CHAPTER I

INTRODUCTION

1.1 The GUI Testing Problem

Because of the ubiquity of Graphical User Interface, GUI, software and the percent of the overall code for many applications that the GUI involves, testing of GUIs is an important part of the testing for most applications. Yet the complexity involved in testing all possible combinations of keystrokes, dialog activations, drop down menu selections, and so on, quickly becomes large. While a traditional approach to verifying the soundness of an application's functioning is to track the percentage of underlying code that is exercised by a suite of tests, this approach is not adequate for ensuring that a GUI will perform as intended. Memon (2001b) points out:

… the space of possible interactions with a GUI is enormous, in that each sequence of GUI commands can result in a different state and a GUI command may need to be evaluated in all of these states… For conventional software, coverage is measured using the amount and type of underlying code exercised. These measures do not work well for GUI testing, because what matters is not only how much of the code is tested, but how many different possible states of the software each piece of code is tested.

1 GUI: Graphical User Interface. For a list of this and all of the acronyms used in this paper turn to the Appendix.
Tracking each resulting state from each sequence is clearly a non-trivial task. In fact further research by Memon, Pollack, and Soffa (2001a) suggests that in a relatively simple application such as WordPad, if one takes into consideration the possible sequence of events between operators of up to only length 3, some 21,000 such event-sequences can be defined. Further, taking into account sequences of up to length 5 leads to nearly 5 million possible event-sequences and by length 6 there are over 84.5 million possible sequences. As the length of an event chain is allowed to increase, the space to be tested quickly becomes unwieldy – too expensive to consider a total coverage solution. Consequently, once the coverage area is extended to beyond a sequence of a few steps into an application, algorithms other than those based on total coverage are needed for efficient GUI testing.

1.2 Goals and Methods

The goal of this research is to define and to demonstrate the effectiveness of a new search algorithm for detecting errors in the event space of the GUI. In this study paths through the event space of the GUI are abstracted as trees with the root of the tree representing a command object on the opening window of the application. Branching from the root represents the different choices available once the root has been activated. Nodes at the end of branches represent the command objects that are now available given the activation of the root object. Each new command object has branches proceeding out of it with new nodes following, and so on. The hypothesis of this research is that the new algorithm,
MSF (Mixed Species Flock), will find errors in these trees equal to or greater than those found by an exhaustive search of the same space. The effectiveness of MSF will be shown by means of a series of side by side tests running in one case MSF and in the other a level-order search of the same event space.

The errors being sought are of four types: uninitialized states, unintended states, hung states, and crash states. Uninitialized states are those which are not ready to be used. For example, if a printer has not been properly set up to work with a text application, an error will occur when attempting to print a document. If the error is caught a warning message is generated prompting the user to take some action to correct the situation. Though it is possible for a hung state or crash state to result from the action of attempting to print the document, for the purposes of keeping this category discrete from others states an uninitialized state is defined as one which produces a warning message to the user requiring further user configuration of the application. The second type, an unintended state, is one which is obviously not intended to occur. Three classes of unintended states are defined: (1) a command object is activated and the resulting action is opposite or different from that indicated (e.g. a cut operation pastes instead of cuts); (2) a command object is activated and the activation causes the button to be corrupted (e.g. a text box which is intended to be a label is written over when activating the edit box so that it no longer serves as a label); and (3), a command object is activated but nothing happens (e.g. a “Zoom” button is clicked but it does not result in font size of highlighted text being increased). A hung state is that of the application under test stuck in a deadlock
condition. An intervention of the user is required to generate an interrupt of the state. Finally, the crash state is one that causes the application to close precipitously, often with the loss of data and records of the state of the application. In addition, crash reports are sometimes also generated which give details about the part of the stack, and the modules, that were affected as the error occurred.

The products of this research are an automated GUI Mapper, a MSF search engine, a GUI Search Iterator, records of the side by side runs of the two error detection algorithms, analysis of those search results, and the conclusions based on the records and their analysis.

The GUI Mapper is an automated configurable engine which creates an abstract model of each frame (window) that appears by way of exercising the AUT, Application Under Test. For each window of the application it encounters, the engine identifies and then stores the identifying properties of the command objects found there, e.g. menu items, combo boxes, buttons, check boxes, list views, and so on. The GUI Mapper then creates a unique key for the window based on the summed availability attribute for all of these objects. This key is later used as input to the Coverage Calculator as it determines how many potential event sequences have been identified and how many have been traversed.

The MSF search engine takes as input the composition of the mixed-species flock, i.e. how many of each type of species. MSF then performs a search for errors based on the combined efforts of each individual agent and its
characteristic search pattern(s). At the completion of each step the MSF calls the
GUI Mapper to examine the objects and properties of all items found there and to
determine if this is a new frame. If it is a new frame the mapper stores this
gathered information and adds the frame and its key to the repository of
discovered frames. The products of the search include: (1) keyed records for
each agent step showing the number and state of objects on the frame it is
starting from before a step is taken and the number and state of objects on the
frame it ends on after a step has been take; (2) screen-shots of error messages
that are keyed to the discovering agent; and, as noted above, (3) an expanded
list of frames and the possible steps to be taken in that frame\(^2\).

The GUI Search Iterator, GSI, takes as input all frames and objects
discovered from a previous n-step bounded search and then generate a template
of the steps needed to progress through that newly discovered space. The GSI
then proceeds to execute that iteration and calls the GUI Mapper after each step
to determine if a new frame has been encountered and if this is the case it then
records all available objects for the possible next searches.

The Coverage Calculator takes as input the records from the GUI Mapping
enabled MSF, or level order searches generated by the GSI, and gives as output:
(1) the number of activation events possible in the GUI space explored; (2) the

\(^2\) Since the steps that precede the activation of a window can determine which objects are
available to be manipulated on that window (such as copying highlighted text to the
clipboard now enables the paste option), a given window may have the same title and
same number of objects on it as the first time it was encountered but those objects may
now be in a different state. For this reason, for the purposes of this research and
evaluating how many available objects are covered in a given frame, a frame with the
same title and number of objects as another but with a different set of objects enabled is
considered to be a new, i.e. different, frame.
number of events actually exercised; and (3), the number of errors found in that space. The size of the event space tested is defined as the sum of all possible steps (as determined by all available command objects reachable in just one step after a previous step has been executed) for a given limit of $n$ steps. For example, for a limit of 1 the number of possible steps is the sum of all active command objects on the opening window of the application. For a limit of 2 the number of steps possible is the sum of all steps of level 1 plus, for each object activated at level 1, the sum of all objects that are reachable in one additional step. An example of a calculation of level 2 for a single command object is that of activating the File-&gt;Open menu option on the initial presenting window of a text editor, a level 1 step, then enumerating all active elements of the combo boxes, list views, buttons, etc. present on the “Open” dialog window which appears.

The side by side testing that is performed is the heart of this research. Multiple tests results will be presented and analyzed given searches of the same application spaces by both the MSF and the GSI search algorithms. The comparisons are based on the ratio of errors detected per steps taken by each search method. At level one the test will compare the algorithms side by side. The Level Order Search, LOS, will be completed first with the MSF then permitted the same number of steps as the LOS required. For level two and above the total number of steps will be calculated by the GSI then each algorithm allowed the same size subset of total steps. In this way the efficiency of each algorithm can be evaluated. Further exploratory tests will compare LOS searches
of a lower depth, e.g. 2, compared to a Multi-Species Flock search of longer sequences. Given a large number of possible configurations of the MSF – as with differing mixes of the types of agents chosen and the numbers of each, a limited number of flock compositions estimated to be typical of the algorithm will be explored.

1.3 Summary of Chapters

This introductory chapter presents in brief the GUI search problem and outlines a new approach for exploring that space for errors. Chapter II, which follows, examines the current research regarding efforts to effectively test the GUI space. The research will be summarized and its findings used to further refine the definition of the search problem as well as to bring into sharper focus the limits of research to date and where further exploration is needed. Chapter III presents the Mixed-Species Flock search algorithm detailing the functioning of mixed-species flocks in nature and the implications for a search algorithm for exploring a large GUI space modeled as a forest of trees. Chapter IV presents in greater detail the tools and methods actually used to implement the GUI Mapper, MSF and LOS search algorithms, and Coverage Calculator. Additionally, Chapter IV summarizes the initial findings and the difficulties encountered in actually attempting to implement the algorithms. The solutions and workarounds discovered that enabled working implementations are detailed at the end of Chapter IV. Chapter V presents the results reached after the issues detailed in
Chapter IV were overcome. Finally, Chapter VI lists the findings and achievements of this research.
CHAPTER 2

CURRENT RESEARCH IN GUI TESTING

2.1 A Typology of Current Research

Given the vastness of the space to be covered, the difficulty in covering all of the event-sequence space, and, at the same time, the importance of the GUI testing endeavor, numerous approaches to the GUI testing task have emerged. These approaches can be classified in terms of two continua: (1) selective coverage through total coverage; and (2), minimal information gathering through comprehensive information gathering. As is likely expected there are pros and cons for both ends of each continuum. The following sections will discuss these continua and list advantages and disadvantages for each.

2.1.1 The Coverage Continuum

An example in regards to the low coverage end of the coverage continuum is that of probably the most popular testing approach, i.e. to insure a core subset of functionality by testing typical user sequences. In terms of a text editor a typical sequence might be opening a document, typing a paragraph, and then saving. For an audio player a typical sequence might be to download a music file then play it. Tests in this scenario cover basic functionality.
this continuum would be a comprehensive test endeavor - to activate every possible sequence of command objects – buttons, menu items, check boxes, and so on, up to a given number of sequences, e.g. 3, to ensure no errors are incurred, or if they do occur that they are handled gracefully.

Part of the case for strategic coverage, the approach of using a relatively low number of test cases to exercise typical use models for the application, is that of validating the core functionality of an application when that application is utilized as expected. However, this approach may miss many errors as the tests designed may not be what a user unfamiliar with the application GUI, a novice, may actually do. Further an approach that targets frequent uses of the application may still miss a significant number of less frequently used operations (usually discovered by key customers of course).

