STAT: Swarm Testing and Analysis Tool

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin–La Crosse

La Crosse, Wisconsin

by

John Cornwell

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

May, 2023

STAT: Swarm Testing and Analysis Tool

By John Cornwell

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

Prof. David Mathias Examination Committee Chairperson	Date
Prof. W. Michael Petullo Examination Committee Member	Date
Prof. Allison Sauppe Examination Committee Member	Date

Abstract

Cornwell, John, D., "STAT: Swarm Testing and Analysis Tool," Master of Software Engineering, May 2023, (David Mathias, Ph.D.).

This manuscript describes the creation of a testing tool to work in tandem with a swarm simulation. Upgrades to the swarm simulation both through its own code, and the creation of the testing tool, allow a user to model the behavior of a swarm as it attempts to satisfy the demand of one or many tasks. The testing tool allows for a convenient way for a tester to create unique swarms and situations that allow for a better understanding of swarm resource allocation. The tool can be used both to create swarms and generate various performace data that can be saved for later review.

Acknowledgements

I would like to express my gratitude to my project advisor Dr. David Mathias for introducing me to the wide world of swarms, and his support in the evolution of this project. I would also like to express my thanks to my family, whom whithout, I would have not been able to pursue further education.

Table of Contents

List of Tables

List of Figures

Glossary

Command Injection

An attck on an application with the goal of executing arbitrary commands on a system.

PLY

Python Lex-Yacc. A Python implementation of the LEX and YACC tools created and maintained by David Beasley.

Flex

Fast Lexical Analyzer Generator. A tool for generating lexical analyzers.

Bison

A tool that converts a LALR CFG into a program that can parse that grammar.

1. Introduction

Maintaining a bee hive involves monitoring and solving many tasks. Worker bees in a hive must protect their queen, produce food, and grow the next generation of bees. It is clear that bee hives have developed efficient ways of solving multiple problems by utilizing their members well [1]. But what if this cooperative problem-solving technique could be used by machines to solve complex problems?

In the past few decades, problem solving techniques using swarms have been proposed and tested as a more efficient problem solving alternative [2]. A swarm is a decentralized system composed of swarm agents that work collectively. An agent is a single member of a swarm that contributes in some way to the collective effort of the larger group. One variation of a swarm is a particle swarm in which agents move around in a search space iteratively until they find their optimal coordinates. Swarms can also be implemented as a group of robots, such as the Swarm-bot project, that uses self-assembling robots working together to solve problems [3].

Swarms can be composed of unique agents that are specialized for a particular task, identical agents that generalize well, or a mixture of both. Swarms that are constructed correctly excel at tackling one or more tasks through careful delegation of available agents. The ability of a swarm to delegate agents gives it the flexibility to accommodate changes in its environment or the problem it is solving. Swarms that are incorrectly constructed will allocate resources too heavily in favor of some tasks while leaving others unsatisfied.

Before humans are able to harness the power of swarm technology, more research must be conducted to understand how to create efficient swarms. Parameterizing swarm behavior is key to understanding how swarms behave, and how they can be constructed to solve various problems. In the same vein, without tools to test the outcome of a particular swarm configuration, it will be difficult improve on the initial swarm design.

Collaboration between The University of Wisconsin-La Crosse and The University of Central Florida has produced a swarm simulation capable of modeling swarm actions. The first version of this software modeled swarm behavior in two-dimensional space. The goal of the swarm in this simulation was to "push" a tracker object onto a moving target. Swarm actions were divided into directional tasks in the North, South, East, and West directions. At any point in the simulation, the swarm would have a demand that it must satisfy in at most two of these tasks. The other two tasks would have opposite (or negative) demand, and would be ignored.

In the Summer of 2021, deveolpment began on two new versions of the swarm simulation. The first was an expansion of the two-dimensional simulation that would model the tracker and target problem in three-dimensional space. The other version sought to shed the constraints of dimensionality and allow a user to define any number of tasks for a swarm to satisfy. This version of the simulation became known as "k-task." This version of the simulation was intended to allow a swarm tester to more accurately model tasks that a swarm might encounter in the real world.

The k-task simulation models tasks through two arguments; a user must give the program a number of tasks to model, and a “demand profile.” These profiles are hard-coded into k-task. A demand profile will split the number of modeled tasks into groups and apply some mathematical function to determine the amount of new demand added to each task in that group every timestep. While this can be useful for starting out, k-task struggles to adapt its task simulation to the specific needs of the tester.

STAT is a tool built on top of the k-task swarm simulation. The goal of STAT is to enhance the testing experience by expanding upon the functionality provided by k-task, and allow for a user-friendly way of generating a swarm test. STAT builds on the previous version of k-task by allowing custom tasks to be defined along with task links. This allows a tester to model tasks that are representative of the real world, along with reintroducing the task dependencies seen in the dimensional simulations. STAT allows for testing of various swarm configurations in an environment that is understandable and organized. Furthermore, STAT provides parameter checking so that the generated swarms are without error and understood by the user. All of these features enhance k-task to provide a testing platform which improves the understanding of the user and expands their swarm generating capabilities.

2. Requirements

This section describes how the software requirements were determined, the life cycle model used during development, and the functional and non-functional requirements of the program.

2.1. The Incremental Process Model

Finding a suitable software process model is paramout to producing a program that is functional and maintainable. Generating the proper documentation will help developers improve their understanding of the software, lead to fewer bugs during implementation, and prevent problems during maintenance and upgrades[4, Section 2]. During the planning phase of this project, the Agile and Incremental development processes were considered. These models make sense because they are manageable for small teams. Both processes have repeatable phases of software development that produce software delverables that may be reviewed and improved upon. Because the requirements were well known ahead of time and were not expected to change, the Incremental process was chosen[5][6].

While considering the Incremental process, it was also decided to include the Prototyping model. Incremental Prototyping is useful because it allows the customer to provide feedback during the development process and it allows the developer to better understand their requirements[7]. In a Rapid Protyping iteration, it is common to quickly develop a prototype for a customer that is later thrown away and rebuilt. In this project, the developers create an incremental product that is developed, presented, and kept as part of the Incremental Prototyping process. This process relates strongly to the Waterfall Model[8], and it begins with a thourough design of the system and consideration of the reqirements. The implementation phase of this project included weekly meetings and reports to Dr. Mathias, which contributed to three prototype releases as the software improved. The feedback from these prototypes did **not** have substantial impact on the requirements, but were instead used to improve the user experience and make implementation decisions.

2.1.1. Stakeholders

Software stakeholders are individuals or groups that have a vested interest in the outcome of a software's development. They may be directly involved in the development process or they may only use the software after it is completed. Below is a list of STAT's stakeholders.

- John Cornwell: Developer of STAT and author of this paper.
- Dr. David Mathias: Project Manager.
- Testers: The group of swarm researchers using STAT.

The developer will be responsible for all planning, developement, and testing of STAT. The Project Manager role in this project functions as both a customer, who will review completed prototypes, and as someone who will evaluate the developer. Finally, the end users are testers who are expected to use STAT to further their understanding of swarms.

2.2. Functional Requirements

The functional requirements below are provided as user stories. Each story represents one scneario in the system that must be possible by the end of the project. Each story describes a user and an action that user wishes to perform.

2.2.1. Use Cases as User Stories

In a normal test scenario, a user (hereafter referred to as tester) of the system will login to their test directory and have the option to begin a new test from scratch, or load in previous test data. A tester will then be taken through the process of setting up their task demand profiles. Decribing task demand profiles may include providing a custom demand function and linking to other tasks' demand. After the tasks are finialized, a tester must provide values for various swarm properties including population size and task selection. Once all properties are entered, the tester may execute their test and view results in both text and graph form. From these use cases, we can gather the following user stories.

1. As a tester, I want to login.
2. As a tester, I want to view my previous test runs.
3. As a tester, I want to create a new test.
4. As a tester, I want to load a previous test into a new test.
5. As a tester, I want to choose the number of tasks to simulate.
6. As a tester, I want to provide demand functions for each task.
7. As a tester, I want to reference another task's demand in a custom demand function.
8. As a tester, I want to set swarm properties.
9. As a tester, I want to view important statistics about a test.
10. As a tester, I want to view graphs relating to a test.

