x2 in zip(base.genome, p1.genome, p2.genome):
            new_genes.append(b + F * (x1 - x2))
        yield type(base)(new_genes, base)

SYSTEM_DEFINITION = r'''
FROM random_real(length=3,lowest=-5,highest=5) SELECT 30 population
YIELD population

BEGIN generation
  JOIN population, population, population INTO bases USING random_tuples(distinct)
  FROM bases SELECT mutants USING mutate_de

  JOIN population, mutants INTO parents
  FROM parents SELECT trials USING crossover_tuple

  JOIN population, trials INTO target_trial_pairs
  FROM target_trial_pairs SELECT population USING best_of_tuple

  YIELD population
END
'''

config = {
    'system': { 'definition': SYSTEM_DEFINITION, 'mutate_de': mutate_DE },
    'landscape': { 'class': esec.landscape.real.Sphere, 'parameters': 3 },
    'monitor': {
        'report': 'brief+local+time_delta',
        'summary': 'status+brief+best_genome',
        'limits': { 'iterations': 50 }
```

```
    },
}
```

ECJ parameter file for DE

```
parent.0 = ec/ec.params

eval.problem = ec.app.ecsuite.ECSuite
eval.problem.type = sphere

state = ec.simple.SimpleEvolutionState
init = ec.simple.SimpleInitializer
finish = ec.simple.SimpleFinisher
exch = ec.simple.SimpleExchanger
eval = ec.de.DEEvaluator
stat = ec.simple.SimpleStatistics
generations = 50
quit-on-run-complete = true
pop = ec.Population
pop.subpops = 1
pop.subpop.0 = ec.Subpopulation
pop.subpop.0.duplicate-retries = 0
pop.subpop.0.species = ec.vector.FloatVectorSpecies

pop.subpop.0.species.fitness = ec.simple.SimpleFitness
pop.subpop.0.species.ind = ec.vector.DoubleVectorIndividual
pop.subpop.0.species.mutation-bounded = true

pop.subpop.0.species.min-gene = -5
pop.subpop.0.species.max-gene = 5
pop.subpop.0.species.genome-size = 3

breed = ec.de.DEBreeder
breed.f = 0.8
breed.cr = 0.5

pop.subpop.0.species.mutation-prob = 0.5

pop.subpop.0.size = 30

public DoubleVectorIndividual createIndividual(EvolutionState state,
    int subpop,
    int index,
    int thread)
    {
    Individual[] inds = state.population.subpops[subpop].individuals;

    DoubleVectorIndividual v = (DoubleVectorIndividual)(inds[index].clone());
    do
        {
        int r0, r1, r2;
        do
            {
            r0 = state.random[thread].nextInt(inds.length);
            }
        while( r0 == index );
        do
            {
```

```
            r1 = state.random[thread].nextInt(inds.length);
            }
        while( r1 == r0 || r1 == index );
        do
            {
            r2 = state.random[thread].nextInt(inds.length);
            }
        while( r2 == r1 || r2 == r0 || r2 == index );

        DoubleVectorIndividual g0 = (DoubleVectorIndividual)(inds[r0]);
        DoubleVectorIndividual g1 = (DoubleVectorIndividual)(inds[r1]);
        DoubleVectorIndividual g2 = (DoubleVectorIndividual)(inds[r2]);

        for(int i = 0; i < v.genome.length; i++)
            v.genome[i] = g0.genome[i] + F * (g1.genome[i] - g2.genome[i]);
        }
    while(!valid(v));

    return crossover(state, (DoubleVectorIndividual)(inds[index]), v, thread);
    }


public DoubleVectorIndividual crossover(EvolutionState state, DoubleVectorIndividual target, »
    «DoubleVectorIndividual child, int thread)
    {
    int index = state.random[thread].nextInt(child.genome.length);
    double val = child.genome[index];

    for(int i = 0; i < child.genome.length; i++)
        {
        if (state.random[thread].nextDouble() < Cr)
            child.genome[i] = target.genome[i];
        }

    child.genome[index] = val;

    return child;
    }
```

FakeEALib program for DE

```
using System;
using System.Collections.Generic;
using FakeEALib;

namespace FakeEALib.Experiments {
    class DifferentialEvolution {
        public static void Run() {
            var population = new Populations.Real(3, -5.0, 5.0);

            population.Select.Add(new Selectors.Clone());
            population.Variation.Add(new MutateDE(0.8));
            population.VariationMode = Operators.VariationMode.Tournament;
            population.Evaluator = new Evaluators.Real.Sphere();

            population.CreateRandom(30);

            population.Evaluate();
```

```
            Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
            Console.WriteLine("{0} | {1} | {2} | {3}",
                0, population.Best.Fitness, population.MeanFitness, population.Worst.Fitness);

            for (int iteration = 1; iteration <= 50; ++iteration) {
                population.Step();

                population.Evaluate();
                Console.WriteLine("{0} | {1} | {2} | {3}",
                    iteration, population.Best.Fitness, population.MeanFitness, population.»
    «Worst.Fitness);
            }

