8-2013

Discovering Divergence: A Framework for Finding Unexpected Behavior Using Directed Exploration

Heath G. Roehr
University of Nebraska-Lincoln, h.roehr@huskers.unl.edu

DISCOVERING DIVERGENCE: A FRAMEWORK FOR FINDING UNEXPECTED BEHAVIOR USING DIRECTED EXPLORATION

by

Heath Roehr

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Myra B. Cohen and Sebastian Elbaum

Lincoln, Nebraska

August, 2013

DISCOVERING DIVERGENCE: A FRAMEWORK FOR FINDING UNEXPECTED BEHAVIOR USING DIRECTED EXPLORATION

Heath Roehr, M. S.

University of Nebraska, 2013

Adviser: Myra B. Cohen and Sebastian Elbaum

Systems that are written to achieve the same high level specifications can vary in subtle ways. Depending on a programmer's objective, using one variant of a program or algorithm over another may be beneficial, and this objective may change over time. However we do not have sufficient techniques to compare two different system variants side-by-side to find specific behavioral differences, particularly in the absence of source code. Assuming two system implementations take the same inputs and produce the same outputs or exhibit the same behavior under most conditions, we want to find input instances where the behavior diverges for a given objective. In this paper we present a framework called UDivE to fill this gap. UDivE accepts a model of the input space and system constraints, as well as an objective measure for the output behavior that is of interest. It then uses a genetic algorithm to explore the input space of two implementations, guiding the search towards divergent behavior. We have implemented a prototype of UDivE and evaluate it on three different software case studies, each with different input spaces and objectives. In all three cases we find 'unexpected' divergent behavior. In addition, we take a first-step towards applying UDivE to a cyber-physical system by providing a feasibility study in which UDivE interacts with a simulation of an unmanned aerial vehicle (UAV), the results of which are validated on the UAV itself. We show that UDivE can produce promising results, even in the presence of a simplistic simulator.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisors, Myra Cohen and Sebastian Elbaum. Over the past two years, they have helped transform me from an eager new graduate student into a researcher capable of asking insightful questions and evaluating complex choices. I am excited to continue learning about the field of computer science (a journey on which I have only just begun), and they have provided me with the first step that will guide me throughout my career. Another professor that has had a large impact on my graduate education is Carrick Detweiler. I would like to thank him for all that he has taught me, and for serving on my thesis committee.

Further, being part of both the ESQuaReD Lab and Nimbus Lab during my time as a graduate student has been a tremendous advantage. Not only have I gained a diverse skill-set, I have been exposed to a wide array of computer science and engineering applications that have had a major impact on me. I would like to thank all of those involved in both of the labs.

To the rest of the faculty and staff that are a part of the Department of Computer Science and Engineering here at the University of Nebraska-Lincoln, thank you for the encouragement throughout my curriculum. All of the courses that I have taken (both at the undergraduate and graduate level) have shown me, from a first-hand perspective, the incredible world of computer science. In particular, I would like to thank Stephen Scott; it was during his machine learning course that I realized how excited I was about advanced topics in computer science, and t that realization helped guide me in the direction of graduate school. Thank you to Charles Riedesel for all of the advice during my undergraduate career. Finally, I would like to thank LaRita Lang for all of her help planning department events. Without her, the events that I helped to plan would have never happened.

To my Mom and Dad, I would have never made it this far without your continued support and encouragement. Not only have you helped me in any way that you could, without question, you never let me lose sight of my end goals and ambitions. Both of you have taught me, by example, what it means to be successful and to never give up. Thank you for everything you have done for me.

To my friends and lab mates, I would like to thank you for the countless hours of collaboration and discussion. This list includes Josh, Eric, Katie, Cory, Brady, Charlie, Brent, Dongpu, Amanda, and many others. To Andrew and Alex, you have been excellent friends, now and for the past several years, and you have been a great source of support.

To Erin, you have been an unwavering source of support and encouragement. Thank you for everything you have done, and everything you have taught me.

This work was supported in part by the Air Force Office for Scientific Research, Award #FA9550-10-1-0406, and #FA9550-09-1-0687.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A single software specification is often implemented by many programs and different approaches emerge and evolve to address the same problem. This is becoming particularly true in program families such as software product lines [9], application programming interfaces (APIs), or even methods within a given API. Given multiple implementations of a software specification, developers deciding which one to use must understand when and how they can differ in their behavior for a particular objective. Often the assumption is that they perform equivalently, because in most situations they will. But there may be a small set of inputs that deviate from this norm and a key to achieving such an understanding is identifying inputs that lead to such unexpected divergent behaviors between the candidate implementations.

Support for finding such divergent behavior is currently lacking. Efforts that do exist focus on the analysis and validation of versions of the same implementation of software-based systems, and exclude the consideration of cyber-physical systems. This has resulted in techniques to better understand the differences and impact of changes between software versions (e.g, [1, 10, 11, 16, 19, 20, 25, 26]). But these techniques may not scale when applied to completely different implementations, because

the size of the "delta" means they will report a large number of irrelevant differences and most of these differences will not lead to divergent behavior. In addition, many of these techniques require code or some intermediate program representation. Instead, what is needed is an input-output based approach that can identify where divergence occurs based on a given behavioral objective.

To address these challenges, we have developed a framework called UDivE (Unexpected Divergence Explorer). Given two implementations sharing a common specification and the same interface, and an objective measure, the framework's goal is to find a set of inputs where the objective measure diverges. As we shall see, the framework is simple yet powerful. It is simple because it operates on the implementations as black boxes, using a standard genetic algorithm guided by a fitness function that rewards larger divergent behavior. It is powerful because even though the effectiveness of the search depends on how the input space is modeled, very simple models are sufficient to reveal unexpected behavior in the four studies we conduct. In addition, UDivE's most expensive phase is trivially paralellizable.

1.1 Research Contributions

This thesis is motivated by the problem that a solution does not exist for identifying divergent behavior between two implementations of the same specification, particularly if the only method of interaction with those implementations is from a black-box perspective. In an effort to address this problem, we present UDivE, a framework for the automated identification of inputs that may cause two implementations' behavior to diverge. Not only do we provide a design of UDivE, but also an instantiation of the framework. We evaluate UDivE's effectiveness in three different software domains: polynomial root-finding, image scaling, and aircraft collision resolution. During this

evaluation we search for divergent behavior and present the results, demonstrating the effectiveness of UDivE. In addition, we extend this evaluation to a cyber-physical system, providing a first step in allowing UDivE to conjecture about the behavior of a cyber-physical system by interacting with a simulator of the system, and then verifying UDivE's results on the cyber-physical system itself. Summarized, this research makes the following contributions:

1. Recognition of the problem of identifying divergent behavior between two implementations of the same specification and its definition in the context of automated guided input generation.

2. Design and instantiation of UDivE: a framework for the automated identification of inputs that may cause two implementations' behavior to diverge.

3. A study of UDivE on three different software domains (root-finding, image scaling, and aircraft resolution) illustrating its application and potential.

4. A feasibility study exploring a first step in the application of UDivE to a cyber-physical system, including the interaction with a simulator and the verification of UDivE's results on the physical system.

1.2 Overview of Thesis

The remainder of the thesis is laid out as follows. Chapter 2 presents a motivating example that further illustrates the usefulness of UDivE, as well as background material on genetic algorithms and other related work. Chapter 3 introduces UDivE, defines its objective formally, and describes its architecture in a component-wise fashion. In Chapter 4 we evaluate UDivE on three different software case studies, and present the

results of the evaluation. Chapter 5 explores UDivE's application to a cyber-physical system, and provides a feasibility study in which UDivE interacts with a simulation of such a system, the results of which are verified on the cyber-physical system itself. Chapter 6 concludes the thesis and discusses future work.

Chapter 2

Motivation and Background

In this chapter we present a motivating example based on one of the artifacts analyzed in Chapter 4, Aircraft Collision Detection and Resolution, to illustrate the potential of our approach, and provide some background information on genetic algorithms and other related work.

2.1 An Example of Divergent Behavior

We begin with an example of a system that could be implemented as a cyber-physical system, however we only consider it only from a software perspective, evaluating only the algorithmic implementation of the system. We present the domain of the system, and discuss the components that we consider in this example. We also discuss related existing approaches, and why they are inadequate. Finally, we present our proposed approach and discuss how it applies to this example.

2.1.1 Domain

Aircraft Collision Detection and Resolution (CD&R) algorithms are one of the key components to handle increasingly congested air spaces. These algorithms aim to detect critical loss of separation between aircraft and recommend modifications to an aircraft's flight plan to avoid a collision. Not surprisingly, many CD&R algorithms exist and many more are emerging [13, 14, 18]. Just within NASA's Airborne Coordinated Conflict Resolution and Detection (ACCoRD) framework there are 15 algorithms that can be parameterized to implement 38 operational resolution techniques [23].

These algorithms take different approaches to collision detection and resolution, yet they all operate on a pair of flight plans (one for the main vehicle, *ownship* vehicle, and one for the intruder, *traffic* vehicle). A flight plan consists of a series of 4-dimensional points known as trajectory change points (TCPs), each of which encode latitude, longitude, altitude and time of arrival at the respective TCP. Given the criticality of these algorithms, it is surprising that many of their trade-offs are not well characterized.

Unknown Divergence. Consider for example two of the most basic algorithms provided by ACCoRD, RRGS and RRTRK, which account for aircraft ground speed and track angle, but ignore more complex factors such as three-dimensional maneuvers or directional constraints. These two algorithms have the same collision detection functions but employ distinct resolution approaches. It is not clear from the literature or from the ACCoRD implementations and documentation how much they may differ in their proposed flight plan adjustments. This is important as these adjustments can translate into longer flying distances or durations, and potentially into conflict propagation with other aircraft.

2.1.2 Existing Approaches are Inadequate

Existing approaches are inadequate to provide much insight into how the implementations of these algorithms may differ. Approaches that perform a syntactic comparison of the implementations, such as the Eclipse IDE Java Source Comparison tool, reveal that 92% of the lines of code between the implementations are syntactically different. Although this may be valuable in some contexts, it provides limited insight about the behavioral differences between these implementations.

Existing test suites for the implementations have the potential to identify some differences, if they are rich enough. In practice, however, test suites are usually not exhaustive enough to detect subtle differences. In the context of ACCoRD, running the 12 test flight scenarios that come with the package does not reveal any differences between the implementations of RRGS and RRTRK. Automatic approaches for test generation may help to mitigate that limitation. Since they are not focused on detecting diverging behavior they spend most of their effort generating tests that expose the same behavior on the implementations. For the example scenario described, 25000 pairs of flight plans generated in 6.5 hours using bounded random input generation did not result in any test that caused a difference in flight plans when attempting to resolve a conflict with RRGS or RRTRK. In fact, the vast majority of randomly generated inputs did not even contain a conflict to resolve, despite the fact that they were valid flight plans generated within a valid range. Utilizing a more sophisticated test case generation tool that uses mixed concrete and symbolic execution, CATG [6], did not render better results. Not only did it require us to simplify RRGS and RRTRK in order to successfully perform the necessary instrumentation, but after 6.5 hours none of the approximately 12,000 generated tests achieved a difference in flight plans. Again, this is not surprising given that the goals of these tools is to achieve higher

levels of coverage, not to find subtle (and rare) behavioral differences. But it helps to illustrate that these types of approaches are not cost-effective for identifying divergent behavior.

Further, approaches that generate tests for code that changed between versions (e.g., [11, 16, 26, 27, 34]) are not helpful in this setting because the delta between the implementations is often large, negating the value of the textual differences. These approaches require access to the implementation source, which in the case of third party libraries, for example, may not be available. Still, our work is in part inspired by this family of techniques that guide test case generation.

2.1.3 Our Proposed Approach: UDivE

Instead of an exhaustive test generation method, which will not scale, we propose to perform an exploration of the implementations' behavior space that favors the generation of tests which reveal divergent behavior. In the context of RRGS and RRTRK we want to favor the generation of flight plans that produce the greatest difference between the implementations in terms of additional distance travelled from the original flight plan. The proposed approach does not analyze the implementations' structure, only their inputs and outputs in an effort to converge towards inputs that reveal the greatest difference.

Let's suppose we are interested in minimizing the difference in flight path distance created by the two implementations when compared to the original flight path. Given this objective, we want to explore whether, given two aircraft, RRGS and RRTRK can return results that diverge by more than some percent difference. For this example we choose 5%. If we find such a flight path, then we can use this to guide us in our selection of algorithms.

UDivE employs five elements:

- A representation of the inputs that can be manipulated. In our example we have two flight plans, one for the *ownship* aircraft and one for the *traffic ship* aircraft, both consisting of 11 TCPs [1].

- Optional constraints on the inputs to be explored. In our example, we target an area of 100 by 100 miles with starting airports located at the southwest and southeast corners and the ending airports at the northwest and northeast corners.

- Initial inputs. These inputs can be generated randomly, obtained from an existing test suite, or crafted to explore particular scenarios. In our case, we select the simplest flight plans consisting of a straight path between airports in opposite corners.

- An objective function based on the implementations' observable behavior. In our example, the function consists of the difference between the extra distance travelled by the adjusted flight plans that result from each implementation.

- An algorithm that guides the input generation toward divergent behavior based on the objective function (in our framework we utilize a genetic algorithm). This algorithm will operate iteratively, with the goal that each iteration will move the generated inputs closer to divergent behavior.

[1] Note that only the *ownship* can maneuver to avoid a conflict, however the inputs for both the *traffic ship* and *ownship* flight plan are manipulated to find the conflicting flight plans.

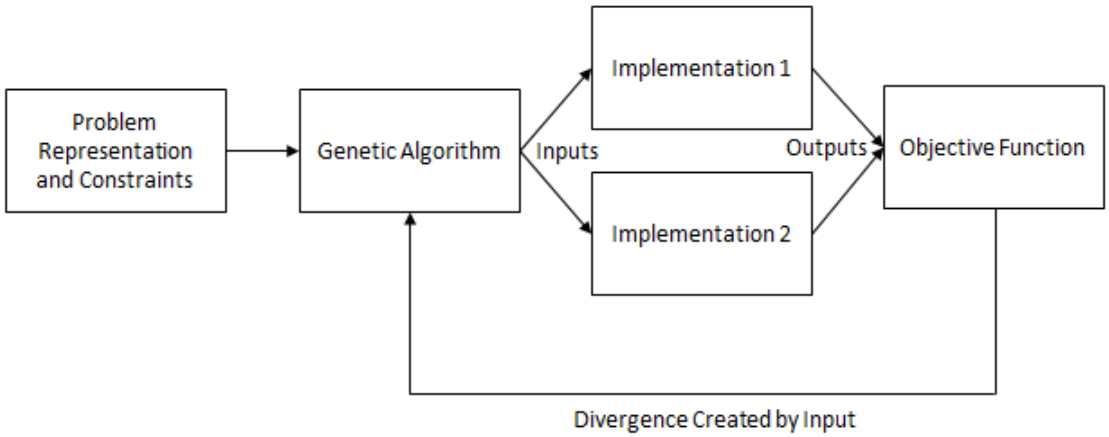

Figure 2.1: High-level conceptual depiction of UDivE's operation.

As shown in Figure 2.1, the problem representation and any optional constraints are provided to a genetic algorithm that generates an initial input (based on the problem representation and constraints) that are consumed by each implementation. The outputs produced by the implementations are consumed by the objective function that evaluates the amount of divergence created by the input supplied to the implementations. This data is consumed by the genetic algorithm and, together with the problem representation and any optional constraints, is used to guide the generation of subsequent inputs as the process repeats.

Given the above instantiation, as shown in Figure 2.2, after an exploration of 116 iterations (20 tests per iteration) taking a total of 6.5 hours, UDivE succeeds in detecting divergent behavior between RRGS and RRTRK. On this graph the `x-axis` shows the number of iterations and the `y-axis` shows the divergence. Furthermore, most of UDivE's execution is trivially parallelizable; the test cases in an iteration can be run on different machines, reducing the exploration time to less than 10 minutes when 40 nodes (2 nodes for each test, one for each implementation) are available.

