DEVELOPING AN APPLICATION FOR EVOLUTIONARY SEARCH FOR COMPUTATIONAL MODELS OF CELLULAR DEVELOPMENT

by

Nicolas Scott Cornia

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2015

© 2015
Nicolas Scott Cornia
ALL RIGHTS RESERVED

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Nicolas Scott Cornia

Thesis Title: Developing an Application for Evolutionary Search for Computational Models of Cellular Development

Date of Final Oral Examination: 20 February 2015

The following individuals read and discussed the thesis submitted by student Nicolas Scott Cornia, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Timothy Andersen, Ph.D.	Chair, Supervisory Committee
Jeff Habig, Ph.D.	Member, Supervisory Committee
Elena A. Sherman, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Timothy Andersen, Ph.D., Chair of the Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

ABSTRACT

VPEvolve is a free and open source application that utilizes a Visual Programming Environment (VPE) for the setup of the Genetic Algorithm (GA), for optimization of computational models. Specifically, the User Interface uses connected glyphs to represent the genetic operators of mutation, reproduction, fitness and selection. These glyphs give the user an intuitive way to set the parameters for the GA, and better visualization of the population's flow through these operators.

VPEvolve is currently being developed alongside research being done in Biocomputing to create models of cellular regeneration based on the regenerative properties of Planaria or flatworms. Since these models are difficult to produce by hand, GAs can be particularly useful to facilitate the process of model creation and validation. VPEvolve is a client-side application that allows the user to setup the parameters for the GA, runs a search using the GA, utilizes a modeling platform, such as CellSim (Cellular Simulator), to perform simulations, evaluates the fitness of each individual with user-defined fitness evaluators and presents the fitness values of the individuals in a population to the user as the evolutionary search is performed.

TABLE OF CONTENTS

ABSTRACT iv

LIST OF TABLES vii

LIST OF FIGURES viii

1 Introduction 1

2 Background 8

2.1 Computational Modeling 8

2.2 Biological Modeling 10

2.3 Evolutionary Search 11

2.4 Visual Programming Languages 12

2.5 Previous Work 14

3 VPEvolve 16

3.1 Motivation 16

3.2 Design Features 18
3.2.1 Visual Programming Environment 19
3.2.2 Configuration File 21
3.2.3 Genetic Algorithm Parameter Setup 21
3.2.4 Evolutionary Search Status View 24
3.2.5 Console Mode 26
3.2.6 Integration with a Modeling Platform Through Middleman Software 29
3.2.7 Evolutionary Search Checkpointing 30
3.3 Architecture 31
3.3.1 Modular Design 31
3.3.2 Implementation of Genetic Operators 33
4 Results 36
4.1 Comparison 36
4.2 Experiments 43
5 Conclusions 46
5.1 Future Directions 46
REFERENCES 48

LIST OF TABLES

4.1 Average Time Elapsed for One Generation in Seconds (over 1000 runs)...42

LIST OF FIGURES

1.1 Planaria 6-cut Experiment 5

2.1 Multi-agent Computational Model of a DNA Chain 10

3.1 Cantata VPE 17

3.2 VPEvolve VPE 19

3.3 Genetic Operator Parameter Setup Window 22

3.4 XML Editor for Individuals 23

3.5 Custom Python Fitness Evaluator Editor 25

3.6 Evolutionary Search Status Window 26

3.7 VPEvolve Evolutionary Search in GUI 27

3.8 VPEvolve Evolutionary Search in Console Mode 28

3.9 VPEvovle Client-Server Individual Processing 30

3.10 VPEvolve Module Diagram 32

3.11 VPEvolve Class Diagram 34

4.1 CSGA GA Parameter Setup View (Replacement Operator) 37

4.2 CSGA Translated Evolutionary Search 38

4.3 CSGA Console Mode 39

4.4 Client-Server Individual Processing 40

4.5 Wild-Type Worm 43

4.6 Planarian Cellular Models 43

4.7 Single Lateral Cut Experiment 44

4.8 “Shark-bite” Cut Target Worm 45

4.9 1.0 Fitness Worm Produced by VPEvovle 45

CHAPTER 1

INTRODUCTION

The use of advanced laboratory techniques, such as high-throughput screening, cause an accumulation of massive amounts of data describing complex systems. This data is exponentially increasing due to the advancement of these laboratory techniques and has the tremendous potential to accelerate and aid scientific inquiry, and reveal insights into how complex systems operate in a way that has heretofore been impossible. However, there are significant challenges that must be overcome in order to realize this potential. The volume and complexity of the data is so great that an unaided human has little chance to assimilate it to the degree necessary to reliably deduce higher-order, governing principles. As an example of this challenge, robotic microarray systems are used for pharmaceutical drug screenings to test thousands of molecules. The test data are then accumulated for analysis. Similar experiments conducted by an unaided human would be impossible to accomplish in the same amount of time.

Software tools can provide a way to overcome the challenges with data. Software tools can be used by researchers to be more productive with their work by aiding them with identifying correlations and causal factors, analyzing and visualizing data, and suggesting further experiments. For example, calculation-based tools can speed up the process of analyzing data and creating summaries; clustering tools can find

new, meaningful groupings of data; and visualization tools can present data in ways that make it easier to understand. In addition to these types of software tools, there are also more advanced tools that can aid researchers in simulating complex systems. For example, software simulations can be run using models that range from single-cell biochemical reactions to multi-cellular environments that simulate tissues, organs or even whole organisms. Such simulations have the potential to be extremely powerful tools for wet-bench researchers, enabling them to develop and validate models that are hypothesized from experimental data, and even to use such models as the predictive basis for further experimentation.

For example, a perfect computational model of the influenza virus, one that reacted just as the virus would to every possible perturbation, could be used to identify drug targets for inhibiting the reproduction of the virus in the body, or to prevent infection. However, developing models that fully explain how these types of systems work is an imposingly difficult task. Even relatively simple biological systems can be astoundingly complex in their interactions, with highly non-linear responses, and complex self-regulatory feedback loops. Moreover, these systems are difficult to elucidate, requiring large amounts of experimental data gathered under varying conditions in order to fully characterize them. This is particularly true when trying to model whole cells or groups of cells, where there are elaborate biochemical pathways and interactions, including processes that maintain homeostasis, extra-cellular signaling, and cell division.

We believe that data driven software tools can facilitate model derivation of complex systems. These tools can help researchers derive models by leveraging the massive amounts of data produced from experiments and simulations, performing what has traditionally been the work of humans (i.e. developing theories and model

derivation), and automate this process. Automated model derivation has the potential to clarify currently unexplained problems, unveil unknown processes, and lead to new biological discoveries.

Embryogenesis is an area of research where automated model derivation could help clarify the mechanism which describes how an embryo forms and develops. This process begins as a single cell, which develops into separate lineages over a number of cellular divisions. These groups continue to develop and eventually become specific tissues and organs such as the central nervous system, digestive system, and musculature. Throughout this process each cell must determine whether to maintain its pluripotent potential or differentiate into particular cell type using information location and neighbors based upon short and long-range cellular communication [11].

Many of the mechanisms that organisms use to guide the process of embryogenesis are used in a similar manner to manage the process of cellular regeneration (i.e. tissue repair in response to damage). Planaria, more commonly known as flatworms, are model organisms in this area of research. Planaria have the impressive ability to regenerate large amounts of damaged or missing cells and tissues, due in part to the abundant stem cell population throughout the organisms [1]. Their regeneration potential is impressive considering that Planaria are complex organisms possessing bilateral symmetry, musculature, an intestinal tract, and a central nervous system including a brain [21, 19]. Remarkably, an excised piece of a flatworm no larger than 0.4% the size of the whole organism can regenerate and regrow into a fully-functioning flatworm including all of the differentiated tissue removed during the excision [14]. This is demonstrated in Figure 1.1, where an intact worm has been dissected into seven independent pieces as a result of six lateral cuts. Between a period of one and two weeks, each fragment independently responds to the perturbation and reshapes

its morphology to produce smaller replicas of the adult worm that can then grow into larger versions akin to the original worm. An astounding feature of this regeneration is that the worm appears to maintain the original head and tail orientation following regeneration, even when the excised region no longer contains either head or tail tissue. These creatures and their impressive regenerative capabilities have fascinated and perplexed biologists for centuries [16]. Despite being the subject of many regenerative experiments, including drug treatments [15, 5], amputations [19], irradiation, and gene ablation or silencing using RNA interference [17, 20, 18, 24, 23],there are major gaps in our understanding of the mechanisms underlying this regenerative potential [2]. Even after hundreds of years and extensive experimentation and research there is still a limited understanding of planarian regeneration and no comprehensive model has been found that can explain more than one or two aspects of this regenerative ability [13].

A number of pathways have been identified that have complementary roles in regeneration. For example, regeneration occurs normally upon elimination of either direct cellular communication (i.e., gap junctions) or neuronal signaling, but not both. Such complementary and complex signaling mechanisms make it challenging to understand regeneration processes and to model their behavior, particular if the models are developed manually. Our team has developed various simple computational regeneration models to explore the strengths and limitations of their use in explaining planarian regeneration. One model uses a molecular concentration gradient extending longitudinally throughout the worm to provide head and tail orientation to the virtual organism. Another model attempts to identify whether or not a head or tail region is present in current worm by essentially responding to a ping sent to the head and tail, respectively. A third model develops polarity at the cellular level based upon

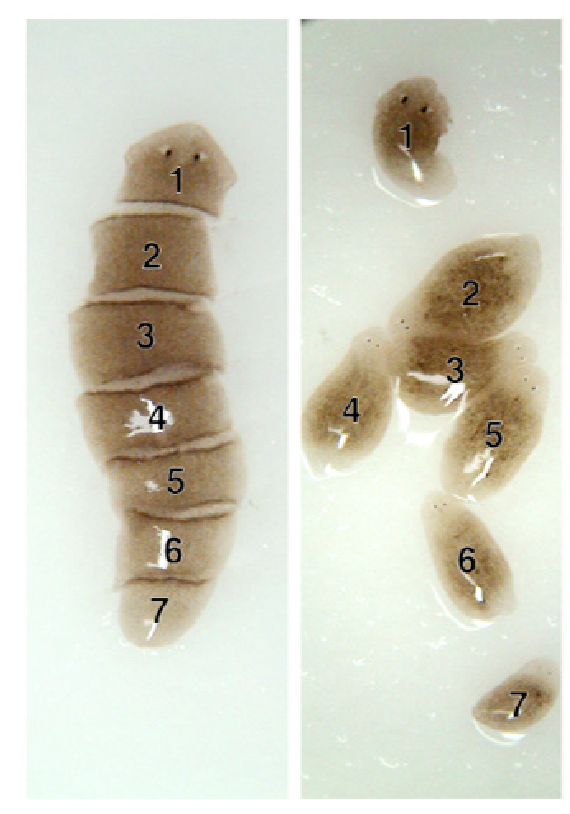


Figure 1.1: Planaria 6-cut Experiment

the orientation of head and tail in the intact worm. Upon excision, it “knows” which side of the cell is oriented towards the head and which side orients towards the tail, much as a bar magnet “knows” the orientation of its poles after being cut into smaller pieces. Each of these models provides valuable information to the virtual organism, but they all have short-comings that limit their ability to respond to a large dataset of experimental outcomes.

Our ultimate goal is to develop a model that can faithfully simulate many, if not all, of the available experiments performed on this organism over the past centuries, with the hope that it can provide predictive capabilities. To aid in this process, we are looking to develop and use automated search to find suitable models. The search

is based upon a powerful agent-based cell modeling platform, CellSim, which allows simulation of multicellular organisms. The current version of this software contains a number of useful features to support this endeavor, which includes a 3-D interface for visualization as well as tools for performing experimental manipulations within the client-server architecture. I have developed an evolutionary search application, VPEvolve, to enable model discovery using an automated search process. The design, implementation, and use of which will be the focus of this thesis.

THESIS STATEMENT

VPEvolve is a software tool that can be used by researchers for automated model derivation and can be an integral part in their experimental research. VPEvolve utilizes a visual programming environment (VPE) that provides a simple and intuitive way to create a genetic algorithm (GA) setup and is implemented in a generic way that allows any problem space to be explored. These features make VPEvolve an applicable tool for researchers in disciplines other than biology.

CHAPTER 2

BACKGROUND

This chapter provides an introduction to computational modeling, the use of modeling in the biological sciences, the role of evolutionary search, and the potential benefits of visual programming environments when developing software tools.