These represent the core of STAT. Other actions may be performed by the system such as validation and interacting with the upgraded k-task simulation, but these are not included here because they are not actions taken by a tester.

2.3. Non-functional Requirements

This section describes functionalities that are not necessarily behaviors, but are important properties for the system to have.

2.3.1. Cross-platform Code

An important goal of STAT is the ability of the system to operate fully on any machine a tester chooses. This goal was clearly communicated at the beginning of the software development process and was kept in mind throughout. There are a few cases where access to low-level functionality of the operating system will require special attention by the developer to ensure STAT is compatible with Mac, Windows, and Linux. These following scenarios are listed below.

1. STAT traverses the filesystem, creates, deletes, reads, and writes to files.
2. STAT executes the upgraded k-task simulation and captures its output.
3. STAT creates graphs as .png files and displays them to the tester.

2.3.2. Security Concerns

STAT is a program that is intended to be used by multiple testers. This provides two general scenarios. In the first, a tester has their own machine and is the only user of the system. From a security standpoint, developers of STAT should be aware of threats that come from an outside network. STAT does not make use of any resources outside of the system it is running on. While STAT does make use of some third party libraries, there is no connection to an outside network. So, the developers need only worry about attacks that may come from users who already have access to the machine that STAT is running on.

This leads to the second scenario: a tester is sharing a machine with other testers. STAT's domain does not require it to store sensitive or personal information. It is a program that is meant to generate, save, and display information relating to test files. These files are not trivial to generate, but their overall value is minimal to the user. Therefore, the login process that is described is simply for STAT to locate a directory in which to operate. The developers and users of STAT do not consider the information generated and saved by the program to be private or worthy of heightened security measures. In fact, users of STAT may wish to share their test data with other users.

One final security consideration that was made during development was the use of a child program. If this program was spawned through a shell, STAT could be vulnerable to a command injection attack. This is also not an issue for STAT as it uses a command that does not spawn a shell and all arguments are hard-coded. Any child process or program spawned by STAT is done without using elevated permissions. Any actions that these processes take will occur within the directory that STAT was executed in.

3. Design and Implementation

This section describes the planning that was done before implementation. This includes a description of the architectural model that was used, along with the envisioned design of the finished product.

3.1. Limitations and Motivations for Change

The k-task simulation was built for the purpose of modeling scenarios that could be found in the real world. It aimed to force swarms to work on a variety of unrelated tasks so that the tested swarms would generalize. The simulation was developed from the previous two-dimensional simulation and provided the same output, but with graphs per task.

However, k-task had several flaws. Among them, k-task only provided preset "demand profiles" that applied to the entire set of tasks. That meant that a tester could not choose to model a specific scenario unless that scenario was already coded into k-task. While effort was made to provide a tester with a variety of unique and interesting task scenarios, k-task was still very limiting in the test cases it could provide. Custom tasks were considered as part of the development of k-task, but the complexity of such an upgrade eventually removed custom tasks from consideration. This ultimately limits the choice and understanding of a tester that is working to understand swarms.

Second, k-task has no task dependencies. This was originally thought of as a benefit to k-task. Without the constraints of dimensionality, it was thought that a tester would be free to model any situation they could dream of. Later, along with the constraints of hard-coded tasks, the developers found that it would be useful in many cases for new demand in one task to be related to new demand in another task. This would be a great way to prevent a set of tasks from spiking in demand at the same time and overwhelming the swarm.

Finally, k-task provides no user interface. All arguments to k-task are provided through two text files. These files specify locations of output files and the arguments that create both the tasks and swarm. As k-task grew (and continues to grow), the arguments that are provided to it also grew. This creates a challenge for any tester or developer who is attempting to understand how arguments affect the simulation. It can be difficult for a user to provide a complete swarm specification without missing an argument or making a mistake in the specification. This created a need for an easy-to-use interface that provides explanations and error checking of the arguments.

Together, these motivations provide the foundation of STAT. This software is intended to provide a solution to the areas that were lacking in the previous k-task simulation. The production of STAT is not meant to completely change the functionality of k-task, but rather improve upon the functionality that it provides. Through the next sections on STAT's design and implementation, the reader will learn how STAT provides a solution to the problems listed above.

In order to solve the problems above, STAT will need to provide a way for users to supply custom demand functions. This will allow the user to model more realistic scenarios and stress the swarm in ways not possible with preset tasks. This will mean that the k-task simulation must be upgraded with a parser and lexer program to recognize these custom demand functions. In order to provide task dependencies, STAT will need to provide a cycle-checking algorithm to identify dependencies and ensure the dependency graph is not cyclic. Finally, STAT must provide a user interface to improve user understanding and interaction with the k-task simulation.

3.2. Architecture Overview

As part of the Incremental Prototyping process, much work is done in the early stages of development to understand the problem and model a solution. The Incremental Prototyping process model requires a design of the software during the Requirements phase of development. A detailed design is possible because the requirements are not expected to change much during development. Before development began on STAT, a layered architecture was developed along with a GUI wireframe. The details for each of these is listed below.

3.3. Layered Architecture

A layered architecture is a way of organizing software into several layers where each layer is composed of modules that perform a specific task. STAT's layered architecture is composed of three layers: a Presentation layer, an Application layer, and a Persistent layer. Each of these layers can be further broken down into modules. The goal of the Presentaton layer is to provide a user interface for the program. All of the modules that exist in this layer are only concerned with collecting and displaying information. The Application layer is responsible for doing computations and checking constraints. Finally, the Persistent layer is concerned with gathering and storing data. In a layered architecture, it is possible for layers to interact with layers that are not neighbors. STAT's layered architecture and final implementation enforces that each layer must only interact with layers that are neighbors to it. This means that the Presentation layer can only access the Persistent layer by first going through the Application layer. This model will only describe software development related to STAT. Any changes that must be made to the k-task simulation are not included in this architecture. Figure 1 provides the notation that will be used for all layered acrchitecture diagrams to follow. Figure 2 shows the highest level of STAT's layered architecture.

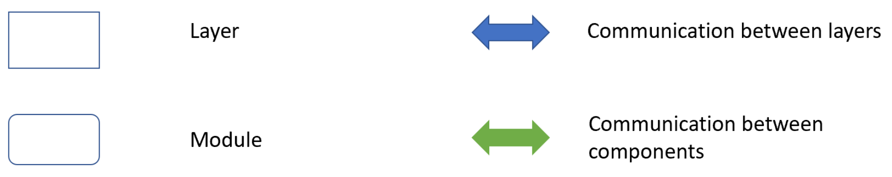

Layer
Communication between layers
Module
Communication between components

Figure 1. Architecture Notation

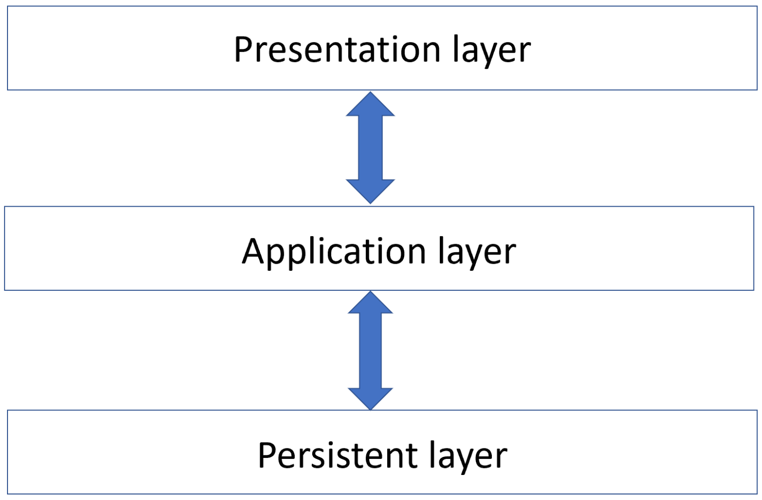


Figure 2. Layered Architecture

3.3.1. Presentation Layer

The Presentation layer is split into several modules. Each module represents a screen that a user can interact with. These screens will provide or request information, but they will not do any computation. This layer also describes the interactions between screens in this layer and which screens use resources from the Application layer. This allows the developers to see how a user will progress through the program and which screens will integrate with the Application layer. Each screen has a singular purpose related to its name. For example, the Previous Run Browsing page is considered the home screen and is concerned only with displaying old tests. From there, a user can choose to begin a test, or navigate to the Previous Run Review screen to view more detailed information from a prior run. No further refinement is necessary in this layer, because this level of detail has broken down the design to a level just above coding implementations. This layer will satisfy the requirement of a user interface for STAT.