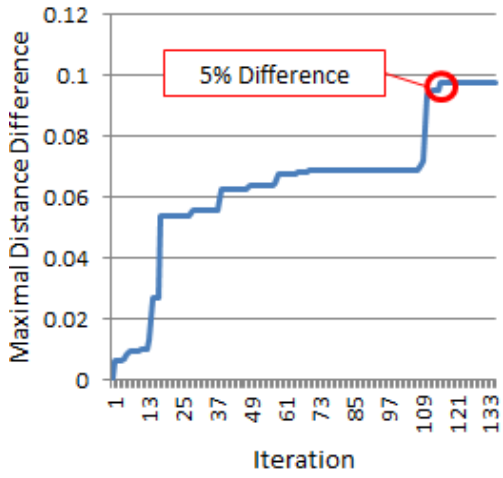

Figure 2.2: Evolving flight plans while favoring exploration of diverging behavior.

UDivE evolved the flight plans that serve as inputs to RRGS and RRTRK to reveal differences of more than 8 miles (5%) between the adjusted flight plans they generate.

Figure 2.3 shows the input flight plans leading to this difference, and the outcome of the implementations in terms of the adjusted plan. In this figure the trafficship aircraft is a triangle. The ownship aircraft is a circle. In the first frame the two aircraft have a conflict (the markers are superimposed). In the next two frames there is a distance between the triangle and circle marker showing that the conflict has been resolved. The third dimension of time is not explicit on these graphs; the location of the markers show where the planes are located at the same time. The changed plan (shown as a dashed line) is different in each frame. On the left is the RRGS and on the right is the RRTRK algorithm. It is evident that the adjusted plan produced by RRGS (which flies ahead of the traffic ship) is more efficient than that by RRTRK (which adjusts the flight plan to go beyond the traffic ship). Interestingly enough, the ACCoRD Stratway implementation which uses multiple CD&R implementations in sequence, always attempts to employ RRTRK before RRGS, which seems inappropriate at least for some cases.

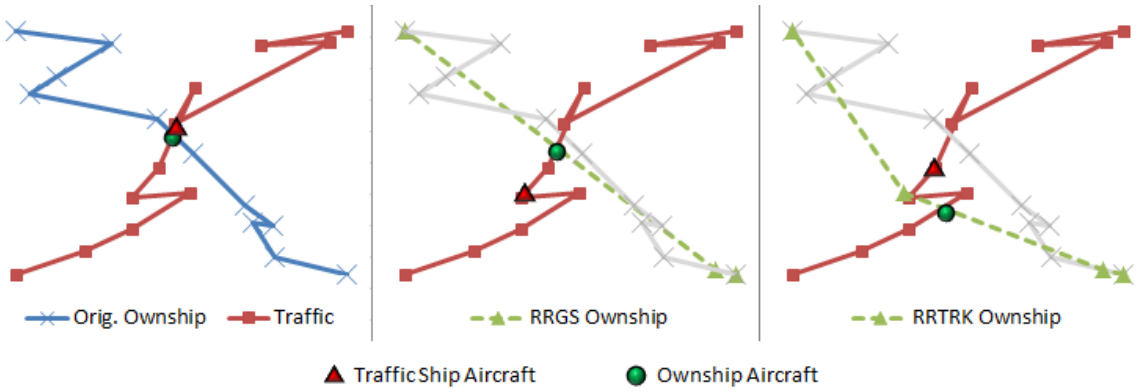

Figure 2.3: Input flight plans leading to collision, adjusted plan by RRGS and by RRTRK (in dotted lines). The adjusted plan of RRTRK results in 8 more travelled miles (5%) than RRGS

This scenario has illustrated the potential and some of the unique dimensions of UDiVE. Still, it simplifies the problem domain input (e.g., airports' location and area size), considers only one objective function (others may include the time travelled or the number of disturbances caused to other flight plan) and ignores many constraints (e.g., aircraft must maintain a minimal speed and cannot change velocity instantaneously). In the next sections we explain how UDivE can support more sophisticated representations of the domain's input, allow for more complex objective functions, and handle families of constraints throughout the exploration process.

2.2 Background

This section presents background information on genetic algorithms and how they function. In addition, it presents several other areas of related work that are relevant to UDivE.

2.2.1 Genetic Algorithms

In this work we leverage a genetic algorithm. A genetic algorithm (GA) is a population based meta-heuristic search technique that aims to optimize a given objective function, while obeying a set of constraints [30, 31].

An objective function (implemented as a concrete "fitness function") can minimize or maximize the objective depending on the problem. For instance, in our aircraft collision avoidance avoidance example, we would choose to search for maximum distance between the new flight paths and the original conflicted one.

GAs have been used to solve many software engineering problems such as test generation, reverse engineering, and refactoring [15]. A GA encodes the problem space as a chromosome which is made up of a set of genes - the primitive elements. In other words, each chromosome encodes a candidate solution to the problem at hand. Each gene within a chromosome has a domain of valid values that it may hold. This domain may be the same for all of the genes in a chromosome, or it may differ depending on the gene. A population consists of a set of chromosomes.

In order to guide a GA's search, GAs make use of a variety of evolutionary operators that are designed to mimic biological evolution (e.g. elitism, selection, crossover, mutation). The fitness function is responsible for measuring the quality (or "fitness") of each chromosome based on the given objective. At each iteration (or generation), each of the evolutionary operators are applied to produce the following iteration (or generation). The evolutionary operators we consider are described below.

Elitism The elitism operator selects the n most fit chromosomes from the population and allows them to propagate to the next generation. Because they are the most fit, we would like to guarantee that they exist in the next generation. Depending on the configuration of the GA, chromosomes selected by the elitism operator may also

be spared from mutation (described below). The value of n is problem specific, and defined by the user of the GA.

Selection The selection operator is responsible for selecting chromosomes from the population that will propagate into the next generation. Two common approaches to selection include a simple rank selection (during which the most fit are always selected), or a probabilistic approach in which the most fit chromosomes have the largest probability of being selected, but it is not guaranteed. This type of selection is often termed roulette-wheel selection. Note that if chromosomes have already been selected by the elitism operator, they will not be available for further selection because they have already been chosen to propagate to the next generation.

Crossover The crossover operator is responsible for mating parent chromosomes to produce child chromosomes. During crossover, two parent chromosomes exchange genes to form two child chromosomes. The parent chromosomes will persist, for a total of four chromosomes after mating two parents. The method of gene exchange varies. Single-point crossover involves genes being exchanged around a single point in the parent chromosomes. Multi-point crossover involves genes being exchanged between multiple points in the parent chromosomes. The location at which crossover occurs in the parent chromosomes is either fixed (e.g. always the midpoint of the chromosome), or may be selected randomly at the time of crossover.

Mutation The mutation operator is responsible for increasing diversity in the population. Mutation occurs at the gene level. Generally, only a small percentage of the genes in the population are selected for mutation. The type of mutation that occurs depends on the encoding of the chromosome, and the domain of the genes. In its most simplistic form, if each gene is simply a binary digit, mutation will flip the digit

from 0 to 1, or vice-versa. In the event the domain of the genes is more complex, say for example, each gene may take any integer value in a predefined range, mutation might simply select a new value from the range (termed full-range mutation), or it could increase or decrease the gene's current value by some percentage (termed creep mutation). Another possibility is that a gene chosen for mutation will be swapped with another gene in the chromosome (termed swap mutation). Further, a check must be performed to ensure mutation does not introduce gene values that appear outside of the allowed domain. The rate and type of mutation used are problem-specific and must be determined by the user of the GA.

The GA continues to iterate until either it has met a predefined goal (such as reaching a predefined fitness value), is stuck in a local optimum (or, ideally, the global optimum), or a predefined maximum number of iterations have been completed.

2.2.2 Other Related Work

This section presents other types of related work. We first discuss techniques that are primarily concerned with the analysis and characterization of differences between two versions of the same system. Next, we present related, but distinct, uses of genetic algorithms, N-version programming, and program refactoring techniques. We then discuss the use of meta-heuristic search methods in the context of cyber-physical systems, including unmanned aerial vehicles (UAVs). Finally, we discuss why UDivE is unique and not constrained by some of the limitations that hinder related approaches.

The idea of looking for divergence in programs is not new, although to date the focus has been on different versions of the same system. Techniques include impact analysis [2, 19, 25], program differencing [1, 17], and differential test case generation [16, 26, 27].

During impact analysis two versions (v and v') of the same system are analyzed, most commonly by reasoning about their control flow, to identify code affected by modifications made to v in v'. Impact analysis can be used to determine what effects a change has created in v' after the change has taken place. Another, more proactive, type of impact analysis is predictive impact analysis, during which the effects of a change are predicted before they are instantiated in v' [2, 19, 25]. However, many of these approaches require access to source code for analysis or instrumentation [25]. Further, some traditional impact analysis techniques such as call graph based analysis, static program slicing, and dynamic program slicing do not adapt to evolving software releases efficiently, and must recompute a large amount of information in order to re-analyze new releases of a software system [2, 19]. Other impact analysis techniques that leverage whole-path profiling improve these limitations by accommodating software evolution with a lower cost and requiring access only to system binaries, rather than source code, but still require that the binaries be instrumented [19].

Program differencing is a lighter weight technique that finds changed portions of code in two versions of a software system by operating on the source code or on source code abstractions such as abstract syntax trees (ASTs) [1, 17]. Certain approaches to program differencing focus on the special considerations that must be made for object oriented software systems, due to the complexity that object oriented features and the relationships between them may add to a software system, especially when making syntactic changes that could create subtle and unintended effects [1].

Differential test case generation approaches analyze different versions of software systems to identify changed sections on which the generated tests should focus. For example, given a software system version v, when a new version of the software system v' is released, its existing test suite of often executed against it to ensure that regression faults were not introduced between v and v' [16]. However, if the existing

test suite is not rich enough to exercise all of the changes that have occurred between v and v', regression faults may be unintentionally introduced into v'. Therefore, tools that focus on the portions of v' that differ from v, and automatically generate tests to exercise those differences can allow regression faults to be exposed and presented to developers. However, some of these techniques do not perform well when large changes occur between versions. A limitation of automated behavioral regression testing is that it is designed to interact with localized changes that involve less than a few classes and may not be ideal in the case of extensive changes.

The use of directed symbolic execution (DSE) allows for the discovery of inequivalence between two versions of a program and creates a behavioral delta that characterizes the input values that lead to different behavior between the two versions [26]. This type of tool can support the evolution of software system test suites [26]. Another approach that combines the efficiencies of static analysis with the precision of symbolic execution is directed incremental symbolic execution (DiSE) [27]. This type of technique is capable of generating path conditions that characterize the differences between two versions of a software system v and v'. The goal is then to cost-effectively direct symbolic execution on v' to explore path conditions that may be effected by the changes between v and v' [27]. Path conditions that are deemed effected by DiSE can be used during regression testing by supplying the solved path conditions as inputs to test cases [27].

All of the techniques discussed above assume that most of v and v' are the same, to be cost-effective. In systems such as the ones we have evaluated with UDivE, the whole program, or the vast majority, would be marked as impacted, changed, or inequivalent. This would provide little or no information when using the above techniques, and their execution would be cost-ineffective. This is expected because they are designed to operate on differing versions of the same system, whereas UDivE

is capable of operating on systems that are entirely different in their design and implementation.

Our work is also related to the use of genetic algorithms to generate pseudo-oracles by finding inputs which produce differences in program output. The difference is that they focus on a single program at a time [20] and the use of program transformations is required. UDivE focuses on two programs at a time, and does not require program transformations. There has also been research on plagiarism detection [37], with the aim of finding portions of algorithms that have re-used without permission, but this work looks for algorithm similarity as opposed to whole program behavior and divergence of that behavior.

N-version programming is also related to our approach in that multiple, different program implementations are written for the same specification. The focus of N-version programming, however, is to introduce diversity to increase system reliability, not to look for differing behavior. In fact, N-version programming presents an interesting context for UDivE to explore.

Work on testing program refactoring is related to UDivE [11, 34]. In this line of research the goal is to find behavioral differences in re-structured code. The behaviors should be identical, however, since refactoring is supposed to preserve program semantics. This line of work is more akin to testing. For example, refactoring engines are tools that automatically apply software refactorings. These engines are often found in integrated development environments (IDEs) such as Eclipse or NetBeans [11]. Automated testing of these refactoring engines can take place by generating complex programs that serve as test inputs (in the form of Java ASTs) that are then checked against an oracle after the refactoring has taken place [11]. The goal is to generate test inputs that, when checked with an oracle, show a behavioral difference in the refactoring indicating the presence of an error with the refactoring engine.

There has been work in the area of applying genetic algorithms and multi-objective genetic programming to various cyber-physical systems [7, 8, 24, 32]. For example, using a genetic algorithm to optimize the power consumption of a cyber-physical system model that requires various actuators [7]. Or, the use of genetic algorithms for UAV path planning and routing [8, 32]. Multi-objective genetic programming has been used to control a fixed-wing UAV as it attempts to accomplish a goal (such as hovering hear a radar emitter) [24]. However, none of these approaches use a genetic algorithm to search for divergent behavior between varying implementations of a cyber-physical system. Rather, they are concerned with controlling or optimizing only a single system.

UDivE is unique because it is capable of interacting with program implementations from a black box perspective. It expects completely different program implementations and designs, and runs the implementations in their original form, deriving new inputs only through the assessment of output differences between the implementations. UDivE does not require the generation of an intermediate or alternative program representation (such as an AST); it only requires an executable of each implementation. Further, UDivE is not restricted by the language in which the implementations have been written (in fact, each implementation could be written in a different language). We further discuss UDivE's execution and applicability in Chapters 4 and 5.

Chapter 3

The UDivE Framework

In this chapter we begin with a formulation of the general problem that we are trying to solve. We then present the architecture for our directed exploration framework, UDivE.

3.1 Unexpected Divergent Exploration

We begin by defining a function $measure : (P, Input_p) \rightarrow \mathbb{Z}$ that maps the behavior of a program P exercised by an input $Input_p$ to a metric M that characterizes that behavior. Let us also define a function $diverge : (Relation, \mathbb{Z}, \mathbb{Z}) \rightarrow Boolean$ that returns $True$ if the specified relation (e.g., greater than, less than, equal, within a range of) between the two $\mathbb{Z}$ metrics holds, and returns $False$ otherwise. Given a specification S, programs P_j^s and P_k^s that are supposed to implement S and accept a set of inputs I_p, and an expected relation r between the behavior of P_j^s and P_k^s, the problem of identifying divergent behavior consists of identifying $I_{div} \subset I_p$ such that:

$$I_{div} = \{\forall i \in I_{div} | diverge(r, measure(P_j^s, i), measure(P_k^s, i)) == True\}$$

In the context of our motivating example, the programs in question are RRGS and RRTRK, both of which match the general specification of a conflict detection and resolution algorithm, and consume two flight paths as input. The *measure* function maps the behavior of these programs to the distance traveled as defined by the adjusted flight path. In our case, the developer was interested in exploring whether RRGS and RRTRK behavior could diverge by more than 5%. For these particular implementations, measure, and relation, *diverge* will return $False$ for most inputs so the challenge is to cost-effectively explore the space of flight paths to identify a divergence.

3.2 UDivE Architecture

Figure 3.1 shows the overall architecture of UDivE. At a high level, UDivE takes as input a problem model (1) and through the use of a generation (2) and execution driver (3) runs a genetic algorithm to search for inputs that make diverge true (4). In the event divergence has not yet been identified (i.e. the hypothesis is still valid), the fitness of the generation's metrics is computed (5) and the process repeats. The framework iteratively evaluates a hypothesis until invalid. The returned input(s) represent I_{div}. In the following subsections, the number of each subsection corresponds to a number assigned to each of the components shown in Figure 3.1.

3.2.1 Problem Model

A problem model PM encodes the requirements and configuration of a UDivE application through a 5-tuple of the form $PM = \{CT, Pop, C, Op, Ftn\}$ where CT is a chromosome template, Pop is the population size, C is a set of constraints, Op is a set of evolutionary operators, and Ftn a fitness function.