The total coverage side of the coverage continuum has clear advantages in light of the drawbacks just mentioned of the strategic approach. Total coverage insures that novice attempts at using the software are much more likely to be covered by testing as well as the less frequently used features of an application and even unusual pairings of command object activations. Yet seeking total coverage quickly approaches a limit, an issue which will be explored in depth at the end of this chapter. In short, however, the overhead costs in terms of time and storage capacity begin to mushroom as coverage is increased from two steps deep to three. Difficulties emerge in many areas as coverage expands – namely involving large increases in the time, effort, and expense of resources (in terms of both computer and human resources) involved in designing then
executing tests, logging and storing validating information obtained from them, and finally in analyzing the vast number of records resulting in order to determine which records likely indicate errors. The time taken to execute the full test suite taken together with the computing resources needed to keep the found data causes those implementing this approach to settle for only a few steps into an event-sequence (for all possible event-sequences) instead of covering longer sequences.

2.1.2 The Tracking Continuum

As for the information gathering continuum an example of low information gathering is that of streaming thousands of valid events to the system event queue of an application in order to trigger possible error events (Forrester and Miller 2000). Not surprisingly this approach is able to cause most applications to hang – but the difficulty encountered is that of trying to determine which subset of the thousands of sequences of events sent caused the problem.

In contrast to approaches with a low level of retrievable information, other approaches seek to keep detailed logs and records of each step and its results. This saved data provides a way to later pin-point which sequence caused the error to manifest – a great aid in beginning to debug an application. Yet, as with the total coverage pole of the coverage continuum, the more detailed the records of tests run the larger the amount of resources in terms of time, computing memory, and analysis needed to make sense of the data.
2.2 Problem Restated

Given both sets of polarities the following quadrants are suggested:

```
<table>
<thead>
<tr>
<th>II</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Info</td>
<td>High Info</td>
</tr>
<tr>
<td>Low Cov</td>
<td>High Cov</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Info</td>
<td>Low Info</td>
</tr>
<tr>
<td>Low Cov</td>
<td>High Cov</td>
</tr>
</tbody>
</table>
```

In the following section we will examine significant testing efforts in each of these areas and highlight the pros and cons of the given approaches. Following these cases these continua and the paradoxes they imply will be further examined with the goal of stating with finer granularity the boundaries of the testing problem and the regions remaining for further exploration.

Ironically it appears that both high information without high coverage and high coverage without high information are relatively easy to achieve. If, on the other hand, the effort is made for both high coverage and high information to make sense of that coverage, limits are quickly reached in terms of the total
length of the execution time needed to run all the tests required and the storage space needed to keep the records generated by those tests.

### 2.3 Related Work

An interesting and relatively easy to implement approach to testing software is that of J. E. Forrester and B. P. Miller. Their approach was to not care about tracking which event caused a crash but rather to stream thousands of valid keyboard and mouse events to the system event queue for an application and hundreds of thousands of random Win32 messages to the thread message queue of the AUT. Forrester and Miller found that with system event loading they could cause about 20% of applications tested to fail (crash or hang) and with message loading about 95% (Forrester and Miller 2000).

While this streaming testing likely provides useful information about the robustness of handling of messages from queues, no effort is made to identify which sequence of messages causes an application to hang or crash. Though hundreds of thousands of messages are sent, perhaps representing some quite long event-sequence lengths, there really is no way to tell how the application interprets these messages. This approach is clearly high coverage but gives little information. In terms of addressing errors discovered, little can be done given this approach.

#### 2.3.1 Strategic Testing

As stated earlier, a common approach to GUI testing is to develop test cases around tasks commonly performed on the application. These efforts often
utilize record and playback tools such as Rational Robot, the Pounder, and others which provide a way to perform a common task, record it, and then have it available for playback as part of a test suite. By first recording a variety of common tasks and putting in validation points the tester can create a set of tests to exercise core functionality of an application.

Though these tests will provide some coverage, they have some built-in limitations. These may or may not be part of a comprehensive plan to cover a large set of core functionality; this issue will be determined by the test designer. Further, playback is sometimes affected by small changes in the next version of a GUI. Unless tests are dynamically recreated with each build, a significant amount of time will likely be given to test maintenance. Also, as tests are likely being developed by persons familiar with the GUI, they are unlikely to cover interactions more typical of less experienced or even novice users.

A number of tools are available for unit testing of applications – e.g JUnit and Pounder and HTTP Torture Machines. However, these tools tend to either require manual creation of tests related to the overall architecture of the application (JUnit) or, if not requiring manual creation, to limit testing to certain common requirements such as link verification or load testing. Though this simplifies testing it does not fully exercising the structure of the GUI involved (Xie and Memon 2005).

A variation of strategic testing of an application by the familiar is that of Kasik and George (Kasik and George 1996). These researchers developed an approach to mimic novice testing based on defined user tasks and the use of
genetic algorithms to deviate from the defined path by a step or two or many on the way toward a typical goal. They defined alleles in genes to be able to be interpreted as GUI events—such as entering text into a text field, activating a button, checking a check box, etc. Next they set a reward system for novice-like behavior—rewarding genes that triggered events on the same window. The assumption here was that a novice was likely to experiment with a window or dialog to understand its use.

Two basic approaches were used by Kasik and George: (1) staying with only genetic generated paths; and (2), beginning with an expert defined script then allowing a controlled use of the switching over to the genetic approach with rewards for moving back to the expected windows of the expert script. In this way deviation from the expert was allowed with progress toward the expert task completion rewarded. The first method proved unsuccessful. “Getting a script to accomplish anything meaningful was unlikely” (Kasik and George 1996, 250). The second approach, however, proved more fruitful with novice scripts progressing further and moving into areas the team judged to shed new light on what a typical first user might do. Nevertheless at the end of the study the authors themselves noted that, “Measures must be developed to determine when enough novice testing has occurred” (Kasik and George 1996, 250).

A final refinement of strategic testing, similar to that of the genetic approach above, which takes familiar tasks and fully exercises variations on them, is that of White and Almezen (White and Almezen 2000). Their approach is to model common tasks in the form of finite state machines. First is the
elaboration of as many paths as possible in these common transition sequences and then to refine this rather large set of models set down to a more manageable one. This is done using two rules: (1) abstracting strongly connected components; and (2) merging states that have structural symmetry (White et al 2000, 111). From this point two sets of tests are created, one that exercises that transitions just defined and a second more elaborate set which seeks to explore alternate pathways to the final goal state. While this approach refines down to a core a reduced set of tests that should test typical use models and was demonstrated to find a significant number of errors in applications it was used to test, its manual analysis and refinement stage is quite time intensive; it is not clear how the success of these tests compare to other approaches.

Taken as a whole, the above strategic approaches to testing provide more useful information than the random testing mentioned above, though the amount of overall coverage of the event-sequence space remains unknown. These testing approaches tend to be higher on information than the random approach and low to moderate in regards to coverage – placing them in or near the second quadrant of our typology: low coverage and moderate to high information. Overall they also present the drawback of needing manual input for the design of models used or the creation of the test cases involved.

2.3.2 Comprehensive Testing

How does a test manager or developer know when a sufficient amount of testing has been done to thoroughly test an application? While there has been general acknowledgement that covering the event space of a GUI is an
enormous task, Memon, Pollack, and Sopha (Memon, Pollack, and Soffa 2001a) in a seminal paper, sought to define and explore the test space of a typical text editor, WordPad. They created their own version implementation of WordPad to allow easy analysis of the underlying code. Then in addition to talking about the underlying code exercised by a set of GUI tests they added three other categories: events coverage, event-interaction coverage, length-n event-sequence coverage.

Event coverage is the testing of each event in a component, each button, menu item, etc available in a window. Event-interaction converge relates to testing all possible pairs of events in a component such that for example (Memon, Pollack, and Soffa 2001a, 260) “after an event e has been performed, all events that can interact with e should be executed at least once; length-n coverage takes into account the state of the application caused by previous events (the context) before a new event is executed. Since any number of events may precede an event, an event may be executed within an infinite number of contexts, it makes sense to set a definition length-n to designate that context. Their definition is as follows (Memon, Pollack, and Soffa 2001a, 260): “A set P of event-sequences satisfies the length-n event-sequence coverage criterion if and only if P contains all event-sequences of length equal to n. “

Given these definitions and a manual representation of the GUI into event sequence pairings, the number of possible event-sequence interactions were now calculated for various levels of depth (up to length 6). The research clearly demonstrated that for a relatively uncomplicated application such as WordPad
literally millions of event sequences are included in total coverage at 5 steps, 85 million at 6.

Once the scope of these possible sequences were established further testing was performed which contrasted the number of underlying code lines exercised with the coverage of various length-n event-sequence limits. An intriguing result was noted. For events sequences of length 1 almost 92% of the underlying code of the application were exercised. By the time length 3 is reached only 5% additional underlying code has been touched. A conclusion at this point is that “high statement coverage of the underlying coded may be obtained by executing short event sequences” (Memon, Pollack, and Soffa 2001a, 264).

Interestingly, the finding about code coverage is emphasized in a manner somewhat at odds with the premise of the research with which the investigators began. Namely, that the measure of test code coverage is fundamentally inadequate in terms of insuring event-sequence coverage.

To be fair, the rest of the article describes evaluating coverage results for a suite of 72 typical tasks each representing a unique function of WordPad. Taking into consideration only those events possible on the Main window, the test suite was shown to cover 88 % of length 1 sequences, about 41 % for length 2, 11% for 3, .4% for 4 and practically 0% by 5 (Memon, Pollack, and Soffa 2001a, 265). This finding puts into perspective the earlier discussion of strategic test suites. In terms of covering possible sequences it appears that common task path testing may only be adequate for level 1 or level 2 length-n coverage. While
a couple of steps may in fact exercise a high percent of the underlying code (all of length 1 exercised 92%), the miniscule % coverage for all event-sequences of levels 5,6, and above raises large questions about whether strategic testing is adequate in itself. A related question emerging from this finding is that if it is a given that a large number of tests are required for adequate GUI testing beyond a few steps into the event sequences of the GUI, what is to be done? Are there other tools and criteria for an efficient testing of this space given that global coverage may be impossible? This testing sets a high mark for subsequent investigations either to strive for higher and higher coverage and to find innovative ways to handle the expense this involves or to discover other criteria that can lead to a viable reduced set of tests which demonstrate the ability to detect a high percentage of defects residing in longer length event sequence space. In terms of the typology mentioned earlier this work clearly aims for the first quadrant – high information and high coverage.