2.1 Computational Modeling

Computational modeling is used to study the behavior of complex systems by utilizing mathematics, physics and computer science, to run computer simulations. Researchers can use the results of simulations to make better predictions about what will happen in real systems that are being studied in response to changing conditions. Therefore, research can be expedited with computational modeling by allowing scientists to create and run thousands of simulated experiments in parallel, in order to identify which wet-bench experiments will be most likely to produce valuable information for the problem being studied.

Furthermore, computational modeling allows experiments to be repeated again exactly the same. The experiments can also be forked at any point, which allows a researcher to change a parameter then proceed with both options or even rewind and

change a parameter. Computational modeling also allow the system to be inspected in detail during the experimentation process, where inspection of a biological system often perturbs its progression. Moreover, operational theories of the model can be derived from these computational experiments.

Computational models are generally complex nonlinear systems, and therefore intuitive, direct, analytical solutions are not readily available. This model complexity means that there are a number of parameters that characterize the system being studied. Therefore, model discovery often requires adjusting the parameters of the system, and studying the differences in the results of the experiments, instead of mathematically deriving an analytical solution to the problem.

Since biological models are often simulating the actions and interactions of autonomous agents, agent-based modeling (ABM) is well suited to create these computational models. ABMs are composed of interacting intelligent agents within an environment. Therefore, they can be used to solve problems that are difficult or impossible for an individual agent or a monolithic system to solve. The goal of an ABM is to search for explanatory insight into the collective behavior of agents, which do not necessarily need to be intelligent, typically in natural systems and obeying simple rules. Figure 2.1 is derived from a computation model of a region of double-stranded DNA. Each atom, listed and colored, is an autonomous agent whose interactions are simulated based upon a set of biochemical rules. Specifically, the bars connecting each atom represent a covenant bond, and the “ladder” rungs of the DNA molecule are held in place by Hydrogen bonds.

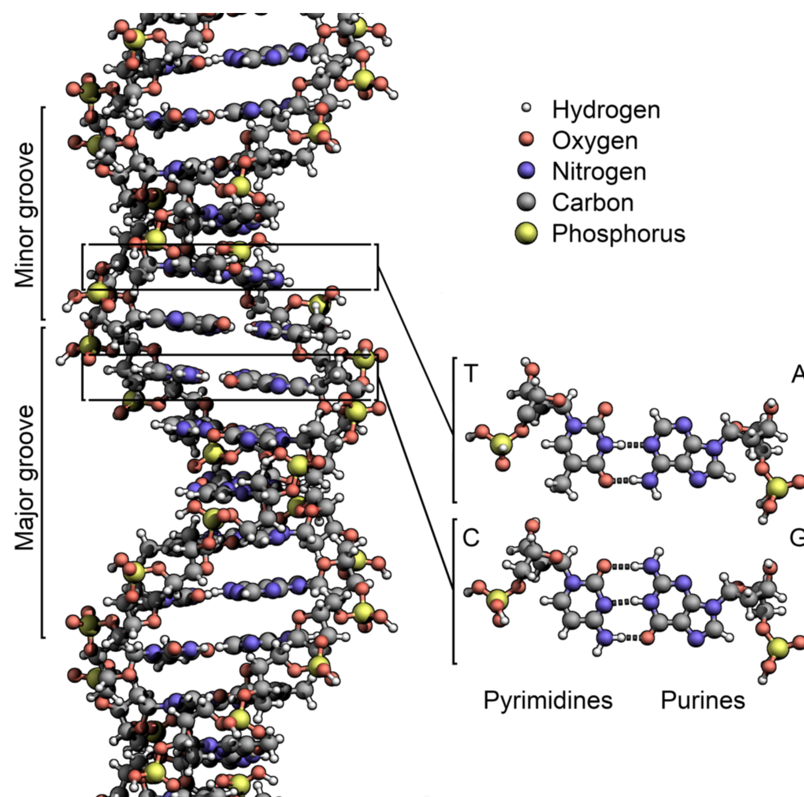


Figure 2.1: Multi-agent Computational Model of a DNA Chain

2.2 Biological Modeling

Multi-cellular systems biology is an emerging field aiming at a mechanical understanding of the processes of physiology, pathology, and development, which is done by studying the multi-scale interactions at the subcellular, cellular and tissue levels. In order to explore and predict a system's behavior and to integrate quantitative data, there is an increasing need for computational simulation.

The aim of multi-scale modeling is to represent the behavior of complex systems across a wide range of spatial and temporal scales. This requires the use of cost-effective and efficient computational techniques. Over the past decades, several multi-scale methods have been developed in other fields of science to meet this requirement.

Recently, a number of modeling and simulation platforms have become available that facilitate the construction and simulation of multi-scale models of multi-cellular systems. There is a range of multi-scale modeling methods that could potentially be employed in systems biology. Shah and Wambaugh have classified these methods as either continuous or discrete methods, based on the strategy the methods use to integrate the various scales [22]. These models simulate cellular behavior in a multi-cellular context. Cellular behavior is modeled with process diagrams using deterministic and/or stochastic model elements.

Creating a model that explains cellular behavior and cellular interaction is complex enough even without introducing the fact that tissues encompass a number of different cell types that behave differently. Moreover, the model definitions themselves for such a system can be complicated since each cellular process needs to be noted and explained. It makes manual attempts intractable.

2.3 Evolutionary Search

Evolutionary search helps automate definition, testing, and modification of these complicated models. Evolutionary searches employ a genetic algorithm (GA), which is a heuristic solution-search or optimization technique, originally motivated by the Darwinian principle of evolution through (genetic) selection. A GA uses an abstract version of evolutionary processes to find solutions to given problems, by operating on a population of artificial individuals. Each individual represents a solution to a problem and has a fitness, a real number which is a measure of how good a solution it is for the particular problem.

The beginning of an evolutionary search starts off with a randomly generated population of individuals. Then to produce a successor population, a GA carries out a process of fitness-based selection and recombination, which creates the next generation. Parent individuals are selected and their descriptions are recombined to produce child individuals during recombination. These child individuals then pass into the successor population. Through this iterative process, a sequence of successive generations evolves and the average fitness of the individuals tends to increase until some stopping criterion is reached. In this way, a GA evolves a best solution to a given problem.

The development and success of GAs have greatly contributed to a wider interest in computational approaches based on natural phenomena and it is now a major strand of the wider field of Computational Intelligence, which encompasses techniques such as Particle Swarm Optimization, Neural Networks, Artificial Immunology and Ant Colony Optimization [10].

2.4 Visual Programming Languages

Visual programming languages (VPL) can provide benefits to understanding the definition of a GA by providing a more intuitive visual indication of the parameter setup of the GA in a graphical user interface (GUI). A VPL is used within a visual programming environment (VPE) where the user can manipulate the program elements visually instead of defining them textually. A VPL has a different convention of programming where tokens are defined in a VPE, using more than one dimension, as any potentially significant object or relationship and expressions are defined as

a combination of one or more tokens. This convention is in contrast to the more traditional approach of textual programming languages that use a single dimension (top to bottom) and where the tokens are defined as words. There are a number of examples of visual expressions in VPLs, such as demonstrations of actions performed by graphical objects, diagrams, icons, or even hand-drawn sketches. Additionally, many VPLs are based on the design of screen objects such as boxes, treated as entities and connected by connections such as lines, arcs or arrows, representing the relationship of the entities [8].

It might be argued that traditional textual programming languages employ two dimensions in defining the code syntax and semantics where the x-dimension represents the language as a strings and the y-dimension is used to make distinction between code sections and statements. This description can only be considered multi-dimensional in a limited way, since the spatial representation in the y-dimension doesn't actually add to the languages syntax and semantics, but specifically allows clearer understanding of the language through that separation. This extra dimension can be remove and the language can be written on a single line without any loss of syntax or semantics, although it would be very difficult to write and read at that point [9].

The use of visual expressions in a VPE allow an editing or creation shortcut which can generate code that may or may not have syntax differing from that written textually. Today VPEs are used commercially to help professional programmers by providing tools, referred to as Integrated Development Environments (IDE), that ease the creation and maintenance of traditional textual languages. IDEs allow programmers to use textually languages that they are already used to and know, but gives a graphical interface that helps increase productivity through code completion from the documented APIs, or adding boiler plate code that is easily created automatically.

Use of VPEs for textual languages help to further research in VPLs, and provide a conduit to putting research into practice, which can provide a gradual migration to using VPLs.

2.5 Previous Work

The earliest work in visual programming was in two directions: visual approaches to traditional programming languages (e.g., executable flowcharts), and approaches that deviated significantly from traditional approaches (such as programming by demonstrating the desired actions on the screen). There were advantages that many of these early systems had, which seemed exciting and intuitive on demonstrated example programs, however when attempts were made to extend them to programs that were more realistically sized, it caused a number of problems. These problems caused many to believe that visual programming was unsuited to real work, that it was just an academic exercise, leading to an early disenchantment with visual programming.

To solve these problems, visual programming researchers began to develop ways to use visual programming for only selected parts of software development, thereby increasing the number of projects in which visual programming could help. By following this approach, straightforward visual techniques were widely incorporated into programming environments that support textual programming languages, to visually combine textually-programmed units to build new programs, to support electronic forms of software engineering diagrams for creating and/or visualizing relationships among data structures, and to replace unwieldy textual specification

of GUI layout. Since then there have been many successful VPEs released and used in the commercial industry of programming including, VisualWorks, which supports the Smalltalk language, and Visual Basic for the programming language Basic [6], while other VPEs have instead focused on coarse programming, such as the Computer-Aided Software Engineering (CASE) tools. These tools support having a visual specification, by way of diagrams and relationships among the program modules, which allow it to automatically generate composition code [9].

Other visual programming researchers took the approach to develop domain-specific visual programming systems, which would increase the kinds of projects suitable for visual programming. Through this approach, the number of projects that could be programmed visually increased as each new supported domain was added. This approach quickly produced a number of successes both in research and in the marketplace. Today there are commercial VPLs and VPEs available in many domains; examples include programming laboratory data acquisition (National Instruments' LabVIEW), programming scientific visualizations (Advanced Visual Systems' AVS), programming telephone and voice-mail behavior (Cypress Research's PhonePro), and programming graphical simulations and games (Stagecoach Software's Cocoa) [9]. A number of software-agent generators are creating allowances for macros that assist with repetitive tasks to be inferred from end-user manipulations, and are starting to become embedded in personal computing software as well.

CHAPTER 3

VPEVOLVE

3.1 Motivation

VPEvolve is a client user interface that utilizes a visual programming environment (VPE) that allows setup of the GA parameters, manages the evolutionary search, and gathers and summarizes the progression of the search. Some motivations for VPEvolve were to create an application that could identify computation models in an automated fashion, use computer resources, such as network bandwidth and computational calculations, efficiently and have an interface that is simple and easy to use for researchers. Since creating complicated models is such a difficult manual task, VPEvolve uses evolutionary search to identify models that explain specific real world experiments. VPEvolve also utilizes distributed solutions to perform the search efficiently and with high resource utilization.