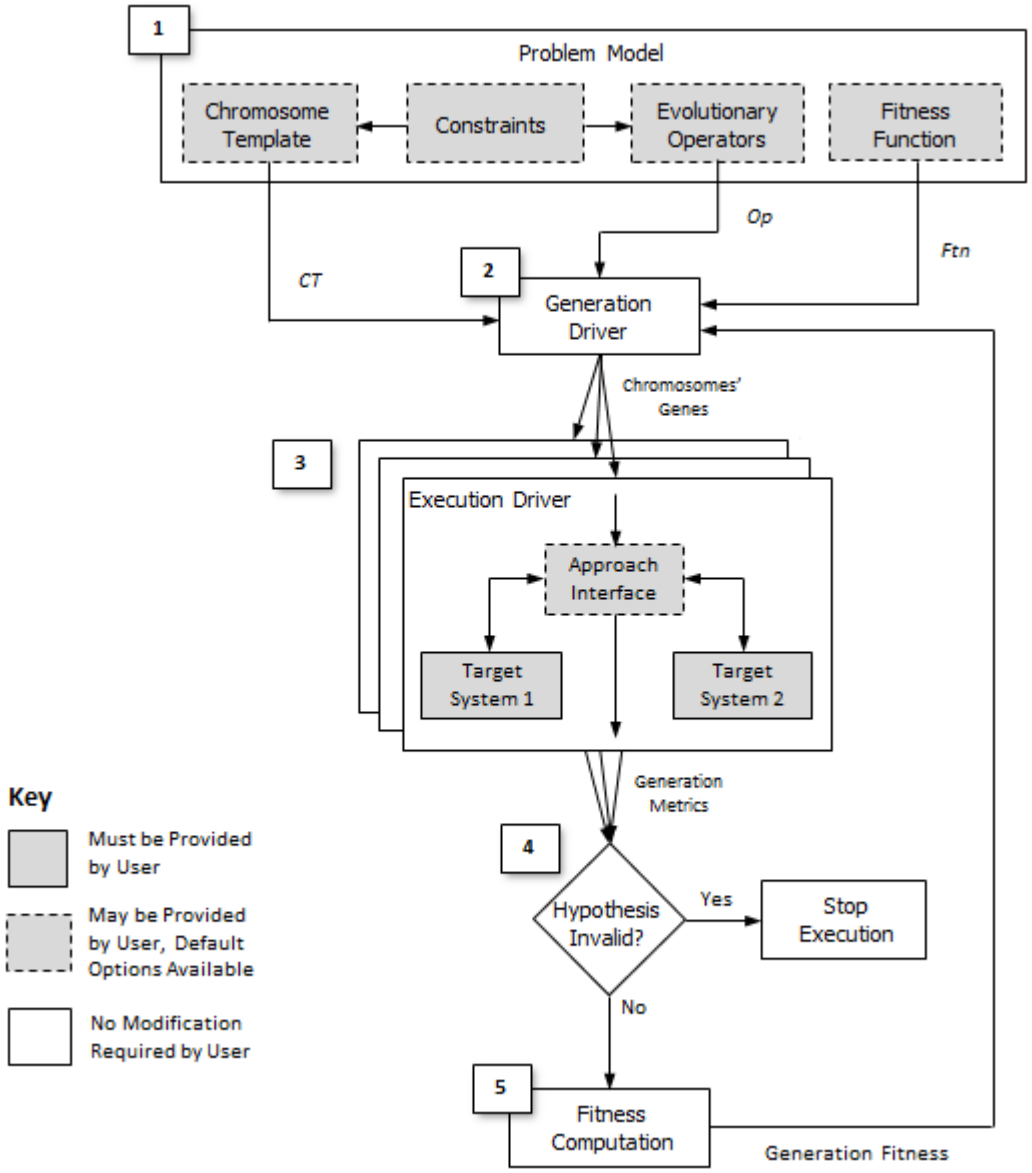

Figure 3.1: UDivE Architecture

Evolutionary Operator	Available Options
Elitism	{n-Best, None}
Selection	{Roulette Wheel, Rank}
Crossover	{One-Point, Two-Point}
Mutation	{Full-Range, Creep, Swap}

Table 3.1: Supported Evolutionary Operators

CT contains a data structure for the chromosome and its genes, constraints for initialization $C_{init} \in C$ and permanent constraints C_{perm} that are obeyed during evolution. Op is a set of evolutionary operators to be used during evolution; the framework's built-in operators are shown in Table 3.1. UDivE currently supports elitism (where the value n determines how many elite chromosomes are retained from generation to generation), two standard selection and crossover operators (Roulette Wheel and Rank), and three types of mutation [30]. Roulette wheel selection is a probabilistic approach where the most fit chromosomes have the highest probability of being selected, but this is not guaranteed. Rank selection simply selects the most fit chromosomes. Full range mutation selects a new (random) value for the gene from its domain. Creep mutation increments or decrements the current allele value by some random number in the range $(0, N_{creep}]$. A swap mutation selects two genes within a chromosome and swaps their values. R_{mut}, N_{creep} and the method of mutation are part of the PM supplied to UDivE. Ftn is a fitness function that computes the fitness of a chromosome. The genes of each chromosome represent the input to the implementations. The fitness is based on the *diverge* function which was defined as an input to the framework.

In the context of the collision detection and resolution example from Chapter 2, we show part of the PM_{cdr} as [1]:

$$
\begin{aligned}
&CT = flightpath = [2]\{ownship, traffic\},\\
&ownship = \{TCP_{origin}, TCP_{path}[0..8], TCPdest\},\\
&traffic = \{TCP_{origin}, TCP_{path}[0..8], TCPdest\}\\
&TCP = \{longitude, lattitude, altitude, time\}\\
&Pop = 20,\\
&C_{init} = \{TCP_{origin} \neq TCP_{dest}...\}\\
&C_{perm} = \{...\}\\
&Op = \{...\},\\
&Ftn = |M_{RRTRK} - M_{RRGS}|
\end{aligned}
\tag{3.1}
$$

3.2.2 Generation Driver

When invoked for the first time, the generation driver will instantiate the chromosome template CT Pop times to create the population P_{init}. As we can see we have 2 parts to our chromosome, the ownship and the traffic portion. For each of these we have one TCP for the origin, one for the destination and 9 that define the path between the origin and destination. The genes consist of the primitive elements of each TCP, longitude, latitude, altitude and time. We describe the fitness function (Ftn) later. The genes in the chromosome produced by the instantiation will be populated with values according to C_{init}. We only show one constraint for now. The origin cannot equal the destination. Other constraints such as the input space of possible latitudes and longitudes will be included later in Chapter 4. Then, and for every subsequent invocation, the generation driver will take a set of fitness values F_i for all chromosomes in the current generation G_i, and using Op and C_{perm}, it will evolve the chromosomes in that population to produce G_{i+1}.

[1] A full problem model is provided in Chapter 4.

Chromosome	G_i	Fitness	G_{i+1}
1	[1,3,4,7,2,10]	10	[1,3,4,7,2,10]
2	[1,6,6,8,4,10]	8	[1,**7**,6,3,4,10]
3	[1,8,4,1,7,10]	7	[1,3,4,**3**,4,10]
4	[1,9,6,8,2,10]	2	[1,6,6,**6**,2,10]

Table 3.2: Sample Evolution in the Generation Driver

If the elitism operator is being used, the *n-Best* most fit chromosomes in the population are selected and added to the next generation G_{i+1}. Next, the selection operator will begin choosing among the remaining chromosomes until G_{i+1} contains a total of $Pop/2$ chromosomes. The crossover operator will then conduct pair-wise mating of the selected chromosomes until G_{i+1} contains a total of Pop chromosomes. The final step is the application of the mutation operator. By default, any chromosomes selected by the elitism operator are spared from mutation.

To illustrate this process consider the four chromosomes in Table 3.2 representing a simplified flight path consisting of just 6 latitudes. The chromosomes in generation G_i with their respective fitness values are shown in the second column, and the next generation G_{i+1} is shown in the last column. Using the elitism operator, the maximally fit chromosome in G_i, the one with a fitness of 10, is placed in G_{i+1}. Using rank selection, the next most fit chromosome in G_i, the one with a fitness of 8, is kept for further evolving the chromosome pool. The two least fit chromosomes in G are discarded from the population. Next, 1-point crossover mates the two retained chromosomes to create two new child chromosomes. In this example, the crossover point is chosen to be in the middle of the chromosome so the resulting chromosomes retain half of each source chromosome. Finally, mutation changes a single randomly selected gene per chromosome - shown in bold - and then the chromosomes are placed into G_{i+1}. Throughout this process, C_{perm} were enforced so that the starting and ending location of the flights paths was retained.

3.2.3 Execution Driver

The execution driver prepares, manages, and measures the execution of the target implementations. It takes a chromosome as input, it runs the implementations on that input, and it translates their behavior into metrics (the *measure* function from Section 3.1).

This component contains three subcomponents, the approach interface and the two target implementations, all of which must be provided by the user of UDivE. The approach interface takes as input a set of genes from a chromosome and builds the input in the format required by the target implementations. The approach interface then collects the metric values produced by the target implementations and does any required processing before producing the metric values as output. Depending on the metric needed, the value could be the result produced by the target implementations, an expression over those results, or some other type of value that represents the target system's behavior (perhaps produced by instrumenting the systems).

This component can operate serially, processing the genes of one chromosome at a time, or multiple execution drivers can be launched in parallel in order to process an entire generation at once. Because we had the available computational resources, we leveraged parallel execution when conducting the case studies presented in Chapter 4.

In the context of the collision detection and resolution implementations, the approach interface receives as input two flight paths (encoded in the genes of a chromosome) and then calls the target implementations with these paths. The output of the implementations consists of adjusted flight paths, which are then translated by this interface into extra distance travelled by subtracting the distance travelled in the adjusted flight path versus the original flight path.

3.2.4 Divergent Behavior Checker

The divergent behavior checker instantiates the *diverge* function from Section 3.1 to check the developer's hypothesis about divergence. It determines whether the metrics produced by the implementations under any of the chromosomes of a generation are different enough that their behavior can be considered divergent. If the behavior is considered divergent, then the process is stopped. Otherwise, the exploration continues by passing the metrics to the fitness computation component. By default, this component simply checks for metric equality.

For the collision detection and resolution implementations, the check function consists of a single predicate that evaluates to true when the distance travelled by the adjusted flight paths produced by the implementations was greater than 5%. This happens after 116 iterations in our scenario at which point the exploration is stopped.

3.2.5 Fitness Computation

The fitness computation routine is responsible for computing a fitness value for each chromosome in a generation. It takes as input a collection of metrics for the current generation G_i and applies the fitness function Ftn. It then produces as output the fitness values for each chromosome in G_i. The computation of fitness for our collision detection and resolution consists of the absolute value of the difference between the distance metrics of the implementations of RRGS and RRTRK (the actual fitness for this problem is slightly more complex; this will be discussed in Chapter 4). As illustrated in Table 3.2, the fitness value assigned by this component to each chromosome will determine how the population of flights is evolved.

Chapter 4

Case Studies

We perform three studies to evaluate the feasibility of UDivE. We have several goals when setting up these studies. First, we want to generate artifacts that present divergent behavior that would be hard to identify through the use of a random approach. Second, we want to study a range of problems with known, partially known and unknown divergent behavior. This allows us to validate our framework (on the known and partially known problems), but it also has potential to generate some interesting results (on the partially known and unknown problems). Finally, we want to select three very different domains to show the breadth of applicability.

These criteria led to the selection of the following three problems. Our first case study, polynomial root finding, has a known (but possibly hard to find) divergence so we selected it to validate whether our approach would find it. The second study is based on a question we found on Stack Overflow regarding image scaling algorithms. It is a domain for which we had little intuition. Even though some general characteristics of the underlying algorithms are known, we were unsure if we would find divergent behavior. The last study returns to our motivating example and looks for divergence under the more complex scenario presented by aircraft collision detection

and resolution. For this problem we lack documentation to know if there are any possible divergences and/or if our hypothesis should hold.

The studies aim to explore three questions about UDivE:

RQ1: Can UDivE identify divergent behavior?

RQ2: What is the search space UDivE explores?

RQ3: What is the cost of running UDivE?

The definition of divergent behavior varies so it is explained in each study separately. The metric to assess the complexity of the problem space explored by UDivE is measured by the size of the genetic algorithm's search space. Cost refers to the amount of time UDivE requires to execute.

4.1 Setup

We use UDivE as illustrated in Figure 3.1. Each study has a distinct chromosome template, constraints and fitness function to match the problem domain, but most of the evolutionary operators used are similar across the studies. The $2 - Best$ Elitism and *Rank* evolutionary operators were used for all studies. The population size was selected by running short experiments with various sizes between 12 and 100, and choosing the smallest size at which maximal gains were observed. In other words, a population size larger than the chosen size did not appear to produce better results. These values were 52 for the first two studies and 20 for the ACCoRD study. Further details for each study are given in the next sections.

We ran our studies using a parallel cluster with AMD 6128 2GHz, Quad-Processors (8-Core) and 128GB RAM on each core. Each chromosome in a population ran on its own node during an iteration. We used either 108 nodes or 40 nodes (depending on which size population was being run). The UDivE framework has 1138 source

lines of code (SLOC). This does not include the approach interface which is supplied separately for each problem. The approach interfaces range from 248 SLOC for the polynomial root finding approach interface to 409 SLOC for the image scaling approach interface. These figures do not include any libraries on which UDivE or the approaches interfaces relied upon.

We now present each of our studies in order. We begin with a problem description on the divergent behavior definition, and follow with the problem definition and the results of each of the three research questions.

4.2 Univariate Polynomial Root Finding

Several algorithms and implementations exist that compute the root values of a univariate polynomial function within a given `x-axis` range $[x_1, x_2]$. We begin with this problem because it is relatively simple to encode and understand, and there is a known oracle (albeit one that would not be obvious without knowledge of the mathematical domain). We consider two implementations using different algorithms: the secant method and the Ridders' method [28]. Given a polynomial function, each implementation is designed to compute the number of roots n within a given `x-axis` range $[x_1, x_2]$, along with the location (x_i, y_i) of each root for every $1 \leq i \leq n$. The implementations of the root finding algorithms used for this study were taken from Numerical Recipes, 3rd Edition [28]. The implementations were used with their default parameters.

4.2.1 Divergent Behavior

Both the secant method and Ridders' method are designed to compute the roots of a polynomial function within a given `x-axis` range. A small variation may exist in the

values of roots reported by the algorithms, but this variation should be negligible when viewing a graph of the function. However, there is a known problem, documented with the secant method as stated in [28], p. 449. In the case when functions are "not sufficiently continuous", the algorithm may not converge causing what has been unofficially termed a *secant explosion*. We therefore define our divergent behavior as a difference in the roots returned. We chose our hypothesis as a 15% deviation since we believe this would be large enough to see a difference when viewing a graph of the function. We define the null hypothesis as:

Null Hypothesis: Given a univariate polynomial, the secant and Ridders' method will report values that are within 15% of one another on the x-axis. In the event UDivE identifies a polynomial that causes the root finding algorithms to report roots that differ by more than 15% from one another on the `x-axis`, our null hypothesis will be invalidated.

4.2.2 Problem Model

For this study, each chromosome is encoded as a six-term univariate polynomial. A 6-term polynomial was chosen because a wide range of polynomials can be produced with six terms. The problem model PM_{root} for this study is defined as

$$\begin{aligned}
&CT = term = [6]\{coefficient, exponent\}\\
&Pop = 52\\
&C_{init} = \{-15 \leq coefficient \leq 15 \wedge\ 0 \leq exponent \leq 15\}\\
&C_{perm} = \{-15 \leq coefficient \leq 15 \wedge\ 0 \leq exponent \leq 15\}\\
&Op = \{\text{2-Best, Rank, One-Point, Creep}[0..6], MutRate = 0.10\}\\
&Ftn = |M_{secant} - M_{Ridders}|
\end{aligned} \tag{4.1}$$

In our encoding each of the 6 terms of the polynomial is a pair containing the coefficient and exponent value. For simplicity, and to prevent polynomials that are

difficult to compute from being introduced into the study (e.g. those with exceedingly large exponent values), coefficient values were limited to integers in the range $[-15, 15]$ and exponent values were limited to integers in the range $[0, 15]$. We begin with initial chromosomes that are randomly generated within this range. We use one-point crossover and creep mutation. We use a creep range of [0,6] and a mutation rate of 0.10. Creep mutation adds or subtracts with a 50% probability, a value chosen randomly from the given range to 10% of the genes chosen randomly. The permanent constraints prevent mutation from moving outside of the allowed ranges of values. We use modular math to stay within the allowed value ranges.