Given the exploration of the current research so far it can be seen that likely the most useful set of test cases are those which provide both high coverage and high information. However the development and maintenance of the necessarily vast number of test cases, the computing power and memory required to validate each execution of a test, and the ability to determine at what point enough testing has been performed are limiting factors. In the remainder of this chapter recent advancements on the path toward achieving efficient comprehensive testing are explored along with also exploring the current limits of such testing.
2.4 At the Limits of Comprehensive GUI Testing: Steps Being Taken and Areas to Be Explored

Since 2001 a number of significant additions have been made to the body of knowledge that Memon et al presented regarding comprehensive coverage. These refinements are in two areas particularly: (1) automation of some of the human resource intensive aspects of performing comprehensive tests; and (2), exploration of what test oracles are most cost effective, i.e. how to determine when enough testing has been performed in terms of coverage given cost constraints.

2.4.1 Automation of Related Comprehensive Testing Tasks

2.4.1.1 PATHS

Already in 2001, Memon et al were using PATHS (Memon, Pollack, and Soffa 2001b), Planning Assisted Tester for graphical user interface Systems. This AI Planning type tool was used to automate test suite creation given an operators list which includes preconditions and effects for each command object. Once these initial artifacts were created the planner (PATHS) is able to automatically create thousands of tests for the now defined GUI space. However, the amount of manual work required is extensive - involving the definition of an options file of the pre and post state for any event. Additionally, the beginning and goal state for events leading to a desired end state, i.e. the completion of a task, need also to be provided to the planner. Further, in terms of the ability to provide comprehensive coverage, though the test is capable of
creating thousands of tests, only a reduced number of these can be executed in a reasonable amount of time. For example the length of time needed to execute the tests required to cover all event-sequence steps to level 3 in WordPad was 30 hours (Memon, Pollack, and Soffa 2001a).

2.4.1.2 GUI RIPPER

Ishan Banerjee, Adithya Nagarajan and Memon (2003) have developed the GUI RIPPER which has helped to eliminate much of the manual work needed by the PATHS planner. This tool dynamically creates the operators file needed by the planner by grabbing all readable objects from the GUI, saving their properties, and by means of analyzing these and other properties of the GUI is able to establish Event-Flow-Graphs (EFGs) for the application under test. The operators and EFGs are then input to the planner style test generator which can produces thousands of valid tests for the application. Obviously, this tool can save the test suite developers many hours of time formerly dedicated to the creation of the operators file. However, in a related work (Xie 2006) it is noted that the GUI RIPPER itself may make errors, and thus its findings may need to be manually reviewed. Further the tool's ability to see custom defined control objects can be quite limited and whole classes of objects may be unreadable. A certain amount of manual work may still be required to enable AI Planning tools such as PATHS to automatically generate tests. (Xie 2006; Memon and Xie 2005, 890)
2.4.1.3 GUI Test Case Repairer

A known problem encountered by attempting to use automated test suites for validating GUIs is that of the fluid nature of the GUI particularly at early stages of development. The number and order of objects may change as well as their outcome states when activated. Underlying code is also changing. So tests that succeeded before no longer pass the previously designed automated test - not because these operators are now malfunctioning but because their design, and to a certain extent their implementation, has changed.

A GUI Test Case Repairer has been implemented by Soffa and Memon (2003). This tool gleans changes of the GUI structure, control-flow, and call graphs by examining the operating GUI. It compares the discovered structure to the previously represented structure and makes decisions about whether to drop or repair test cases based on these representations. Additionally sometimes new test cases are generated when there is more than one way to repair the original case.

2.4.2 Exploration of Efficient Oracles and Their Limits

One of the cost factors of comprehensive testing is that of how much information it is necessary to gather after a test execution to determine if an error, a result which is not intended, has occurred or not. Is it enough to examine only the object just activated or is it necessary to search the entire application for all the states of all the objects present? Memon and Xie tested 5 levels of information gathering and comparison for which they examined: (1) the effected widget only, (2) all the widgets on the active window, (3) all the widgets for all
visible windows, (4) all the widgets of all windows after each step, and (5), all the
widgets for all windows but only after the last step (Memon and Xie 2005). Interes-
tingly they found that many instances of method 1 were less effective than
fewer instances of higher information gathering methods. Methods 3-5 were
reported as particularly effective.

In particular, 200 errors were seeded, i.e. injected, into known good
programs, in this case a WordPad replica. One error was injected per one
instance of the replica to insure there were not interactions between one injected
error and another. To cover the event-sequence space of lengths 1, 2, and 3
(112, 1253, and 16012 respectively) a test suite of 5,000 test cases was
generated. These mapped 112 for the 112, 1253 for the 1253, and 3880 for the
16012 just mentioned (Memon and Xie 2005,891). When the test suites were run
about 83% of the injected errors were found (Memon and Xie 2005, 892-894).
An earlier study also using seeded errors had reported nearly 100% detection
success when running 600 test cases for 100 versions of an application each
with one error injected (Memon, Banerjee, and Nagarajan 2003).

While upon first examination it may appear that tests covering all
sequences up to only a couple of event-sequences may be adequate in detecting
possible errors, several cautions need to be raised. First, a more recent study
revealed that one of the criteria for seeding is to seed a fault “in code that is
covered by an adequate number of test cases, e.g., they may be seeded in code
that is executed by more than 20% and less than 80% of the test cases”.
(Memon and Xie 2005, 891) This method puts a different perspective on 5000
test cases finding over 150 errors, given that the errors were entered only if at least 20% of those 5,000 cases covered them. So it was known before beginning that 1000 to 4000 test cases should quite likely discover these errors. Though it is not certain, it may be possible that the earlier study (Memon, Banerjee, and Nagarajan 2003) also followed this method of seeding errors.

A second caution is that as more and more test cases are used to provide coverage of event-sequence lengths above 2, a time limit is approached. Note from examining the number of tests used to cover the over 16,000 possible event-sequences for level 3 was only about 4,000, whereas for the previous levels the ratio of possible event-sequence to test case was 1 to 1. The authors stated this reduction was because “Not all length 3 test cases were generated for some applications since DART (test execution environment engine) would not be able to run them on one machine in one night” (Memon and Xie 2005, 891). Even with this reduced set, the time required per test case to execute, gather oracle information, and then analyze that information gathered varies from between 30 sec to about 80 sec per case. This means a suite of 5,000 test cases could take about 14 hours to run. Given similar time requirements per case the time required to cover a similar application for up to an event-sequence of 4 would be in the time frame of about one month.

A third caution is that of space requirements for storing all the information needed per test oracle. In the 2003 research just mentioned, the space required to store information needed to validate tests for a test suite of 600 tests was between 100 to 225 MB for the most effective test oracles (Memon, Banerjee,
and Nagarajan 2003, 9). Again, simple extrapolation shows space needs growing large as more test are added for more complete coverage at higher sequence-n levels. Recall from the above paragraph the estimate of 16,000 tests to cover level 3. This could represent from between 2.5 to 6 GB of space to store data. By level 5 this means approaching 2 terabytes for the most extensive coverage.

2.5 Summary of Advancements and the Current State of Limits to GUI Testing

In the past five years significant gains have been demonstrated in the domain of GUI Testing. Specifically, it is now possible to create a model of the GUI automatically by dynamically gleaning the properties of the widgets and their relationships to each other using a tool like the GUI RIPPER. This advancement means thousands of test cases can be generated by planners such as PATHS with little human time required. Further test cases generated for GUI coverage known to be relatively easily broken can, in many cases, be automatically repaired. It has also been shown that, given a few caveats for tests not being able to be run in an automated way (because of custom controls not readable to the GUI RIPPER), complete coverage of all event sequence tests to levels 1 and 2 can be run in a couple of hours. Further, because of the ability to create the test cases dynamically, new test suites can be created and run each evening for effective smoke testing of rapidly evolving code during periods of rapid product development – i.e. most of the time (Memon and Xie 2005).

At the same time, limits to effective coverage of event-sequences above 2 in length have also been identified. Both in terms of the time and the memory
space involved, completely testing a GUI above an event-sequence level of 2 is now seen as impractical (Xie 2005, 475) As a result, the most recent studies are along the lines of running different subsets of level 3 sequences, or of further refining what are considered to be the significant subsets of tests to be run.

2.6 The Case for Explorations into Coverage Space of Event Sequence Level 3 and Above

While a shift away from comprehensive GUI testing for event-sequences greater than 2 appears to be occurring, the importance of continued efforts to cover this area remains important. The effort to further refine subsets of level three tests to run is evidence of their importance. Further, testing to determine effective oracles demonstrates a marked increase in error detecting ability for event-sequences over 2 in length with steady increases for progressively longer sequences. Additionally the recent research indicating that most errors can be detected in three steps may be flawed given the error seeding methodology used. Finally, the applications mentioned in testing so far by the researchers including Memon and those following after have largely been focused on in-house software which is said to be similar in complexity to the WordPad application. As testing seeks to include more complex application such as for example Microsoft Word, it is likely that the possibility of encountering errors along longer path sequences will increase. For example, consider the myriad interaction sequences possible in Word when making use of complex semi-autonomous features with many configuration settings such as tables, pictures, drawings, databases, and visual basic macros.
Given the importance of testing for errors on GUIs beyond the limit of 2 event sequences and the limits encountered both in terms of time and storage for moving into this testing area, this paper proposes a new method for exploring this space. The following chapter describes in some detail a search algorithm coming out of the field of multi-agent search and informed by observations and research in field ornithology. It describes how this algorithm can be implemented into code and how the agents cooperate together to explore a large space.
CHAPTER 3

The MIXED-SPECIES FORAGING FLOCK ALGORITHM

The model for creating an algorithm to efficiently search a graphical user interface comes from the field of ornithology. Field observations and research show that mixed-species foraging flocks exhibit efficient search strategies. The first part of this chapter explores the characteristic behaviors of such flocks and the search algorithms suggested. The second part of the chapter describes how the mixed species search algorithms can be implemented into a test framework for testing certain types of GUI applications. The following chapter describes particular technical problems that present themselves during the actual testing of GUIs using this algorithm as well as methods for resolving these obstacles.