             Console.WriteLine("Best: {0} [{1}]", population.Best.Fitness, population.Best.»
    «ToString());
        }
    }

    class MutateDE : Operators.IOperator {
        public double F { get; private set; }

        public MutateDE(double F) {
            this.F = F;
        }

        public void Vary(IList<Populations.IIndividual> source, int index) {
            int j = Random.Integer(source.Count);
            while (j == index) {
                j = Random.Integer(source.Count);
            }

            int k = Random.Integer(source.Count);
            while (k == index || k == j) {
                k = Random.Integer(source.Count);
            }

            var result = (Populations.IRealIndividual)source[index];
            var p1 = (Populations.IRealIndividual)source[j];
            var p2 = (Populations.IRealIndividual)source[k];

            for (int i = 0; i < result.Count; ++i) {
                if (Random.Probability(0.5)) {
                    result[i] += F * (p1[i] - p2[i]);
                }
            }
        }
    }
}
```

Full C# program for DE

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FullAlgorithms {
    static class DifferentialEvolution {
```

```
private class Individual : IComparable<Individual> {
    public readonly double[] Genome;
    public double Fitness;

    public Individual(int length) {
        Genome = new double[length];
    }

    public int CompareTo(Individual other) {
        return other.Fitness.CompareTo(Fitness);
    }

    public override string ToString() {
        return String.Join(", ", Genome);
    }
}

private static void PrintSummary(int iteration, List<Individual> indivs) {
    indivs.Sort();
    double total = 0;
    foreach (var indiv in indivs) {
        total += indiv.Fitness;
    }
    double average = total / indivs.Count;

    Console.WriteLine("{0} | {1} | {2} | {3}",
        iteration, indivs[0].Fitness, average, indivs[indivs.Count - 1].Fitness);
}

public static void Run() {
    var rnd = new Random();
    var population = new List<Individual>();

    for (int i = 0; i < 30; ++i) {
        var indiv = new Individual(3);
        for (int j = 0; j < indiv.Genome.Length; ++j) {
            indiv.Genome[j] = rnd.NextDouble() * 10.0 - 5.0;
        }
        population.Add(indiv);
    }

    foreach (var indiv in population) {
        double total = 0;
        for (int j = 0; j < indiv.Genome.Length; ++j) {
            total += indiv.Genome[j] * indiv.Genome[j];
        }
        indiv.Fitness = -total;
    }

    Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
    PrintSummary(0, population);

    for (int iteration = 1; iteration <= 50; ++iteration) {
        for (int i = 0; i < population.Count; ++i) {
            int i1 = rnd.Next(population.Count);
            while (i1 == i) {
                i1 = rnd.Next(population.Count);
            }
```

```
                int i2 = rnd.Next(population.Count);
                while (i2 == i || i2 == i1) {
                    i2 = rnd.Next(population.Count);
                }

                var child = new Individual(population[i].Genome.Length);
                var p1 = population[i1].Genome;
                var p2 = population[i2].Genome;

                for (int j = 0; j < child.Genome.Length; ++j) {
                    if (rnd.NextDouble() < 0.5) {
                        child.Genome[j] = population[i].Genome[j] + 0.8 * (p1[j] - p2[j]);
                    } else {
                        child.Genome[j] = population[i].Genome[j];
                    }
                }

                double total = 0;
                for (int j = 0; j < child.Genome.Length; ++j) {
                    total += child.Genome[j] * child.Genome[j];
                }
                child.Fitness = -total;

                if (child.Fitness > population[i].Fitness) {
                    population[i] = child;
                }
            }

            PrintSummary(iteration, population);
        }

        Console.WriteLine("Best: {0} [{1}]", population[0].Fitness, population[0].»
«ToString());
    }
  }
}
```

F.5 Genetic Programming

The code for GP was not compared in Chapter 6 because the influence of the library design outweighs the effect of the method of description. For completeness, the descriptions for `esec` and ECJ are included here. Both are complete and describe the same experiment, though rely heavily on the libraries included with each framework.

`esec` configuration file for GP

```
from esec import esdl_eval
from esec.species.tgp import Instruction

SYSTEM_DEFINITION = r'''
FROM random_tgp(instructions,deepest=15,terminals=1) SELECT (size) population
EVAL population USING eval_expression
deepest_result = 7
YIELD population
```

```
BEGIN generation
  FROM population SELECT (size) parents USING fitness_proportional
  FROM parents SELECT (size*0.9) p1, (size*0.02) p2, p3

  FROM p1 SELECT o1 USING crossover_one(deepest_result)
  FROM p2 SELECT o2 USING mutate_random(deepest_result)

  FROM o1, o2, p3 SELECT (size) population
  YIELD population
END
'''

@esdl_eval
def eval_expression(individual):
  from esec.context import rand
  error_sum = 0.0
  for _ in range(20):
    x = rand.random() * 2.0 - 1.0
    expected = x**3 + x**2 + x + 1
    actual = individual.evaluate(individual, terminals=[x])
    error_sum += (actual - expected) ** 2

  return 1.0 / (1 + error_sum)

config = {
    'system': {
        'definition': SYSTEM_DEFINITION,
        'instructions': [
            Instruction(lambda a, b: a+b, param_count=2, name='+'),
            Instruction(lambda a, b: a-b, param_count=2, name='-'),
            Instruction(lambda a, b: a*b, param_count=2, name='*'),
            Instruction(lambda a, b: (a/b) if b else 0.0, param_count=2, name='/'),
        ],
        'size': 50
    },
    'monitor': {
        'report': 'brief+local+time_delta',
        'summary': 'status+brief+best_phenome',
        'limits': { 'iterations': 50 }
    },
}
```