Because we are searching for reported roots with different respective values, the metric value returned by each of the algorithms is the sum of its roots. No instrumentation of the root finding algorithms is required to compute the sum of reported roots. The approach interface handles this computation by consuming the respective implementations' output. Formally, the metric value M for each root finding algorithm is defined as $M = \sum_{i=1}^{n} x_i$, where n is the number of roots and x_i is the value of the ith root. The fitness function Ftn computes the absolute value difference of the metric values M_{secant} and $M_{Ridders'}$.

4.2.3 Results

RQ1. UDivE was able to identify a univariate polynomial that invalidated the null hypothesis within 13 generations. The identified polynomial, $(-2x^{12} - 6x^{15} + 10x^{11} + x^{10} + x^{15} - x^{10})$, contains three reported roots, the second of which was identified by the secant method as $(-9999,6 \times 10^{60})$ and identified by the Ridders' method as $(-0.09, 0)$. This represents an x-axis value percent difference well above our requirement of 15%. A graph of the polynomial that invalidates the hypothesis, along

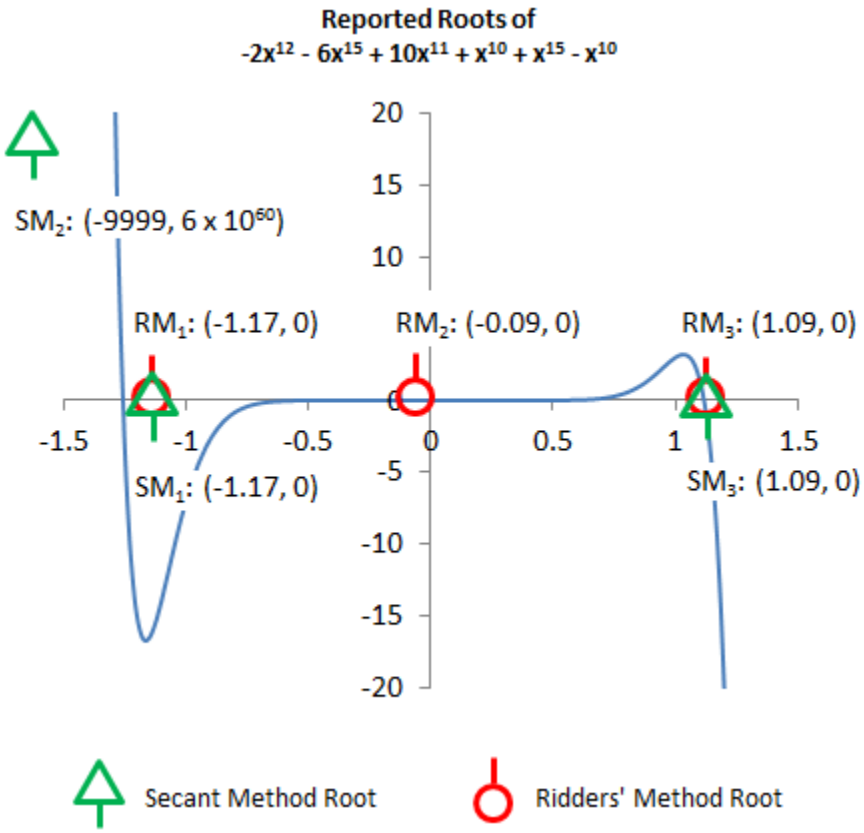

Figure 4.1: Input polynomial leading to divergent behavior between Secant's and Ridder's Methods for root finding

with the root values reported by the respective algorithms, is shown in Figure 4.1. In this figure we show the roots for the secant method with triangles and Ridder's with circles. As shown in the Figure, the first and third roots found by the secant method and Ridders' method are the same, however the second root reported by the secant method is significantly different than the (more accurate) root reported by Ridders' method.

RQ2. The size of the search space for this study is defined as the total number of univariate polynomials that can be encoded using the given problem model when satisfying the constraints in the set C_{perm}. This equates to a search space size of 1.489×10^{16} polynomials.

Trial	Hypothesis Invalidated?	Maximum % Difference	Generations
1	Yes	SE	13
2	Yes	SE	101
3	Yes	SE	2
4	Yes	SE	50
5	No	1.1%	500
6	No	1.0%	500
7	Yes	SE	28
8	Yes	SE	115
9	No	4.6%	500
10	Yes	SE	10

Table 4.1: Root Finding Repeatability Trials

RQ3. Each generation required an average of 4.57 seconds of CPU time per node to execute. We used 108 nodes per generation. Our hypothesis was invalidated in 13 generations, which required approximately 6,416 seconds or 1.8 machine hours.

4.2.4 Repeatability

We are interested in determining if UDivE is capable of repeatedly discovering a polynomial that invalidates our hypothesis. Therefore, we repeat the trial 10 times with the goal of invalidating the hypothesis defined in Section 4.2.1.

UDivE discovered a polynomial that invalidates the hypothesis in 7 out of 10 trials. Table 4.1 shows the results of each trial, the maximum percent difference discovered during the trial, and the number of generations required to invalidate the hypothesis. In the event a "secant explosion" is discovered (i.e. a polynomial that prevents the secant method from converging), "SE" is listed. If, after 500 generations, the hypothesis has not been invalidated, execution is stopped and the result is reported.

In the event the hypothesis was invalidated, it was observed to occur within the first 115 generations. Based on the repeatability data presented in Table 4.1, if the hypothesis was not invalidated within the first 115 generations, it would not be

Trial	Hypothesis Invalidated?	Maximum % Difference
1	No	7.4%
2	No	1.9%
3	Yes	SE
4	No	2.6%
5	No	0.00%
6	No	10.0%
7	No	6.0%
8	No	1.6%
9	No	2.1%
10	No	4.0%

Table 4.2: Root Finding Random Trials

invalidated before the trial was stopped at 500 generations. Further, the maximum percent difference achieved for these trials (5, 6, and 9) is well below the threshold of 15%, as required by the hypothesis.

4.2.5 Randomness

We are interested in determining if randomly generated polynomials are capable of invalidating the hypothesis defined in Section 4.2.1. Therefore, we run 10 trials with the goal of evaluating how randomly generated polynomials compare to those discovered by UDivE.

To remain consistent with the repeatability trials executed in Section 4.2.4, 500 iterations were executed and the maximum percent difference discovered is reported. Table 4.2 shows the results of each trial and the maximum percent difference discovered. Randomly generated polynomials were unable to invalidate the hypothesis in 9 out of 10 trials. In the event a "secant explosion" is discovered (i.e. a polynomial that prevents the secant method from converging), "SE" is listed.

	Polynomial
1	$-3x^{14} - 1x^{13} + 3x^{15}$
2	$-3x^{14} + 3x^{15} - 1x^{13}$
3	$-3x^{15} + 1x^{13} + 3x^{14}$
4	$-3x^{15} + 3x^{14} + 1x^{13}$
5	$-1x^{13} - 3x^{14} + 3x^{15}$
6	$-1x^{13} + 3x^{15} - 3x^{14}$
7	$1x^{13} - 3x^{15} + 3x^{14}$
8	$1x^{13} + 3x^{14} - 3x^{15}$
9	$3x^{14} - 3x^{15} + 1x^{13}$
10	$3x^{14} + 1x^{13} - 3x^{15}$
11	$3x^{15} - 3x^{14} - 1x^{13}$
12	$3x^{15} - 1x^{13} - 3x^{14}$

Table 4.3: Polynomials in Restricted Search Space

As shown in Table 4.2, the randomly generated polynomials that did not invalidate the hypothesis produced a maximum percent difference between 0.0% and 10.0%, shy of the 15% required by the hypothesis.

4.2.6 Search Space Enumeration

We are interested in understanding the landscape of the search space for this case study. However, because the search space for the given problem model (Section 4.2.2) is too large to effectively enumerate, we enumerate the search space of a more restricted polynomial. This allows us to understand the landscape of a similar search space, without requiring an intractable enumeration.

The restricted search space we consider is composed of three-term univariate polynomials with coefficients in the range $[-3..3]$ and exponents in the range $[0..15]$, with a size of 1404928 polynomials. This search space contains 12 polynomials that invalidated the hypothesis defined in Section 4.2.2. These polynomials are listed in Table 4.3. For this restricted search space, each of the polynomials that invalidate

the hypothesis have a coefficient of either -1, 1, -3, or 3 and an exponent in the range [13..15].

Only $8.57 \times 10^{-6}\%$ of the polynomials in the restricted search space invalidate the hypothesis, which shows how difficult it is to find such divergent behavior, and those polynomials appear to be clustered around the higher level, i.e. polynomials with large exponent values.

4.3 Image Scaling Algorithms

For our second study we wanted to identify a problem instance raised by developers that did not seem to have a clear answer, and that is domain specific. We turned to Stack Overflow to identify such problem, performing a search with the query *difference between algorithms*. The fourth question in the list met our requirements [1] and stated: "Could anyone explain to me the difference between these scaling algorithms? i.e. Which ones are better for upscaling or downscaling, which are better for photos and which are better for 2-bit images, and the relative speed of each, etc..."

We selected the two most common algorithms (bilinear and bicubic) from among the five listed and focused on the downscaling problem. We used open source implementations of these algorithms written in Java from the OpenCV library [4]. After some additional searches for information on the differences between these two algorithms we found the following posted statement: "For certain resizing values, e.g. (depending on software) 25% 33% 50% 67% 75% and 200%, bilinear and bicubic produce identical results " [2]. From this we set up our divergent behavior hypothesis.

[1] `http://stackoverflow.com/questions/8322788/whats-the-difference-between-these-scaling-algorithms`

[2] `http://photo.net/digital-darkroom-forum/00BjsZ`

Note that we are not arguing that these claims by the posters are the ground truth, but rather just using this as a hypothesis a developer may have.

4.3.1 Divergent Behavior

We define divergent behavior in terms of pixel differences between downscaled images, and the null hypothesis as:

Null Hypothesis: Given an image, the bilinear and bicubic image scaling algorithms will create scaled down images at 25% scaling that are identical. If UDivE identifies an image that causes the two implementations to generate different images, our null hypothesis will be invalidated. We selected a difference of 2.5% when performing a pair-wise pixel comparison. This percentage represents at least an accumulated difference of 1000 in a target image of 200×200 pixels.

4.3.2 Problem Model

We set up a 200×200 pixel image. Lacking a benchmark or much intuition to serve as a starting point, we decided to define the problem model in terms of shapes with different locations, sizes, and colors that may appear in an image. More specifically, we populated the image randomly with 9 shapes. The domain of shapes used included circles, squares and triangles, which could each be a solid color, an outline of the shape (unfilled), or a blank shape (where the entire shape including the border is set to be all white). Each shape can appear anywhere on the canvas (if it runs off of the canvas it is truncated) and can be any size within the 200×200 pixel space. We chose this configuration since we are unfamiliar with the image domain and wanted to use images that are discrete and finite, while still allowing for a wide range of images to be produced.

The problem model PM_{image} for this study is defined as

$$
\begin{aligned}
&CT = shape = [9]\{x, y, width, height, type, intensity_1, intensity_2, intensity_3\} \\
&Pop = 52 \\
&C_{init} = type = \{solidsquare|solidtriangle|solidcircle|opensquare| \\
&opentriangle|opencircle|blanksquare|blanktriangle|blankcircle\}, \\
&x = [0..134], y = [0..134], width = 66, height = 66, intensity_i = \{0|85|170\}\} \\
&C_{perm} = type = \{solidsquare|solidtriangle|solidcircle|opensquare| \\
&opentriangle|opencircle|blanksquare|blanktriangle|blankcircle\}, \\
&x = [0..200], y = [0..200], width = [20..120], height = [20..120], intensity_i = \{0|85|170\}\} \\
&Op = \{\text{2-Best,Rank,Two-Point,}Swap|FullRange_{x,y}|FullRange_{width,height}| \\
&FullRange_{inensity[0|1|2|}|FullRange_{type}, MutRate = 0.10\} \\
&Ftn = |M_{Bilinear} - M_{Bicubic}|
\end{aligned}
\tag{4.2}
$$

Each shape has an `x` and `y` value denoting the upper left location, a width, height, a type from within the set of 9 shapes described above, and three intensity values. The set of operators Op contains the 2-Best elitism operator, rank selection and two point crossover. For mutation we randomly select 10% of the population's image locations and then randomly select from among one of 5 mutation operators. This includes a swap (randomly select a second location and switch images), a full range mutation on the `x` and `y` values (randomly select a new value for each of `x` and `y`), a full range mutation on width and height, a full range mutation on one (randomly selected) intensity value, or a full range mutation on the shape type. We did this to allow a wide range of diversity to appear in our images.

The metric value M for this problem is defined as $M = \sum_{i=1}^{n} x_i$, where n is the number of pixels and x_i is the value of the i^{th} pixel.

The fitness function Ftn computes the absolute value difference of the metric values M_{secant} and $M_{Ridders'}$.

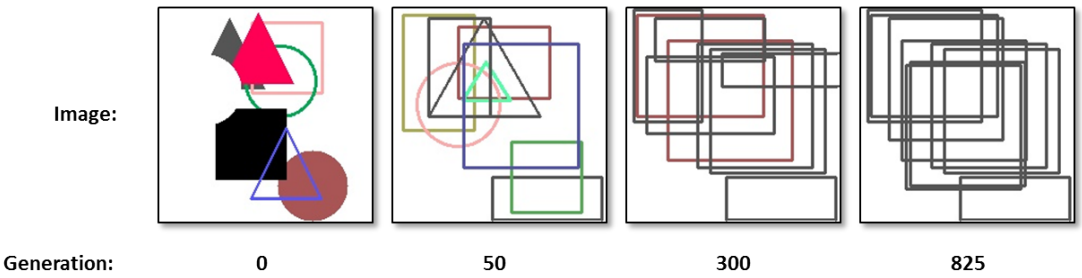

Figure 4.2: Inputs evolved by UDivE for the Bilinear and Bicubic image scaling implementations. After 825 generations, the input image lead to divergent behavior greater than the 2.5% target.

4.3.3 Results

RQ1. We found an image after 825 generations that exceeds the 2.5% difference. We show the fittest input picture in each generation in Figure 4.2 after 0, 50, 325, and 825 iterations. It is interesting to note how, as more iterations are performed, the input image leading to divergent behavior consists of a more uniform set of shapes (rectangles) of gray color with a lot of juxtaposition. We conjecture that the bilinear algorithm implementation struggles to maintain the definition of images with lines and angles in such proximity when downscaling.

Without domain knowledge and a clear oracle we ran a few additional studies. First, we repeated this study and each time we saw a similar input (all open rectangles that were grey-scale). We then removed the rectangle image from our image set. This attempt converged on an image packed with open triangles (although the percentage difference did not exceed the 2.5% threshold).

RQ2. There are 200 possible values for `x`, `y`, `width`, and `height`. There are 9 possible image types, 3 channels of color and 9 shapes in total. This means our search space is is $(200^4 \times 9 \times \times 3^3)^9 = 2.03 \times 10^{104}$.

Trial	Hypothesis Invalidated?	Maximum % Difference	Generations
1	Yes	2.6%	825
2	Yes	2.57%	194
3	Yes	2.5%	725
4	Yes	2.51%	565
5	Yes	2.52%	1072
6	No	2.48%	2000
7	Yes	2.53%	532
8	Yes	2.52%	1016
9	Yes	2.5%	1262
10	Yes	2.5%	1992

Table 4.4: Image Scaling Algorithms Repeatability Trials

RQ3. Each generation took an average of 3.34 seconds per node to execute. We used 108 nodes for 825 generations for a total of 297,594 seconds or 82.7 hours (or 3.44 days) of machine time.