3.1 The Search Algorithms

Typically during the fall and winter seasons certain species of birds form mixed-species foraging groups. (In North America these usually have at the core several pairs from the family Paridae, i.e. chickadees or titmice.) Other species join the Paridae such as woodpeckers, creepers, and nuthatches (Sibley 2001; Stokes 1979). Each species has its characteristic foraging strategy and it is clear
to the observers in the field that these combine in complimentary ways to form a successful shared search paradigm (Dolby and Grubb 1998).

3.1.1 Species Specific Search Strategies

The Paridae core pairs have a particularly vigorous search strategy consisting of an intense local search – rapid head movements side to side, a brief pause, and than a flit or jump to the found seed or insect, or, failing a new discovery, flight to a new likely place to search. This pattern consisting of repeated sequences of:

    search -> pause-> flit,   search -> pause-> flit,   ...

is repeated for long periods of time in a local area when food sources are found but following several failed attempts in a local area the birds quickly move several trees off or further away to begin a new search.

The other species which join the Paridae, (creepers, woodpeckers, and nuthatches) show their complementary search patterns. Combinations of species filling out the above categories and found in the Pikes Peak region of Colorado are Brown Creeper, Pygmy Nuthatch, and Downy Woodpecker. The Brown Creeper is often seen as part of the flock and rather than flitting from branch to branch can be observed starting near the base of the trunk of the tree inching up the trunk exploring holes and crevices for insects. Though it can work its way up the tree into branchy areas, the creeper characteristically flies back down at a certain point to inch back upwards again. The Downy Woodpecker also moves up the trunk but can also be found at the tips of branches thus performing a larger search of the particular tree. Like the creeper it searches cracks and
crevices but adds as well drilling into the bark. The Pygmy Nuthatch, in contrast to the creeper and woodpecker, can be observed moving down the tree from higher up a branch to lower down and then often moving down the trunk of the tree.

It is clear that each of these species display a particular search strategy. It is believed each strategy explores a somewhat different part of the tree environment thus decreasing competition between species when foraging together.

### 3.1.2 Group Search Characteristics

The former section serves as a brief introduction to each particular species included in the mixed-species flock. However, certain behaviors of the group as a whole can be noted. The mixed-species flock is observed to more or less stay together. The cohesiveness of the flock is observed to be facilitated both by means of the frequent vocalizations of the chickadees and also the sight observations of the following birds. Further, though the Parids are the leaders and the mixed-species group tends to follow them, individuals may lag behind in a nearby tree after the leading Parid has moved to an adjoining or nearby tree. Thus a particular flock is frequently searching several trees located next to each other at the same. Such a search area can be conceptualized as more or less a sphere which gradually moves through the search space defined by the perimeter of the wooded area and the forest floor to the height of the trees being searched.
Further characteristics of the mixed-species flock include its ability to improve protection and food collection ability for its members. In the wild actual improvements in weight and feather length can be observed for the non-Parid species when they are found in these mixed species foraging groups compared to where the Parids are not present (Dolby and Grubb 1998, 501-509). Additional studies of various species note seemingly large capacities for spatial memory. Chickadees are known to hide seeds for later use and flocks are known to return to areas of known supplies of food.

In summary then, the overall search pattern of a Parid led mixed-species flock is that of a roving search sphere which demonstrates an intensely explorative local search coupled with a low toleration for frustration and a quick movement to a new area upon discovering the immediate area lacks sustenance. At the same time, the search pattern includes the ability to maximize the benefits from previous searches by allowing returns to areas which have previously been discovered to be productive.

### 3.2 The Mixed-Species Flock Search Algorithm

In this section we introduce the algorithms for converting these patterns observed from the flock into implementations of similar patterns for searching the GUI space of one particular type of graphical user interface, that of text and media editors. The goal of the search is that of finding error states either in the GUI or the underlying code.
3.2.1 A Forest of Trees

At the highest level one way to conceptualize the search of the GUI space is to view it as a search of trees - trees formed by the paths taken through the structure. The initial click of a button or selection of a menu item or any other activation of a control structure on the GUI forms the root and all other subsequent moves form a tree. Branches are filled out on further iterations as different choices are made at decision points already having been visited. If the tree root were for example "&Insert" below each menu item following in the drop down menu represents a different branch choice. Examine the following figure showing for example how many branches can be taken from the initial root choice of "&Insert".

Figure 1. From WORD, any menu choice at the top level opens many choices. For example activate "Insert" and note the many additional choices opening up.
Unlike a step-by-step movement in a straight line fashion with one move leading to just one choice for a future move and that choice once taken leading to only one further choice and so on, branching begins right away. It is precisely this feature which suggests a tree as the structure to represent the sequence of moves possible after an initial step. In Figure 1, note that each upper level menu item opens many second level options. As events are not independent but past events effect the options present in the future and the state of the application in the future, it is clear that knowing the previous sequence of events prior to the current state is crucial for determining what went wrong when an error is detected. (The error may have been set in motion several steps earlier). Consider, then, a tree root for each command object occurring on the opening window of an application as a quick way to define a forest of trees for exploration. Though side by side command objects may describe areas of similar functionality, it is also possible for nearby objects to differ in function and area of access in the GUI. Nodes are then added on each root as an agent makes subsequent moves from the root then gathers information about all objects present after that move. These new objects, unless identical to the objects detected on the previous step, comprise the next set of nodes to be added to the tree.
3.2.2 The Species Specific Search Algorithms

3.2.2.1 The Chickadee (Parid) Algorithm

The species specific algorithms are fairly easy to describe with the chickadee pattern perhaps being the most interesting. The characteristic search for the chickadee is that of a vigorous search of local GUI command structures, buttons, etc., all within the same window. The agent then picks one command object, activates it, and thereby most often moves to a new location (window). If no reward (error) is found the agent may return to the original starting point and select a different command object in the original window. This type of search is repeated in the same manner unless too long of a period ensues without finding a reward. As a restless search agent the chickadee will then moves out of the local area – this could be to a different part of the tree or to a nearby tree. For the purposes of this study, the movement once the frustration limit is reached will be to a new window from which to explore.

The outline of the algorithm for the chickadee agent is now described. Let us name the restlessness limit alpha, $\alpha$. In the search of a GUI structure $\alpha$ represents the number of steps taken in the local window before a move to a different area must occur. Similarly, a limit exists for the number of moves allowed for traversing a given tree. Once the chickadee agent reaches this limit the next move will be to a new tree to explore. Let us name the limit for the number of moves within a tree in the case where no reward is found to be tau, $\tau$.

The pseudocode for chickadee (Parid) would then be:
Do while x < total_moves_allowed
Do while y < total_moves_allowed
Search to identify local window control structures
Form lists of toolbar items
menu items
buttons
list view items
and so on.
If preferred type of control structure available
Choose one of preferred type of control structures
Else
Choose any of the above available control objects
Endif
Move to and select the structure
Save record of the step
x = x + 1
y = y + 1
z = z + 1
Record new state (new window or possible error message)
If error message or error state achieved then
Save data on tree nodes traversed to this point
Save data on command object just activated
Save content of error message if any
Reset y to 0
Endif
# Limit Checking
If x = then
Select and Move To New Tree
Reset x to 0
# Prepare to begin a new search in a new tree
Exit Do
Endif
If y = then
Move back to original window
Reset y to 0
# Prepare to begin a new search
EndIf
Loop
If z = total_moves_allowed
Record State of Agent
Exit Do
End if
Loop

3.2.2.2 The Brown Creeper and Downy Woodpecker Algorithms

The creeper and woodpecker algorithms are similar to each other in that each progress step by step through the tree structure taking the next move directly before them. An example would be an agent searching from the main
GUI window and progressively choosing toolbar buttons 2 then 3 then 4, etc. Their search pattern is thus more restricted as it performs a thorough investigation of the area they are immediately in with the difference being that the woodpecker algorithm will do a more aggressive search, "drilling" into the same button or menu item repeatedly. The creeper algorithm is just as thorough but searches for less noticed objects, e.g. hotkey shortcuts or accelerator key paths to tools or windows. Each progresses step by step in their local area but with different styles. However, both will move with the flock leader (chickadee) should it move out of the area. In most cases, should the leader move it will be to a tree outside of the local area.

The Brown Creeper and Downy Woodpecker algorithms are described bellow. First we describe the creeper algorithm. The algorithm for the creeper agent is similar to the chickadee with the following changes: (1) at the beginning of each search the agent needs to begin in a tree within one step from the primary agent tree (chickadee); (2) if available the agent selects the next command object in order from where it began (e.g. selecting the second toolbar button after having selected the first, the 22 combo box item after the 21st, and so on); and (3), where possible the agent selects an alternative path to an object (e.g. the hotkey route to a command button or the accelerator key sequence to perform the cut operation instead of using the activation of the “cut” toolbar button). Differences are highlighted by using italics.
Check for distance of current tree root from tree root selected by chickadee agent
If distance > 1 then
    Move to same tree as lead chickadee or a tree directly adjacent to it
End if
Do while x < total_moves_allowed
    Do while y < total_moves_allowed
        Search database for id of last command object activated by agent
        Determine next item in order from last – e.g. next toolbar button, etc.
        If available
            Select this next object
        Else
            Search to identify local window control structures
            Form lists of toolbar items
            menu items
            buttons
            list view items
            If preferred type of control structure available
                Choose one of preferred type of control structures
            Else
                Choose any of the above available control objects
            End if
        End if
    Move to and select the structure
    Save record of the step
    x = x + 1
    y = y + 1
    z = z + 1
    Record new state (new window or possible error message)
    If error message or error state achieved then
        Save data on tree nodes traversed to this point
        Save data on command object just activated
        Save content of error message if any
        Reset y to 0
    End if
    # Limit Checking
    If x = total_moves_allowed
        Select and Move To New Tree
        Reset x to 0
        # Prepare to begin a new search in a new tree
        Exit Do
    End if
    If y = total_moves_allowed
        Move back to original window
        Reset y to 0
        # Prepare to begin a new search
    End if
Loop
If z = total_moves_allowed
    Record State of Agent
Exit Do
End if
Loop
Next the woodpecker algorithm is described with differences from the chickadee algorithm including checking first to insure being in an allowed tree and utilizing a drill of a command object.