ECJ parameter file for GP

```
parent.0 = ec/ec.params

eval.problem = dower.gp.Regression2
eval.problem.data = ec.app.regression.RegressionData
eval.problem.size = 20

state = ec.simple.SimpleEvolutionState
init = ec.gp.GPInitializer
finish = ec.simple.SimpleFinisher
exch = ec.simple.SimpleExchanger
eval = ec.simple.SimpleEvaluator
stat = ec.gp.koza.KozaStatistics
generations = 50
pop = ec.Population
```

```
pop.subpops = 1
pop.subpop.0 = ec.Subpopulation
pop.subpop.0.size = 50
pop.subpop.0.species = ec.gp.GPSpecies
pop.subpop.0.species.fitness = ec.gp.koza.KozaFitness
pop.subpop.0.species.ind = ec.gp.GPIndividual
pop.subpop.0.duplicate-retries = 0
pop.subpop.0.species.ind.numtrees = 1
pop.subpop.0.species.ind.tree.0 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.0.tc = tc0

pop.subpop.0.species.pipe = ec.breed.MultiBreedingPipeline
pop.subpop.0.species.pipe.generate-max = false
pop.subpop.0.species.pipe.num-sources = 3
pop.subpop.0.species.pipe.source.0 = ec.gp.koza.CrossoverPipeline
pop.subpop.0.species.pipe.source.0.prob = 0.9
pop.subpop.0.species.pipe.source.1 = ec.gp.koza.MutationPipeline
pop.subpop.0.species.pipe.source.1.prob = 0.02
pop.subpop.0.species.pipe.source.2 = ec.breed.ReproductionPipeline
pop.subpop.0.species.pipe.source.2.prob = 0.08

breed = ec.simple.SimpleBreeder
breed.reproduce.source.0 = ec.select.TournamentSelection

gp.koza.xover.source.0 = ec.select.TournamentSelection
gp.koza.xover.source.1 = same
gp.koza.xover.ns.0 = ec.gp.koza.KozaNodeSelector
gp.koza.xover.ns.1 = same
gp.koza.xover.maxdepth = 17
gp.koza.xover.tries = 1

gp.koza.mutate.source.0 = ec.select.TournamentSelection
gp.koza.mutate.ns.0 = ec.gp.koza.KozaNodeSelector
gp.koza.mutate.build.0 = ec.gp.koza.GrowBuilder
gp.koza.mutate.maxdepth = 17
gp.koza.mutate.tries = 1

select.tournament.size = 7

gp.koza.grow.min-depth = 15
gp.koza.grow.max-depth = 15

gp.problem.stack = ec.gp.ADFStack
gp.adf-stack.context = ec.gp.ADFContext

gp.koza.ns.terminals = 0.1
gp.koza.ns.nonterminals = 0.9
gp.koza.ns.root = 0.0

gp.fs.size = 1
gp.fs.0 = ec.gp.GPFunctionSet
gp.fs.0.name = f0
gp.fs.0.size = 5
gp.fs.0.func.0 = ec.app.regression.func.X
gp.fs.0.func.0.nc = nc0
gp.fs.0.func.1 = ec.app.regression.func.Add
gp.fs.0.func.1.nc = nc2
gp.fs.0.func.2 = ec.app.regression.func.Mul
```

```
gp.fs.0.func.2.nc = nc2
gp.fs.0.func.3 = ec.app.regression.func.Sub
gp.fs.0.func.3.nc = nc2
gp.fs.0.func.4 = ec.app.regression.func.Div
gp.fs.0.func.4.nc = nc2

gp.type.a.size = 1
gp.type.a.0.name = nil

gp.tc.size = 1
gp.tc.0 = ec.gp.GPTreeConstraints
gp.tc.0.name = tc0
gp.tc.0.fset = f0
gp.tc.0.returns = nil

gp.tc.0.init = ec.gp.koza.FullBuilder
gp.koza.full.min-depth = 7
gp.koza.full.max-depth = 7

gp.nc.size = 3

gp.nc.0 = ec.gp.GPNodeConstraints
gp.nc.0.name = nc0
gp.nc.0.returns = nil
gp.nc.0.size = 0

gp.nc.1 = ec.gp.GPNodeConstraints
gp.nc.1.name = nc1
gp.nc.1.returns = nil
gp.nc.1.size = 1
gp.nc.1.child.0 = nil

gp.nc.2 = ec.gp.GPNodeConstraints
gp.nc.2.name = nc2
gp.nc.2.returns = nil
gp.nc.2.size = 2
gp.nc.2.child.0 = nil
gp.nc.2.child.1 = nil
```