VPEvolve uses a VPE that was inspired by Cantata, which was developed by Kubica for the Khoros system [25]. Previous work [12] shows that VPEs are more intuitive to use by non-software engineers when applied to a system-dependent environment. This is especially important because most users of VPEs will belong to the same category of not be software domain professionals. As shown in Figure 3.1,

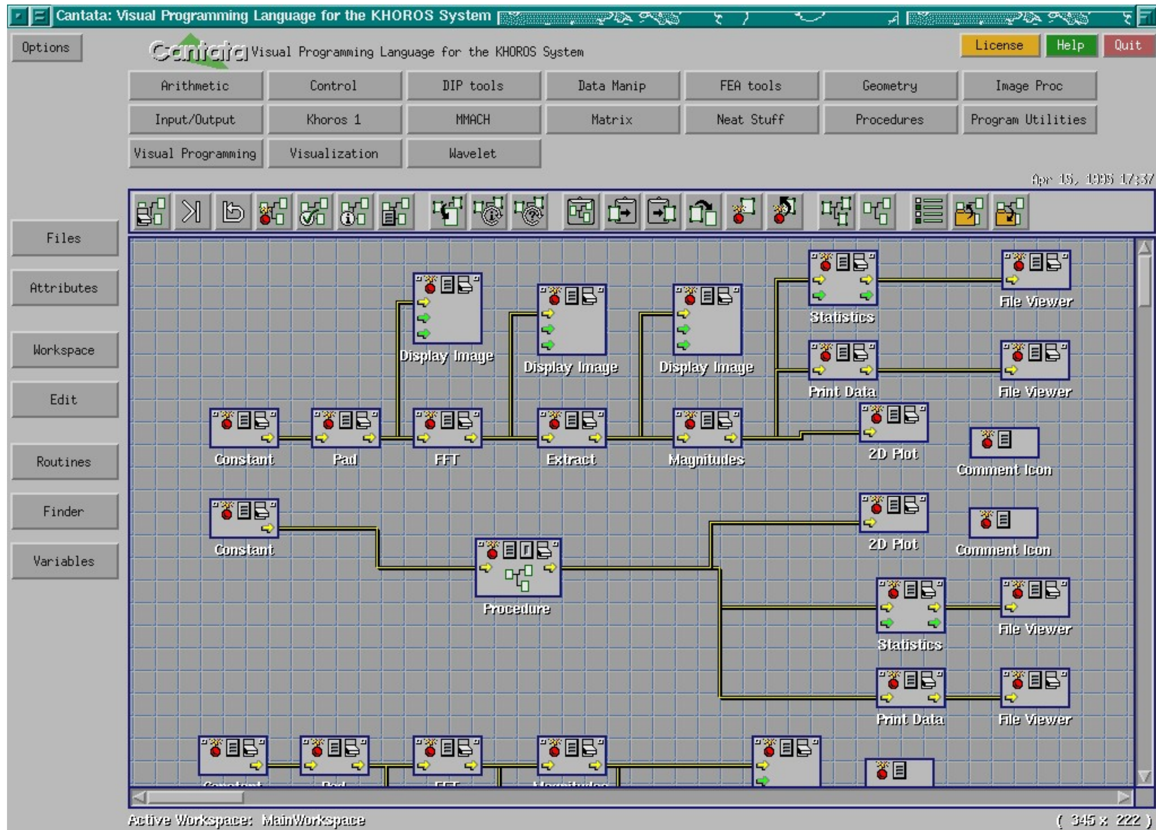


Figure 3.1: Cantata VPE

Cantata uses components called glyphs that are connected together to provide its work-flow, which was used in a similar way in VPEvolve. Specifically, VPEvolve has an emphasis on glyphs and inter-connectivity, which is used to manage the genetic operator work-flow of the evolutionary search. A glyph based VPE reduces the time for new users to become proficient with the user interface (UI) as well as give experienced users a better understanding of their genetic operator work-flow through visualization. In summary, this type of VPE provides a more intuitive environment to design and set up an evolutionary search.

An additional motivation for VPEvolve was to develop an application with significant improvements in usability, flexibility, and resource management compared

with the python-based evolutionary search application (CSGA) that was designed as a companion to Cellsim, and was unable to work with any other modeling platforms.

3.2 Design Features

The mentioned motivations were considered heavily when designing the implementation for VPEvolve. The programming language used was Java, a compiled language, to stream-line its run-time efficiency. VPEvolve was created using the NetBeans IDE and GUI builder, which enables user interface customization at the source code level [7]. Additionally, this customizable interface also allows users to save their client desktop environment (window layout, locations, etc.) according to their preferences and desired work-flow. The decision to use NetBeans and Java makes the client source code simpler, which improves tasks of future maintenance and development.

VPEvolve was designed as a general platform for initializing, managing and monitoring an evolutionary search. This generality is attributed to the modular architecture of the application, which theoretically allows it to be used as an evolutionary search application in any problem space. Another aspect of VPEvolves modularity is the weak coupling between the user interface code and the core logic that manages the GA.

VPEvolve was designed as a collection of modules to allow re-implementation of either specific GUI components, the console-mode, or genetic algorithm implementation with minimal effect to any other modules. Such architecture specifically allows another implementation of the genetic algorithm setup substituted in place of the current implementation. Additionally, the current genetic operators can be re-implemented for another modeling platform.

3.2.1 Visual Programming Environment

VPEvolves main window is the VPE, where the genetic operator components are created, connected and the GA parameters are set. The connected components (i.e., glyphs) are used to represent the genetic operators of the GA. Since a GA includes the application of genetic operators to individuals from a population over a given generation, the VPE window is organized to highlight this workflow (Figure 3.2).

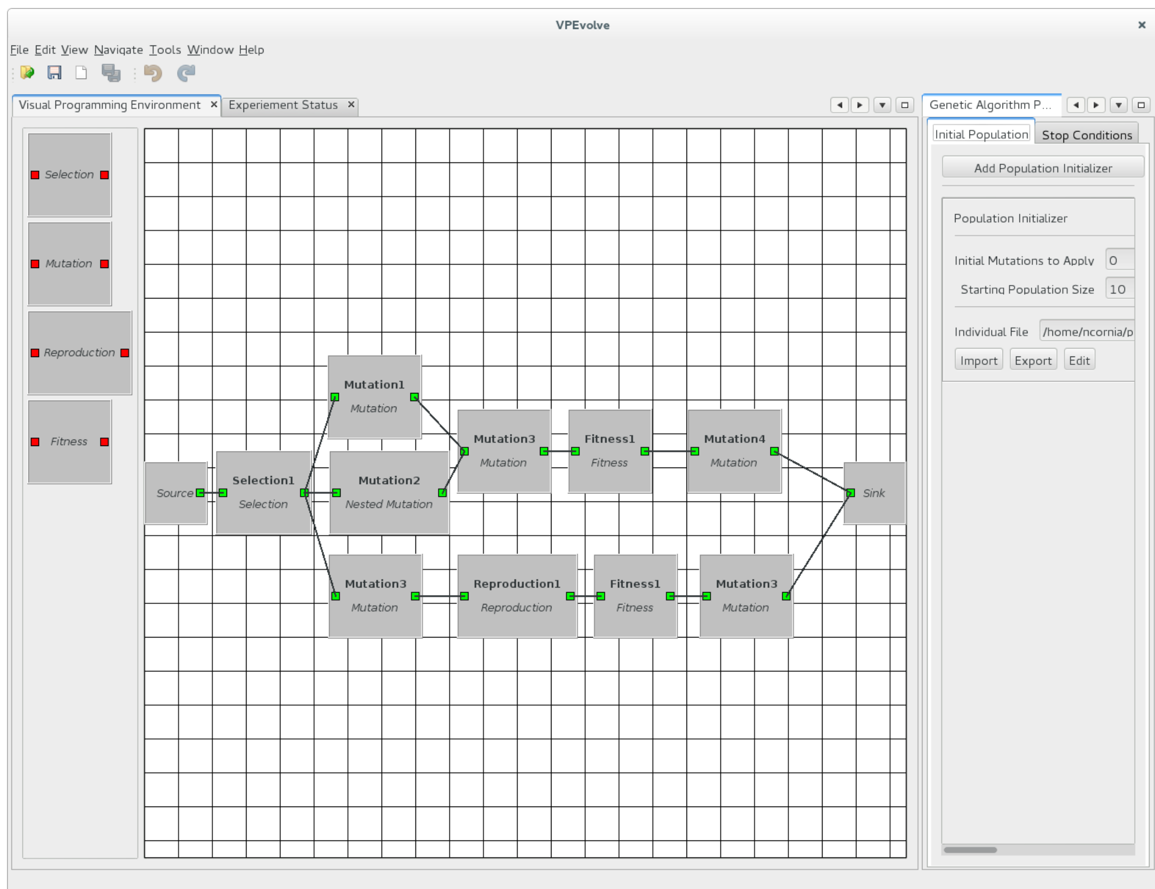


Figure 3.2: VPEvolve VPE

There is a list of genetic operators in the VPE window, which use the drag and drop feature to pick up and place the operators in any arrangement that specifies

the desired genetic operator work-flow. If used with CellSim, as the simulation server, there are four available operators to begin with, namely Selection, Mutation, Reproduction, and Fitness. Each of these operators have input and output points. Connections can be made between the output of one glyph and the input of another. In particular the Mutation operator allows the user to choose it to be a Nested Mutation, where the user can pick more than one mutation to apply at that point in the genetic operator work-flow. The user interface was designed to allow easy removal of glyphs or their connections by choosing the delete option following a right mouse click.

Glyphs are initially given a unique numbered name, which allows them to be identified in the operator network. These names can be changed at anytime through the pop-up menu, making them easier to identify. Additionally, mutation operators also display which part of the individual description to be mutated for quick differentiation. Upon starting an evolutionary search, VPEvolve does a simple validation to check that all components are connected, (i.e., there exists a path form Source to Sink). If this condition is violated then the tool reports the violating glyph's name, so the glyph with the error can be quickly found in the VPE and the error corrected.

The VPE window also contains both a Source and a Sink. The Source is where the new generation of individuals begin when the genetic operator application starts and the Sink is where all the individuals for a single generation end up after all the operators have been applied. When a fork, where the output of one glyph connects to the input to more than one other glyph, occurs in the operator network the output individuals of one glyph are copied to the next glyph(s), which can cause the number of individuals in a population to grow quickly. However, populations across generations should remain the same size and to accommodate this a Sink operator is used. The

Sink operator uses a cramming function that will prune the number of individuals down to the predetermined population size before moving to the simulation and fitness evaluation phases. This new set of individuals represents the population for the subsequent generation.

3.2.2 Configuration File

The configuration for VPEvolve file is in XML format, and contains two main sections, Core and GUI. The Core portion is requested from GA logic code, which will be further explained in Section 3.3.2. Meanwhile the GUI portion contains the relative locations for all the glyphs and their inter-connectivity.

3.2.3 Genetic Algorithm Parameter Setup

The parameters for each genetic operator are defined prior to performing an evolutionary search. Since glyphs in VPEvolve represent genetic operators and are visual components, double-clicking any of them creates a temporary window that slides out from the right. Depending on which operator is selected, editable parameter settings will be displayed. For example as shown in Figure 3.3, the VPE is the section in the middle containing gridlines. The left side of the application contains a toolbox of 4 glyphs (Mutation, Selection, Reproduction, and Fitness, respectively), which can be added to the VPE with drag and drop functionality to construct the genetic operator network. Double-clicking on the mutation glyph in Figure 3.3 brings up the Genetic Operator Editor as shown in the right panel. For the mutation operator this window displays the entire schema for the individual in a tree structure. By clicking on any

Figure 3.3: Genetic Operator Parameter Setup Window

of the nodes of the tree, the panel to the right of the tree is populated with either editable parameter fields or a selection box that allows you to add a specific type of mutation that applies to the node selected. For example, if the Effect node under the Genome node is selected, which describes the amount of promotion a molecule will have on a gene assembly, a selection box is populated with Numeric Mutations because effect is the data type float. By using the selection box, a mutation operator such as NormalMutation can be chosen and added, which will show the editable parameters for that mutation. Once focus is lost on the Genetic Operator Editor window, by clicking somewhere outside of its bounds, the window slides to the right and disappears so that the VPE can be clearly seen.

A simple text editor was included in VPEvolve to make certain operations such as editing the individual description file or a custom Python fitness evaluator more fluid while using the VPEvovle application. This text editor allows edit operations

to be done to files without having to leave the application and make edits in another application. The individual description consists of an xml that defines properties such as where cells are located, gene assemblies, chemistry equations and the molecule catalog. In VPEvolve, the individual XML file can be imported and edited in the internal text editor to make desired changes. This editor highlights syntax for convenience of editing and complying with correct XML syntax (Figure 3.4).

Figure 3.4: XML Editor for Individuals

Arguably, the most important components of a GA are the fitness evaluator(s) or fitness function(s). Before the GA can select individuals for the next generation, each individual needs to be compared with the expected outcome of the actual planarian regeneration, by evaluating the individual's fitness after the most recent genetic operation. This is accomplished through the evaluation of a fitness function.

These fitness functions need to be supplied to VPEvolve, which are very specific to the algorithm used and how it is calculated based on the individual description.

Fitness evaluators are very specific to the system and problem under consideration. Due to its wide acceptance and use by biologists, fitness evaluators are user-defined and coded in Python, that is easy to learn and a flexible programming language. Contributing to its flexibility is its ability to invoke library functions from higher performing compiled languages like C and C++. Therefore, a Python file import option is provided in VPEvolve, as well as an editor to make any necessary changes (Figure 3.5). Prior work in our lab has led to development of a number of fitness evaluators implemented as C libraries accessible from Python. These include evaluators such as Difference Distribution, Overlay Distribution and Graph Edit Distribution.

3.2.4 Evolutionary Search Status View

After the GA parameters are defined, the experiment can be started from another tabbed window referred to at the Status Window as shown in Figure 3.6. This window displays the results of the GA as it runs and has options to start the GA, more the GA forward by one generation, pause the GA after it has been started, and restart the GA. If the user clicks the start button, the genetic operator network is validated followed by creation of a configuration file that saves the glyph locations, connectivity, and genetic operator parameters for each glyph. The main portion of the status window consists of two tables. The upper table displays a list of the individuals that comprise the population of the specified generation. The lower table displays a list of all the offspring that are created in the genetic operator application process.