3.3.2. Application Layer

The Application layer is responsible for performing computations on data provided by the user. It will also work on data provided by the Persistent layer after a swarm test is performed. In order to separate the Presentation layer from the Persistent layer, another function of the Application layer is to provide a bridge between these layers. Figure 3 shows how these responsibilities are satisfied by the Application layer. The layer bridging is satisfied by a "Data Access" module that will handle data collection from the Persistent layer and pass it to the Presentation layer. The Sim Communication module is responsible for working on data provided by the user through the Presentation layer. Specifically, this module provides data validation for swarm parameters. Lastly, the Sim Analysis module takes information gathered from the Persistent layer after a simulation is run and generates useful statistics. Together, these modules encompass all of the computation that STAT must do.

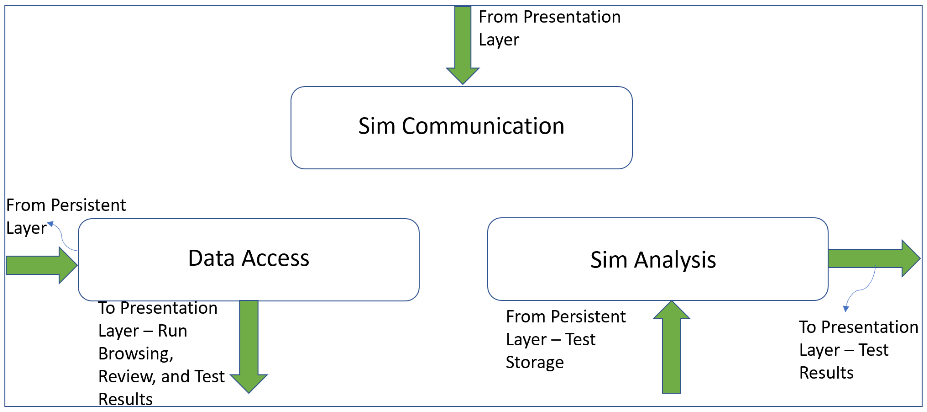

From Presentation Layer
Sim Communication
From Persistent Layer
Data Access
Sim Analysis
To Presentation Layer – Run Browsing, Review, and Test Results
From Persistent Layer – Test Storage
To Presentation Layer – Test Results

Figure 3. Application Layer

3.3.3. Sim Communication

Figure 4 shows more detail on the design of the Sim Communication module in the Application layer. Here, two more modules are revealed: the Cycle Check and Sim Setup modules. These modules satisfy the responsiblilities of Sim Communication to validate swarm parameters and provide that information to the k-task simulation. The Cycle Check module is concerned with ensuring that any custom demand functions provided are legal math expressions and that dependencies between these functions do not create a cycle. The Sim Setup module is concerned with validating ranges of swarm parameters and ensuring that combinations of these parameters are legal. From Figure 4, it is clear that Sim Setup will need to use the Cycle Check module to confirm that all demand functions are valid. Information is then sent back to Sim Setup about the custom function validity, along with a topological ordering of these functions. Because any further refinement would constrain the developers to a specific implementation, no more detail on the structure of these two modules is provided in the design.

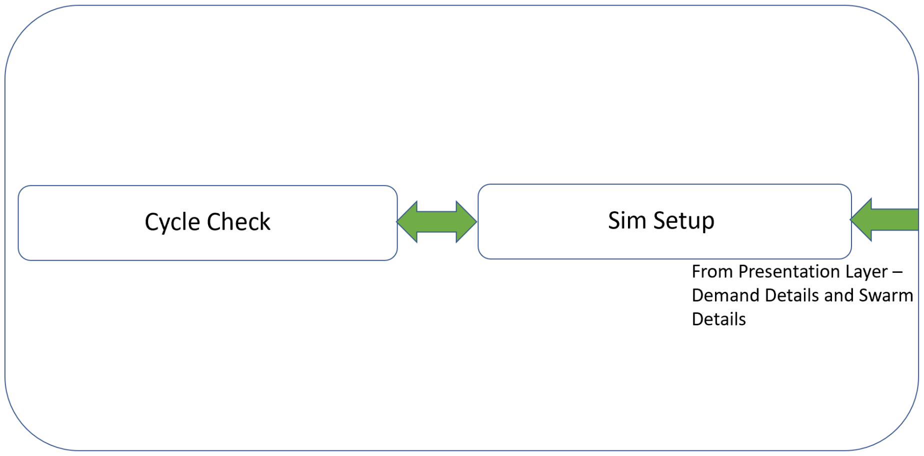


Figure 4. Sim Communication

3.3.4. Sim Analysis

The Sim Analysis module can be broken down further into two modules: Sim Graph and Collect Stats. This is shown in Figure 5. These modules are used for generating graphs from k-task's output files and computing simple stats about total swarm performance. Sim Graph will retrieve data from the Persistent layer and generate graphs that may be later used by the Presentation layer. A similar process is used by the Collect Stats module. Neither of these modules are interacting with the file system; they are only transforming the data they receive from the Persistent layer. Similarly, these modules are not direcly interacting with the user to display the data they have generated. That is left to the Presentation layer. Since these modules have few responsibilities, no further detail has been modeled.

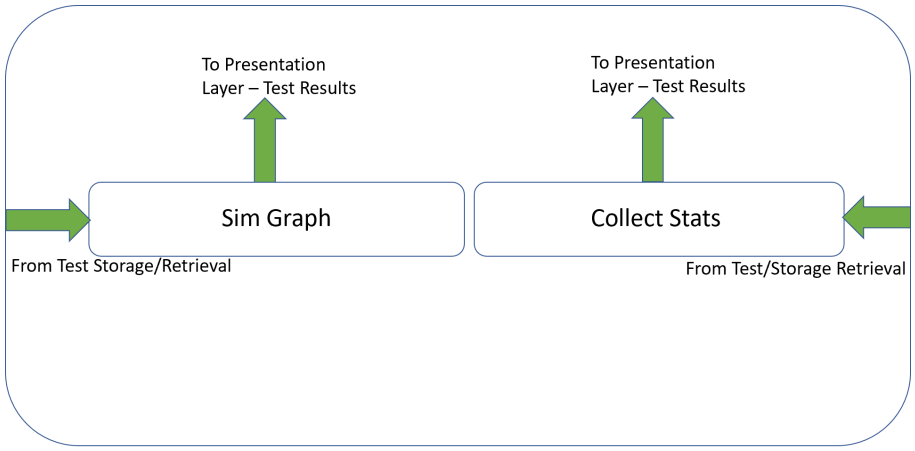

To Presentation Layer – Test Results
To Presentation Layer – Test Results
Sim Graph
Collect Stats
From Test Storage/Retrieval
From Test/Storage Retrieval

Figure 5. Sim Analysis

3.3.5. Persistent Layer

Figure 6 shows a detailed view of the Persistent layer. The Persistent layer is only concerned with gathering data from the filesystem. This layer will not do any computation; it is only concerned with providing formatted data to the application layer. Since the k-task simulation produces output on a per-timestep basis, there is no reason to store data in an object. Therefore, there are no data access objects (DAOs) shown in this model. Instead, this layer need only produce relevant statistics in an array. This is the main function of the Test Access module in the Persistent layer. This module handles retrieving data from a directory that it accesses from the User Directory Access module. User Directory Access is primarily used to set up and access a specific swarm tester's environment. This module takes care of directory creation, access, and deletion. By having all directory and file access occur within the Persistent layer's modules, the developers of STAT will have a much easier time ensuring that the program is able to interact with both Windows and Unix filesystems.