Check for distance of current tree root from tree root selected by chickadee agent
If distance > 1 then
Move to same tree as lead chickadee or a tree directly adjacent to it
End if
Do while x < total_moves_allowed
Do while y < total_moves_allowed
Search to identify local window control structures
Form lists of toolbar items, menu items, buttons and so on
If preferred type of control structure available
Choose one of preferred type of control structures
Else
Choose any of the above available control objects
Endif
Move to and select the structure
Perform test to see if object can be drilled
If test passes
Rapidly activate then restore the original state of the command object
Endif
Save record of the step
x = x + 1
y = y + 1
z = z + 1
Record new state (new window or possible error message)
If error message or error state achieved then
Save data on tree nodes traversed to this point
Save data on command object just activated
Save content of error message if any
Reset y to 0
Endif
# Limit Checking
If x = total_moves_allowed then
Select and Move To New Tree
Reset x to 0
# Prepare to begin a new search in a new tree
Exit Do
EndIf
If y = total_moves_allowed then
Move back to original window
Reset y to 0
# Prepare to begin a new search
Endif
Loop
If z = total_moves_allowed
Record State of Agent
Exit Do
End if
Loop
The Nuthatch Algorithm

The Nuthatch often takes larger moves from step to step as it frequently flies to higher branches after foraging in small hops or moving step by step down the tree. While a direct correlation to moving down a tree is difficult to implement on a GUI unless for example the window searched is a browser with a "back" button, the Nuthatch algorithm can be approximated by having its agent begin with recorded, already traversed, sequences. An example of a move down the tree would be to begin moving by first progressing to the third step of a four step sequence already taken path. From this lower level the Nuthatch algorithm could then look for ways to close the currently open window or activate one object before returning to the original window (a sideways move). To ensure movement back down a traversed path of tree nodes the algorithm would be allowed only one sideways move before needing to move down to the next lower step (in this case to the second step of the recorded path mentioned above). After a limited number of steps down the path or reaching the root of a tree the agent would then choose another tree node sequence to move down in the same fashion. As before the Nuthatch agent would again begin near the end of the recorded path and be allowed one sideways move before going to the parent node of the current node in the selected sequence. As the Nuthatch algorithm is dependent on having some sequences of traversed tree nodes already recorded its algorithm may need to be delayed in starting until after some number of chickadee agent paths have already been recorded. The psuedocode for the Nuthatch algorithm follows:
Check for distance of current tree root from tree root selected by chickadee agent
If distance > 1 then
  Move to same tree as lead chickadee or a tree directly adjacent to it
End if
Do while x < & z < total_moves_allowed
  Do while y < & z < total_moves_allowed
    Select a recorded tree node path sequence from current tree
    Set s = 0
    If sequence found for n-length sequence
      For x = 1 to n
        s = s + 1
        Progress through the steps to the n – s node
        Set selected sequence node as current base i.e. next move is the first step
        # Prepare for sideways move
        Search as any other agent for available controls
        If preferred type of control structure available
          Choose one of preferred type of control structures
        Else
          Choose any of the above available control objects
        Endif
        Move to and select the structure
        Save record of the step
        x = x + 1
        y = y + 1
        z = z + 1
        Record new state (new window or possible error message)
        If error message or error state achieved then
          Save data on tree nodes traversed to this point
          Save data on command object just activated
          Save content of error message if any
          Reset y to 0
        Endif
        # Limit Checking
        If x = then
          Select and Move To New Tree
          Reset x to 0
          # Prepare to begin a new search in a new tree
          Exit Do
        EndIf
        If y = then
          Move back to original window
          Reset y to 0
          # Prepare to begin a new search
        EndIf
      Next x
    Endif
  Loop
  If z = total_moves_allowed
    Record State of Agent
    Exit Do
  Endif
Loop
3.3 The Flock Algorithm

While the previous section has dealt mostly with the search patterns of individual flock members, this section deals with the search patterns of the flock, the members taken together. In the initialization phase of the group search algorithm it is assumed that the starting state will be the just activated application under test. For the focus of this research we are using as an initial area for exploration the testing of text and media editors. An editor considered ready to be tested is one that is opened with a new document or form loaded. A tree root is any command object on the just opened GUI. Initial tree roots are first mapped to a database recording each instance of each command object type. With a typical GUI having scores of control items available on the initial window (especially with menu items included in this set) many tree roots are likely to be identified. The result of the mapping will be the discovery of 50 to 150 command objects (tree roots). For illustration purposes we consider a beginning point of 25 command objects. A way to represent these roots is by means of a 5 X 5 matrix. The movement of the flock over time could then be seen as the movement across this matrix by a smaller square which represents, at the center location, the location of the chickadee agent and around it the roots one step away. As the mixed-species flock follows the chickadee across the space of the possible roots the search area would come to be covered over time. See below.
A further consideration for the group search algorithm is the rewards to be built in for finding “food” i.e. target values, in this case, application error states. For this let $\rho$ represent the amount of reward to be deposited at a tree node whose command when executed results in an error state. Similar to the distribution of trace amounts on a path for the ant search agents to follow for the Ant Colony System algorithm (Dorigo and Gambardella 1997), let nodes in the path sequence closer to the root also receive a reward but discounted to encourage exploration most intensely at the identified “error” node or just before it. For each parent node of the “error” node moving back to some limit of steps, a reward is assigned to the parent node(s).

Additionally, the reward amount must be allowed to degrade over repeated visits by search agents so that further exploratory searching is not truncated at this point. For simplicity of implementation let the assigned value of the selected parent node be discounted by 1 on each new visit. A parent node initially assigned a reward value of 3 then has the value of 2 after the first agent selects it on a re-foraging move. Each subsequent re-foraging move further decrements the reward level until no reward value remains. With these reward considerations in place, a change in how searching is done is needed for each
species specific algorithm – i.e. a check for the highest amount of reward available looking in all nodes must now be added. The search step of the algorithm now becomes:

First search all nodes for reward values
If value > 1 found
    Chose the maximum value or if more than 1 value at max level select 1 at random
    Select this node
Else
    Proceed with usual search
Endif

Just as a mixed flock’s tendency is to stay in an area when foraging proves successful (see above), the flock’s tendency to move out of a non-productive area also needs to be modeled. Modeling of this aspect of the mixed-flocks behavior has been achieved by the restless algorithm of the Parid agent leading the flock. After reaching a limit, the search leader moves to a new tree. Other agents must then move to the same tree or an adjacent tree.

In summary the mixed-species flock search algorithm has been modeled to mimic the characters of an actual mixed-species flock foraging through a forest. The algorithm implements collective behavior as it moves as a group following the lead searcher, the lead Parid of the flock. Additionally, the mixed-flock algorithm implements provisional perseverance at productive areas, and moving away from non-productive areas. Given the above characteristics it is hypothesized that the mixed-species flock search algorithm will perform better than an exhaustive search of the GUI space as it searches intensely in productive areas and moves away from non-productive areas.
CHAPTER 4

IMPLEMENTATION AND PRELIMINARY RESULTS

4.1 Overall Architecture of the Multi-Species Flock Search Algorithm

Chapter 1 gave an introduction to the multi-species search algorithm and some of its components. Chapter III elaborated on the characteristics of the flock and the flock’s member species. This chapter moves from the higher level descriptions given so far to the practical level in order to highlight the decisions made regarding implementation. These decisions are enumerated for three areas: (1) overall architectural requirements; (2) technical considerations for implementing the search agents; and (3), modifications made necessary by problems encountered in prototype development of the multi-species flock search algorithm.

To place the discussion of the overall architectural requirements in context, here is a brief description of the search algorithm. The model for the algorithm is a multi-species flock of birds foraging in a forest. Since it imitates a flock, the algorithm displays characteristics of a flock as a whole as well as characteristics of individual members. The event space of a GUI is modeled as a
forest of trees in which search agents are rewarded for finding errors as they exercise command objects at various nodes on the trees.

The basic pattern of the search is as follows: (1) an agent is activated from the pool of available agents; (2) the agents searches for its specie’s specific preferred type of object and, if available, chooses an object from this group, otherwise it chooses a target object from among the group of any available objects; (3) the agents activates that selected object in its specie’s specific way – creepers, for example, prefer activating a button using a hotkey or accelerator key sequence verses the direct activation by a button click used by a chickadee agent type; (4) the GUI is examined for any evidence of an error occurring and if evidence is found the path to the error is marked and any error message recorded; (5) changes in state of any kind whether evidenced by a new active window or different properties on the current window are tracked with new windows being modeled as a new tree for exploration; and (6), the agent then takes its next step or, if it has reached its limit of allowed steps in a row, a new agent is selected and the overall pattern repeats.

Visually the search sequence of the Mixed-Species Search Algorithm is as follows:
Figure 3  Diagram of the Mixed-Species Flock Search Algorithm
The above chart gives but the highest level overview of the code which drives the MSF algorithm. The language chosen to implement this search, as stated above, is Rational Visual Test. It combines the utility of Basic while adding hundreds of GUI identification and manipulation commands. Essentially a new application, GUI_TST (GUI TEST), was created with this language to run the MSF. In addition to code designed to identify and on occasion activate each of 27 types of objects found on the GUIs examined, the RVT language was used to create interface commands for the MySQL database used to store data gathered from the application. As there was no ODBC available for RVT, multiple functions were created for creating, loading, querying, and modifying each table utilized by MSF.