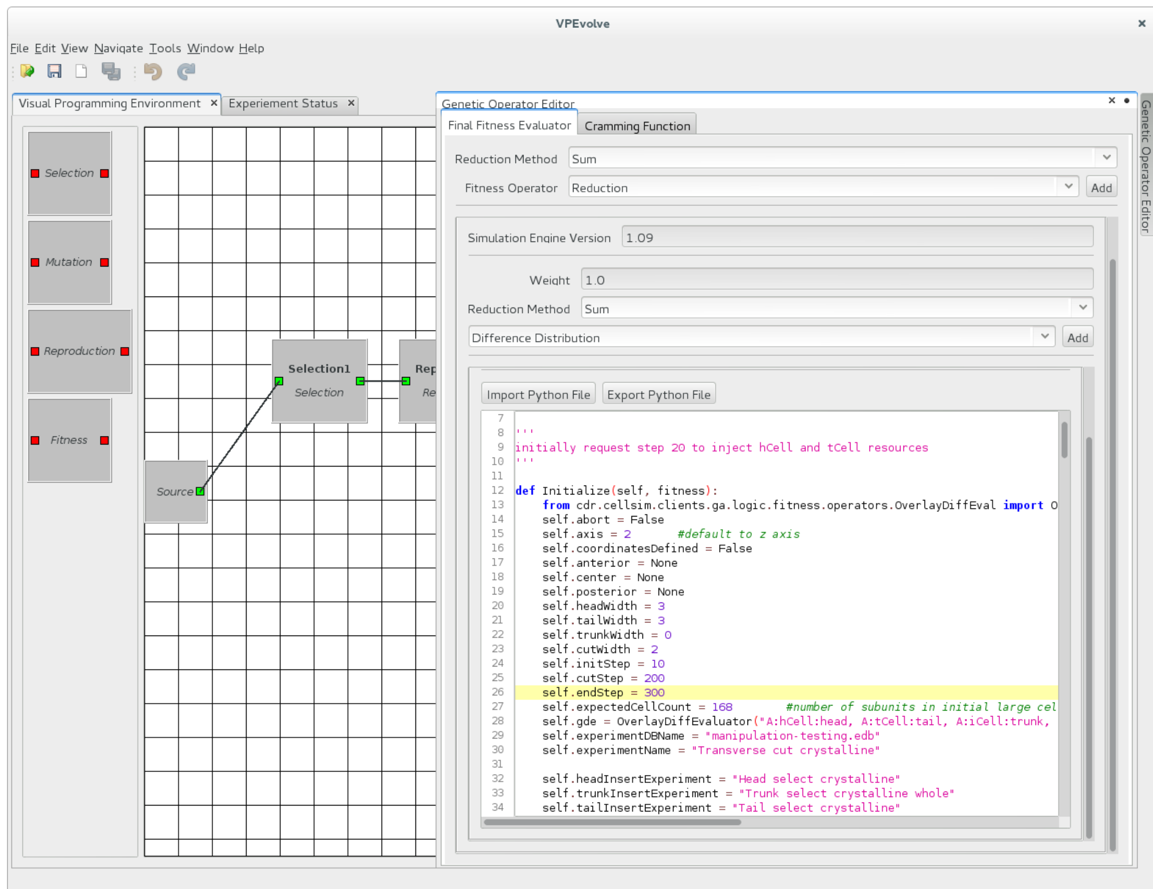


Figure 3.5: Custom Python Fitness Evaluator Editor

At the start of an experiment the individual description file is used to generate the appropriate number of initial individuals as specified by the population size parameter. The fitness values for the initial population are then calculated. After this initial startup phase, each generation is processed in a similar manner, consisting of application of genetic operators followed by a simulation and fitness evaluation until convergence is reached (e.g., a fitness of 1.0 is found). The status window displays both a list of the current generation's individuals and the list of individuals that have been altered by the genetic operators so the progression of the fitness evaluations can be seen (Figure 3.7).

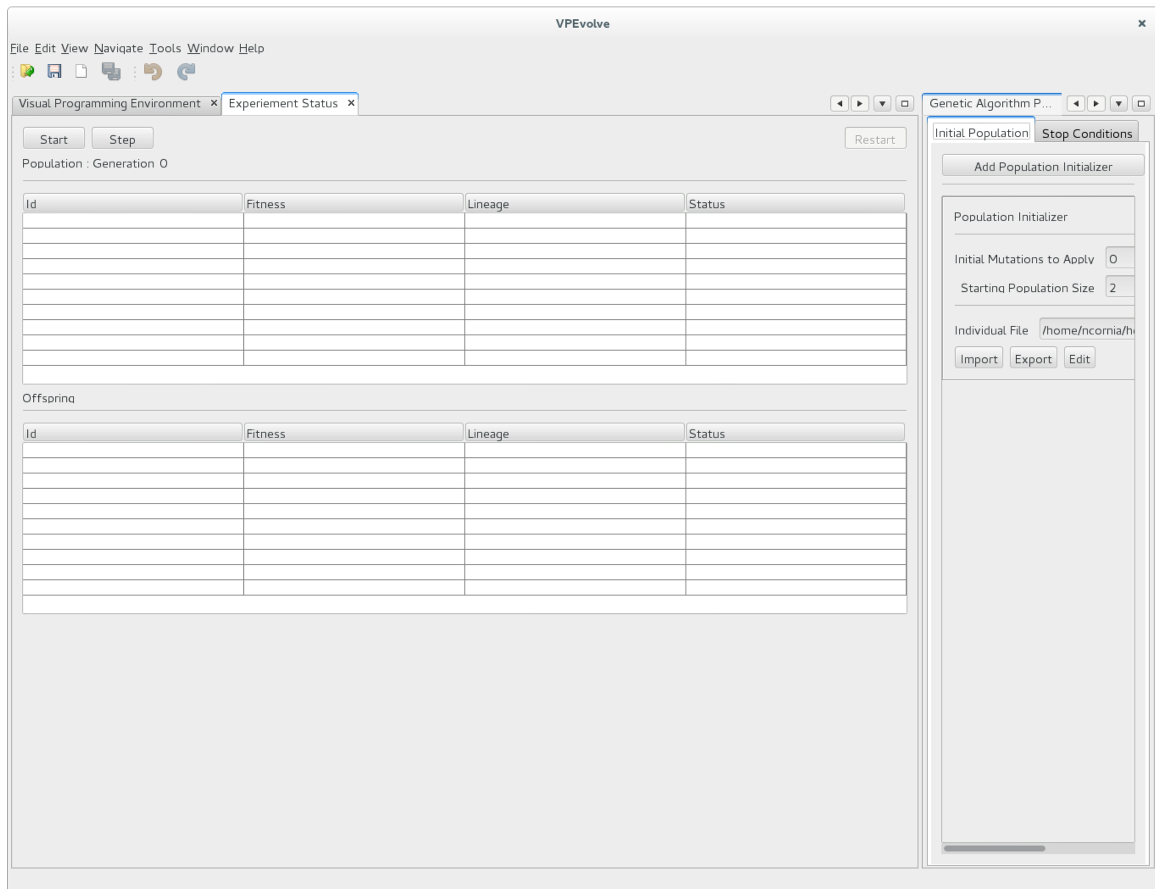


Figure 3.6: Evolutionary Search Status Window

3.2.5 Console Mode

Since VPEvolve uses a GA as its underlying method of performing evolutionary searches, a considerable benefit can be gained by doing some processing and calculations in parallel on a computer cluster, especially when large population sizes are specified for running the GA. Fitness evaluators are one of the most expensive operations when running the GA, and since the evaluator needs to be run for each individual at every generation, there can be a significant bottleneck at this phase of the search. However since the fitness of one individual does not depend on another individual, each individual's fitness can be evaluated separately and in parallel.

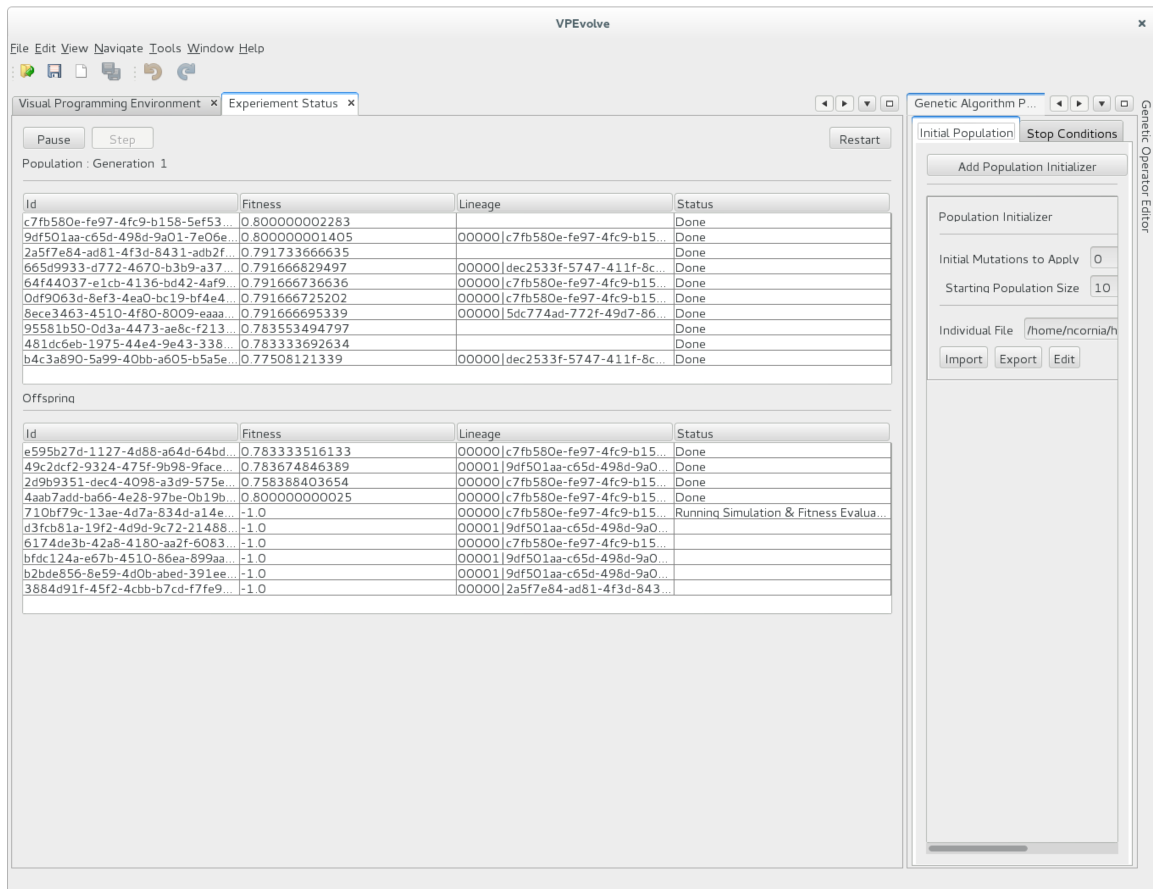


Figure 3.7: VPEvolve Evolutionary Search in GUI

To take advantage of this distributed solution, VPE was designed to be loosely coupled between the GUI and GA logic. The GA logic doesn't need to know anything about the GUI to complete an evolutionary search. Moreover, a console only mode was developed in VPEvolve where only a configuration file and output directory need to be provided to run an evolutionary search. This was designed be straightforward with a distributed solution. For example, by logging into a computer cluster via the console, VPEvolve can start from that node and take full advantage of the clusters processing power. Since most remote computer clusters use a console interface, this design makes VPEvolve more versatile, because it can perform evolutionary searches without the