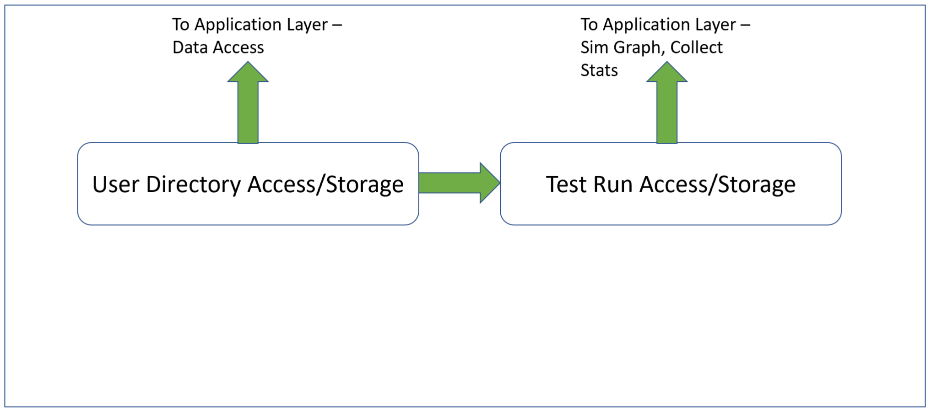

To Application Layer – Data Access
To Application Layer – Sim Graph, Collect Stats
User Directory Access/Storage
Test Run Access/Storage

Figure 6. Persistent Layer

3.4. UI Design

The design phase of this project also included UI modeling prior to development. Because Iterative Prototyping is not as flexible in changing requirements as an Agile process, it is important that all software requirements are understood before development begins. This does not mean that no changes can be made, but drasic changes in requirements are not supported by this process. In order to communicate with the Project Manager that all of the requirements are understood, UI wireframes were created to show how a user might navigate through STAT to fufill each requirement. Not every page in STAT was modeled, because some pages included only a single choice in progression, or they consisted mostly of text fields to be filled. The pages to be modeled were taken from the Persistent layer described previously and include the Previous Run Browsing, Previous Run Review, and Test Results pages.

Figure 7 shows the design of the Previous Run Browsing Page. This page is intended to be the home screen for a swarm tester. Buttons are included for viewing information relating to a previous run, deleting a previous run, or loading a previous run into a new test. There is also a button to generate a completely new test. This screen also includes a scrollable area where all previous test runs created by the current swarm tester can be selected. This screen provides many paths for which a swarm tester may take. They can navigate to the Demand Specification screen with old test data or a new test. They can also go to the Test Review screen for an old test. Finally, the swarm tester could choose to logout. Logout functionality is implemented as a button that appears on every screen. The logout button is not shown to minimize distractions from a screen's purpose.

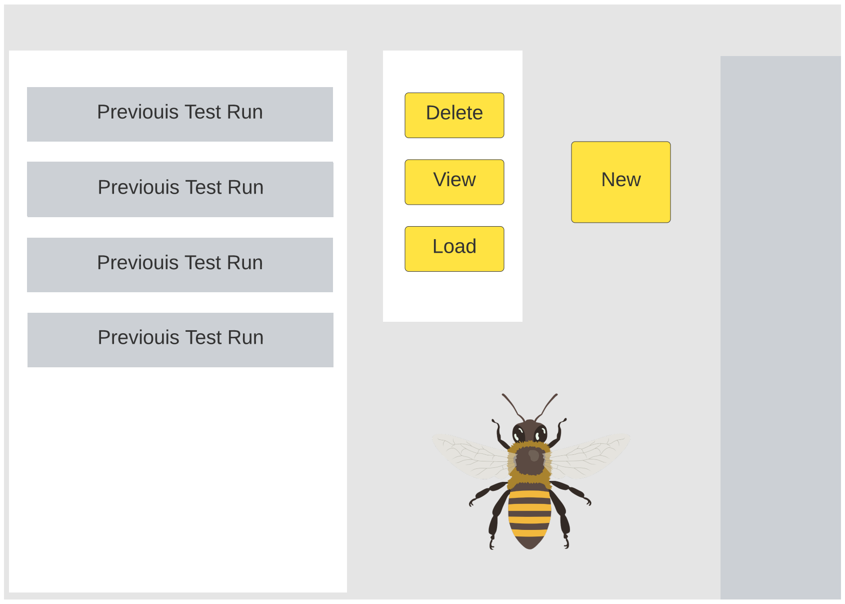


Figure 7. Previous Run Browsing Page

Figure 8 shows the design for the Run Review screen. This screen layout was intended to be used for both displaying details from a previous run and displaying details for a new run prior to execution. This screen features two scrollable areas detailing demand parameters and swarm properties. Functionality for a the Previous Run Review screen includes loading the displayed test into a new test and viewing the Test Results page for this test. A swarm tester may also choose to go back to the home screen. For a Run Review screen that is generated just before a new test's execution, the only functionality will be to start the test or go back.

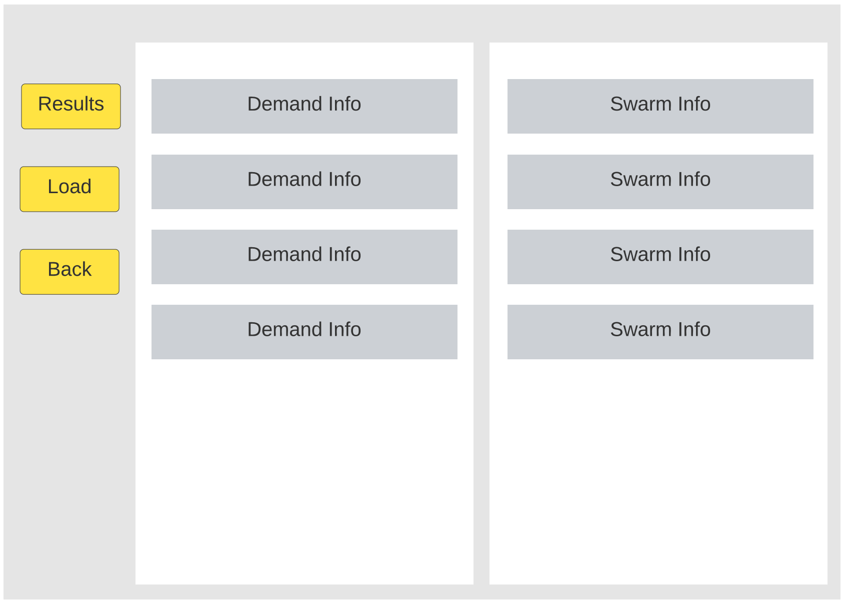


Figure 8. Run Review Page

Finally, Figure 9 shows the design for the Test Results page. This page includes a section with a high-level overview of how the swarm performed and a section for graphs. The graphs that were to be included in STAT had not been fully decided upon, so the names of all available graphs are not included. The swarm performance metrics are to be displayed just above the graph button section. If a swarm tester navigated to this screen from the Previous Run Details screen, they will have the option to go back or go home. In the case where a swarm tester has just completed a test, they will only have the option to navigate home.

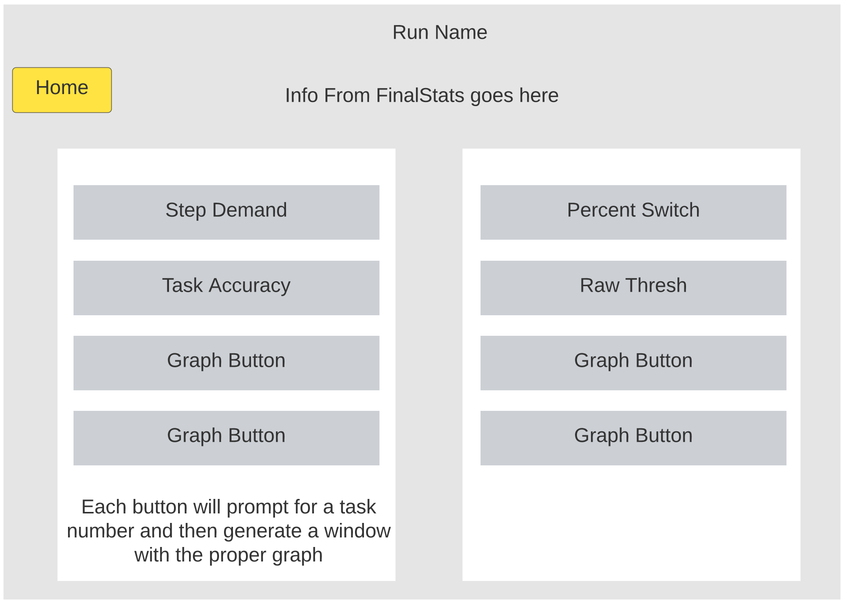

Run Name
Home
Info From FinalStats goes here
Step Demand
Task Accuracy
Graph Button
Graph Button
Each button will prompt for a task number and then generate a window with the proper graph
Percent Switch
Raw Thresh
Graph Button
Graph Button