4.3.4 Repeatability

We are interested in determining if UDivE is capable of repeatedly discovering an image that invalidates our hypothesis. Therefore, we repeat the trial 10 times with the goal of invalidating the hypothesis defined in Section 4.3.1.

UDivE discovered an image that invalidates the hypothesis in 9 out of 10 trials. Table 4.4 shows the results of each trial, the maximum percent difference discovered during the trial, and the number of generations required to invalidate the hypothesis. If, after 2000 generations, the hypothesis has not been invalidated, execution is stopped and the result is reported.

The only trial that was unable to invalidate the hypothesis was Trial 6, shown in Table 4.4. However, the maximum percent difference it was able to discover was 2.48%, which is only slightly lower than the 2.5% required by the hypothesis. The other trials were able to invalidate the hypothesis in 194 to 1992 generations.

Trial	Hypothesis Invalidated?	Maximum % Difference
1	No	0.71%
2	No	0.58%
3	No	0.54%
4	No	0.57%
5	No	0.56%
6	No	0.53%
7	No	0.52%
8	No	0.78%
9	No	0.58%
10	No	0.60%

Table 4.5: Image Scaling Algorithms Random Trials

An interesting aspect of the maximally fit image discovered by UDivE reported in Section 4.3.3 is that it is composed entirely of grey outlined rectangles with a lot of juxtaposition. We are interested in determining if the maximally fit images discovered during the repeatability trials share this trait. In fact, each of the maximally fit images in the 10 trials share this trait to a high degree. For example, the maximally fit images produced by trials 1, 4, 6, 8, 9, and 10 are composed entirely of grey outlined rectangles. Other images are composed entirely of grey outlined rectangles, with the exception of one outlined rectangle that is either green or blue, for example the maximally fit images produced by trials 2 and 7. The image that deviated the most from the trait was that of trial 3, which was composed of all grey outlined rectangles, with the exception of one blue outlined triangle.

4.3.5 Randomness

We are interested in determining if randomly generated images are capable of invalidating the hypothesis defined in Section 4.3.1. Therefore, we run 10 trials with the goal of evaluating how randomly generated images compare to those discovered by UDivE.

To remain consistent with the repeatability trials executed in Section 4.3.4, 2000 iterations were executed and the maximum percent difference discovered is reported. Table 4.5 shows the results of each trial and the maximum percent difference discovered.

Randomly generated images were unable to invalidate the hypothesis in each of the 10 trials. In fact, the maximum percent difference discovered during all of the trials presented in Table 4.5 is 0.71%, which is less than a third of the way to the 2.5% difference required by the hypothesis.

4.4 Aircraft Collision Detection & Resolution

For our last study we return to the motivating problem introduced in Chapter 2, trying to identify divergent behavior but now with a problem scenario that has a larger impact. We set up flight plans using real-world locations. We restrict the ownship flight plan to originate at the Baltimore Washington International Airport and terminate at the LaGuardia Airport (184 miles apart). We restrict the traffic ship flight plan to originate at the Harrisburg International Airport and terminate at the Philadelphia International Airport (83 miles apart). These two flight plans cross one another, providing an opportunity for a conflict.

As described in Chapter 2, we use unmodified third-party implementations of the CD&R algorithms RRGS and RRTRK from the Airborne Coordinated Conflict Resolution and Detection (ACCoRD) Framework developed by NASA [23].

4.4.1 Divergent Behavior

Because RRGS and RRTRK perform strategic conflict resolution, it can be assumed that they will attempt to resolve a given conflict in such a way that the distance the

ownship aircraft deviates from its original flight plan is minimized. We expect some variation due to the different approaches taken by CD&R algorithms, but want to find out if there are situations where these diverge significantly. Therefore, we would like to identify flight path resolutions that have a percent difference in distance of more than 15%. Consider a hypothetical example that involves two resolution flight paths. Assume the first resolution flight path is the same distance as the original ownship flight path, 184 miles, and assume the second resolution flight path is 214 miles. This is a difference in distance of 30 miles, and a percent difference in distance of approximately 15%. We consider this type of difference to be significant, especially because RRGS and RRTRK perform strategic conflict resolution.

For this problem we do not have guidance on how large of a difference in resolution would be considered significant in this domain, therefore we simply selected a value that we deemed as reasonable; in practice this threshold would be determined by a domain expert.

We define our hypothesis as:

Null Hypothesis: Given a flight plan for both an ownship and traffic aircraft that contains at least one conflict, the implementations of RRGS and RRTRK should produce resolutions that do not have a percent difference in distance of more than 15%.

4.4.2 Problem Model

Each chromosome encodes a flight plan for both the ownship aircraft and the traffic aircraft. There are a total of 11 TCPs in each flight plan. This value was chosen empirically to be large enough to allow sufficient variation in the flight plan but small enough so the flight plans would not contain so many TCPs that they were at risk

of becoming congested and mangled during evolution. Each TCP is composed of a 4-tuple containing a value for latitude, longitude, altitude, and time.

The problem model $PM_{collision}$ for this study is defined as

$$
\begin{aligned}
&CT = flightpath = [2]\{ownship, traffic\}, ownship = \{TCP_{origin}, TCP_{path}[0..8], TCPdest\},\\
&traffic = \{TCP_{origin}, TCP_{path}[0..8], TCPdest\}\\
&TCP = \{longitude, lattitude, altitude, time\}\\
&Pop = 20\\
&C_{init} = \{TCP_{origin} \neq TCP_{dest}, orgin \leq TCP_{path} \leq dest \wedge onStraightLine(TPC)\\
&longitude = [39N..41N], lattitude = [73E..79E], altitude = [23000],\\
&ACCoRD.check(ownship),\ ACCoRD.check(traffic)\}\\
&C_{perm} = \{TCP_{origin} \neq TCP_{dest} \wedge immutable(TCP_{origin}) \wedge immutable(TCP_{dest}),\\
&longitude = [39N..41N], lattitude = [73E..79E],\\
&ACCoRD.check(ownship),\ ACCoRD.check(traffic)\}\\
&Op = \{\text{2-Best, Rank, One-point}, Creep_{lat} = [0..0.1] | Creep_{long} = [0..0.1] | Creep_{alt} = [0..0.2] |\\
&Creep_{time} = [0..5.0], MutRate = 0.1\}\\
&Ftn = \begin{cases} if\ Penalty, & \text{Penalty} \\ else & |M_{RRTRK} - M_{RRGS}| \end{cases}
\end{aligned}
\tag{4.3}
$$

The constraints on this study are more complex than our other two studies presented previously. First the origins and destinations cannot be the same. Next, we check that all of the TCPs are on a straight line. At initialization we fixed the altitude to 23,000 but later this is allowed to change. Finally, we did checks on the ownship and traffic TCPs using the ACCoRD framework to ensure that our models were valid. These constraints are required for both initialization and as permanent constraints, although in practice we found that the small size of creep during mutation, did not require us to enforce any other than mutability; they were never violated.

The set of evolutionary operators Op contains the 2-Best elitism operator, rank selection operator, one point crossover, and creep mutation operator. For latitude

values, we use a creep range of [0, 0.1], for longitude values [0, 0.2], altitude values [0, 1000], and time values [0, 5]. We use a mutation rate of 0.10.

In the event that the algorithm resolves the conflict it will return a modified conflict-free ownship flight plan. The metric value is defined as the absolute value of the difference in miles between the original conflicted ownship flight plan and the new conflict-free ownship flight plan. Formally, the metric value M for each CD&R algorithm is defined as

$$M = |distance(ownship_{orig}) - distance(ownship_{new})|.$$

In order to accurately calculate flight plan distances, we apply the result of applying the haversine formula [5] to each pair of TCPs on the flight plan and sum them up. To ensure that we do not evolve away from conflicted paths or end up without a resolution for both algorithms (we are only interested in divergence in this study of distance between the resolved paths) we employ penalties for this situation. If no conflict occurs between the ownship and traffic flight path, a value of -200 is assigned to the metric. If on the other hand if one of the algorithms does not return a resolution, its metric is assigned a value of -100. These values will be used as penalties in the fitness calculation. The fitness function Ftn first checks for the existence of any penalty. If this exists it uses that value. If not, it computes the absolute value difference of the metric values M_{RRTRK} and M_{RRGS}.

4.4.3 Results

RQ1. UDivE was able to identify an ownship and traffic ship flight plan that invalidated our null hypothesis in 10 iterations. Figure 4.3 summarizes the results. The left most graph shows the flight plans and the conflict that UDivE found that results in

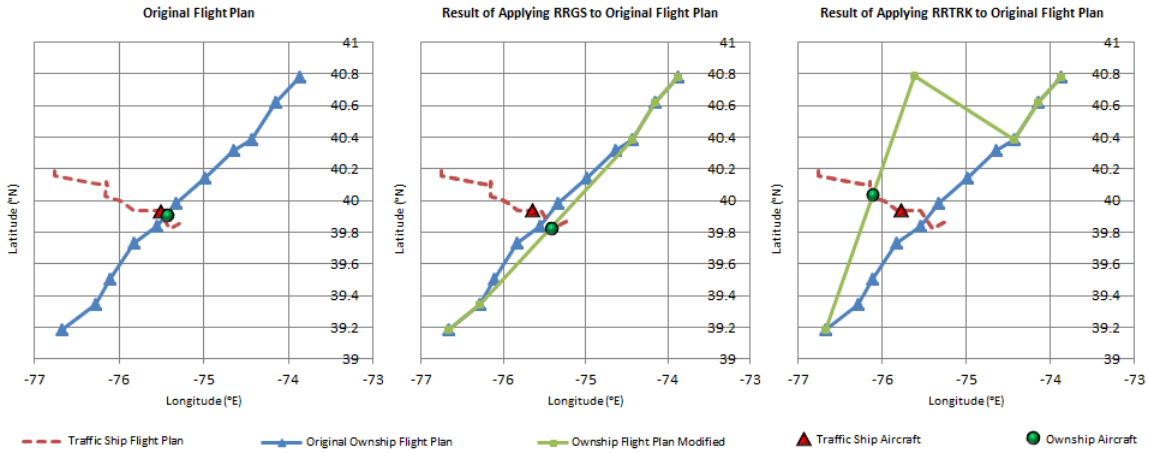

Figure 4.3: The figure on the left shows the ownship (circle) and its flight path (solid line with triangles) and the trafficship (triangle) and its flight path (dotted line) evolved by UDivE. These flights path show a collision and lead to divergent behavior in RRGS and RRTRK. The middle and right figures incorporate the adjusted flight paths (solid lines without triangles) generated by RRGS and RRTRK to resolve the collision. The difference from the original flight paths proposed by the RRTRK and RRGS show a percent difference of 16.8%.

divergent behavior. The flight path identified by UDivE for the ownship is 187 miles. The next two graphs show the resolution paths that avoid the conflict at different costs. The original ownship flight plan is shown in each of these graphs, with the respective resolution flight paths shown as an overlay. RRGS resolved the conflict with a flight path of 185 miles, while RRTRK created a resolution flight path of 219 miles. The difference is 34 miles which represents a percent difference of 16.8%.

RQ2. In the context of this study, the size of the search space was bounded by the latitude/longitude points (41N, -79E), (41N, -73E), (38N, -79E), and (38N, -73E). This is the area covering a portion of the East coast of the United States. In addition, based on the distances between the airports appearing in this space, and the range of valid speeds of common commercial aircraft, the maximum time value expected in this study was approximately 3600 seconds. The altitude of the aircraft in this

study is bounded at 40000 feet. Finally, the amount of precision considered for each of these non-integer numerical values is 4 decimal places.

Therefore, for a given TCP, each latitude value can take one of 30000 values. Each longitude value can take one of 60000 values. Each time value can take one of 3.6×10^7 values, and each altitude value can take one of 4.0×10^8 values. There are a total of $30000 \times 60000 \times (3.6 \times 10^7) \times (4.0 \times 10^8) = 2.59 \times 10^{25}$ possible values for each TCP. Because there are 9 TCPs per chromosome that are mutable (the origin and destination TCPs are immutable), each chromosome could hold one of $(2.59 \times 10^{25})^9 = 5.24 \times 10^{228}$ values. This means there is a search space size of 5.24×10^{228} flight paths.

RQ2. Each generation required an average of 5.8 seconds per node to execute. We used 40 nodes per generation and we ran 10 generations for a total of 2,320 seconds or 39 minutes.

4.5 Summary of Results

We proposed three research questions in this study. We summarize the results here.

RQ1: Can UDivE identify divergent behavior? In all three scenarios UDivE was able to find inputs that violated the null hypothesis. We therefore answer this research question in the affirmative.

RQ: What is the search space UDivE explores? All three of our problems have large search spaces. The smallest space, that of the image processing implementations was in the order of 10^{11}. Our largest search space was for the collision avoidance algorithms, in the order of 10^{26}.

RQ2: What is the cost of running UDivE? The sequential running times of each of these case studies varied from between 39 minutes (conflict avoidance), to 3.44 days (image processing). In practice we ran UDivE in parallel with one node per chromosome meaning that the clock runtime was actually a fraction of this cost (45.9 minutes for image processing).

Discussion. We believe that the case studies show that UDivE has found divergent behavior of practical significance. In the first study, we examined a problem with a known divergence. But the divergence itself is not easy to predict without knowledge of how particular polynomials will behave ahead of time. For the image processing application, the discussions on Stack Overflow and other discussion groups tell us that not only the question we studied is of interest, but that the divergence is not easy to find and the answer is not common knowledge. In our last case study, the results seem compelling– they reveal undocumented behavior.

If we examine the relationship between RQ2 and RQ3 there does not seem to be a strong correlation between the search space and run times. In fact our largest search space ended up having the shortest run time. We believe that the run time is a factor

of both the constraints on the system and the number of, or distribution of, divergent solutions in the given search space. We plan to explore this more in future work. Understanding the landscape of the search space, and how system constraints effect that landscape will be key in discovering these types of correlations, if they exist. See Chapter 6 for a more detailed discussion of future work.

Chapter 5

Extending UDivE to Cyber-Physical Systems

So far, the case studies we have discussed have considered target systems that exist purely in the software domain (in the case of the root finding and image scaling studies), or have application in the physical domain, but the results of the study have not been extended beyond an algorithmic representation of the problem (in the case of the aircraft collision detection and resolution study).

Here, we consider a feasibility study involving a cyber-physical system that can be simulated using software, and the results of which can be verified in the physical domain. This is valuable because it allows us to consider the extensions to UDivE that are required for such a study to take place.

The Need for Simulation Many cyber-physical systems require a number of items to operate successfully. These items may include a "world" on which to sense and actuate, and resources such as power, sensors, actuators, operating personnel, and

potential (and costly) repair time. This is problematic since UDivE requires the target systems to perform tasks repeatedly, potentially thousands (or more) of times.

Therefore, the need to simulate cyber-physical systems to combat cost becomes evident. By simulating the system, we can still conjecture about cyber-physical systems, simulating their behavior until our hypotheses have either been validated or invalidated, and then perform verification of UDivE's results on the physical system.

Extending UDivE to Interact with Cyber-Physical Systems In order to allow UDivE to accurately and successfully explore the behavior of cyber-physical systems, and by extension, simulations of those systems, a consistency check must be introduced. Once UDivE produces a result, with the aid of a simulator, the result must be checked on the physical system. Depending on the accuracy of the simulator, the noise present in the physical system itself, and the cost of physical execution to check, this process can be non-trivial. We explore such issues in this chapter.

5.1 Background

This section provides the necessary background information that will be referenced throughout Section 5.2. In addition, it provides a simple introduction to quad-rotor UAVs, as well as an explanation of common terms that are related to UAVs. Finally, it provides a simple introduction to proportional-integral-derivative (PID) controllers.