need to forward a GUI, which often makes interaction slow or unresponsive. Moreover, all the same status information provided in the GUI is displayed to the console except a table format (Figure 3.8).

```
server@genesis:~/nicTest
File  Edit  View  Search  Terminal  Help
*** Starting Generation 10 ***
Applying GA Operators
GAOp Time: 0.361327
Evaulating Fitness of Dirty Individuals

==========================================================================================================================
|                                                Population Generation: 00010                                            |
==========================================================================================================================
|                 ID                   |     Fitness       |                              Lineage                        |
==========================================================================================================================
| bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 |   0.800000003527  |                                                             |
| 05da3bcd-fd0a-466f-9f83-33be29286295 |   0.791804312691  | 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 - 00002|3a4b8c94-cb0e-4ca0-bcc2-91b5f2f50f9f |
| 9f8e028d-9d0c-4bbf-8472-46d34a387ba6 |   0.791766196969  | 00004|0cc2ab9a-898a-4943-8e0c-e09f8f9b73be - 00004|0cc2ab9a-898a-4943-8e0c-e09f8f9b73be |
| e44fbc1d-9833-42fc-be6e-58d9c0b3028a |   0.791666718377  | 00004|0cc2ab9a-898a-4943-8e0c-e09f8f9b73be - 00005|05da3bcd-fd0a-466f-9f83-33be29286295 |
| 5df94b12-bfcf-4bec-8435-27119b8ac9f9 |   0.791666702406  | 00004|0cc2ab9a-898a-4943-8e0c-e09f8f9b73be - 00004|0cc2ab9a-898a-4943-8e0c-e09f8f9b73be |
| 05c4f052-322f-4137-96ee-0509c51f5606 |   0.783464623668  | 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 - 00007|e44fbc1d-9833-42fc-be6e-58d9c0b3028a |
| 78d919d3-3e7c-4811-bb1f-4bb8c817705f |   0.783333743157  | 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 - 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 |
| e4c080d5-d90c-4057-89aa-409fc06f4bee |   0.783333743157  | 00005|05da3bcd-fd0a-466f-9f83-33be29286295 - 00006|9f8e028d-9d0c-4bbf-8472-46d34a387ba6 |
| d110ef36-f88a-40e4-84bc-445bece18157 |   0.783333743157  | 00006|9f8e028d-9d0c-4bbf-8472-46d34a387ba6 - 00009|05c4f052-322f-4137-96ee-0509c51f5606 |
| 1c1bbb82-ee13-480a-984d-0cfe1a46fbcc |   0.783333743157  | 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 - 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 |
==========================================================================================================================

Checkpoint Time: 0.023818
Generation Time: 21.967115
*** Generation 10 is DONE ***
*** Starting Generation 11 ***
Applying GA Operators
GAOp Time: 0.364567
Evaulating Fitness of Dirty Individuals

==========================================================================================================================
|                                                Population Generation: 00011                                            |
==========================================================================================================================
|                 ID                   |     Fitness       |                              Lineage                        |
==========================================================================================================================
| bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 |   0.800000003527  |                                                             |
| 05da3bcd-fd0a-466f-9f83-33be29286295 |   0.791804312691  | 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 - 00002|3a4b8c94-cb0e-4ca0-bcc2-91b5f2f50f9f |
| 9f8e028d-9d0c-4bbf-8472-46d34a387ba6 |   0.791766196969  | 00004|0cc2ab9a-898a-4943-8e0c-e09f8f9b73be - 00004|0cc2ab9a-898a-4943-8e0c-e09f8f9b73be |
| 7c7d50bb-20f2-4469-9f16-6cbf34a45181 |   0.791666738270  | 00005|05da3bcd-fd0a-466f-9f83-33be29286295 - 00005|05da3bcd-fd0a-466f-9f83-33be29286295 |
| e44fbc1d-9833-42fc-be6e-58d9c0b3028a |   0.791666718377  | 00004|0cc2ab9a-898a-4943-8e0c-e09f8f9b73be - 00005|05da3bcd-fd0a-466f-9f83-33be29286295 |
| 05c4f052-322f-4137-96ee-0509c51f5606 |   0.783464623668  | 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 - 00007|e44fbc1d-9833-42fc-be6e-58d9c0b3028a |
| 61b19406-71d8-4086-8dd9-7808c025a2ec |   0.783338401841  | 00005|05da3bcd-fd0a-466f-9f83-33be29286295 - 00006|9f8e028d-9d0c-4bbf-8472-46d34a387ba6 |
| 0a68b6a3-232d-4b0b-8a39-bfb09973a006 |   0.783333743157  | 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 - 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 |
| 1c1bbb82-ee13-480a-984d-0cfe1a46fbcc |   0.783333743157  | 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 - 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 |
| 78d919d3-3e7c-4811-bb1f-4bb8c817705f |   0.783333743157  | 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 - 00000|bc03bbd0-197f-4b63-8dd4-91d25fcf5ad1 |
==========================================================================================================================
```

Figure 3.8: VPEvolve Evolutionary Search in Console Mode

Additionally, the GUI mode can be used on a local workstation to create and edit the GA configurations, parameters, and individual specification, which can be saved to disk, then transferred over to a remote system (i.e. computer cluster), after which VPEvolve can use that configuration to conduct the evolutionary search on the remote system in console only mode.

3.2.6 Integration with a Modeling Platform Through Middleman Software

Since VPEvolve starts processes on separate nodes of a computer cluster, therefore it was advantageous to delegate as much work to the compute nodes and keep GA logic processing the same node as VPEvolve. The diagram in Figure 3.9 shows the process of starting the middleman software on a separate node, doing a model simulation of a single individual, calculating the fitness value and sending it back to VPEvolve. Since VPEvolve was designed to be generic and be able to use any platform to run simulations, middleman software was needed to correctly integrate with CellSim. The middleman software interacts with CellSim as the simulation is in process, and once the simulation is completed it calculates the fitness of the individual with the specified fitness evaluator. This separation of functionality allows the simulation and fitness evaluation, the most expensive calculations, to be done on a different node than VPEvolve. Therefore, VPEvolve needs to simply create the new individuals for each generation by applying the genetic operators, collect the fitness values, and report on the progress of the search.

Once VPEvolve reaches the phase where it calculates the fitness values for the individuals, it uses Swarm, a program that sends jobs to other nodes. These jobs start a separate process of the middleman software at a specified node with an XML file location that contains the description of the individual, the description of the fitness evaluator, and the IP address and port where VPEvolve is running and waiting to receive the calculated fitness values from the middleman software. The middleman software then starts a CellSim instance and passes the individual description which runs a specified simulation. CellSim will take one or more snapshots of the individual,

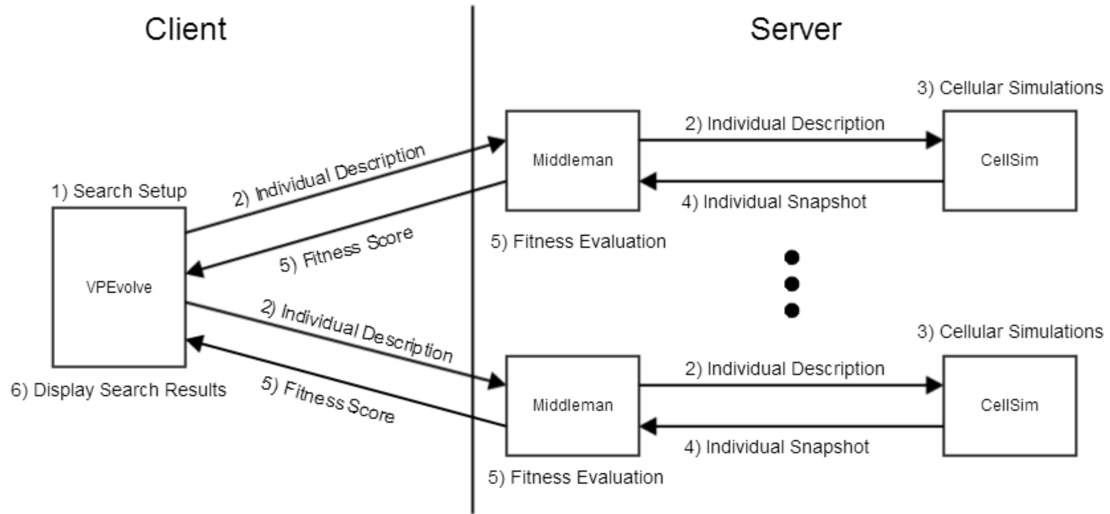


Figure 3.9: VPEvovle Client-Server Individual Processing

which are used by the middleman software in the fitness evaluator to calculate the fitness value. The middleman software then sends the fitness value for that individual back to VPEvolve. If VPEvolve identifies the fitness value to be unique, it tells the middleman software to save both the snapshot and individual description in a shared folder for later analysis.

3.2.7 Evolutionary Search Checkpointing

As the evolutionary search progresses VPEvolve will save a list of individual descriptions that comprise the population at the end of that generation and the GA configuration in XML format, which is called a checkpoint, at specified intervals. The default checkpoint operation in VPEvolve is every generation, however this in configurable. These checkpoint files are useful if an evolutionary search needs to be restarted at a particular point.

When using the open operation in VPEvolve, either a GA configuration file can be chosen which will load the setup parameters for the GA, or a checkpoint file can be used to load both the GA configuration and the starting population where the correct information about generation and population is preserved. During this process of loading the file, VPEvolve will detect which file is being used and load the search appropriately. The individual descriptions for our experiments typically consist of a sheet of about 120 cells, which take up about 100KB of disk space. Therefore, a typical GA setup with a population size of 100 individuals, each checkpoint file will be about 10MB, and over 100 generations it will grow to 1GB of disk space. This is obviously a problem if disk space is limited, so the user can specify at what generation interval to checkpoint or choose to disable check pointing all together.

3.3 Architecture

3.3.1 Modular Design

VPEvolve was designed and implemented as a modular application, to ensure that the GA logic and GUI are loosely coupled and to allow each module to be a stand-alone portion of the interface. Moreover, because of this modularity, each module can be replaced with another implementation of similar logic. For example, the main window contains the VPE component, which was created as its own module; therefore, a different implementation of the visual representation can be created and used in its place. A ConsoleMode module was also created so that VPEvolve can run an evolutionary search on a remote computer cluster where either graphical forwarding is disabled or network bandwidth cannot support a GUI. Additionally, VPEvolve

could be deployed on a remote system without any of the GUI modules which would save disk space.

Figure 3.10 shows the critical modules that are used in VPEvolve where lines represent internal dependencies. For simplicity and readability of the figure, some

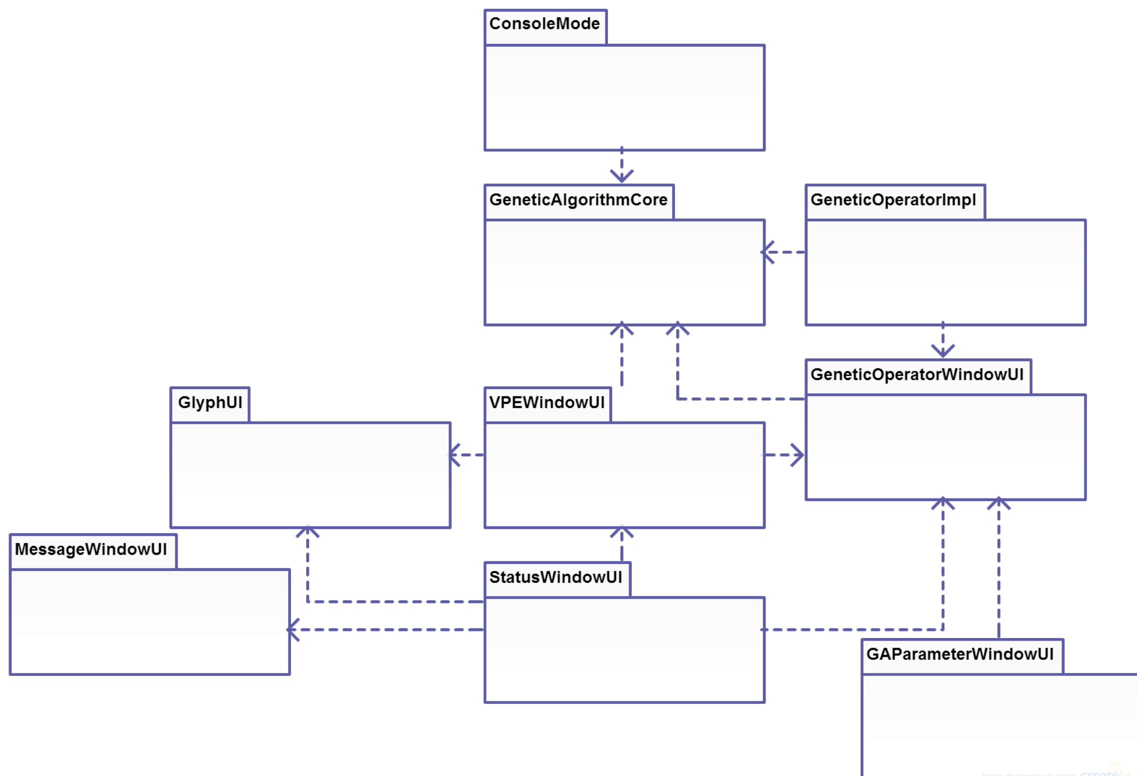


Figure 3.10: VPEvolve Module Diagram

modules were left out, such as the Utilities module, because most of the modules depend on it. In Figure 3.10 the UI is partitioned into distinct modules according to their functionality. This abstraction makes maintenance more convenient and less expensive, and module swapping for any of these components can easily be done. An important note is that the ConsoleMode module has no dependencies to the UI modules. Specifically, the ConsoleMode module and UI modules are only indirectly

connected through the GeneticAlgorithmCore module, which in turn doesn't depend on any UI modules either. The GA logic actually runs self-contained and broadcasts notifications of its progress to any processes waiting for those notifications.