F.6 Steady State Genetic Algorithms

esec configuration file for SSGA

```
import esec.landscape.real

SYSTEM_DEFINITION = r'''
FROM random_binary(length=64) SELECT (100) population
YIELD population

BEGIN generation_equivalent
  REPEAT 100
    FROM population SELECT 2 parents, rest \
        USING binary_tournament(without_replacement)
    FROM parents SELECT offspring \
        USING crossover(per_pair_rate=0.98, two_children), \
              mutate_bitflip(per_gene_rate=1/64)
```

```
    FROM offspring, rest SELECT population
  END
  YIELD population
END
'''

config = {
    'system': { 'definition': SYSTEM_DEFINITION },
    'landscape': { 'class': esec.landscape.binary.OneMax, 'parameters': 64 },
    'monitor': {
        'report': 'brief+local+time_delta',
        'summary': 'status+brief+best_genome',
        'limits': { 'iterations': 100 }
    },
}
```

ECJ parameter file for SSGA

```
parent.0 = ec/ec.params

eval.problem = dower.evaluators.OneMax

state = ec.steadystate.SteadyStateEvolutionState
init = ec.simple.SimpleInitializer
finish = ec.simple.SimpleFinisher
exch = ec.simple.SimpleExchanger
eval = ec.steadystate.SteadyStateEvaluator
stat = ec.simple.SimpleStatistics
breed = ec.steadystate.SteadyStateBreeder
generations = 100
quit-on-run-complete = true
pop = ec.Population
pop.subpops = 1
pop.subpop.0 = ec.Subpopulation
pop.subpop.0.duplicate-retries = 0
pop.subpop.0.species = ec.vector.VectorSpecies
pop.subpop.0.species.pipe = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.source.0 = ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.source.1 = same

select.tournament.size = 2

pop.subpop.0.species.fitness = ec.simple.SimpleFitness
pop.subpop.0.species.ind = ec.vector.BitVectorIndividual

pop.subpop.0.species.genome-size = 64
pop.subpop.0.species.crossover-type = one
pop.subpop.0.species.crossover-likelihood = 0.98
pop.subpop.0.species.mutation-prob = 0.015625

steady.deselector.0 = ec.select.TournamentSelection
steady.deselector.0.size = 1
steady.deselector.0.pick-worst = true

pop.subpop.0.size = 100
```

FakeEALib program for SSGA

```
using System;
using System.Linq;
using FakeEALib;

namespace FakeEALib.Experiments {
    class SteadyStateGeneticAlgorithms {
        public static void Run() {
            var population = new Populations.Binary(64);
            var parents = new Populations.Binary(64);
            population.Evaluator = new Evaluators.Binary.OneMax();
            population.Sort.Add(new Sorters.FitnessProportional());
            parents.Select.Add(new Selectors.Clone());
            parents.Variation.Add(new Operators.Crossover(1, 0.98));
            parents.Variation.Add(new Operators.MutateBitflip(1.0, 1.0 / 64));
            parents.VariationMode = Operators.VariationMode.Replace;

            population.CreateRandom(100);

            population.Evaluate();
            Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
            Console.WriteLine("{0} | {1} | {2} | {3}",
                0, population.Best.Fitness, population.MeanFitness, population.Worst.Fitness);

            for (int iteration = 1; iteration <= 50; ++iteration) {
                for (int step = 0; step < 100; ++step) {
                    population.Step();
                    parents.Clear();
                    parents.AddRange(population.Take(2));
                    parents.Step();
                    population.RemoveAt(0);
                    population.RemoveAt(0);
                    population.AddRange(parents);
                    population.Evaluate();
                }

                Console.WriteLine("{0} | {1} | {2} | {3}",
                    iteration, population.Best.Fitness, population.MeanFitness, population.»
    «Worst.Fitness);
            }

            Console.WriteLine("Best: {0} [{1}]", population.Best.Fitness, population.Best.»
    «ToString());
        }
    }
}
```

Full C# program for SSGA

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FullAlgorithms {
    static class SteadyStateGeneticAlgorithm {
        private class Individual : IComparable<Individual> {
```

```
        public readonly bool[] Genome;
        public double Fitness;

        public Individual(int length) {
            Genome = new bool[length];
        }

        public int CompareTo(Individual other) {
            return other.Fitness.CompareTo(Fitness);
        }

        public override string ToString() {
            return String.Join(", ", Genome);
        }
    }

    private static void PrintSummary(int iteration, List<Individual> indivs) {
        indivs.Sort();
        double total = 0;
        foreach (var indiv in indivs) {
            total += indiv.Fitness;
        }
        double average = total / indivs.Count;

        Console.WriteLine("{0} | {1} | {2} | {3}",
            iteration, indivs[0].Fitness, average, indivs[indivs.Count - 1].Fitness);
    }

    public static void Run() {
        var rnd = new Random();
        var population = new List<Individual>();

        for (int i = 0; i < 100; ++i) {
            var indiv = new Individual(64);
            for (int j = 0; j < indiv.Genome.Length; ++j) {
                indiv.Genome[j] = rnd.NextDouble() < 0.5;
            }
            population.Add(indiv);
        }

        foreach (var indiv in population) {
            double total = 0;
            for (int j = 0; j < indiv.Genome.Length; ++j) {
                total += indiv.Genome[j] ? 1.0 : 0.0;
            }
            indiv.Fitness = total;
        }

        Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
        PrintSummary(0, population);

        for (int iteration = 1; iteration <= 100; ++iteration) {
            double fitnessTotal = 0;
            foreach (var indiv in population) {
                fitnessTotal += indiv.Fitness;
            }

            for (int repeat = 0; repeat < 100; ++repeat) {
```

```
        var parents = new List<Individual>();

        while (parents.Count < 2) {
            var prob = rnd.NextDouble() * fitnessTotal;

            for (int i = 0; i < population.Count; ++i) {
                prob -= population[i].Fitness;
                if (prob <= 0) {
                    parents.Add(population[i]);
                    fitnessTotal -= population[i].Fitness;
                    population.RemoveAt(i);
                    break;
                }
            }
        }

        var p1 = parents[0];
        var p2 = parents[1];

        int cut = rnd.Next(1, p1.Genome.Length - 1);
        var c1 = new Individual(p1.Genome.Length);
        var c2 = new Individual(p1.Genome.Length);

        for (int j = 0; j < p1.Genome.Length; ++j) {
            if (j < cut) {
                c1.Genome[j] = p1.Genome[j];
                c2.Genome[j] = p2.Genome[j];
            } else {
                c2.Genome[j] = p1.Genome[j];
                c1.Genome[j] = p2.Genome[j];
            }

            if (rnd.NextDouble() < (1.0 / 64.0)) {
                c1.Genome[j] = !c1.Genome[j];
            }
            if (rnd.NextDouble() < (1.0 / 64.0)) {
                c2.Genome[j] = !c2.Genome[j];
            }
        }

        double total = 0;
        for (int j = 0; j < c1.Genome.Length; ++j) {
            total += c1.Genome[j] ? 1.0 : 0.0;
        }
        c1.Fitness = total;
        total = 0;
        for (int j = 0; j < c2.Genome.Length; ++j) {
            total += c2.Genome[j] ? 1.0 : 0.0;
        }
        c2.Fitness = total;

        population.Add(c1);
        population.Add(c2);
        population.Sort();
        fitnessTotal += c1.Fitness + c2.Fitness;
    }

    PrintSummary(iteration, population);
```

```
            }

            Console.WriteLine("Best: {0} [{1}]", population[0].Fitness, population[0].»
    «ToString());
        }
    }
}
```

F.7 Particle Swarm Optimisation

esec configuration file for PSO

```
import esec.landscape.real

def random_particle(length=10, low_limit=0, high_limit=1):
    from esec.context import rand, context
    for indiv in context['random_real'](length=length, lowest=low_limit, highest=high_limit):
        indiv.strategy = [0] * int(length)
        yield indiv

def update_velocity(_source, global_best=None, low_limit=-100, high_limit=100,
                    c1=2, c2=2):
    from esec.context import rand
    g_best = global_best[0]
    for indiv, p_best in _source:
        new_velocity = []
        for (x, v, p_i, p_g) in zip(indiv, indiv.strategy, p_best, g_best):
            r1 = rand.random()
            r2 = rand.random()

            new_v = v + c1*r1*(p_i-x) + c2*r2*(p_g-x)

            if new_v < low_limit:
                new_velocity.append(low_limit)
            elif high_limit < new_v:
                new_velocity.append(high_limit)
            else:
                new_velocity.append(new_v)

        yield type(indiv)(indiv.genome, indiv, strategy=new_velocity)

def update_position(_source, low_limit=-100, high_limit=100, bounce=False):
    for indiv in _source:
        new_position = []
        new_velocity = []
        for x, v in zip(indiv.genome, indiv.strategy):
            new_x = x + v
            new_v = v
            if bounce:
                if new_x < low_limit:
                    new_x = 2 * low_limit - new_x
                    new_v = -v
                elif high_limit < new_x:
                    new_x = 2 * high_limit - new_x
                    new_v = -v
            new_position.append(new_x)
```

```
            new_velocity.append(new_v)

        yield type(indiv)(new_position, indiv, strategy=new_velocity)

SYSTEM_DEFINITION = r'''
FROM random_particle(length=2, low_limit=-5, high_limit=5) SELECT 50 swarm
FROM swarm SELECT p_bests
YIELD swarm

BEGIN iteration
  FROM swarm SELECT 1 g_best USING best
  JOIN swarm, p_bests INTO particles_with_pbest
  FROM particles_with_pbest SELECT swarm USING \
       update_velocity(global_best=g_best, low_limit=-100, high_limit=100), \
       update_position(low_limit=-5, high_limit=5, bounce)
  YIELD swarm
END
'''

config = {
    'system': {
        'definition': SYSTEM_DEFINITION,
        'random_particle': random_particle,
        'update_velocity': update_velocity,
        'update_position': update_position,
    },
    'landscape': { 'class': esec.landscape.real.Sphere, 'parameters': 2 },
    'monitor': {
        'report': 'brief+local+time_delta',
        'primary': 'swarm',
        'summary': 'status+brief+best_genome',
        'limits': { 'iterations': 50 }
    },
}
```

ECJ parameter file for PSO