5.1.1 Quad-Rotor Unmanned Aerial Vehicles

A quad-rotor unmanned aerial vehicle (UAV) is an aircraft with four rotors that is capable of moving in three dimensions. Quad-rotor UAVs are becoming increasingly common in a variety of domains. In addition, the semi- autonomous or fully-

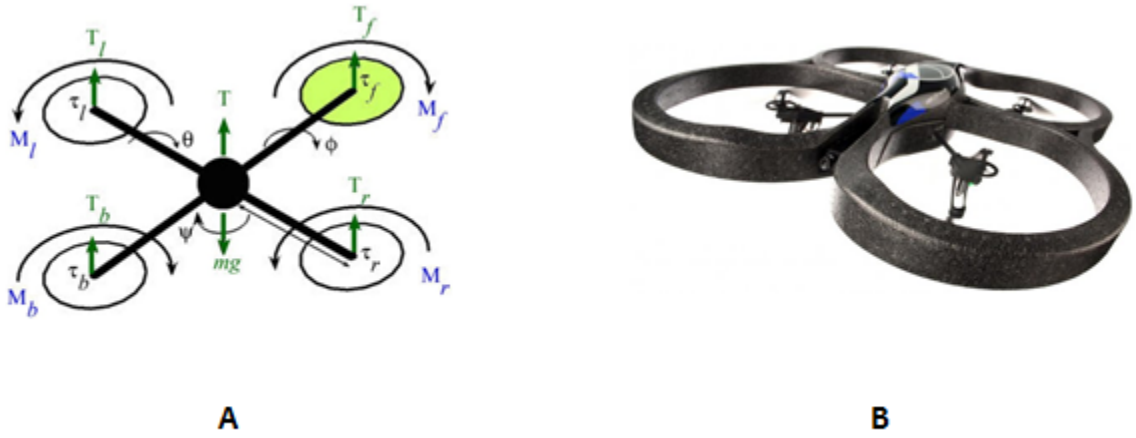

Figure 5.1: Shown is (A) a schematic view of a typical quad-rotor UAV[1]and (B) a Parrot AR.Drone[2], the quad-rotor UAV considered in this feasibility study.

autonomous use of these UAVs is an active area of research [3, 12, 21, 22, 33, 36] that has the potential to revolutionize the application of UAVs to everyday activities.

Quad-rotor UAVs are either remotely controlled by a human operator, semi-autonomous, or fully autonomous. Semi and fully autonomous UAVs leverage either, or a combination of, on-line and off-line processing. On-line processing refers to computation that takes place on-board the UAV, and off-line processing refers to commutation that takes place remotely, and the result of which is broadcast to the UAV. Off-line processing is common when resources are constrained onboard the UAV, yet it requires complex computation to navigate through its environment.

In order to navigate through its environment, a quad-rotor UAV modifies its position with four types of movements. They are termed pitch, roll, yaw, and altitude change. Pitch allows the UAV to tilt forward or backward and achieve translation in the forward or reverse direction. Roll allows the UAV to tilt to the left or right

[2]http://www.intechopen.com/source/html/6587/media/image4.jpeg

[2]http://mrl.isr.uc.pt/experimentaldata/datasets/handle/files/images/devices/Parrot_AR_Drone_09.jpg

and achieve translation in the left or right direction. Yaw allows the UAV to spin in either direction. An altitude change allows the UAV to ascend or descend. The combination of these movements allows for complex flight capability, as well as a high degree of maneuverability.

When a quad-rotor UAV is to pitch or roll, the degree to which the motion occurs is measured as an Euler angle. The further the angle from 0 radians, the more substantial the effect on the UAV. For example, pitching forward with an Euler angle of 0.2 radians will allow the UAV to travel forward with a greater speed than pitching forward with an Euler angle of 0.1 radians. Typically, a maximum Euler angle is enforced that prevents the UAV from pitching or rolling to a degree that would cause unstable flight. The UAV risks flipping over or becoming uncontrollable if this angle is too high.

One major advantage of a quad-rotor UAV when compared to a fixed-wing or single-rotor UAV is the simplicity of its design. As shown in Figure 5.1, a quad-rotor UAV accomplishes flight with four rotors, each pair of which counter-rotate (a rotor is paired with the rotor opposite its location). Simply adjusting the rotation rate of one rotor allows the UAV to pitch or roll (and achieve translation). Adjusting the rotation rate of a pair of rotors allows the UAV to yaw. Modifying the rotation rate of all four rotors at the same time modifies lift and allows the UAV to ascend or descend.

5.1.2 PID Controllers

A PID controller is used to direct a system toward, and then hold it, at a given set-point. For example, the "cruise control" feature of an automobile may make use of this type of controller. Because the automobile is incapable of maintaining a desired speed

(i.e., the set-point) without constant acceleration and brake commands, a controller is required to calculate the error between the current speed of the automobile and the desired speed, and then apply the appropriate amount of acceleration or brake until the automobile is within an acceptable threshold of the set-point (ideally exactly at the set-point). Because of noise in the environment, such as wind, hills, or uneven road surfaces, the behavior of the automobile will frequently change, therefore the controller must have a feedback loop that allows it to continue to correct for changes.

In order to accomplish this type of control, a PID controller makes use of three terms that combine to produce an output fed into the system being controlled (following the previous example, the output would be acceleration or break commands). Each of these terms relies on the current magnitude of error. The error is defined as the distance between the current state and desired the set-point. The proportional (P) term produces output proportional to the current error. If the current error is large, the P term will produce a large output, and vice-versa. The integral (I) term considers accumulated error from the past and produces output that helps eliminate the accumulated error. The derivative (D) term attempts to predict system behavior and produces an output that is designed to reduce future error. The values of these terms (also referred to as gains or coefficients) are defined during a tuning process and are dependent upon the system being controlled.

Variations of a PID controller may exist depending on the requirements of the system. For example, some systems may only require a P controller, which uses only the P term described above. Other systems may require a PD controller, which uses only the P and D terms described above. These types of controllers are referenced throughout Section 5.2 when describing the controllers used to direct quad-rotor UAVs.

5.2 Feasibility Study: Quad-Rotor UAV Configuration

We now introduce a feasibility study in which UDivE interacts with a simulator, and the simulated results produced by UDivE are checked using a cyber-physical system. The cyber-physical system we consider are quad-rotor UAVs.

As UAVs become more sophisticated, so too do they become more configurable. In fact, it is often a simple operation to reconfigure a UAV's flight capability (within a predefined range to preserve flight stability). Some high-level examples of UAV reconfigurations include UAVs operating with or without an altimeter, or a UAV navigating with one of several types of localization techniques. Two common examples of localization techniques include the use of a global positioning system (GPS) chip, or an inertial measurement unit (IMU); a device that is capable of estimating distance and direction travelled by tracking accumulated movements. Some lower-level examples of UAV reconfigurations include modification of the maximum Euler angle at which the UAV is permitted to pitch and roll, or the coefficients that the UAV uses in its control system (such as a PID controller). This feasibility study explores the effects of different configurations of a UAV, that in turn produce different types of behavior, as the UAV traverses along a flight path.

For this feasibility study, we consider two styles of UAV configuration: passive and aggressive. These two styles of UAV configuration serve as our "target systems." Informally, a UAV in an aggressive configuration will execute its flight commands abruptly and quickly, whereas a UAV in a passive configuration will execute its flight commands in a slow, gentle fashion. These configurations are described in more detail in Section 5.2.2.

The type of UAV considered in this feasibility study is the Parrot AR.Drone[3]. The Parrot UAV is a low-cost quad-rotor that is becoming increasingly common in both hobbyist and research settings. In addition, the Parrot UAV can be easily reconfigured using the freely available Parrot SDK[4]. A Parrot AR.Drone is shown in Figure 5.1.

The goal of this feasibility study is to determine if UDivE can identify divergent behavior, discuss the search space UDivE explores, calculate the cost of running UDivE, and determine the feasibility of checking UDivE's results on a cyber-physical system.

5.2.1 Setup

This section discusses the simulator chosen for this feasibility study, its differences when compared to a physical UAV, and the experimental configuration used for the study.

In order to simulate the behavior of UAVs in different configurations, we leveraged the Nimbus Simulator (NimSim). The NimSim simulator was created by us and is designed to provide basic simulation functionality of a quad-rotor UAV during flight. We created NimSim to enable the simulation of a large number of flights without the need to manually collect flight data. The physical characteristics and limitations of the UAV (e.g. mass, maximum acceleration and speed, maximum Euler angle) to be simulated can be supplied via a configuration file or updated dynamically at runtime. For a more detailed discussion of NimSim, see Appendix A.

[3] `http://ardrone2.parrot.com/usa/`
[4] `https://projects.ardrone.org/`

5.2.1.1 NimSim

There are two primary differences between the way NimSim simulates the flight of a UAV, and the way in which a physical UAV flies in its environment. We assume that all flight commands (sent both to NimSim and the physical UAV) are provided in the range $[-1.0, 1.0]$ for each of the x, y, and z directions. For simplicity, this feasibility study does not consider yaw motion.

The first difference is the way in which translation is accomplished. When configured, NimSim is supplied with a maximum Euler angle. When a translational flight command is supplied to NimSim (in either, or both, the x and y directions), the magnitude of the flight command is used to compute the percentage of the maximum Euler angle at which the simulated UAV is to pitch or roll. For example, if the maximum Euler angle of the simulated UAV is 0.52 radians, and a flight command of 0.5 in the x direction is received, the simulated UAV will pitch 0.26 radians. This allows the simulated UAV to respond to a translational flight command of any magnitude. This type of design was chosen for its simplicity.

The physical UAV is also configured with a maximum Euler angle. However, when the physical UAV receives a translational flight command, it will begin to pitch or roll until its maximum Euler angle has been reached, and will not pitch or roll any further. This means that if a translational flight command of magnitude M causes the physical UAV to pitch or roll to its maximum Euler angle, any translational flight command with a magnitude greater than M will produce the same effect (because the physical UAV will reach its maximum Euler angle and will not pitch or roll any further).

The second difference is the use of controllers. Because NimSim creates a simulated environment in which the simulated UAV will fly, some real-world disturbances and

noise can be controlled or eliminated. NimSim provides only a very basic model of real-world forces that act upon the simulated UAV, such as drag and gravity. Therefore, NimSim does not use controllers (such as a PID controller) when flying to a waypoint. Rather, NimSim sends the simulated UAV towards its target at a constant velocity. The speed of the simulated UAV as it travels to the waypoint is determined by its maximum Euler angle. When the simulated UAV reaches it target, it simply stops. The simulated UAV will overshoot by a certain amount (depending on its simulated velocity, and by extension, its simulated momentum), however once it stops moving it will remain in place.

The physical UAV, however, must operate in the physical world where real-world forces cannot be controlled or eliminated. In addition, the physical UAV itself is often a source of noise (due to potentially inaccurate sensor readings or hardware malfunctions). Therefore, the physical UAV requires the use of controllers to both fly to, and remain at, a waypoint. This means the physical UAV will not fly to its target at a constant velocity, rather it will begin to slow down as it gets closer to its destination, and attempt to correct for any overshoot once it reaches its destination. In order to remain at a waypoint, the physical UAV uses on-board PID controllers. For this feasibility study, moving the physical UAV to a waypoint is accomplished with a P controller in the vertical direction, and a PD controller in the translational direction.

5.2.1.2 Experimental Configuration

The 2-Best elitism, Rank selection, One-Point crossover, and Full-Range mutation evolutionary operators are used for this study. Due to limitations of NimSim and our distributed computing environment, we are unable to execute NimSim in a parallel fashion. Therefore, because only one instance of NimSim can execute at a given

time, UDivE is required to execute in serial mode; processing chromosomes one at a time. Because this study requires real-time simulation, and each chromosome is processed serially, we use a population size of 12. This population size was chosen as a compromise to reduce execution time while still providing as much population diversity as possible.

We ran this study using a HP ProLiant DL580 G7 server with 8 Intel Xeon E7-4820 (2.0GHz/8-core/18MB/105W) processors (64 cores total) and 64GB of memory.

5.2.2 Divergent Behavior

Consider a UAV in either an aggressive or passive configuration. We are interested in determining if UDivE can identify flight paths that either maximize or minimize the difference in distance travelled by the UAV in each of the respective configurations. Similarly, we are interested in determining if UDivE can identify flight paths that either maximize or minimize the difference in time required to execute the said flight path by the UAV in each of the respective configurations. We formalize our definition of aggressive and passive configuration in Section 5.2.2.1, and we introduce hypotheses that relate to these configurations in Section 5.2.2.2.

5.2.2.1 Configuration Definitions

Because we are interested in the behavior of the UAV, and not the specific implementation of that behavior, the way in which passive and aggressive are defined for the simulated UAV and physical UAV differ.

For the simulated UAV, configuration changes relate to the maximum Euler angle supplied to NimSim. The aggressive configuration is defined as the maximum Euler angle permitted by the Parrot UAV, which is 0.52 radians. The passive configuration

is defined as the lowest Euler angle that still permits the simulated UAV to remain responsive when flying between waypoints, which is 0.13 radians. These values were determined empirically by experimenting with NimSim.

For the physical UAV, configuration changes relate to the coefficients supplied to the controller used to move the UAV between waypoints. As discussed in section 5.2.1.1, the controller used to move the physical UAV between waypoints uses two controllers, one for vertical movement and one for translational movement. The vertical controller is a P controller, whereas the translational controller is a PD controller.

Changes were made to the respective P coefficients of these controllers. The aggressive configuration is defined as the largest P values that permit the UAV to fly between waypoints without potentially flipping over or crashing to the ground. These P values are 0.7 and 0.8 for the vertical controller and translational controller, respectively. The passive configuration is defined as the smallest P values that permit the UAV to fly between waypoints without undershooting its target, or becoming unresponsive to flight commands. These P values are 0.6 and 0.38 for the vertical controller and translational controller, respectively. These values were determined empirically by experimenting with the physical UAV.

5.2.2.2 Hypotheses Definitions

We formalize the goals of this feasibility study in the form of 4 hypotheses. We begin by considering hypotheses that relate to the difference in distance travelled by the UAV in the respective configurations. We are interested in determining if UDivE can identify flight paths that either maximize or minimize this difference.

The difference in distance travelled by the UAV in the respective configurations, when executing randomly generated flight paths (as defined in Section 5.2.3), show a percent difference of approximately 10%. Therefore, we consider a difference of

distance to be maximized when the percent difference of the distances travelled by the UAV in the respective configurations is 15%. Similarly, we consider a difference of distance to be minimized when the percent difference of the distances travelled by the UAV in the respective configurations is 5%. Formally, Hypothesis #1 and #2 are defined below.

Null Hypothesis #1: Given two identical Parrot UAVs, one in an aggressive configuration and the other in a passive configuration, that execute the same flight path, the UAVs will not travel respective distances that differ by more than 15%. In the event UDivE identifies a sequence of waypoints that cause the UAVs to travel respective distances that differ by more than 15% from one another, null hypothesis #1 is invalidated.

Null Hypothesis #2: Given two identical Parrot UAVs, one in an aggressive configuration and the other in a passive configuration, that execute the same sequence of waypoints, the UAV with an aggressive configuration will always travel a distance that is greater than 5.0% of the distance travelled by the passive UAV. In the event UDivE identifies a sequence of waypoints that cause the UAVs to fly respective distances that differ by less than 5% of one another, null hypothesis #2 is invalidated.

Next, we consider hypotheses that relate to the difference in time required for the UAV in the respective configurations to execute a flight path. We are interested in determining if UDivE can identify flight paths that either maximize or minimize this difference. The difference in time required by the UAV in the respective configurations to execute randomly generated flight paths (as defined in Section 5.2.3), show a percent difference of approximately 40%.

Therefore, we consider a difference of time to be maximized when the percent difference of the time required by the UAV in the respective configurations to execute a flight path is 50%. Similarly, we consider a difference of time to be minimized when

the percent difference of the time required by the UAV in the respective configurations to execute a flight path is 30%. We allow a larger range of percent difference for hypotheses related to time, when compared to those related to distance, due to the larger percent difference observed in randomly generated flight paths. Formally, Hypothesis #3 and #4 are defined below.