3.3.2 Implementation of Genetic Operators

The Genetic Algorithm Core module contains the GA logic used in an evolutionary search and describes how the genetic operators are applied and how communication between VPEvolve and the middleman software is done. The implementation of the genetic operators is contained in its own module. This separation of logic definition and implementation allows new genetic operators to be implemented and added as needed. If new operators are needed the GeneticOperator abstract class should be implemented and be placed in the GeneticOperatorImpl module. When VPEvolve is first opened, it searches this module for all the source files and registers them with their appropriate factories in the GeneticAlgorithmCore module. Afterwhich, any operator can be requested and returned from the operator's factory. This factory design pattern allows VPEvolve to contain and support new implementations of the genetic operators without needing to change how the GA or VPE component code works.

To allow integration and use with CellSim, the same genetic operators used in CSGA are also implemented in VPEvolve, such as AddListMutation, TargetFitness-Selection, and CrossoverReproduction, etc. Figure 3.11 shows the class architecture of the Mutation genetic operator used in the CellSim implementation. Normal Mutation, Range Mutation, and Relative Mutation, are a few currently implemented operators that belong to the Mutation genetic operator category. These mutations define how

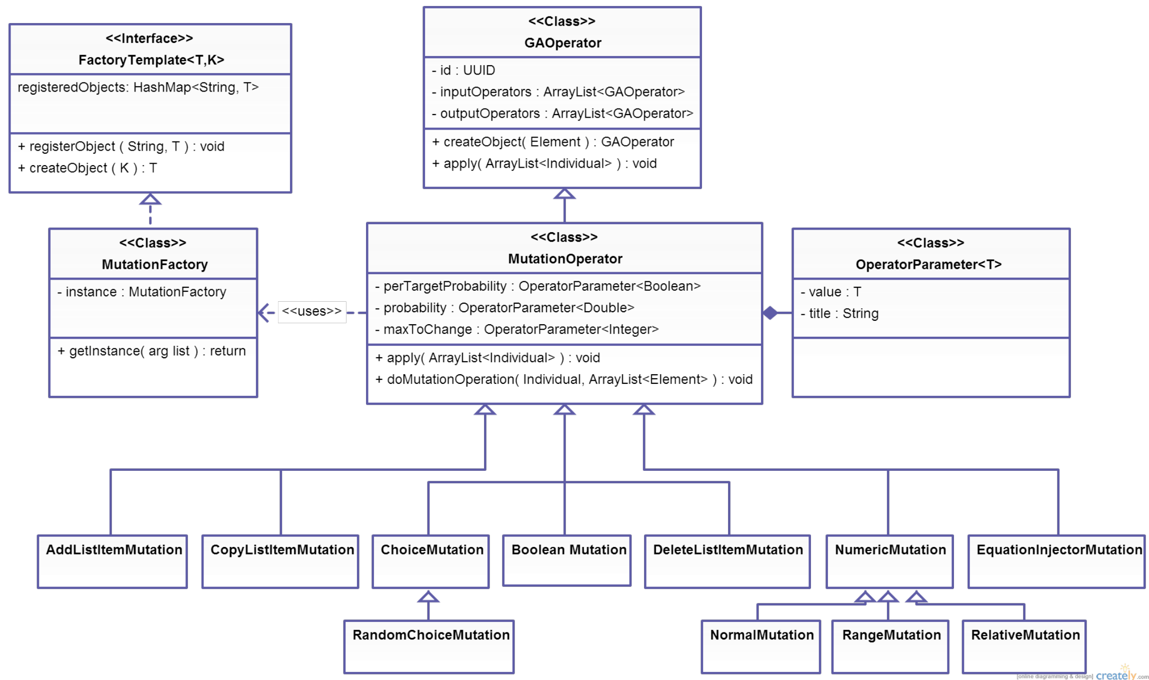


Figure 3.11: VPEvolve Class Diagram

numeric values in the individual description are changed. The Mutation Operator is a subclass of the Genetic Operator class, so that once an evolutionary search is run it performs a single operation of applying all specified genetic operators in the operator network. This approach allows the search to be run without the search understanding all the definitions of subclass operators.

These genetic operators are implemented specifically to make modifications to the scheme used by CellSim. This scheme file is used to identify the location for editing the individual and for genetic operator category, there is a do<category>Operation method, which directly edits xml tags of the individual description. By designing VPEvolve in a modular way, other modeling platforms can be used in place of CellSim.

In order to use with another problem space, there are a few steps to follow:

1. provide a scheme file in xml format that describes the individual (i.e., what the components and properties it has)

2. provide genetic operators that describe how an individual should be manipulated, implemented in Java and implement the corresponding GeneticAlgorithmCore Class

3. provide a way to translate the individual from the xml format used in VPEvolve (i.e., the provided scheme file) to the platform being used for simulation if the scheme file format is incompatible

4. provide fitness evaluator(s) that will determine the fitness value of the individual

5. provide a middleman piece of software that is used to start the simulation platform given an individual from VPEvolve, run the fitness evaluator and report that result back to VPEvolve

Even though VPEvolve currently uses genetic operators that were designed specifically for CellSim, most of the genetic operators are generic and based on the the data type used for a specific individual property (e.g. integer, float, list, boolean, etc.). Therefore, these operators could be easily repurposed to use for other problem spaces.

CHAPTER 4

RESULTS

4.1 Comparison

This section provides a comparison between VPEvolve and CSGA. The following areas will be covered, user interface, architecture, computer resource usage and performance. VPEvolve uses a VPE for its user interface, which provides significant benefit over parameter input in CSGA, since VPEvolve allows the user to visually evaluate how the genetic operators are be applied. In contrast, CSGA only provides a primitive interface to setup the GA, which requires that users specify the genetic operators in tabbed views (Figure 4.1). This can lead to confusion about ther order of operator application, especially since the tabbed views can be reordered and the default order isn't the actual order the operators are applied during the evolutionary search. VPEvolve on the other hand shows precisely where operator application begins (Source operator), how individuals flow through operator application and where it ends (Sink operator), in a visually understandable way. Furthermore, VPEvolve also provides additional flexibility for experiment setup.

Since part of VPEvolve's motivation was to be a replacement for the CSGA client and since the Andersen lab has done extensive research work using CSGA

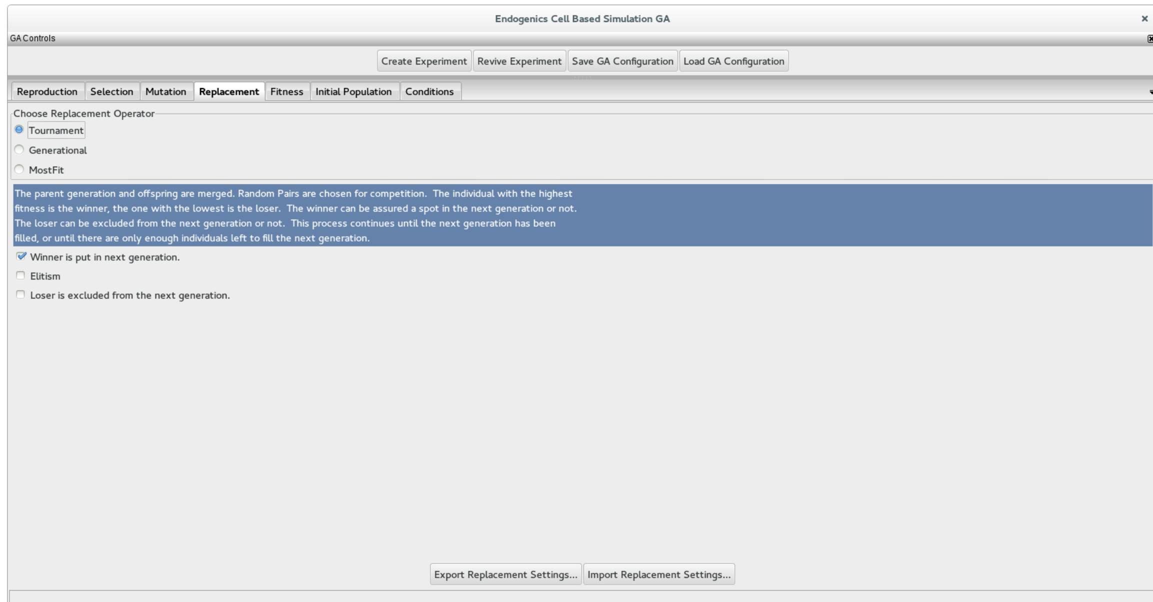


Figure 4.1: CSGA GA Parameter Setup View (Replacement Operator)

to study cellular regeneration in planaria, the ability to accurately convert the GA configurations from previous evolutionary searches from the CSGA format to the VPEvolve format was necessary. Otherwise any incompatibility between the CSGA evolutionary search configuration file and the configuration file of VPEvolve would lead to redoing the GA setup configuration for any revisited evolutionary searches. For this reason, and the fact that the the genetic operator network in VPEvolve is represented in a considerably different fashion than CSGA, a parser and translator were also created. The parser and translator takes the CSGA configuration file and extrapolates the connectivity, which is linear or one-dimensional for CSGA. This linearity in CSGA is because each category of genetic operator (i.e. reproduction) can only be set once, after which CSGA will process each category one by one. VPEvolve assigns the appropriate glyphs for each operator, then they are assigned locations that represent the linear position of the genetic operator work-flow (Figure 4.2). This

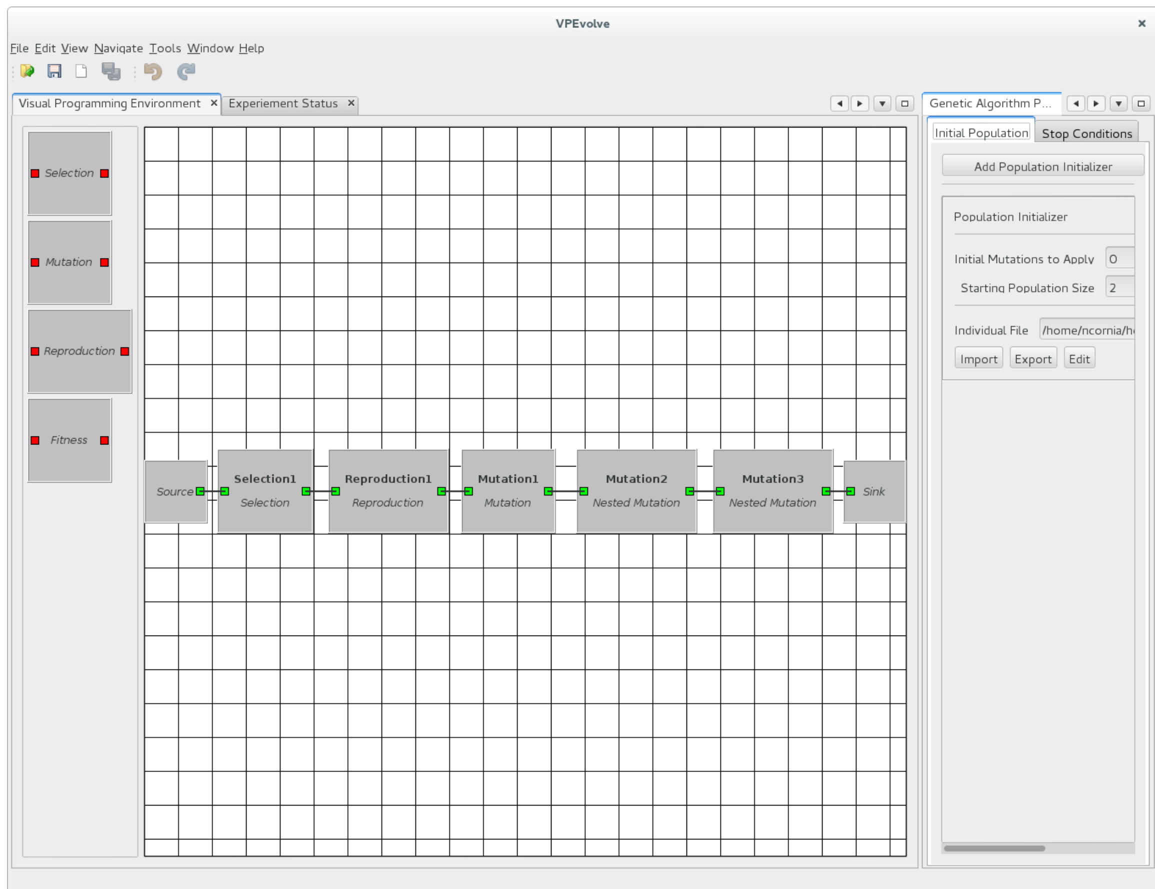


Figure 4.2: CSGA Translated Evolutionary Search

one-dimensionality of the CSGA genetic operator workflow, is in contrast to the flexibility of VPEvolve, which allows a multi-dimensional genetic operator work-flow. This type of work-flow allows VPEvolve to create genetic algorithm setups with greater complexity and that promote to explore parts of a search space that were previously not possible.