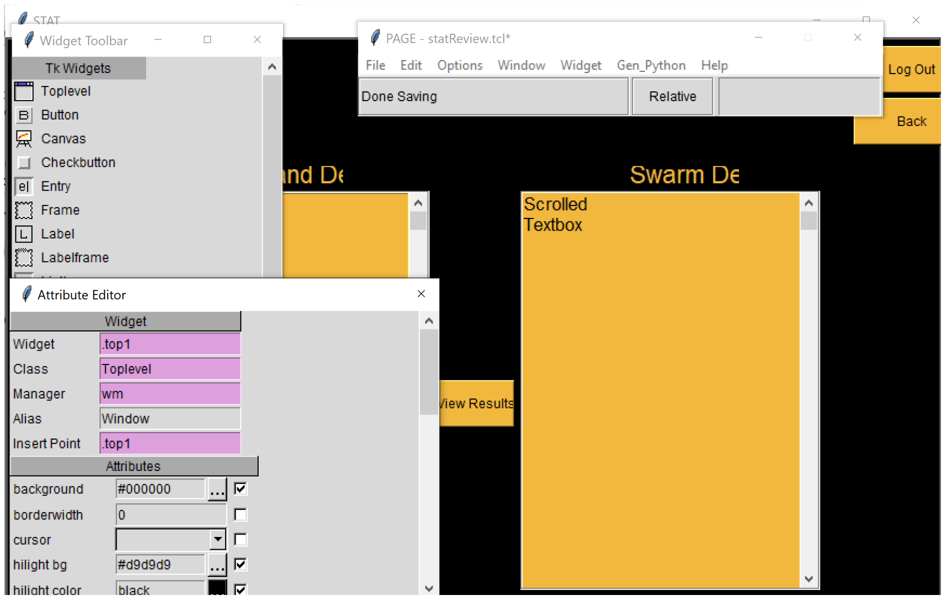
Figure 9. Test Results Page

4. Implementation

This section describes the implementation details of STAT and the new k-task simulation. This section will show how the functional requirements are present in the final software.

4.1. PAGE Tool

The first stage of development and first prototype focused on building a full user interface for STAT. In order to build an interface that was visually pleasing, organized, and consistent, a tool was needed. The PAGE tool is a program that allows a user to construct user interfaces by "dragging and dropping" elements onto the screen[9]. This tool is able to generate Python code from the tkinter library. The PAGE tool is not intended to fully build a user interface. Instead, it works to eliminate the time needed to generate boilerplate code and organize interface elements on a screen. Figure 10 shows an example of a GUI that is built inside of the PAGE tool.



Widget	
Widget	.top1
Class	Toplevel
Manager	wm
Alias	Window
Insert Point	.top1

Attributes		
background	#000000	☑
borderwidth	0	☐
cursor		☐
hilight bg	#d9d9d9	☑
hilight color	black	☑

Figure 10. PAGE Tool

After a screen is built, two files are generated: a support file containing a driver function with a list of callback functions and a class file containing the code to create the window. STAT's user interface structure is a single "toplevel" container that contains several frames, each of which contains a screen for a user to interact with. When a user navigates to a screen, a frame may be created if it does not exist or pushed to the top where it is visible. PAGE provides the code to generate the frames, but it does not provide code for functionality. Additionally, much of the code generated by PAGE needed to be refactored to maintain organization and integrate with the rest of STAT.

Most of the utility of the PAGE tool is from its ability to generate widgets and their placements on the screen quickly. Using the GUI wireframes discussed in Section 3, development of the first iteration of pages in STAT did not take long. Since PAGE is a drag-and-drop tool, changes to the design of a screen can be made very quickly. PAGE's library of tkinter widgets was also very useful in providing a number of scrollable widgets. This was very helpful as there are many places in STAT where an indeterminate-length list is displayed to the user.

4.2. The Filesystem

One of STAT's non-functional requirements is to support multiple operating systems. Many operations that STAT must perform will require it to interact with the filesystem. Python supplies cross-platform methods for moving, deleting, opening, and creating files. However, path separators on Windows and UNIX machines are different. To solve the issue of path traversal in STAT, functions in the Persistent layer make use of the normpath function. This function normalizes a given path to the filesystem of the machine currently running STAT. Figure 11 shows an example of how normpath is used to find and return a test directory.

```python
# This method returns the path of a given test name for the logged
# in user or the empty string
def find_test(name):
    if os.path.exists(normpath(directoryAccess.get_user() + "/" + name)):
        return normpath(directoryAccess.get_user() + "/" + name)
    else:
        return ""
```

Figure 11. Normalize Path

STAT must also execute the k-task simulation. However, executable files on Windows present differently than on other operating systems. Figure 12 shows another example of normpath along with the differences in executable files between Windows and other operating systems.

```python
def run():
    line1, line2, line3 = "", "", ""
    if sys.platform == 'win32' or sys.platform == 'cygwin':
        cmd = [os.path.normpath(os.getcwd() + "/Sim/sim.exe"), "params", "
    opfiles"]
    else:
        cmd = [os.path.normpath(os.getcwd() + "/Sim/sim"), "params", "
    opfiles"]
    process = subprocess.Popen(args=cmd, stdout=subprocess.PIPE, stderr=
    subprocess.PIPE, universal_newlines=True,
                               cwd=os.path.normpath(os.getcwd() + "/Sim"))
    for stdout_line in iter(process.stdout.readline, ""):
        line3 = line2
        line2 = line1
        line1 = stdout_line
        yield stdout_line
    process.stdout.close()
    return_code = process.wait()
    if return_code:
        raise subprocess.CalledProcessError(return_code, cmd, output=line3
    )
```

Figure 12. Program Execution

Finally, STAT must prepare and show graphs pertaining to the tested swarm's performance. All graphs are generated at the end of the test's execution and then stored in the filesystem for later viewing. Opening these files in a separate window requires an "open" command. Once again, for STAT to open a file, it must first determine what filesystem it is using and then provide the proper command. Figure 13 shows how STAT uses different commands to open files. Windows, for example, has its own file opening command called startfile that will use the default application when opening a file. Once again, this function makes use of the normpath method to normalize filepaths for the filesystem STAT is using.

```python
# This function retrieves a graph for the given test and graph name
def open_graph(test_name, graph_name):
    test_dir = testAccess.find_test(test_name)
    if testAccess.file_exists(normpath(test_dir + "/" + graph_name)):
        if sys.platform == "win32" or sys.platform == "cygwin":
            os.startfile(normpath(test_dir + "/" + graph_name))
        elif sys.platform == "darwin":
            subprocess.call("open", normpath(test_dir + "/" + graph_name))
        else:
            subprocess.call(["xdg-open", normpath(test_dir + "/" +
    graph_name)])
        return ""
    return "Could not find the {} graph.".format(graph_name)
```

Figure 13. File Opening

4.3. Changes to K-task

Section 3.1. provided a number of limitations in the k-task simulation were found. The development of STAT addresses the user interface need, but there were also features missing from the simulation. The most important feature that the k-task simulation must implement is custom tasks. Without the ability to define custom tasks, a swarm tester is very limited by the tasks that the simulation provides. They will not be able to test new scenarios that the programmer of the simulation did not think of. Even small changes to the configuration of tasks can have a large impact on the outcome of the simulation. These changes cannot be made easily if the swarm tester must make use of a preset library of tasks. It would be very inefficient for a programmer to code every conceivable task into the simulation.

Similarly, the ability to reference the demand of one task in another task function is important. When the k-task simulation moved away from modeling dimensional problems, it also lost task dependencies. However, there are many real-world instances where task dependencies can be found. For example, a swarm tester might want to recreate the problem modeled by the old simulation in the new k-task simulation.