Null Hypothesis #3: Given two identical Parrot UAVs, one in an aggressive configuration and the other in a passive configuration, that execute the same sequence of waypoints, the UAVs will not require respective amounts of time to complete the sequence that differ by more than 50%. In the event UDivE identifies a sequence of waypoints that cause the UAVs to require respective amounts of time that differ by more than 50%, null hypothesis #3 is invalidated.

Null Hypothesis #4: Given two identical Parrot UAVs, one in an aggressive configuration and the other in a passive configuration, that execute the same sequence of waypoints, the UAV with an aggressive configuration will never require an amount of time that differs by less than 30.0% of the time required by the passive UAV. In the event UDivE identifies a sequence of waypoints that cause the UAVs to require respective amounts of time that differ by less than 30.0% of one another, null hypothesis #4 is invalidated.

5.2.3 Problem Model

For this study, each chromosome encodes a flight path that contains 10 waypoints. Here, a waypoint WP is defined as a three dimensional point in space, containing an numeric value for x, y, and z. More formally, $WP_i = (x_i, y_i, z_i)$. The point in space encoded by each waypoint is relative to the UAVs starting position in millimeters. For example, the waypoint $WP_A = (800, 1200, 500)$ represents the point in space that is

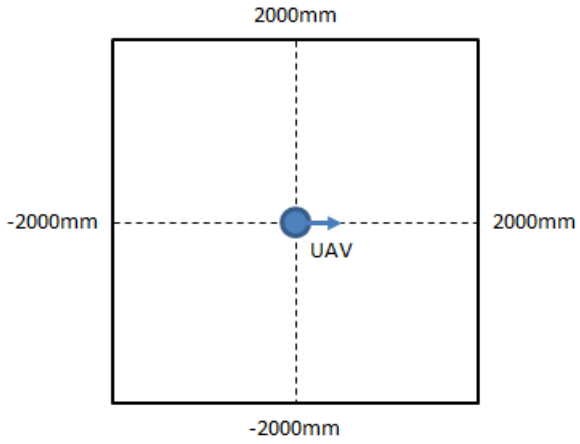

Figure 5.2: View of the XY plane in which the UAV is permitted to fly. The arrow shows the direction in which the front of the UAV is facing.

800mm in the x direction from where the UAV originated. Similarly, WP_A represents the point in space that is 1200mm and 500mm from where the UAV originated in the y and z directions, respectively.

The problem model PM_{UAV} for this study is defined as

$$
\begin{aligned}
&CT = WP = [10]\{x, y, z\} \\
&Pop = 12 \\
&C_{init} = \{WP_0 = (0, 0, 1000), \forall i > 0 | -2000 \leq WP_i\{x\} \leq 2000 \wedge \\
&-2000 \leq WP_i\{y\} \leq 2000 \wedge 1000 \leq WP_i\{z\} \leq 3000\} \\
&C_{perm} = \{WP_0 = (0, 0, 1000), \forall i > 0 | -2000 \leq WP_i\{x\} \leq 2000 \wedge \\
&-2000 \leq WP_i\{y\} \leq 2000 \wedge 1000 \leq WP_i\{z\} \leq 3000\} \\
&Op = \{\text{2-Best, Rank, One-Point, Full-Range}, MutRate = 0.10\} \\
&Ftn = \{|M_{Aggressive} - M_{Passive}|, M_{Aggressive} - M_{Passive}, M_{Passive} - M_{Aggressive}\}
\end{aligned}
$$

The UAV is constrained to remain in a 32 cubic meter box during flight. That means each waypoint WP_i must exist within the bounds of the box. This constraint is applied for two reasons, to prevent infeasibly long flight paths from being introduced into the study, and so that we could leverage our motion capture system during

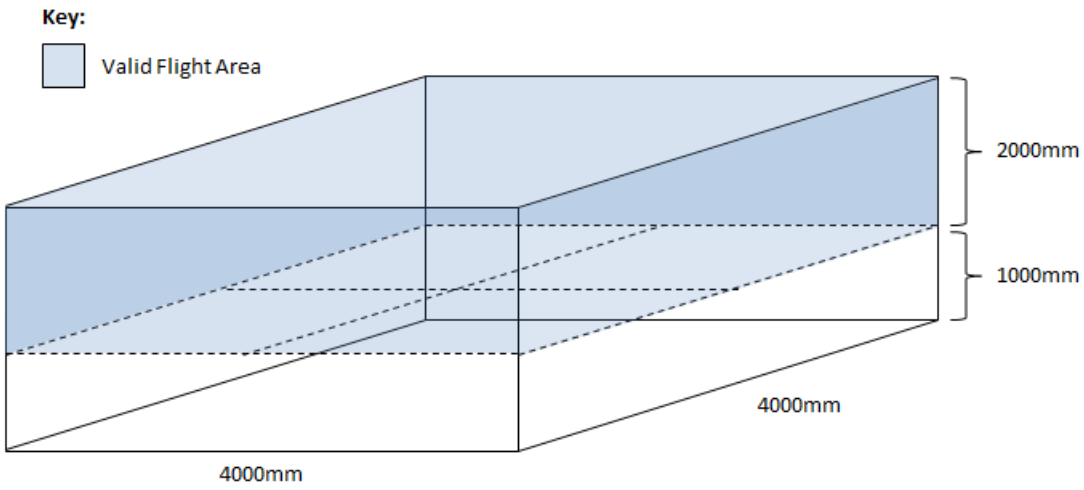

Figure 5.3: Three dimensional view of the area in which the UAV is permitted to fly.

results verification (discussed in Section 5.2.5.1) to provide a ground truth. Our motion capture system is only capable of capturing data in the specified area.

In both the x and y dimensions, the UAV is permitted to fly in the range $[-2000, 2000]$. Because these values are relative to the UAV's starting position, this range represents 4000 millimeters for each dimension (a total of 16000 square millimeters, or 16 square meters). See Figure 5.2 for a depiction of this area.

In the z dimension, the UAV is permitted to fly in the range $[1000, 3000]$ after take off. The UAV begins on the ground at position $(0, 0, 0)$ for each flight, therefore the first waypoint in each chromosome is constrained to be $(0, 0, 1000)$. This allows the UAV to take off and enter the valid range of z dimension values. See Figure 5.3 for a depiction of this area.

There were two types of metrics collected during this feasibility study, defined as M_{dist} and M_{time}. The first metric value is required by hypothesis #1 and #2 and is

the total distance travelled by the UAV in millimeters. Formally, the metric value M_{dist} for each configuration is defined as

$$M_{dist} = \sum_{i=2}^{n} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2 + (z_i - z_{i-1})^2}$$

where n is the number of UAV location samples taken during flight and i is the ith location sample (where $i > 1$).

The second metric value is required by hypothesis #3 and #4 and is the total amount of time required by each UAV in seconds. Formally, the metric value M_{time} for each configuration is defined as

$$M_{time} = Time_{10} - Time_0$$

where $Time_i$ represents the time in seconds at waypoint i. Subtracting the time taken when the UAV reached its final waypoint from the time taken when the UAV started at its first waypoint yields the total time required to execute the flight path.

5.2.3.1 Fitness Function Definition and Biasing

When attempting to maximize the difference in distance travelled, or the difference in time required to execute a flight path for Hypothesis #1 and #3, respectively, Ftn is defined as

$$Ftn = |M_{agressive} - M_{passive}|$$

Because we are interested in simply maximizing the fitness of the chromosome, we are not concerned with which metric value is larger or smaller in the random case. We simply wish to maximize their difference. Therefore, the absolute value difference of the metric values is used to compute fitness for these hypotheses.

However, in practice, one of the metric values is smaller than the other in the random case. In order to minimize the difference in distance travelled, or the difference in time required to execute a flight path for Hypothesis #2 and #4, respectively, this must be taken into account. Therefore, we leverage a biasing feature for these two hypotheses, which allows one of the two metric values to be favored over the other.

For Hypothesis #2 (minimizing the difference in distance travelled), we observe that in the random case, for a given flight path, the passive configuration will travel a shorter distance than the aggressive configuration. Therefore, in order to minimize the difference between the two metric values, Ftn is defined as

$$Ftn = M_{passive} - M_{aggressive}$$

Because the goal of Ftn is to maximize fitness as generations progress, this definition of Ftn will favor flight paths in which the aggressive configuration travels a distance that is closer to the distance travelled by the passive configuration, thereby minimizing the difference in distance travelled.

For Hypothesis #4 (minimizing the difference in time required), we observe that in the random case, for a given flight path, the aggressive configuration will require less time than the passive configuration. Therefore, in order to minimize the difference between the two metric values, Ftn is defined as

$$Ftn = M_{aggressive} - M_{passive}$$

. Because the goal of Ftn is to maximize fitness as generations progress, this definition of Ftn will favor flight paths in which the time required by the passive configuration

is closer to the time required by the aggressive configuration, thereby minimizing the difference in time required to execute the flight path.

5.2.4 Results

Can UDivE identify divergent behavior?

The results for each hypothesis are presented below, as produced by the NimSim UAV simulator. We allowed UDivE to execute until either it invalidated its respective hypothesis, or the search converged. Here, we define convergence as 10 continuous generations during which the maximum fitness does not increase. In order to illustrate how the flight paths evolve during UDivE execution, we show both the generation 0 and maximally fit flight paths for Hypothesis #2 and #4. These hypotheses were selected because Hypothesis #2 is concerned with the difference in distance travelled, whereas Hypothesis #4 is concerned with the difference in time required to execute the flight paths. We wish to illustrate the evolution of the flight paths for each of these metrics.

Hypothesis #1:

UDivE was unable to identify a flight path that invalidated Hypothesis #1 prior to converging. However, we report the maximally fit flight path that UDivE was able to identify, which required 23 generations. When the UAV was in the aggressive configuration, it travelled 36.79 meters. When the UAV was in the passive configuration, it travelled 32.56 meters. This represents a difference of 4.23 meters, or a 12.2% percent difference, shy of the 15% difference required to invalidate Hypothesis #1. Figure 5.4 shows a three-dimensional diagonal view and the XY plane of the maximally fit flight path for both UAV configurations. In addition, the starting location and terminal location of the UAV are denoted in each view with a circle and triangle, respectively.

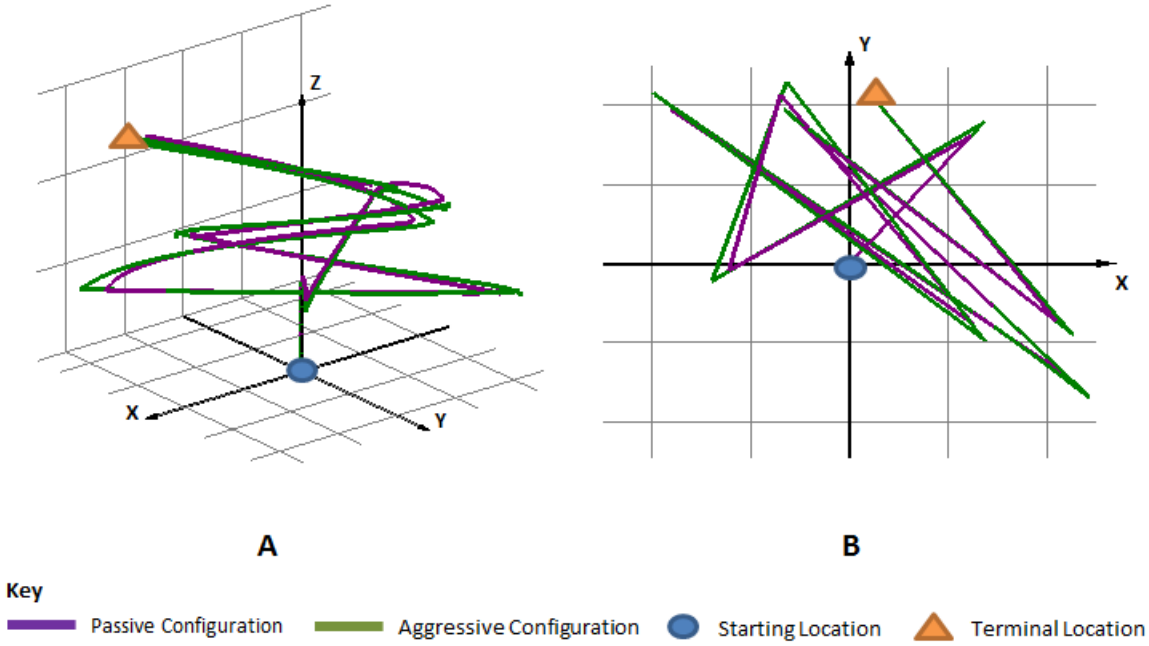

Figure 5.4: Hypothesis #1 Maximally Fit Flight Path. Shown from a (A) three-dimensional diagonal view and (B) as the XY plane. Each grid square represents one square meter.

As shown in Figure 5.4, we observe complex flight paths with many long segments that span the majority of the flight area, as well as several abrupt turns that, in some cases, send the simulated UAV in nearly the opposite direction from which it came.

Hypothesis #2:

UDivE was able to identify a flight path that invalidated Hypothesis #2 in 49 generations. When the UAV was in the aggressive configuration, it travelled 16.81 meters. When the UAV was in the passive configuration, it travelled 16.14 meters. This represents a difference of 0.67 meters, or a 4.06% percent difference. For this hypothesis, we show both the generation 0 flight path, as well as the maximally fit flight path to illustrate how the flight paths change during evolution. Figures 5.5 and 5.6 show a three-dimensional diagonal view and the XY plane of the generation 0 and maximally fit flight path for both UAV configurations, respectively. In addition,

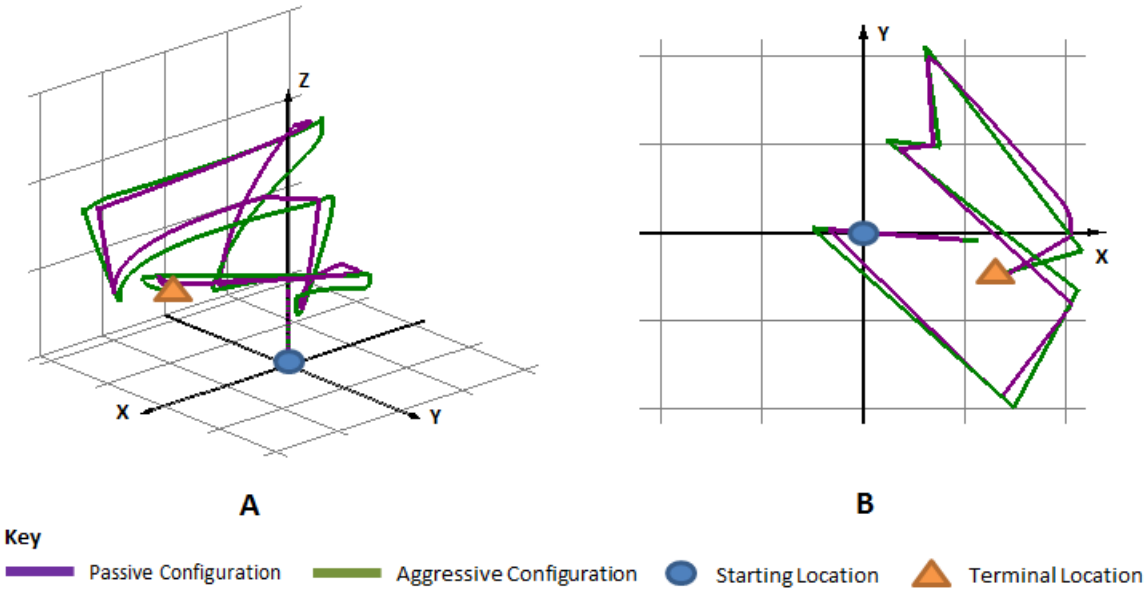

Figure 5.5: Hypothesis #2 Generation 0 Flight Path. Shown from a (A) three-dimensional diagonal view and (B) as the XY plane. Each grid square represents one square meter.