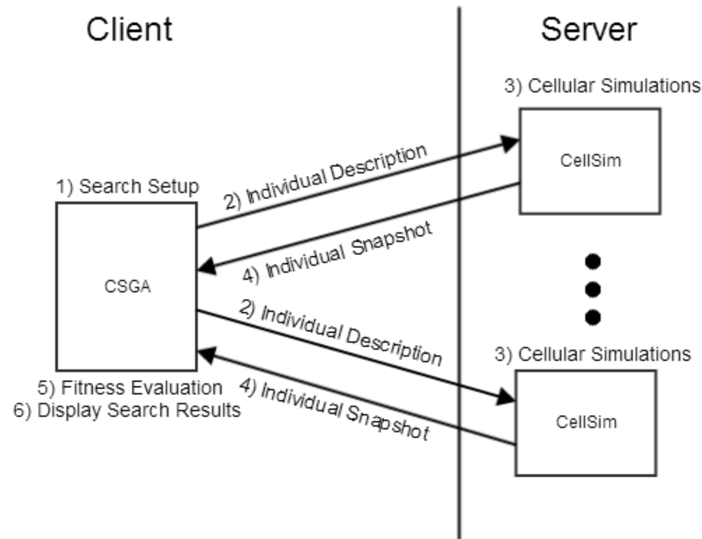
VPEvolve also provides a clean console only mode, which was only recently provided in CSGA. However the information provided in CSGA is difficult to read, and is printed to the screen in a list that has unclear information, which lacks in clarity comparing to VPEvolve. Thus Figure 3.8 shows how CSGA can lead to

misunderstanding about the state of the search process when compared to Figure 4.3 because individuals aren't identified in any way.

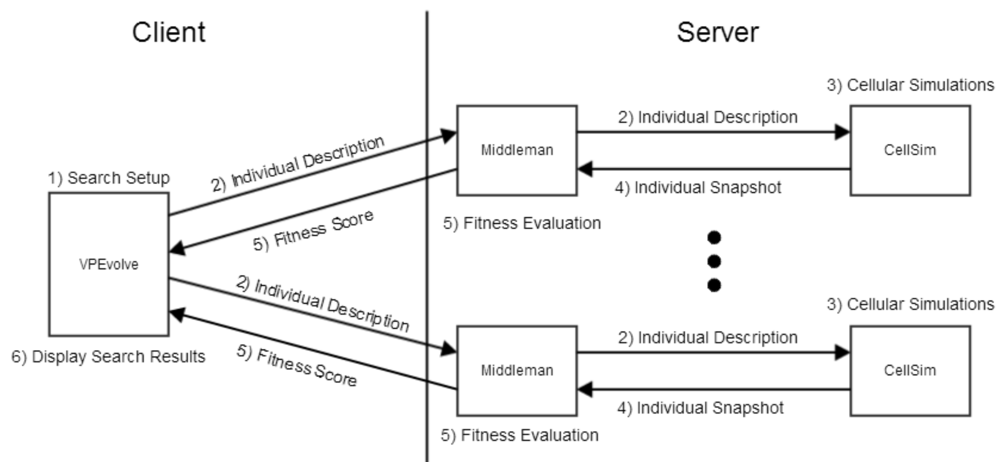
```
server@genesis:~/bicspe/construct/python/src
File  Edit  View  Search  Terminal  Help
[Wed, 11 Mar 2015 11:12:49]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.975406708944 for final value of 0.975406708944
[Wed, 11 Mar 2015 11:12:49]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.931127252969 for final value of 0.931127252969
[Wed, 11 Mar 2015 11:12:49]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.971675403638 for final value of 0.971675403638
NAME IS morph0.975406708944_gen21
NAME IS morph0.975265328545_gen21
NAME IS morph0.931127252969_gen21
[Wed, 11 Mar 2015 11:12:50]::0000::evaluation::INFO    :: generation 21
[Wed, 11 Mar 2015 11:13:29]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.893946323928 for final value of 0.893946323928
[Wed, 11 Mar 2015 11:13:29]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.835297418631 for final value of 0.835297418631
[Wed, 11 Mar 2015 11:13:29]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.945929217509 for final value of 0.945929217509
[Wed, 11 Mar 2015 11:13:30]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.931127252969 for final value of 0.931127252969
[Wed, 11 Mar 2015 11:13:30]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.971118381037 for final value of 0.971118381037
[Wed, 11 Mar 2015 11:13:30]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.971126752597 for final value of 0.971126752597
[Wed, 11 Mar 2015 11:13:30]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.893650557456 for final value of 0.893650557456
[Wed, 11 Mar 2015 11:13:30]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.815001631681 for final value of 0.815001631681
[Wed, 11 Mar 2015 11:13:30]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.975123177686 for final value of 0.975123177686
[Wed, 11 Mar 2015 11:13:30]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.975177723588 for final value of 0.975177723588
[Wed, 11 Mar 2015 11:13:30]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.975058602528 for final value of 0.975058602528
NAME IS morph0.975058602528_gen22
NAME IS morph0.971126752597_gen22
NAME IS morph0.971118381037_gen22
NAME IS morph0.945929217509_gen22
NAME IS morph0.893946323928_gen22
[Wed, 11 Mar 2015 11:13:32]::0000::evaluation::INFO    :: generation 22
[Wed, 11 Mar 2015 11:14:11]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.974921818015 for final value of 0.974921818015
[Wed, 11 Mar 2015 11:14:11]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.97515017027 for final value of 0.97515017027
[Wed, 11 Mar 2015 11:14:11]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.954044624144 for final value of 0.954044624144
[Wed, 11 Mar 2015 11:14:12]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.892916737733 for final value of 0.892916737733
[Wed, 11 Mar 2015 11:14:12]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.957211016117 for final value of 0.957211016117
[Wed, 11 Mar 2015 11:14:12]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.973098406994 for final value of 0.973098406994
[Wed, 11 Mar 2015 11:14:12]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.971646056713 for final value of 0.971646056713
[Wed, 11 Mar 2015 11:14:12]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.81228956229 for final value of 0.81228956229
[Wed, 11 Mar 2015 11:14:12]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.966535073351 for final value of 0.966535073351
[Wed, 11 Mar 2015 11:14:12]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.974892859207 for final value of 0.974892859207
[Wed, 11 Mar 2015 11:14:12]::0000::evaluate::INFO    :: Python : applying multiplier 1.0 to fitness 0.974839807082 for final value of 0.974839807082
NAME IS morph0.97515017027_gen23
NAME IS morph0.974839807082_gen23
NAME IS morph0.973098406994_gen23
NAME IS morph0.971646056713_gen23
NAME IS morph0.957211016117_gen23
[Wed, 11 Mar 2015 11:14:14]::0000::evaluation::INFO    :: generation 23
```

Figure 4.3: CSGA Console Mode

VPEvolve's implementation uses computer resources more efficiently. In particular it provides a significant computational and network bandwidth usage advantage, due to the architecture used in VPEvolve. Figure 4.4 shows a comparison of the architecture design of how the management of running the experiment and deriving a fitness value for an individual is done for both CSGA and VPEvolve. As can be seen CSGA only uses the separate nodes of the cluster to run the CellSim simulation. Next, CSGA is sent the snapshot of the individual and perform the fitness evaluations of each individual sequentially on the node CSGA is running. In contrast VPEvolve uses the cluster node to run both the experiments and the fitness evaluation for an individual. This means that the most time consuming tasks for processing one



(a) CSGA



(b) VPEvolve

Figure 4.4: Client-Server Individual Processing

individual can be run separate from the main software, taking advantage of parallel computation.

Because of this architecture, VPEvolve provides more efficient use of network bandwidth. In both CSGA and VPEvolve the individual description needs to be

sent to run the experiment, which has been about 100KB in size. However, CSGA must be sent an individual snapshot from CellSim at least once for each individual, which can depend on the experiment being performed. The studies being done in the Andersen lab generate snapshots that are about 3MB in size. Therefore with a population size of 1,000 network traffic could peak at 3GB if there are enough nodes to run all simulations for the 1000 individuals in parallel. If CSGA is running on a computer cluster this bandwidth would amount to inter-node traffic, in which case there are a number of high-speed connections available such as InfiniBand which could handle that amount of traffic, but such solutions require expensive equipment. Comparatively, VPEvolve moves the fitness evaluation logic out to the same node as experiment simulation. Therefore, the individual snapshot never needs to be sent back to VPEvolve. After the fitness calculation, VPEvolve will receive a 64-bit number, the fitness value. Therefore, for a population size of 1,000 individuals VPEvolve would use 100MB of network bandwidth compared to at least 3GB for one snapshot per individual with CSGA which is an order of magnitude improvement.

The other area where VPEvolve provides improvement over CSGA is in computational efficiency. Since CSGA only uses parallel processing to run the CellSim simulations, after it receives the snapshot for each individual it must calculate the fitness value for each individual sequentially on the same node that CSGA is running. Therefore, with population size of 100, there is a considerable bottleneck at the end of each generation to calculate fitness values for each individual. Larger populations of 1,000 or more become impractical, since the searches run in the studies performed in the our lab generally take at least 100 generations to converge to a 1.0 fitness. VPEvolve provides a significant improvement over CSGA, as shown in Table 4.1, since the fitness calculations for each individual are done in parallel along with the

simulation before communicating the result back it to VPEvolve. The average times

Runtime Comparison			
Population Size	VPEvolve	CSGA	(CSGA / VPEvovle)
5	8.665	31.214	3.60
10	16.923	34.667	2.00
25	33.473	57.889	1.73
50	60.359	117.899	1.95
100	116.496	185.067	1.59
250	276.577	486.286	1.76
500	556.069	941.791	1.70

Table 4.1: Average Time Elapsed for One Generation in Seconds (over 1000 runs)

shown in Table 4.1 were gathered using a small computer cluster with 8 compute nodes, not including the master node, where each compute node has four 4-core processors. As shown in the fourth column of the table, the ratio of time decreases and converges around 1.7. This convergence happens because at population sizes larger than the number of available cores to run in parallel, the cluster became saturated and the distributed design solution used in VPEvolve stopped being beneficial. In theory, if unlimited resources were available, VPEvolve would run in constant time in relation to the population size because no matter the size of the population, the simulations and fitness calculations could all be run in parallel. Whereas CSGA would be dominated by the fitness evaluation calculation.