```
parent.0 = ec/ec.params

eval.problem = ec.app.ecsuite.ECSuite
eval.problem.type = sphere

state = ec.simple.SimpleEvolutionState
init = ec.simple.SimpleInitializer
finish = ec.simple.SimpleFinisher
exch = ec.simple.SimpleExchanger
breed = dower.pso.PSOBreeder
eval = ec.simple.SimpleEvaluator
stat = ec.simple.SimpleStatistics
generations = 50
quit-on-run-complete = true
pop = ec.Population
pop.subpops = 1
pop.subpop.0 = ec.pso.PSOSubpopulation
pop.subpop.0.duplicate-retries = 2
pop.subpop.0.species = ec.vector.FloatVectorSpecies
pop.subpop.0.species.fitness = ec.simple.SimpleFitness
#pop.subpop.0.species.pipe = ec.vector.breed.VectorMutationPipeline
```

```
#pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline
#pop.subpop.0.species.pipe.source.0.source.0 = ec.select.TournamentSelection
#pop.subpop.0.species.pipe.source.0.source.1 = same

pop.subpop.0.species.ind = ec.vector.DoubleVectorIndividual

pop.subpop.0.species.min-gene = -5
pop.subpop.0.species.max-gene = 5
pop.subpop.0.species.genome-size = 2

#select.tournament.size = 2
#pop.subpop.0.species.mutation-prob = 0.005
#pop.subpop.0.species.crossover-type = one

pop.subpop.0.size = 50
#pop.subpop.0.neighborhood-size = 50
pop.subpop.0.clamp = true
pop.subpop.0.initial-velocity-scale = 0
#pop.subpop.0.velocity-multiplier = 1


package dower.pso;

import ec.Breeder;
import ec.EvolutionState;
import ec.Population;
import ec.pso.PSOSubpopulation;
import ec.util.Parameter;
import ec.vector.DoubleVectorIndividual;

public class PSOBreeder extends Breeder
    {
    public void setup(EvolutionState state, Parameter base)
        {
        // intentionally empty
        }

    public Population breedPopulation(EvolutionState state)
        {
        PSOSubpopulation subpop = (PSOSubpopulation) state.population.subpops[0];

        // update bests
        assignPersonalBests(subpop);
        assignGlobalBest(subpop);

        // make a temporary copy of locations so we can modify the current location on the fly
        DoubleVectorIndividual[] tempClone = new DoubleVectorIndividual[subpop.individuals.»
    «length];
        System.arraycopy(subpop.individuals, 0, tempClone, 0, subpop.individuals.length);

        // update particles
        for (int i = 0; i < subpop.individuals.length; i++)
            {
            DoubleVectorIndividual ind = (DoubleVectorIndividual)subpop.individuals[i];
            DoubleVectorIndividual prevInd = (DoubleVectorIndividual)subpop.»
    «previousIndividuals[i];
            // the individual's personal best
            DoubleVectorIndividual pBest = (DoubleVectorIndividual)subpop.personalBests[i];
```

```
        // the individuals's global best
        DoubleVectorIndividual gBest = (DoubleVectorIndividual)subpop.globalBest;

        // calculate update for each dimension in the genome
        for (int j = 0; j < ind.genomeLength(); j++)
            {
            double velocity = ind.genome[j] - prevInd.genome[j];
            double pDelta = pBest.genome[j] - ind.genome[j];                    // »
«difference to personal best
            double gDelta = gBest.genome[j] - ind.genome[j];                    // »
«difference to global best
            double pWeight = state.random[0].nextDouble();                      // »
«weight for personal best
            double gWeight = state.random[0].nextDouble();                      // »
«weight for global best
            double newDelta = (velocity + 2.0*pWeight*pDelta + 2.0*gWeight*gDelta);

            if (newDelta < -100.0)
                newDelta = -100.0;
            else if (newDelta > 100.0)
                newDelta = 100.0;

            ind.genome[j] += newDelta;
            }

        if (subpop.clampRange)
            ind.clamp();
        }

    // update previous locations
    subpop.previousIndividuals = tempClone;

    return state.population;
    }

public void assignPersonalBests(PSOSubpopulation subpop)
    {
    for (int i = 0; i < subpop.personalBests.length; i++)
        if ((subpop.personalBests[i] == null) || subpop.individuals[i].fitness.betterThan»
«(subpop.personalBests[i].fitness))
            subpop.personalBests[i] = (DoubleVectorIndividual)subpop.individuals[i].clone»
«();
    }

public void assignGlobalBest(PSOSubpopulation subpop)
    {
    DoubleVectorIndividual globalBest = subpop.globalBest;
    for (int i = 0; i < subpop.individuals.length; i++)
        {
        DoubleVectorIndividual ind = (DoubleVectorIndividual)subpop.individuals[i];
        if ((globalBest == null) || ind.fitness.betterThan(globalBest.fitness))
            globalBest = ind;
        }
    if (globalBest != subpop.globalBest)
        subpop.globalBest = (DoubleVectorIndividual)globalBest.clone();
    }
}
```

FakeEALib program for PSO