To solve these problems in k-task, it is necessary to create a parser that can read in custom demand functions and allow k-task to evaluate them every timestep. This parser needs to recognize standard math operations and task references. From the custom task, the parser must produce a parse tree that can be evaluated every timestep by the k-task simulation. In order to implement a parser, the Flex and Bison programs were used to specify rules and generate code. This process is given more detail in the next section.

4.4. Flex and Bison

Flex is a program that creates a lexical analyzer that generates tokens from an input string. The k-task simulation makes use of these tokens in its grammar specification. Bison is a parser generator that turns a context-free grammar into a parser. Using the generated lexical analyzer and parser, the k-task simulation can recognize and store mathematical expressions. Figure 14 shows a portion of the rules written to recognize valid tokens in mathematical expressions given to the simulation.

```
Sin {return SIN;}
Cos {return COS;}
Tan {return TAN;}
Sqrt {return SQRT;}
\+ {return '+';}
\- {return '-';}
\* {return '*';}
\/ {return '/';}
\% {return '%';}
\^ {return '^';}
\( {return '(';}
\) {return ')';}
```

Figure 14. Flex Rules

The Bison-generated parser is responsible for collecting these tokens and building parse trees that represent the expression read in. The grammar rules recognize sequences of tokens and execute some action. The parser will "reduce" the recognized set of tokens into a single token that can be recognized in another rule in the parser. Figure 15 shows a part of the grammar used by the k-task simulation.

```
1 Term      :     Term '%' Factor      {$$ = doOp2($1, $3, " % ");}
2 Term      :     Factor               {$$ = $1;}
3 Factor    :     Num '^' Factor       {$$ = doOp2($1, $3, " ^ ");}
4 Factor    :     Num                  {$$ = $1;}
5 Num       :     SIN '(' Expr ')'     {$$ = doOp1($3, "Sin");}
6 Num       :     COS '(' Expr ')'     {$$ = doOp1($3, "Cos");}
```

Figure 15. ParserRules

Line three shows an example of a reduction where the parser recognizes the token "Num" followed by the "^" token and a "Factor" token. When these tokens are found, the parser will execute the doOp2 function using the recognized tokens as arguments. Then, the parser will reduce these three tokens into a new "Factor" token.

The result of the lexer and parser consuming a mathematical expression will be a parse tree. The purpose of the tree is to store the semantics of the expression in a way that can be easily analyzed by the k-task simulation. These trees can be traversed very quickly by the simulation to determine the value of the expression every timestep. A tree node contains a node type, pointers for up to two children, a math operation to be performed, and a place to store data values. Figure 16 shows one node in the parse tree. Every timestep during execution, the k-task simulation will traverse the tree and perform mathematical operations and variable lookups to determine the value of the tree.

```c
/* struct for storing a parse tree of a functon */
struct MathFunct
   {
   char type;
   struct MathFunct * v1;
   struct MathFunct * v2;
   char op[4];
   Data * data; //for any literals or variables
   };
```

Figure 16. Tree Node

The parse tree described supports basic mathematical operations that can be found on a simple calculator. This parser also supports the classic order of operations which may be circumvented through use of parentheses. Complex mathematical expressions using integrals, piecewise expressions, etc. are not supported by this parser.

4.5. PLY

Before a test is executed by STAT, several checks are performed on the given parameters to ensure that they are allowed by the k-task simulation. Catching errors early on can help a tester find and fix problems in their specification before they run a faulty simulation. An important part of the error-checking process is ensuring that any custom demand functions provided are correct. Any incorrect functions should be flagged for the user to fix. The simplest way to ensure correctness of mathematical expressions is to reproduce the lexer and parser discussed earlier inside of the STAT program. To do this, STAT uses a library called PLY.

PLY stands for Python Lex-Yacc. It is a convienient Python implementation of the parsing tools that were discussed previously. While the syntax is slightly different, the library implements almost all of the functionality of the original C tools. PLY was developed by David Beasly as a teaching tool, but it has improved much since its inception. After providing both a lexer and parser rule file, the PLY library will generate a lexer and parser that can consume input and take action based on matched rules[10].

The lexical analyzer in PLY uses regular expressions just like Flex. Figure 17 shows how the program handles creating tokens for strings defined in a reserved words list. The regular expression that this rule matches is included in the first line in the method. All methods recognized by the lexer must begin with a "t_" followed by an appropriate token name. Because this rule is going to catch any strings that have valid characters, this rule needs to ensure that illegal tokens are marked in a way that the parser can identify. In this case, the token will not receive a type, and the parser will be unable to match it in a rule.

```python
# Handles tokens from the reserved list
def t_ID(t):
    r'[a-zA-Z][a-zA-Z]*'
    if t.value in reserved:
        t.type = reserved[t.value]
        return t
    else:
        # this word is not in the reserved list, so it is an illegal
        # token that will be handled by the parser
        return t
```

Figure 17. PLY Lexer

The parser syntax uses the same comment rule style shown above. In the case of the parser generator, the rule comment contains sequences of tokens following a result token. All rule methods must begin with "p_" followed by a rule name. Just as before with the Bison parser, the PLY parser will build a parse tree from the matched tokens. The parser will also keep track of other demand functions that are referenced in the expression that is currently being parsed. Figure 18 shows how the parser will identify task references and store these references in a graph which will be discussed later.

```python
def p_name_to_id(p):
    '''id : NAME'''
    if p[1] == "$t":
        # this is a reference to the timestep value
        p[0] = pt.Data(p[1], 0)
    elif not 0 <= int(p[1][2:]) < globals.numTasks:
        globals.parserError = True
        globals.errorMsg = "Illegal task reference {}".format(p[1])
    else:
        if p[1] == "$t{}".format(globals.task):
            # This is a self-referential task, so we will not add a cycle
            pass
        else:
            # Add this function as a child of the task it references (
duplicates are ignored)
            globals.graph.find_node(p[1]).add_child(globals.node)
        # add a data node to the tree with the value of the named task
        p[0] = pt.Data(p[1], 0)
```

Figure 18. PLY Parser

4.6. Cycles in Custom Tasks

One challenge that is presented by task references is a task reference cycle. When a swarm tester is creating a list of tasks to simulate, they may choose to have their custom tasks reference the demand of other tasks. There is no limit to the number of times one task's demand function may reference another task's demand. However, when the k-task simulation is determining a task's demand for the current timestep, it must know which tasks to evaluate first. When the k-task simulation is determining the demand to add to a task in a timestep, it must first know the value of any task referenced by the task it is evaluating. This means that the k-task simulation requires a order in which to evaluate tasks that ensures all tasks that are used as references are evaluated before the tasks that use these references.

STAT is in charge of both ensuring the correctness of the custom demand functions and the generation of the task evaluation list. In order for STAT to determine the order of evaluation, it creates a directed graph where each node in the graph represents a task, and each edge represents a task reference. Since each edge in this graph is directed, STAT knows the referenced task, or "parent" task, and the referecing task, or "child" task. After all custom tasks have been parsed, STAT must ensure that the resulting graph is a DAG: a directed acyclic graph. This means that there must be no cycles where a graph is a parent of itself. If the graph contains a cycle, the user must not be allowed to continue in the program. If the graph has no cycles, then STAT must generate an ordering such that every parent appears before any of its children.