Four database tables are heavily utilized by the MSF as it progresses though various stages. These tables are: search_agents, node_info, frames, and agpaths. The search_agents table has fields for the number and type of each agent as well as counters for step, search, local step limit, tree step limit, and max step limit. As each agent begins its run this table is queried for the selection of the next available agent. After each step the active agent’s counters are updated and analyzed. Decisions are made about continuing a search depending on the state of the counters for a given agent.

The node_info table is described in greater detail below. Overall, however, this table is used as the repository for information about all identifiable objects and their properties per window identified by the MSF algorithm. Each new window discovered in the course of the MSF search has information about each
object located on it loaded to this table. The table is the source of information to each agent as it formulates commands to activate an object on the active window before it.

The frames table holds identifying information about each window. Its fields include the title of a window and a unique key used to differentiate windows of the same title from each other. The method used to determine this key is described below in section 4.2.2.

Finally, the agpaths table is the table that holds information about each step each agent takes. This table is also further described below at 4.3.1.1. In brief, this table is used by each agent to pre-record the step it is about to take in terms of the number of the following fields: the specific search and step involved, the title of the current active window, the node id of the object about to be activated, the command about to be used to activate the object, and the parameters that will be used. After the step is taken the resulting window frame id (from the frames table) and the resulting window title are also recorded as well as any resulting warning message. An additional field serves as a marker which is incremented should an error be discovered on this step. As the MSF algorithm progresses the marker fields are examined before each agent takes the first step on a new search. If positive values are found in these marker fields the search agent is presented with the choice to either repeat a path that led to an error being so tagged or the agent is given the choice to continue with an exploratory search.
4.2 Algorithm Architectural Requirements

4.2.1 Application Usability

The application that was developed to implement the MSF algorithm requires the ability to run multiple test cases on target applications with different combinations of agents – e.g. all chickadees, all creepers, and mixed flocks of different compositions. Further parameters for the limits to local, tree, and maximum length need to be frequently reconfigured. Additional configurable parameters include basic data regarding the application under test so that the MFS application can retrieve data concerning it.

Not surprisingly a number of GUIs were developed to make the dynamic configuration of the test application possible. Two frequently used GUIs are as follows: (1) the target AUT configuration screen; and (2) the MSF agent configuration dialog.

![CALIBRATE TEST](image)

**Figure 4**: View of the calibration GUI for the MSF application. Note the ability to enter and save the name and path of the application under test, the ability to configure the log level for MSF, and the ability to set custom control classes for the AUT.
Figure 5: Configuration GUI for MSF search. Note ability to set number and type of agents, to set limits for local area, tree, and maximum search. From this GUI the level of the reward for searching a found error is also configurable.
4.2.2 Window Identification

In early stages of prototype development of the MSF algorithm it quickly became obvious that recording a pre-state and a post-state to an agent move would need to be enhanced. The earliest implementation recorded only the window title of the active window before and after a move. However, this gave no indication of a change of state within the window and would therefore report as "no change" possibly quite significant changes in state for the active window. As the state of the window had changed and the active window would now respond differently to certain command sequences than before, it was decided to treat this active window as a new window, a window with a new property set.

Previously data about the objects in a window had been keyed to the title of the window. Now, with the decision to count a window as a new window, if it had a change of state, a new way of determining a key had to be found. The way found was to create a key for a window depending on the number, type, and state of the command objects it contained. For each type a triad was created consisting of the identifier of the type of object, the count of available objects of that type, and the total object of the type whether enabled or not. An example of this method for keying is that of creating a triad for the check boxes on an active window. Given that the identifier for a check box is the number 3, a window with 7 objects of type check box with 4 enabled check boxes would generate the triad 3-4-7. As 27 types are tracked per window a key with 27 triads or 81 elements is generated per window. If this key matches any other already recorded it is assumed the active window has been seen before and no further action is taken.
If, however, the key has not been seen before the key is entered in the frames
table of the database and the properties of that frame are added to the node_info
table keyed to this frame. This means of determining a key was then able to
provide a reliable system for identifying and tracking the state of active windows.

As window identification became more precise problems that were
occurring earlier came into clearer focus. It was now easier to see where
windows had been misidentified earlier. For example, the opening window for the
WordPad application and the print preview window both have the same title
though obviously quite a different set of active command objects on them.

4.2.3 Command Object Recognition

One of the most basic requirements for an algorithm which navigates
through a GUI space is that it be able to determine the kind and characteristics of
objects contained on the window before it. In this way agents can choose among
preferred types of objects to select from. Further, by recording the state of all
objects on a window before and after a move of an agent, the algorithm can track
the effects of an agent’s moves. The basic tool used for object identification is the
Rational Visual Test scripting language which has numerous window and
command object identification function calls, readily accesses the windows API,
and further provides tools and methods for identifying and activating non-
standard command objects. The mapping function mentioned above relies on
this language to identify from among 27 different types of command objects on a
given window under test. These object are then stored in a MySQL database
table (node_info) with entries keyed to the window being investigated.
Part of the Rational Visual Test suite of tools is a utility called WINFO. It has the feature of being able to identify the properties of an object clicked on by the cursor. The properties identified by this tool include the class of the object being viewed. By determining the class of an object, function calls to the API used to determine properties of the object and to activate the object can then be calibrated to now respond to a new class of custom controls. Without this calibration the mapping function would not be able to recognize many of the objects on some applications. As such the GUI_TST application used to implement the multi-species search algorithm utilizes a calibration utility. The calibration utility leads a tester through a step by step method of determining the class name for non-standard objects by utilizing the WINFO utility. Once the class is determined the calibration program loads the class name to a MySQL table (rootypes) so that future exploration, mapping, and activation of the custom objects in the GUI space may proceed.

On rare occasions an additional step is needed to access objects such as that of creating a wrapper for the Rational Visual Test API calls. On occasion the custom control is only partially enabled by providing the class parameter to the rational visual test function calls. In these cases it is necessary to create wrappers that augment the ordinary API calls by way of positional commands. For example some toolbars are recognized as custom toolbars and yet the buttons are not able to be selected. The wrapper toolbar button commands in this case take as parameters the toolbar number and the number of the button to be activated, then moves the cursor to the position of that button and performs a
click of the mouse. As the buttons are not readable the position of each button relative to the toolbar was recorded and loads dynamically into an array as the toolbar function is called. With this new information the function is able to perform the requested button activation. Fortunately most objects on most applications did not require this level of added programming.

4.2.4 Error States

An obvious requirement for the implementation of the algorithm is that it be able to handle finding what it intends to find, namely error states. Recall error states have previously been defined as being of four types: uninitialized states, unintended states, hung states, and crash states.

The primary means of handling error states is the implementation of a MYSQL database. Two personal computers, one with windows 98 and a second running Windows XP, were used for testing. On each personal computer a MySQL database was loaded and configured. On the Windows 98 computer version 3.23 of MySQL was used and on the XP version 5.0.24. MYSQL was chosen because of its ready availability, ease of use, and it is a database with which this researcher was already familiar. The database was used to record each found object and the paths that agents took to discover those objects. Then if a crash occurred, information about what was found and the steps needed to get there are preserved. As an added feature the step about to be taken is recorded before the actual step is activated so that, in the case that the subsequent step results in a hung state or a application failure, there will be a record in the database regarding what the state was just before the error state
occurred. Further the steps taken to a crash can be examined and repeated to prove reliability of the test.

A second implementation decision regarding how to deal with error states was to implement the use of the screen print utility to take a screen shot of what the search algorithm determined to be a warning type error state (one necessitating further action on the user’s part). When a warning message occurs, the screen print function is invoked and a jpg image stored to a results directory with a name which is a key for the agent – search – step on which the event occurred. For example, should the warning have occurred on the second search of the primary chickadee agent on its 5th step, the key used for the name of the snapshot is 1-2-5. The primary chickadee agent is designated as the number 1 search agent hence, 1-2-5 clearly keys to the 5th step on the second search by the first agent. By examining these images after a run, the tester can easily determine if they are true warning messages and also the path taken to get to the warning.

4.3 Technical Considerations for Implementing the Search Agents

There are four types of agents implemented in the multi-species search algorithm – chickadee, nuthatch, creeper, and downy woodpecker. While implementing these agents several programming challenges were encountered and overcome. This section highlights the nature of the obstacles and how they were overcome.

The basic challenges encountered were: (1) how to implement for any agent the selection and actual traversing of paths through the GUI; (2) how to
implement the nuthatch agent moving down the tree; (3) how to implement the downy agent drilling into an object, i.e. repeatedly selecting the same object over and over again; and (4), how to mimic the restless search the nuthatch agent demonstrates – exploring a local space for a few moves and, if finding nothing, moving to a new area in the current tree, and failing a find there moving to a different tree.

4.3.1 Selection of Nodes and Activation of Steps

The task of implementing the ability of agents to select their path through the GUI is two fold. On the one hand the heart of the MSF algorithm is the ability of agents to find paths through the GUI that match their own particular way of searching. This first requirement suggests a dynamic solution – one that allows agents to determine their paths for themselves on a step by step basis with each new step being a decision point. On the other hand an agent returning to a previously searched area needs, for a few steps at least, to follow the exact same steps that were taken before. This second requirement suggests a path that is laid out in advance with the agent merely following it step by step in a rote manner.

4.3.1.1 Selection

The primary driver for the MSF search algorithm reconciles this tension by the use of a loop which has at its beginning the choice between repeating a previously known search or that of beginning a new undetermined search. The selection of a previous search is based on the records made from the
undetermined searches. Given this dependency the undetermined search is described first.