```
using System;
using System.Collections.Generic;
using FakeEALib;

namespace FakeEALib.Experiments {
    class ParticleSwarmOptimisation {
        public static void Run() {
            var swarm = new Populations.RealWithStrategy(2, -5.0, 5.0, 0.0, 0.0);
            var pBests = new Populations.RealWithStrategy(2, -5.0, 5.0, 0.0, 0.0);
            swarm.Evaluator = new Evaluators.Real.Sphere();

            swarm.Variation.Add(new UpdateVelocity(-100.0, 100.0, pBests));
            swarm.Variation.Add(new UpdatePosition(-5.0, 5.0, true));
            swarm.VariationMode = Operators.VariationMode.Replace;

            swarm.CreateRandom(50);
            swarm.Evaluate();
            pBests.AddClonedRange(swarm);

            swarm.Evaluate();
            Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
            Console.WriteLine("{0} | {1} | {2} | {3}",
                0, swarm.Best.Fitness, swarm.MeanFitness, swarm.Worst.Fitness);

            for (int iteration = 1; iteration <= 50; ++iteration) {
                swarm.Step();
                swarm.Evaluate();

                for (int i = 0; i < swarm.Count; ++i) {
                    if (swarm[i].Fitness > pBests[i].Fitness) {
                        pBests[i] = swarm[i].Clone();
                    }
                }

                Console.WriteLine("{0} | {1} | {2} | {3}",
                    iteration, swarm.Best.Fitness, swarm.MeanFitness, swarm.Worst.Fitness);
            }

            Console.WriteLine("Best: {0} [{1}]", swarm.Best.Fitness, swarm.Best.ToString());
        }
    }

    class UpdatePosition : Operators.IOperator {
        public double LowLimit { get; private set; }
        public double HighLimit { get; private set; }
        public bool Bounce { get; private set; }

        public UpdatePosition(double lowLimit, double highLimit, bool bounce) {
            LowLimit = lowLimit;
            HighLimit = highLimit;
            Bounce = bounce;
        }

        public void Vary(IList<Populations.IIndividual> source, int index) {
            var indiv = (Populations.IRealIndividualWithStrategy)source[index];

            for (int i = 0; i < indiv.Count; ++i) {
```

```
                indiv[i] += indiv.Strategy[i];
                if (Bounce && indiv[i] < LowLimit) {
                    indiv[i] = 2 * LowLimit - indiv[i];
                    indiv.Strategy[i] = -indiv.Strategy[i];
                } else if (Bounce && indiv[i] > HighLimit) {
                    indiv[i] = 2 * HighLimit - indiv[i];
                    indiv.Strategy[i] = -indiv.Strategy[i];
                }
            }
        }
    }

    class UpdateVelocity : Operators.IOperator {
        public double LowLimit { get; private set; }
        public double HighLimit { get; private set; }
        public Populations.IPopulation PersonalBests { get; private set; }

        public UpdateVelocity(double lowLimit, double highLimit, Populations.IPopulation »
    «pBests) {
            LowLimit = lowLimit;
            HighLimit = highLimit;
            PersonalBests = pBests;
        }

        public void Vary(IList<Populations.IIndividual> source, int index) {
            var indiv = (Populations.IRealIndividualWithStrategy)source[index];
            var pBest = (Populations.IRealIndividualWithStrategy)PersonalBests[index];
            var gBest = (Populations.IRealIndividualWithStrategy)PersonalBests.Best;