4.6.1. Graph Coloring

Graph Coloring is a technique used to detect cycles in a directed graph. This method uses colors to indicate which nodes in a graph have been checked, are being checked, and which have yet to be checked. STAT uses graph coloring to satisfy the requirement of ensuring a DAG for the task graph it creates. Figure 19 shows a portion of STAT's cycle checking implementation which recurses over the graph in an attempt to find a grey node. Grey nodes indicate that the cycle checking algorithm is in the process of traversing a route in which the grey node resides. After all nodes in a route are checked, they are colored black and do not need to be checked again.

```
    def cycle_check_util(self, node):
        # A WHITE node has not yet been checked.
        # A GREY node has started being checked.
        # A BLACK node has been checked.
        node.color = "GREY"

        # if any adjacent node is GREY, then there is a loop.
        for child in node.children:
            if child.color == "GREY":
                return True
            elif child.color == "WHITE" and self.cycle_check_util(child):
                return True

        # mark node as fully processed
        node.color = "BLACK"
        return False
```

Figure 19. Graph Coloring

4.6.2. Khan’s Algorithm

Khan’s Algorithm is used to generate a topological ordering from a directed acyclic graph. Given a DAG, Khan’s Algorithm will produce a list where each parent node appears before any of its children. This method is used by STAT to create the order of evaluation used in the k-task simulation. Since STAT has already confirmed that the task graph is a DAG, this algorithm can rely on that fact to find at least one node with no parents. These nodes will comprise the start of the list. This method will then cycle through the process of removing the first node from the list and adding on its immediate children. The nodes that are removed are added in order to the final list. Because Khan’s Algorithm requires a DAG as input, it guarantees that all nodes in the graph will appear in the final sorted list.

```python
    # implements topological sort using Khan's Algorithm
    def sort(self):
        # number of parents per node
        degree = list(range(0, len(self.nodes)))
        # nodes that can be added to the sorted list at any time
        queue = list()
        # list of nodes sorted topologically
        sorted_nodes = list()
        for i in range(0, len(self.nodes)):
            degree[i] = self.nodes[i].num_parents
            if degree[i] == 0:
                # this node has no parents, so it may be added to the sorted list.
                queue.append(self.nodes[i])
        while len(queue) != 0:
            # there will be at least one node with no parents because this is a DAG
            first = queue.pop(0)
            sorted_nodes.append(first)
            for child in first.children:
                # find all children of the node we are adding
                for i in range(0, len(self.nodes)):
                    if self.nodes[i] == child:
                        # subtract degree of node
                        degree[i] -= 1
                        if degree[i] == 0:
                            queue.append(child)
        self.nodes = sorted_nodes
```

Figure 20. Khan’s Algorithm

5. Testing

5.1. Black-Box Testing

Black-box testing is a type of testing where the tester has no access to or knowledge of the code. In this kind of testing, all tests must be run through the interface that the program provides. When considering how to test STAT, the development team found three areas of the software that made good candidates for Black-box testing. STAT has a swarm specification screen that asks the user to enter many values and then validates those entries. From the task and swarm specification, STAT will produce a params file that contains all entered data. Finally, once a simulation is complete, the k-task simulation produces a file containing all of the parameters it was able to read and understand. This provides three places where a tester can check the validation of STAT and the modified k-task simulation: the swarm screen, the original params file, and the result params file.

5.2. Swarm Specification Testing

The text fields on the swarm specification page accept numerical values that are often in a range and may allow for individual values outside of that range. To test that STAT only accepts values in these ranges, a combination of Boundary Value Analysis and Equivalence Class Partitioning was used. Equivalence Class Partitioning beaks up the range(s) of input values into classes. Classes consist of only valid or only invlid inputs for the range they describe. In Boundary Value Analysis, a tester constructs test cases using the boundaries of a range of data[11]. Using the equivalence classes, a tester will use test data that tests both ends of the class, and a central value.

When testing input values for STAT, each input had a number of equivalence classes assigned to it depending on the number of valid ranges of data that the input field would accept. Then, the boundaries of each class were tested using the values at the boundary and a value close to each side of the boundary. Additionally, a "nominal" or middle value was tested that resides inside of the class.

Table 1 shows an example of equivalence classes that are derived for the "Thresh Init" field. This field is responsible for telling the k-task simulation how it should generate threshold values for the swarm agents. Threshold values tell an agent that it should act on a task if the current demand is greater than the threshold. The allowable values for this field are values between zero and one inclusive, two, three, four, five, six, and ten. All other values should be disallowed. This creates a "disallowed" class for values less than zero, greater than ten, and for each range between the allowed constants.

Invalid Input	Valid Input
$\{x \mid x < 0\}$	$\{x \mid x \geq 0.0 \text{ and } x \leq 1.0\}$
$\{x \mid x > 1.0 \text{ and } x < 2.0\}$	$\{2.0\}$
$\{x \mid x > 2.0 \text{ and } x < 3.0\}$	$\{3.0\}$
$\{x \mid x > 3.0 \text{ and } x < 4.0\}$	$\{4.0\}$
$\{x \mid x > 4.0 \text{ and } x < 5.0\}$	$\{5.0\}$
$\{x \mid x > 5.0 \text{ and } x < 6.0\}$	$\{6.0\}$
$\{x \mid x > 6.0 \text{ and } x < 10.0\}$	$\{10.0\}$
$\{x \mid x > 10.0\}$	

Table 1. Equivalence Classes

Table 2 shows the boundary values that are created for each of the classes listed. Some of these values are duplicates and only need to be tested once. For each class, each value is tested with all other input fields having a valid value. If any test case passes STAT's validation where it should fail, or fails where it should pass, then the input field fails the test and the code must be fixed. The table begins with a class of values that is not acceptable, and then alternates between acceptable and unacceptable equivalence classes until the entire range of floating-point values has been tested. This method of testing greatly reduces the number of values that need to be tested to ensure field validation is correct. With this method, each input's range is ensured to stop at the boundary and allow values between the boundary if that range is acceptable. For single values such as those listed in Table 2, it is guaranteed that neighboring values are disallowed, while the single-value class is accepted. While Boundary Value Analysis and Equivalence Class Partitioning does not provide an exhaustive list of test values, it identifies the important test cases where a coding mistake is likely to be made.

Input Name:	Class	Min-	Min	Min+	Mid	Max-	Max	Max+
Thresh Init	{< 0}	-11	-10	-9	-5	-2	-0.1	0
	{0.0 – 1.0}	-0.1	0	0.1	0.5	0.9	1	1.1
	{> 1.0 and < 2.0}	1	1.1	1.2	1.5	1.8	1.9	2
	{2}	NA	NA	NA	NA	1.9	2	2.1
	{> 2.0 and < 3.0}	2	2.1	2.2	2.5	2.8	2.9	3
	{3}	NA	NA	NA	NA	2.9	3	3.1
	{> 3.0 and < 4.0}	3	3.1	3.2	3.5	3.8	3.9	4
	{4}	NA	NA	NA	NA	3.9	4	4.1
	{> 4.0 and < 5.0}	4	4.1	4.2	4.5	4.8	4.9	5
	{5}	NA	NA	NA	NA	4.9	5	5.1
	{> 5.0 and < 6.0}	5	5.1	5.2	5.5	5.8	5.9	6
	{6}	NA	NA	NA	NA	5.9	6	6.1
	{> 6.0 and < 10.0}	6	6.1	6.2	8	9.8	9.9	10
	{10}	NA	NA	NA	NA	9.9	10	10.1
	{> 10}	10	10.1	10.2	15	18	19	20

Table 2. Boundary Values

5.3. Params File Test

Before a test may be run in the k-task simulation, STAT must gather all of the test parameters and provide them to the simulation via a params file. This provides a convienient way for a tester to ensure that STAT provides all input to the simulation. It is a trivial task to walk through the params file and ensure all swarm input and task specifications are present. Furthermore, STAT organizes the params file to separate the swarm and task specification for help with debugging.

If a user of STAT wants to rerun a previous test, the params file from that test will be loaded into STAT. This provides another simple test case to ensure that STAT is reading from the proper test directory, reading the params file correctly, and writing the params file with the correct data. To perform this test, a sample test was created and executed. Then the same test was rerun without any modification of the swarm or tasks. Then the params files from both tests were compared. No differences between these files means that STAT was able to read a previous test's parameters and exactly recreate that test.

5.4. K-task Params

Several modifications were made to the k-task simulation to allow for the upgrades that STAT provides. This required changes to the portion of the simulation responsible for reading in parameters. For example, the previous version of the k-task simulation accepted a single “Demand Profile” argument that described what kind of behavior eack task would have. With the new version of the simulation, each task is independent. This means that the argument must now accept a list of length k describing the behavior of each task. Many other parameters were added or modified to support individual task specification.

To test the new input functionality is very simple; The k-task simulation will produce a params file containing all of the parameters it was able to read into memory before execution. If any parameters are different between the provided params file and the final params file, then there is an error in the pameter reading functionality. This test was performed several times with varying types of demand specifications. While error checking was added to the simulation, and was tested through other means, it is expected that the input to the k-task simulation is validated by the user or STAT.