The path of an undetermined search is dynamically created. Recall that after each agent step the active window of the AUT is scanned to see if it has been seen before. If the current window has not been seen before the results of the scan are entered into a node information table, node_info. The scan involves examining the current window for all instances of 27 different types of command objects using Rational Visual Test’s command object identification functions. These objects are: buttons, check boxes, combo boxes, combo box items, edit boxes, list boxes, list box items, list views, list view items, menus, parent menu titles, menu items, option boxes, spin boxes, status bars, status bar items, tab views, tabs, toolbars, toolbar items, trees, tree items, system menu, system menu items. If you are counting you note this adds up to 24 types to this point. The additional three types are alternative categories for custom buttons, list views, and list view items.

For each command object found 21 parameters (including a unique key associated with the object) are identified and loaded into the node_info table. These parameters, also gathered by use of Rational Visual Test functions, include: the type of command object, the object’s window class, the ordinal position of the object in relation to other objects of its same type, the text (if any) associated with the object, any label associated with the object, the object’s id number, the object’s parent ordinal number, a flag to indicate if the object is visible, a flag to indicate if the object is enabled, and finally, four parameters for
the relative position of the object in the active window (left, top, right, and bottom). Two additional counter fields are utilized a further steps are made - a counter to indicate how many times an object has been selected, and a counter for the number of times the selection of the object has resulted in a crash. Two more additional fields tell the sub routines called using this table which parameters to choose to most reliably activate the objects described. A frame ID is also entered for each object which is a key linking the object to the frames table which contains a listing of all the windows discovered while testing the AUT.

Once all this data has been entered for the properties of the command objects observable on the active window of the application, the selection of a command object is a simple process. Each agent searches the enabled objects on the current frame before it (window) for its preferred type of object by means of querying the node_info table entries for matches keyed to the current frame. If a match is found a selection from all the instances of the preferred type are selected by means of a query on these instances.

Agents have their own distinctive selection criteria for both the type and instance of objects preferred. In the field chickadees are noted for preferring, to a certain extent, searching near the ends of twigs. Given this preference the implementation of the selection of a node to activate for a chickadee type agent is made by the selection in terms of a 2 to 1 preference of an object type which has a parent. Objects which have parents tend to have longer sequences of ordinal numbers associated with them and it is therefore easier to choose objects whose ordinal numbers are higher thus imitating a selection near the end of a
branch. For example toolbar objects have by definition a number of toolbar buttons as their children. Additionally, combo boxes and list boxes or list views are all parent objects types with usually many children items associated with each parent.

The nuthatch has a similar search pattern in terms of type and instance as the chickadee, but in a more rigorous manner. If any instance of a parent type object exists on the currently active window the nuthatch agent selects a child of a parent type object in the top third of the range of possible selections so that it can start at the top and move backwards.

Creeper agents first select a type where a hot key option for selection is available. Among these available types a lower order sequence number is selected from instances available so that the creeper may inch up the tree representing available objects. Subsequent steps for the creeper involve saving the id of the node just previously selected and then choosing the next node in sequential order if it is available.

Downy agents prefer buttons, menu items, toolbar items, and tree items. If available downy agents will select instances of the above which have clues that they could be good candidates for opening and closing a dialog (e.g. having key words associated with them like ‘open’, ’close’, ’save as’, and ‘save’).

If the preferred type and instance of an object is not available for a given agent it selects, as a backup move, any remaining available object on the currently active window by means of a random selection.
If the beginning of the search loop described above selects taking a previous path (i.e. decides to search where errors have already been found) instead of choosing to begin a fresh search, a previous path is selected from the agpaths table. Agpaths is another MySQL table which has an entry for each step taken in the MSF search. Each entry in the table records three integers - an integer standing for the particular agent taking the step, another for the respective search it is enacting, and a third for the sequence number of the step being taken. The agpaths entries also include a node field which is keyed to the node items in the node_info table. An additional field serves as a marker field to indicate if a particular step has proven successful in that when it is executed an error state occurs.

To begin a repeat of a previous search an agent queries the agpaths table to find available sequences as indicated by entries which have step values matching the highest number of steps that can be taken in a local search. Once a particular entry of a final step is selected its previous steps are reconstructed by means of a query to find all steps for that agent given its specific search. The previous path can now be executed by means of activating in order the command objects (node field) associated with steps 1 through step limit.

4.3.1.2 Activation

The previous section has described in some detail how a particular node or command object is chosen. Once a command object has been identified by an agent for activation, the object is activated using the appropriate Rational Visual Test command. Specifically a sub routine, runAgentStep, is called which takes as
parameters the name of the active agent, its current search and specific step numbers, and, of particular note, a string containing the table entries for the node selected (resulting from a query of the node_info table for that node). The sub routine is essentially a case statement which has options for each command type (the 27 types listed above). Each case has four options possible one for each of the four types of agents. In this manner the appropriate Rational Visual Test object activation command can be associated with both the type of object to be activated and the preferred manner of activation specific to each type of agent. The activation commands are grouped in Rational Visual Test according to the class of the object being activated and use as their possible parameters identifiers associated with the object e.g. name, id, ordinal number, parent ordinal number, hot key, accelerator keys, or nearby level. As these associated identifiers are all contained in the string passed to this sub routine, it is easy for the appropriate RVT command to be loaded with the necessary parameters and then called.

4.3.2 Nuthatch

While modeling moves up a tree by an agent as moves beginning from a root and progressing onwards proved no particular challenge, modeling moves down a tree prove more difficult. An actual nuthatch can forage in a tree by merely flying to a certain height in a tree and then begin moving down the trunk or limb, but in order for the programmed nuthatch to perform similar behavior there needs to be some way to begin with a certain height already reached, e.g. to a level of 6 steps. However, this necessitates first traversing to that level and
so the agent is actually moving up the tree instead of down. While initial steps could potentially be ignored or not counted, what if an error was encountered on the way? Should these steps be recorded a second time so that errors can be properly investigated? Yet if this second recording is done they present the nuthatch agents as taking more than the allowed steps per agent. Chapter 3 already described in some detail the algorithm for mimicking movement down the tree. The added coding needed to handle possible errors for this first going up the tree to then go down was handled by not recording the preliminary steps but by adding a key to a field in the paths table which keyed the steps to a previously recorded path. Given the key, the tester is then able to see from the previous record what the preliminary steps are so that the likely steps to the error can be traced.

4.3.3 Downy

The challenge for implementing the downy search pattern is that of allowing “drilling”, the rapid activation of the same command object over and over again. If the applications had a forward and back arrow enabled there might be an easy solution to the problem. However, the applications tested did not have this functionality. To imitate this agent activity the activation of an object by a downy agent was coded with an exploratory feature. The exploratory feature is a function which is called after a given command object is activated by the downy agent. The function takes as parameters the state of the window before the command is used, the command and parameters just used, and the state of the window after the command is used. The test is to determine if the new window
can be closed resulting in the original window in the same state as before the command was issued. If this test is successful, a drill is then executed with the rapid repetition of activation of command object and closing the new window which results, then activating the command object and closing the resulting window again, repeated up to 10 times.

### 4.3.4 Chickadee

The Chickadees restless search was implemented by using two limits – local search and tree, and respectively. The search limit is the number of steps the chickadee was allowed before it needed to begin a new search. The tree limit is the number of steps allowed before the agent needs to change trees. By tracking the progress to both limits the chickadee agent could be implemented as searching frequently within the same tree and changing to a new tree after a given number of steps. The ability to keep searching in the same area if an error is found in a tree was implemented by way of a reward parameter, rho, which was associated with a successful search on a given tree node. As each new step is implemented, the chickadee first searches for a food reward. If found, the agent may repeat a search in the tree the node is associated with or the agent may chose to continue exploring. The choice is weighted equally. Therefore, given a choice situation, the agent is just as likely to continue foraging in a productive area as it is to continue exploring for new productive areas.
4.4 Modifications Made Necessary by Problems Encountered

Numerous problems were encountered during early runs of the MSF search algorithm. The following sections deals with the most significant in terms of understanding how the results found may have been affected.

Many text editor applications make available the choice of non-English scripts when selecting font types. To keep the testing simpler and to enable the investigator to understand the results obtained, it was decided to limit the script types to those used in English.

The open file dialog on most editors does not limit itself to certain directories. As an automated search program could modify or even damage crucial files, the directories which can be opened from and saved to were strictly limited to a single import and a single export directory respectively within the test directory structure.

After a few introductory runs, it was discovered that the number of roots actually selected by the MSF search algorithm may become clustered in certain areas leaving the majority of roots unexplored. As the cluster areas were not particularly productive areas in terms of finding errors, it was decided to modify the implementation. The fix for this problem was to track which roots had been selected (with entries to the node_info table), and then to force selection from the pool of under utilized nodes on a predictable basis. As flock members in the field are likely to look elsewhere after foraging in a non-productive area, this seems a reasonable solution. To allow the return to previously searched areas (also a likely behavior in the field), access to any available roots was allowed during
alternating sets of 100 steps per agent. In this way, during the first 100 steps of an agent, only roots non-previously selected, or selected fewer times than others, may be chosen. During the second 100 steps any root may be chosen, and during the third 100 under selected, etc. This implementation resulted in a more even distribution of selected roots.

An additional problem that was encountered early on was that of agents overly preferring to re-forage a food reward area to the detriment of continuing exploring behavior. Two reasons were quickly discovered as contributing to this situation: (1) the choice to pursue a reward was coded as the only choice after a search for a reward should prove successful; and (2), the value set for the reward was set too high. In practice, a reward of 1 or 2 set to both the direct parent of the node where an error was found and to the parent’s parent was found to encourage active re-exploration of a potentially error prone area without side-tracking the exploration of new areas for too long.
CHAPTER 5

RESULTS

Two text editors were selected for the side by side trails between the LOS and MSF search algorithms. The editors represent both a mature product that has been around for a long time and widely used and generally seen to be a reliable medium size program, i.e. WordPad Version 5. The second editor, HTML Writer, was chosen as it represents a program in early development. A copy of the 0.9 beta 4a version was obtained. Brief initial testing revealed HTML Writer to likely contain a number of errors whereas it preliminary testing found none with WordPad.