4.2 Experiments

In order to derive a comprehensive model of planarian cellular regeneration, simulations are being done in the Andersen lab using CellSim, where the worms are simulated as shown in Figure 4.5, consisting of a simple sheet of cells having a head region (purple) and a tail region (blue). The models in Figure 4.6, are able to reproduce

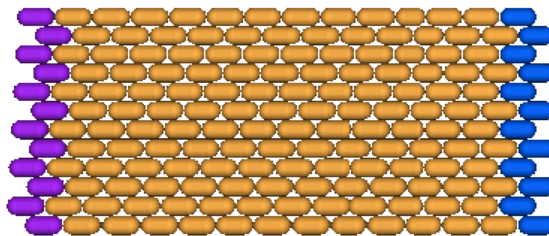
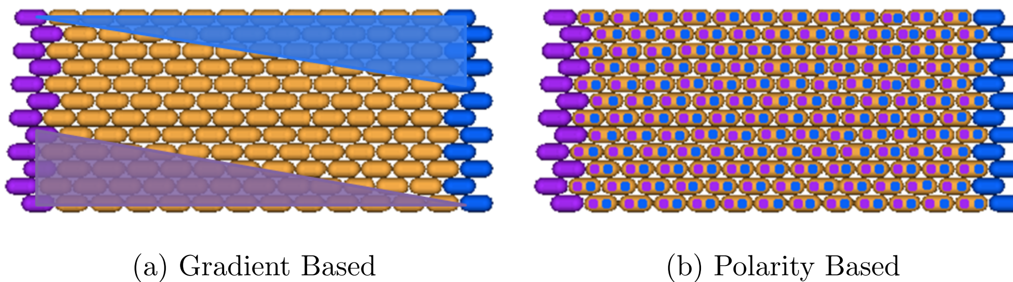


Figure 4.5: Wild-Type Worm

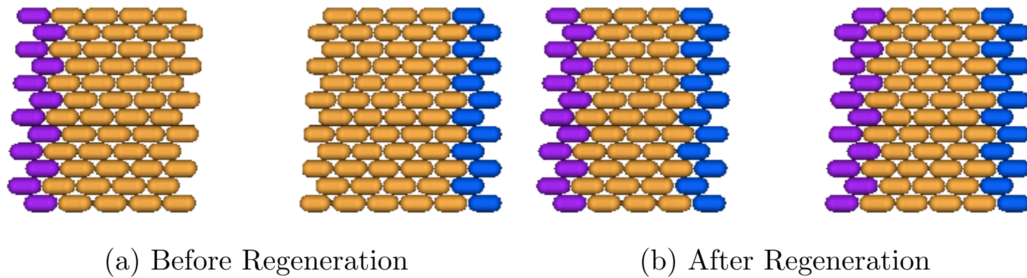
wet-bench experimental results from a single lateral cut as shown in Figure 4.7a. After the single cut the worm is able to regenerate the missing tail on the portion left with the head and vice versa as shown in Figure 4.7b.



(a) Gradient Based (b) Polarity Based

Figure 4.6: Planarian Cellular Models

Figure 4.6a is a model that uses a molecular concentration gradient to signal to cells what cell type needs to be regenerated in the event of cell death. When the single lateral cut is made, the side with the head (purple) knows that it needs to



(a) Before Regeneration (b) After Regeneration

Figure 4.7: Single Lateral Cut Experiment

generate tail near the cut, because it still has the head molecular concentration and the tail concentration has decayed away. Figure 4.6b depicts a model that has two cellular subunits. One subunit points towards the head of the worm, and is labeled with head polarity, and the other subunit points towards the tail and is labeled with tail polarity. In this case, when the single lateral cut happens the correct regions of head and tail are regenerated correctly because the cells along the cut know their orientation and can turn into the correct cell type.

In another experiment where a "shark-bite" portion is removed from the side of the worm, shown in Figure 4.8, the regeneration of neither a head region nor a tail region is expected because the worm is still considered whole. However, both the gradient and polarity models are not able to reproduce the wet-bench experimental results (Figure 4.8) for this experiment, which means there are critical mechanisms missing from either model that keep them from accurately representing planarian cellular regeneration.

To show the ability of VPEvolve to automatically find models of cell regeneration a new experimental setup was used which combines both the gradient and polarity models.The experiment begins by removing the "shark-bite" of cells from the worm, then the worm is allowed to achieve stability. Afterwhich, the worm is completely cut

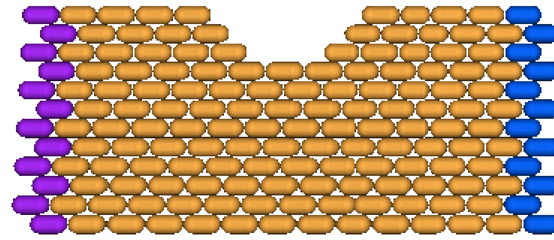


Figure 4.8: “Shark-bite” Cut Target Worm

in half laterally, where a line of cells are removed and then again allowed to achieve stability. For this experimental setup the expected outcome would be the following: After the “shark-bite” cut there should be no regeneration of head or tail because the worm is considered to still be whole, but after the single lateral cut the side portion with the head should regeneration a new tail region and the portion with the tail should regeneration a new head region. When VPEvolve was used to run the GA for this setup a model was was found that produced a 1.0 fitness, meaning it was able to reproduce the wet-bench experimental results for both the “shark-bite” experiment, and the single later cut experiment in tandem as shown in Figure 4.9.

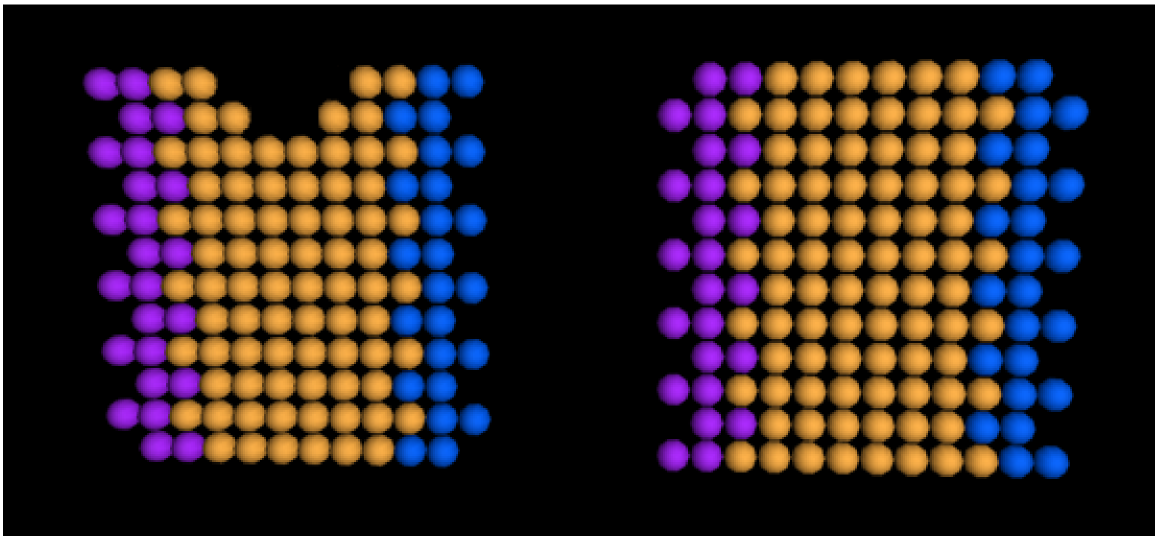


Figure 4.9: 1.0 Fitness Worm Produced by VPEvovle

CHAPTER 5

CONCLUSIONS

VPEvolve is a desktop application aimed to facilitate initialization, running, and monitoring the progress of an evolutionary search. It was created as a free and open source application, which utilizes a VPE to make GA setup more convenient and efficient. Specifically, glyphs and connections are used to represent how individuals flow through the genetic operator application of the GA, which is an intuitive visual representation of how this work-flow proceeds.

VPEvolve offers improvements over a currently used GA setup implementation in the Andersen lab (CSGA), which allow evolutionary searches to be carried out faster and in a more efficient manner. Additionally, these improvements allow VPEvolve to scale better when more resources are available for an evolutionary search with 1) computational power and 2) network bandwidth. This is because VPEvolve manages the calculations for fitness evaluations on the nodes of a computer cluster and send the minimal amount of data over the network.

5.1 Future Directions

A few future directions where VPEvolve can be extended and other features added are outlined below:

Customizable Glyphs

Selection of a set of glyphs in the UI could give an option to create a custom glyph that contains the glyphs and connections of the selection. This would allow sections of repeating glyphs and connections to be more concise. The additional benefit would be that the glyphs and interconnections could be better organized and labeled more concisely, which would make the setup easier to understand.

GUI Hooking and Unhooking

Since the GA logic is completely separate from the UI, a feature could be introduced to allow detaching the evolutionary search from either the console or the GUI. After which you could log out of the remote machine, then login from somewhere else, and reattach the GUI or console to the GA process running on the remote machine. This would allow more flexibility as to how an evolutionary search could be monitored. Allowing researchers to start an evolutionary search at one device, then check the status of the search from another device or location (e.g., home/work).

Proof of Concept and Guide for Using Modeling Platforms

Because researchers might have a preference for a particular platform, or they might want to compare two or more platforms' results, VPEvolve was developed to be loosely connected to CellSim or any modeling platform; therefore, users have the option to choose not only CellSim, but any other modeling platforms for cellular simulations. VPEvolve uses XML to describe an individual, which is what most modeling platforms require to run their simulations. A proof of concept for how this would work with a second modeling platform, and step-by-step guide would improve implementation approachability.

REFERENCES

[1] A. Aziz Aboobaker. Planarian stem cells: a simple paradigm for regeneration. *Trends in Cell Biology*, 21(5):304 – 311, 2011.

[2] T. Adell, F. Cebrià, and E. Saló. Gradients in planarian regeneration and homeostasis. *Cold Spring Harbor perspectives in biology*, 2(1), January 2010.

[3] W. S. Beane, J. Morokuma, D. S. Adams, and M. Levin. A chemical genetics approach reveals h,k-atpase-mediated membrane voltage is required for planarian head regeneration. *Chemistry & Biology*, 18(1):77 – 89, 2011.

[4] Marat Boshernitsan and Michael S. Downes. Visual programming languages: a survey. Technical Report UCB/CSD-04-1368, EECS Department, University of California, Berkeley, Dec 2004.

[5] Tim Boudreau, Jim Glick, Simeon Greene, Jack Woehr, and Vaughn Spurlin. *NetBeans: The Definitive Guide*. O'Reilly Media, Sebastopol, CA, USA, 2003.

[6] Steven D. Bragg and Carl G. Driskill. Diagrammatic-graphical programming languages and dod-std-2167a. In *AUTOTESTCON'94. IEEE Systems Readiness Technology Conference.'Cost Effective Support Into the Next Century', Conference Proceedings.*, pages 211–220. IEEE, 1994.

[7] Margaret M. Burnett. Visual programming. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1999.

[8] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[9] Scott F. Gilbert. *Developmental Biology*. Sinauer Associates, 2000.

[10] Gabor Karsai. A configurable visual programming environment: A tool for domain-specific programming. *Computer*, 28(3):36–44, 1995.

[11] D. Lobo, W. S. Beane, and M. Levin. Modeling planarian regeneration: A primer for reverse-engineering the worm. *PLoS Comput Biol*, 8(4), 2012.

[12] T. H. Morgan. Experimental studies of the regeneration of planaria maculata. *Arch Entwickelungsmech Org*, 7:364–397, 1898.

[13] N. J. Oviedo, J. Morokuma, P. Walentek, I. P. Kema, M. B. Gu, J. M. Ahn, J. S. Hwang, T. Gojobori, and M. Levin. Long-range neural and gap junction protein-mediated cues control polarity during planarian regeneration. *Developmental Biology*, 339(1):188 – 199, 2010.

[14] P. S. Pallas. Spicilegia zoologica quibus novae imprimis et obscurae animalium species iconibus, descriptionibus atque commentariis illustrantur. *Berolini, Prostant, Apud Gottl.*, 8(4), 1774.

[15] C. P. Petersen and P. W. Reddien. Smed-βcatenin-1 is required for anteroposterior blastema polarity in planarian regeneration. *Science*, 319(5861):327–330, 2008.

[16] C. P. Petersen and P. W. Reddien. Polarized notum activation at wounds inhibits wnt function to promote planarian head regeneration. *Science*, 332(6031):852–855, 2011.

[17] P. W. Reddien and A. Sánchez Alvarado. Fundamentals of planarian regeneration. *Annual Review of Cell and Developmental Biology*, 20(1):725–757, 2004. PMID: 15473858.

[18] J. C. Rink, K. A. Gurley, S. A. Elliott, and A. Sánchez Alvarado. Planarian hh signaling regulates regeneration polarity and links hh pathway evolution to cilia. *Science*, 326(5958):1406–1410, 2009.

[19] H. B. Sarnat and M. G. Netsky. The brain of the planarian as the ancestor of the human brain. *The Canadian journal of neurological sciences. Le journal canadien des sciences neurologiques*, 12(4):296–302, November 1985.

[20] Imran Shah and John Wambaugh. Virtual tissues in toxicology. *Journal of Toxicology and Environmental Health, Part B*, 13(2-4):314–328, 2010.

[21] J. M. Sikes and P. A. Newmark. Restoration of anterior regeneration in a planarian with limited regenerative ability. *Nature*, July 2013.

[22] Y. Umesono, J. Tasaki, Y. Nishimura, M. Hrouda, E. Kawaguchi, S. Yazawa, O. Nishimura, K. Hosoda, T. Inoue, and K. Agata. The molecular logic for planarian regeneration along the anterior-posterior axis. *Nature*, July 2013.

[23] Mark Young, Danielle Argiro, and Steven Kubica. Cantata: visual programming environment for the khoros system. *ACM SIGGRAPH Computer Graphics*, 29(2):22–24, 1995.