6. Conclusions and Future Work

6.1. Technological Considerations

One of the most impactful technologies utilized in the development of STAT was the PAGE tool. This tool saved a lot of time in developing an interface that was appealing. Even though Python contains libraries for creating user interfaces, a from-scratch user interface would have taken considerably more time to develop and likely would not have been as presentable. The PAGE tool allowed for several changes to STAT's screens throughout development without much effort.

Another tool that was very useful was the PLY library. This library allowed the reuse of the grammar that was developed for the k-task simulation without much modification. The PLY library allows for the creation of very powerful parsers that were more than enough to satsfy this project's needs. This library also facilitated the creation of parse trees in STAT which can be used to preview demand before starting the simulation.

6.2. Software Engineering Considerations

The Incremental Prototyping model was the best choice for this project because it allowed the development team to fully design the software before beginning the project. Since there was little change in the requirements, Incremental Prototyping worked well in helping the developers understand all requirements up front. One difficulty that was faced during development was the integration phase of development. This software model allows for modules of the project to be developed separately and integrated later. Some of the connections between modules were not fully understood during the analysis phase. More care should have been taken in the analysis phase of each increment to ensure that the integration phase would be successful. This lead to more time being needed at the end of the project to ensure that all modules worked together.

6.3. Future Work

STAT has been developed with the hope that it will be used by users of k-task for many years to come. In order to keep STAT relevant and useful, much thought was put into the added functionalities that could be implemented later. As part of the implementation of STAT's parser, a parse tree was added that allows STAT to model provided custom demand functions. In the future, STAT can be upgraded to include a demand preview page where the swarm tester can see all of the tasks' demands in graph form before running the simulation.

Another future addition to STAT is a functionality to develop threshold values. These values tell each agent what the minimum current demand for a task must be before the agent can consider acting on that task. Developing a set of threshold values for each agent is important to discovering if a generalized swarm can be described. A machine learning tool has already been developed in Python which could easily be integrated into STAT to allow for development of these threshold sets.

7. References

[1] Anja Weidenmüller. The control of nest climate in bumblebee (Bombus terrestris) colonies: interindividual variability and self reinforcement in fanning response. *Behavioral Ecology*, 15(1):120–128, 01 2004. ISSN 1045-2249. doi: 10.1093/beheco/arg101. URL `https://doi.org/10.1093/beheco/arg101`.

[2] Sebastian Vehlken and Valentine A. Pakis. *Zootechnologies*, pages 297–316. Amsterdam University Press, 2019. URL `http://www.jstor.org/stable/j.ctvswx8f2.10`.

[3] The European Commision. Swarm-bots, 2014. URL `https://www.swarm-bots.org/index.php@main=1.html`.

[4] Reussner, Goedicke, Hasselbring, Volgel-Heuser, and Martin, editors. *Managed Software Evolution*. Springer International Publishing, 2019.

[5] Incremental model of software development life cycle, Sep 2021. URL `https://newline.tech/incremental-model-of-software-development-life-cycle/`.

[6] Jin. Software development framework — incremental model, Jan 2023. URL `https://medium.com/geekculture/software-development-framework-incremental-model-640719a6dbc`.

[7] Roger Pressman. *Software engineering*. Mcgraw-Hill, 2014.

[8] Waterfall methodology: A complete guide, Mar 2022. URL `https://business.adobe.com/blog/basics/waterfall`.

[9] Don Rozenberg. Page - python automatic gui generator - version 7.6. URL `https://page.sourceforge.net/`.

[10] David Beasly. Ply (python lex-yacc). URL `https://www.dabeaz.com/ply/`.

[11] Software Teting Help. What is boundary value analysis and equivalence partitioning?, 2023. URL `https://www.softwaretestinghelp.com/what-is-boundary-value-analysis-and-equivalence-partitioning/`.

8. Appendices

Input Name:	Class	Min-	Min	Min+	Mid	Max-	Max	Max+
Population	$\{\leq 0\}$	-11	-10	-9	-5	-1	0	1
	$\{> 0\}$	0	1	2	50	99	100	101
Thresh Init	$\{< 0\}$	-11	-10	-9	-5	-2	-0.1	0
	$\{0.0 - 1.0\}$	-0.1	0	0.1	0.5	0.9	1	1.1
	$\{> 1.0$ and $< 2.0\}$	1	1.1	1.2	1.5	1.8	1.9	2
	$\{2\}$	NA	NA	NA	NA	1.9	2	2.1
	$\{> 2.0$ and $< 3.0\}$	2	2.1	2.2	2.5	2.8	2.9	3
	$\{3\}$	NA	NA	NA	NA	2.9	3	3.1
	$\{> 3.0$ and $< 4.0\}$	3	3.1	3.2	3.5	3.8	3.9	4
	$\{4\}$	NA	NA	NA	NA	3.9	4	4.1
	$\{> 4.0$ and $< 5.0\}$	4	4.1	4.2	4.5	4.8	4.9	5
	$\{5\}$	NA	NA	NA	NA	4.9	5	5.1
	$\{> 5.0$ and $< 6.0\}$	5	5.1	5.2	5.5	5.8	5.9	6
	$\{6\}$	NA	NA	NA	NA	5.9	6	6.1
	$\{> 6.0$ and $< 10.0\}$	6	6.1	6.2	8	9.8	9.9	10
	$\{10\}$	NA	NA	NA	NA	9.9	10	10.1
	$\{> 10\}$	10	10.1	10.2	15	18	19	20
Prob Check	$\{< 0.0\}$	-11	-10	-9	-5	-2	-1	0
	$\{0.0 - 1.0\}$	-0.1	0	0.1	0.5	0.9	1	1.1
	$\{> 1.0\}$	1	1.1	1.2	5	9	10	11
Response Prob	$\{< 0.0\}$	-11	-10	-9	-5	-2	-1	0
	$\{0.0 - 1.0\}$	-0.1	0	0.1	0.5	0.9	1	1.1
	$\{> 1.0$ and $< 2.0\}$	1	1.1	1.2	1.5	1.8	1.9	2
	$\{2.0\}$	NA	NA	NA	NA	1.9	2	2.1

	$\{> 2.0$ and $< 3.0\}$	2	2.1	2.2	2.5	2.8	2.9	3
	$\{3.0\}$	NA	NA	NA	NA	2.9	3	3.1
	$\{> 3.0\}$	3	3.1	3.2	5	9	10	11
Spontaneous Response	$\{< 0.0\}$	-11	-10	-9	-5	-2	-1	0
	$\{0.0 - 1.0\}$	-0.1	0	0.1	0.5	0.9	1	1.1
	$\{> 1.0$ and $< 2.0\}$	1	1.1	1.2	1.5	1.8	1.9	2
	$\{2.0\}$	NA	NA	NA	NA	1.9	2	2.1
	$\{> 2.0$ and $< 3.0\}$	2	2.1	2.2	2.5	2.8	2.9	3
	$\{3.0\}$	NA	NA	NA	NA	2.9	3	3.1
	$\{> 3.0\}$	3	3.1	3.2	5	9	10	11
Aging Min	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11
Aging Max	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11
Aging Up	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11
Aging Down	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11
Thresh Increase	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11
Thresh Decrease	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11

Hetero Range Max	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11
Hetero Range Min	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11
Hetero Radius Max	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11
Hetero Radius Min	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11
RP Gaussian MU	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11
RP Gaussian STD	$\{< 0\}$	-11	-10	-9	-5	-0.2	-0.1	0
	$\{\geq 0\}$	-0.1	0	0.1	5	9	10	11
Kill Number	$\{< 0\}$	-11	-10	-9	-5	-2	-1	0
	$\{\geq 0\}$	-1	0	1	5	9	10	11
First Extinction	$\{< 0\}$	-11	-10	-9	-5	-2	-1	0
	$\{\geq 0\}$	-1	0	1	5	9	10	11
Extinction Period	$\{< 0\}$	-11	-10	-9	-5	-2	-1	0
	$\{\geq 0\}$	-1	0	1	5	9	10	11

Table 3. Swam Specification Tests