To date two series of tests, one for each text editor, have been run comparing the two search algorithms. The test design for the two editors is somewhat different and is explained below. Recall the hypothesis of this research is that the MSF search algorithm will prove to be, on a step by step basis, as effective in finding errors as the LOS algorithm.

5.1 The WordPad Comparisons

Two primary means of comparing search efficiency on a step by step basis have been devised. The first method is to show that, given the same limit
for an n-length sequence, each algorithm will find the same or comparable numbers of errors for each of the four classes of errors targeted. Though the MSF is not meant as a total coverage algorithm, testing its usefulness over the same search space as a comprehensive search algorithm is hoped to validate its potential usefulness for a larger search space. A second method of testing usefulness is to allow the MSF search to venture into longer n-length sequences. However, the search will have as a limit the maximum steps taken for a comprehensive search of a given lower n-sequence length. This second method is meant to provide evidence to determine if, given a limited amount of resources, the MSF could discover comparable results for the same cost as the already widely used comprehensive tests of length 2 and length 3 sequences.

Both methods are scheduled to be performed on both text editors. Given limitations of time it was decided to begin testing WordPad in the second manner – as it has had much use and errors at lower length-n sequence lengths are probably unlikely to occur. Conversely, given the early state of development for the HTML Writer application, it was viewed to be a more likely candidate to find errors even in the shorter sequence explorations of the GUI space. As such, it was thought this second application would be a good candidate for side by side tests that confine both algorithms to the same search space.

The level order search for sequences of length 2 in WordPad takes slightly over 5,000 steps. The implementation used in this testing performed the comprehensive coverage of all possible steps of length two in 5,032 steps. The results of the errors found by this search are seen below:
Figure 6: Results for level order search of all length 2 sequences in WordPad. The hang states were of two cases: (1) attempting to print either from the toolbar or from the main menu but without the printer having been set up, and (2), attempting to insert a video object that was purported to already be in use.

When the steps of the search are broken down into 1,000 step increments the distribution of when the discoveries occurred can be better seen.

Figure 7: The same single initialization error and the two hang states were discovered in each set of 1,000 steps. The two additional hang states were discovered in the first 1000 steps only.

In contrast the MSF algorithm did not discover the same errors in such an even distribution as seen below.
A couple of points are of note from the above chart. First, though the number of hang states discovered was less than that of the level order search, the chart does not show enough specificity as to whether these are the same states or not. This will be left to the next chart. The most significant obvious result, however, is that five distinct crash states were found by the MSF that were not found by the comprehensive search. Further analysis shows that these crash states were triggered by nuthatch and downy agents. Four of the crash states were generated while trying to insert objects into the application via the Insert -> Object … menu option. These four crashes were dual crashes, i.e. both WordPad and the objects attempted to be inserted crashed (Paint and PowerPoint). The next view indicates how many discrete errors were discovered.
The above chart better helps to display the results of the two searches. The same uninitialized error state was found by both algorithms. LOS discovered the same two hung states as did MSF and two more. MSF discovered all the crash states.

Though not enough iterations of these tests have been run yet to show what results are going to be statistically significant, if these trends hold the MSF may be shown to excel in discovering crash states and to perform slightly less better in detecting other states. These possible trends are being further investigated with the HTML Writer tests.

5.2 The HTML Writer Comparisons

As with the WordPad testing, this section reports only on the first part of testing planned. This section reports on the side by side testing of just the initial command objects. Further testing is planned to compare LOS to MSF as we did
for WordPad, i.e., with LOS covering all level 2 sequences and MSF taking the same number of total steps but venturing into level 6 territory.

5.2.1 Level Order Search of HTML Writer VS Multi-Species Flock Search For Length 1 Sequences

All the command objects on the presenting window of the application are activated by the LOS algorithm for 4 separate runs of 145 steps each. Each search, with one exception, found the same errors summarized below:

The LOS searches each discovered the same states over and over: an unintended state (a warning message with wrong information); two uninitialized states, i.e. calls to Netscape Navigator to test the html produced (but Netscape could not load the html), three hung states (two when attempting to do a next or a previous find query, respectively, when no previous query had been made; and one attempting to print with no printer set up); and two crash states (the printer crashed after hanging and after a particular menu selection the application would crash when objects were being mapped). The third crash is of
undetermined cause again occurring while just counting objects after an activation (which had not caused a crash).

The results from five MSF searches of 145 steps are shown below:

![Errors Per Run Chart]

As can readily be seen the MSF algorithm performed poorly in comparison to the comprehensive approach. More tests are being run but at this point it appears coverage is more important than a creative search given errors likely widely distributed throughout an initial GUI interface.

Again, as with the WordPad searches, the unique errors found over all runs were compared for the two search algorithms. The chart of this comparison follows.
It appears from this comparison that although the MSF does not perform as well in terms of several of the types of errors tracked, it performs better than the level order search in terms of finding crash states.

5.2.2 Level Order Search of HTML Writer VS Multi-Species Flock Search For Length 6 Sequences

Additional tests were run matching runs of length 4 to 6 sequences MSF but limited to the 145 total numbers of steps per run needed to cover all the objects on the initial window of the HTML Writer application. The chart showing the results follows below:
With the data so far, again not yet enough to run statistical validity tests on, it appears the trend is toward uneven performance for finding a wide distribution of errors with the MSF. At the same time it appears that the ability of the MSF to find crash states is still strong. The following chart shows the results of an examination of how many unique errors were found by the summed efforts of each algorithm (LOS vs. 4-6 step MSF Runs).
As concerning the pattern last time when the results of the searches are summed an interesting finding appears, i.e. for almost all cases the MSF finds all the errors the LOS finds, however in a more stochastic manner. The MSF also displays a somewhat greater ability to find crash states. However, all these tentative findings need further data to have a high level of confidence in their validity.

One additional tentative finding is that the additional error states discovered appear to be largely crash states so far mostly by the downy and nuthatch agents. Though it is not certain what the greater incidence of crash states by these agents means, a possible finding may relate to the pattern of the search. Both of the agents display a shorter time between activation of successive command objects than the other agents. The nuthatch is allowed to repeat up to n-1 steps in a row as a preparation for then moving down the tree (see Chapter 3). These preparatory steps occur without the need to take a snapshot after each step (since they are repetitions of an earlier recorded path). The downy specializes in rapidly activating then resetting a command object, activating, resetting, etc. It appears to likely be this quality is what triggered the newly discovered crash states when using HTML Writer. The same quality would explain the crash while attempting to load, stop, load, and stop another application object to WordPad as earlier noted.

To further explore the possibility that the downy agents are perhaps particularly good at detecting crash type errors a side by side comparison was run for a flock consisting of two downy agents and a chickadee versus the level
order every command object in order search. Again the MSF search is allowed to venture into the longer sequence runs (6 steps in this case) but is limited to the same number of steps needed to cover all the object at level 1 (146 steps). The following two charts show the results. The first chart shows the results of each of 5 runs by the MSF. The second chart shows the results of the 5 runs summed for each error type. Given this view it appears the MSF may prove more successful searching a particular application as mixes of agents are fine tuned to match the vulnerabilities of the given application.

![ERRORS PER RUN](chart.png)
CHAPTER 6

CONCLUSIONS

In the previous chapter results were shown from some initial side by side tests between the LOS and MSF search algorithms. These initial tests need to be supplemented by additional tests to further prove or disprove the trends that appear to be emerging from the data gathered so far comparing the two algorithms. Solid conclusions will need to wait for this data. However, if the data holds the following may be shown to be the case: that over a run or two the MSF may not perform as well as the exhaustive search given especially a confinement to the same n-length search; over the long run the MSF appears to gain ground nearly matching the discoveries of the LOS item per item; and further, the MSF tends to find additional errors not discovered by the LOS.

The ability of the MSF to gain ground and potentially discover a broader range of errors in the long run may perhaps be explained by three somewhat divergent views that present themselves as areas for further exploration. The first hypothesis is that the MSF gains ground through repetition merely because it is achieving a higher percentage of coverage. This may be found to be true especially when comparing runs of longer sequence to those of the shorter sequence (but broader coverage) LOS runs.
A second hypothesis is that errors tend to reside in the longer sequences where dialogs that present themselves have more of their configuration parameters exercised than is possible with very short sequence runs. An alternative to this second view is that more errors are eventually found not so much because of the larger space opened up but because of the strengths of the MSF algorithm. It can be argued that the MSF potentially presents a powerful combination of the ability to achieve coverage plus the ability to stress an application. The different agents stress different needed qualities in testing. The chickadee a restless moving to divergent part of the GUI, the creeper methodically moving from one command object to the next adjacent one in a local area, the downy stressing event queues, and the nuthatch looking in unlikely places.

Clearly research comparing side by side runs of the same sequence length will help to determine which of the above views is more correct. It will be especially helpful to find out if the possible trend detected from these early explorations proves out. Namely, that all, or nearly all, the errors detected by the exhaustive search are eventually discovered by the MSF when both are operating at the same sequence level; and further that additional crash error states are found that are not discovered by the LOS.

Finally, much research remains to be done in exploring the relative efficiencies of various groupings of agents as was done in the case explored above that utilized mostly downy agents. What if the flock were mostly all creeper agents, or chickadees, or … Clearly much further testing needs to occur to
determine what kinds and mixes of agents are best suited to test which types of applications. Many further explorations need to be made.
# APPENDIX

## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AUT</td>
<td>Application Under Test</td>
</tr>
<tr>
<td>EFG</td>
<td>Event Flow Graph</td>
</tr>
<tr>
<td>GSI</td>
<td>GUI Search Iterator</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>GUI_TST</td>
<td>GUI TEST</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>LOS</td>
<td>Level Order Search</td>
</tr>
<tr>
<td>MSF</td>
<td>Multi-Species Flock</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open Database Connectivity</td>
</tr>
<tr>
<td>PATHS</td>
<td>Planning Assisted Tester for graphical user interface Systems</td>
</tr>
<tr>
<td>RVT</td>
<td>Rational Visual Test</td>
</tr>
<tr>
<td>WINFO</td>
<td>Window Information</td>
</tr>
</tbody>
</table>
Works Cited