            for (int i = 0; i < indiv.Count; ++i) {
                indiv.Strategy[i] += 2 * Random.Uniform(0, 1) * (pBest[i] - indiv[i]) +
                                     2 * Random.Uniform(0, 1) * (gBest[i] - indiv[i]);
                if (indiv.Strategy[i] < LowLimit) {
                    indiv.Strategy[i] = LowLimit;
                } else if (indiv.Strategy[i] > HighLimit) {
                    indiv.Strategy[i] = HighLimit;
                }
            }
        }
    }
}
```

Full C# program for PSO

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FullAlgorithms {
    static class ParticleSwarmOptimisation {
        private class Individual : IComparable<Individual> {
            public readonly double[] Genome;
            public readonly double[] Velocity;
            public double Fitness;

            public Individual(int length) {
                Genome = new double[length];
```

```
            Velocity = new double[length];
        }

        public int CompareTo(Individual other) {
            return other.Fitness.CompareTo(Fitness);
        }

        public override string ToString() {
            return String.Join(", ", Genome);
        }

        public Individual Clone() {
            var indiv = new Individual(Genome.Length);
            Genome.CopyTo(indiv.Genome, 0);
            Velocity.CopyTo(indiv.Velocity, 0);
            indiv.Fitness = Fitness;
            return indiv;
        }
    }

    private static void PrintSummary(int iteration, List<Individual> indivs) {
        indivs.Sort();
        double total = 0;
        foreach (var indiv in indivs) {
            total += indiv.Fitness;
        }
        double average = total / indivs.Count;

        Console.WriteLine("{0} | {1} | {2} | {3}",
            iteration, indivs[0].Fitness, average, indivs[indivs.Count - 1].Fitness);
    }

    public static void Run() {
        var rnd = new Random();
        var swarm = new List<Individual>();
        var p_bests = new List<Individual>();

        for (int i = 0; i < 50; ++i) {
            var indiv = new Individual(2);
            for (int j = 0; j < indiv.Genome.Length; ++j) {
                indiv.Genome[j] = rnd.NextDouble() * 10.0 - 5.0;
                indiv.Velocity[j] = 0.0;
            }
            swarm.Add(indiv);
            p_bests.Add(indiv);
        }

        foreach (var indiv in swarm) {
            double total = 0;
            for (int j = 0; j < indiv.Genome.Length; ++j) {
                total += indiv.Genome[j] * indiv.Genome[j];
            }
            indiv.Fitness = -total;
        }

        Console.WriteLine("Iter | Best Fit | Mean Fit | Worst Fit");
        PrintSummary(0, swarm);
```

```
            for (int iteration = 1; iteration <= 50; ++iteration) {
                var g_best = swarm[0].Clone();

                for (int i = 0; i < swarm.Count; ++i) {
                    var pos = swarm[i].Genome;
                    var vel = swarm[i].Velocity;
                    var p_i = p_bests[i].Genome;
                    var p_g = g_best.Genome;
                    for (int j = 0; j < pos.Length; ++j) {
                        double newV = vel[j] + 2.0 * rnd.NextDouble() * (p_i[j] - pos[j]) +
                            2.0 * rnd.NextDouble() * (p_g[j] - pos[j]);

                        if (newV < -100) {
                            newV = -100;
                        } else if (newV > 100) {
                            newV = 100;
                        }

                        pos[j] += newV;

                        if (pos[j] < -5) {
                            pos[j] = -10 - pos[j];
                            newV = -newV;
                        } else if (pos[j] > 5) {
                            pos[j] = 10 - pos[j];
                            newV = -newV;
                        }

                        vel[j] = newV;
                    }
                }

                for (int i = 0; i < swarm.Count; ++i) {
                    var indiv = swarm[i];
                    double total = 0;
                    for (int j = 0; j < indiv.Genome.Length; ++j) {
                        total += indiv.Genome[j] * indiv.Genome[j];
                    }
                    indiv.Fitness = -total;

                    if (indiv.Fitness > p_bests[i].Fitness) {
                        p_bests[i] = indiv.Clone();
                    }
                }

                PrintSummary(iteration, swarm);
            }

            Console.WriteLine("Best: {0} [{1}]", swarm[0].Fitness, swarm[0].ToString());
        }
    }
}
```

Glossary

blackboard
: A software architecture using a central storage repository that is freely accessible from any component.

block (ESDL)
: A sequence of statements representing one iteration of the algorithm.

C++ AMP
: C++ Accelerated Massive Parallelism

credit assignment
: Determining the fitness of individuals that were evaluated as components of a joined individual.

CSV
: Comma Separated Value. Typically refers to the file format used for storing tables as plain text.

DE
: Differential Evolution

DSL
: Domain-Specific Language. A language of limited expressiveness that provides greater fluency for a specific field or subject.

EA
: Evolutionary Algorithm

EC
: Evolutionary Computation

EP
: Evolutionary Programming

ES
: Evolution Strategies

ESDL
: Evolutionary System Definition Language

evaluator (ESDL)
: The calculation that determines the fitness of an individual.

fitness
: The suitability of a set of input values for the problem to which they are applied.

framework
: A program that may be extended by a developer. See also, *library*.

GA
: Genetic Algorithms

GPU
: Graphical Processing Unit

group (ESDL)
: A list containing a subset of the individuals existing in a given search space.

GUI
: Graphical User Interface

HTML
: HyperText Markup Language

individual (ESDL)
: A single set of input values for a problem.

joined individual (ESDL)
: An association between multiple individuals, conceptualised as a distinct individual.

library
: A collection of code that can be used to create new programs. See also, *framework*.

merge (ESDL)
: Combine multiple groups into a single group containing all the individuals. Equivalent to concatenation of lists.

MIMD
: Multiple-Instruction Multiple-Data. Most computers with multiple processors or cores use this model.

OOP
: Object-Oriented Programming

operator (ESDL)
: A composable element providing joining, filtering, selection or variation of groups.

partition (ESDL)
: Separate a group by taking groups of individuals from the start.

problem
: A mapping from input values to output values, representing the system or function that is to be optimised or 'solved.'

SIMD

Single-Instruction Multiple-Data. Most GPUs are based on SIMD processors.

species

A class of individuals, typically specifying the representation and valid operations.

SQL

Structured Query Language

stream (ESDL)

A temporary sequence of individuals used to pass them between operators.

XML

eXtensible Markup Language