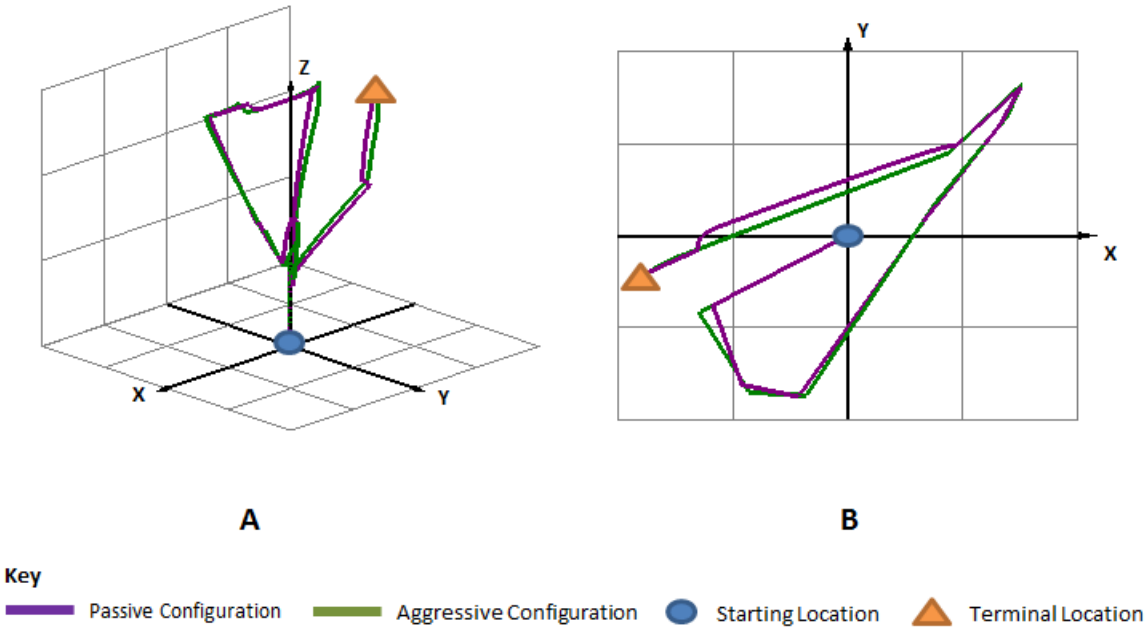

Figure 5.6: Hypothesis #2 Maximally Fit Flight Path. Shown from a (A) three-dimensional diagonal view and (B) as the XY plane. Each grid square represents one square meter.

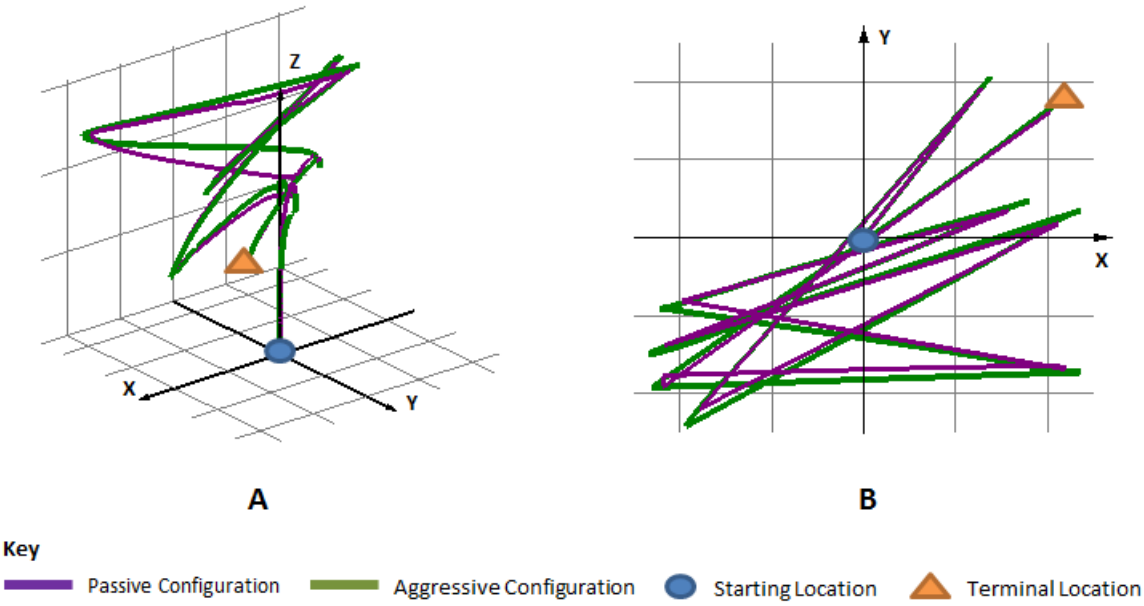

Figure 5.7: Hypothesis #3 Maximally Fit Flight Path. Shown from a (A) three-dimensional diagonal view and (B) as the XY plane. Each grid square represents one square meter.

the starting location and terminal location of the UAV are denoted in each view with a circle and triangle, respectively. Between generation 0 and generation 49, we observe flight paths that appear to become simpler and show fewer abrupt turns and a reduction in complexity.

Hypothesis #3:

UDivE was able to identify a flight path that invalidated Hypothesis #3 in 21 generations. When the UAV was in the aggressive configuration, it required 154 seconds to execute the flight path. When the UAV was in the passive configuration, it travelled 91 seconds. This represents a difference of 63 seconds, or a 51.42% percent difference. Figure 5.7 shows a three-dimensional diagonal view and the XY plane of the maximally fit flight path for both UAV configurations. In addition, the starting location and terminal location of the UAV are denoted in each view with a circle and triangle, respectively. As shown in Figure 5.7, we observe complex flight paths with

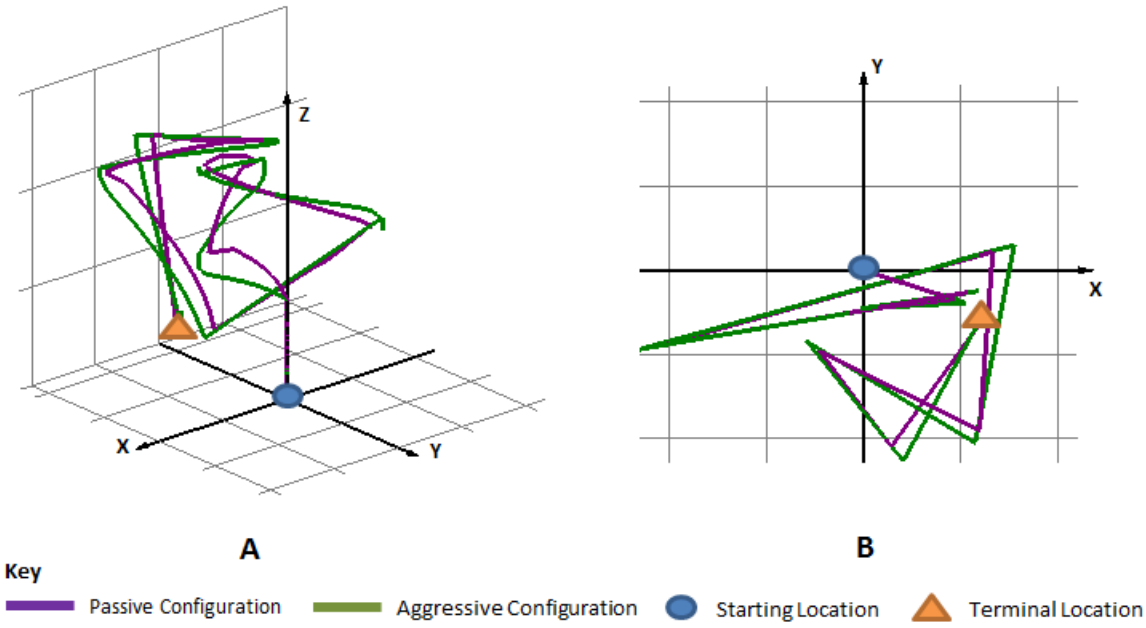

Figure 5.8: Hypothesis #4 Generation 0 Flight Path. Shown from a (A) three-dimensional diagonal view and (B) as the XY plane. Each grid square represents one square meter.

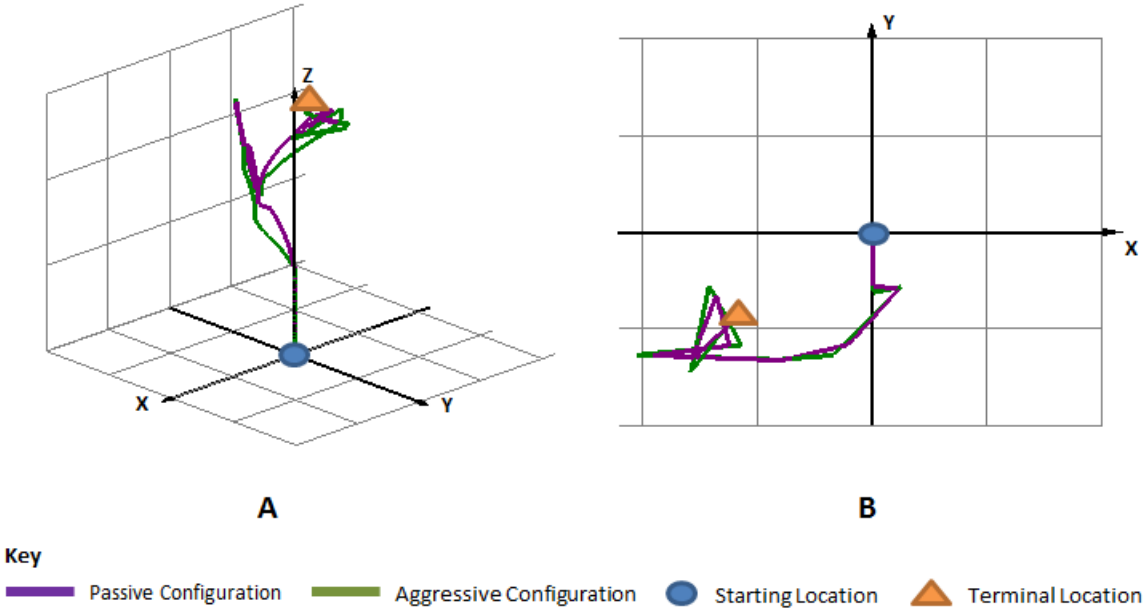

Figure 5.9: Hypothesis #4 Maximally Fit Flight Path. Shown from a (A) three-dimensional diagonal view and (B) as the XY plane. Each grid square represents one square meter.

several abrupt turns and long flight segments that send the simulated UAV back and forth across its flight repeatedly.

Hypothesis #4:

UDivE was able to identify a flight path that invalidated Hypothesis #4 in 41 generations. When the UAV was in the aggressive configuration, it required 57 seconds to execute the flight path. When the UAV was in the passive configuration, it travelled 73 seconds. This represents a difference of 16 seconds, or a 24.6% percent difference. For this hypothesis, we show both the generation 0 flight path, as well as the maximally fit flight path to illustrate how the flight paths change during evolution. Figures 5.8 and 5.9 show a three-dimensional diagonal view and the XY plane of the generation 0 and maximally fit flight path for both UAV configurations, respectively. In addition, the starting location and terminal location of the UAV are denoted in each view with a circle and triangle, respectively. Between generation 0 and generation 41, we observe flight paths that become much simpler, with fewer abrupt turns and a significant reduction in the length of flight segments.

What is the search space UDivE explores?

Because the problem model is the same for all four hypotheses (the only difference being the type of metric value under consideration), the search space is also the same. Both the x and y value for each waypoint are permitted to take one of 4000 values (i.e. any value in the range $[-2000, 2000]$). The z value for each waypoint is permitted to take one of 2000 values (i.e. any value in the range $[1000, 3000]$). The only exception is the first waypoint, which is constrained to always be defined as $WP_0 = (0, 0, 1000)$.

Therefore, the first waypoint can hold only one value. The remaining nine can each take one of $4000 \times 4000 \times 2000 = 3.2 \times 10^{11}$ values. Because there are nine waypoints that can hold any of these values, the total search space size is

Hypo	Num of Gen	Avg Time/Gen (sec)	Total Time (Hrs)	Total Sim Flights
#1	23	3014.92	19.26	552
#2	49	2551.0	34.72	1176
#3	21	2622.13	15.23	504
#4	41	1881.42	21.42	984
		Total	90.63	3216

Table 5.1: Cost of Running UDivE

$3.2^{11} \times 9 + 1 = 2.88 \times 10^{11}$. That means there are 2.88×10^{11} valid flight paths that could be selected by UDivE for exploration.

What is the cost of running UDivE?

Each hypothesis requires its own run, therefore the cost of running UDivE is presented for each of the four hypotheses in Table 5.1, as well as the total number of simulated flights that were required during each respective run.

It is important to consider the comparison of the cost of this feasibility study with that of manual flight data collection. As shown in Table 5.1, during all of the runs for all four hypotheses, UDivE explored a total of 3216 flight paths. Note that each chromosome had to be executed twice, once for each UAV configuration.

Here we provide a simple estimation of the cost of manual flight data collection, assuming only a single researcher is collecting the data. If we assume that each flight takes an average of 30 seconds (this number is an estimation based on actual flight data collected using physical UAVs), that means 26.8 hours of constant flight time would be required. However, between flights there is set-up and repair time. If we assume each flight requires an average of 3 minutes of set-up and repair time (this number is an estimation based on actual flight data collected using physical UAVs), then an additional 160.8 hours of set-up and repair time would be required. If the researcher did nothing but collect flight data for 8 hours per day, a total of 23.45 days

would be required execute the same number of flights NimSim was able to simulate in a total of 3.77 days (note that NimSim is capable of simulating flights 24 hours per day).

Regarding the batteries required for flight, a fully charged Parrot UAV battery provides approximately 12 minutes of flight time. If, as above, we assume each flight requires an average of 30 seconds, 134 fully charged batteries would be required. Each battery requires approximately 90 minutes of time to recharge, which would require 8.37 days of charge time. Granted, the researcher would most likely have access to multiple batteries and chargers, however if the researcher did not have access to a sufficient number of these resources, then waiting for batteries to recharge could further increase the cost of manual flight data collection.

5.2.5 Validation of Results with Physical UAV

An important component of this feasibility study is the physical validation of the results obtained in our simulated environment. In order to physically validate the results produced in our simulated environment, we execute various flight paths produced by the simulator using physical Parrot UAVs.

This section discusses the environment in which physical validation was performed, presents the results of the validation, and compares the results with those obtained in the simulated environment.

5.2.5.1 Physical Validation Procedure

The UAV was placed in a Vicon motion capture cage[5] for this experiment. This type of motion capture system provides a ground-truth measurement of the UAV's

[5] `http://www.vicon.com/products/bonita.html`

location, accurate to 0.5 millimeters[6]. This location information is supplied to the controller described in Section 5.2.1.1.

The UAV then executes the flight path twice, once in each configuration (as defined in Section 5.2.2.1), using the controller to move the UAV between waypoints. When a waypoint has been reached the controller moves the UAV to the next waypoint, and so forth, until the entire flight path has been executed. We are able to determine the distance the UAV flew, as well as the time it required, by analyzing the location information provided by the motion capture system.

Although the motion capture cage in which we performed the validation is 3 meters in height (the same height as the virtual cage used for the simulated UAV), in practice we found that allowing the UAV to fly to this altitude caused either the motion capture system to stop reporting positional data about the UAV (because it could no longer see it), or it caused the UAV to crash into the ceiling of the cage. Therefore, to preserve flight stability we truncated the z component of all waypoints to a maximum altitude of 2 meters. Although the UAV often flew higher than 2 meters (due to overshoot), this allowed the flight paths to be executed without frequent crashes.

5.2.5.2 Variation in Physical Results

In order to determine the amount of variation present in the physical validation results, we repeated the flight paths considered in Hypothesis #1 five times using the same procedure outline in Section 5.2.5.1 using a physical UAV. Although the flight paths identified by UDivE were unable to invalidate Hypothesis #1, we selected it because it showed the largest variation in distance travelled, as shown in Table 5.2.

[6] `http://www.vicon.com/products/bonita-features.html`

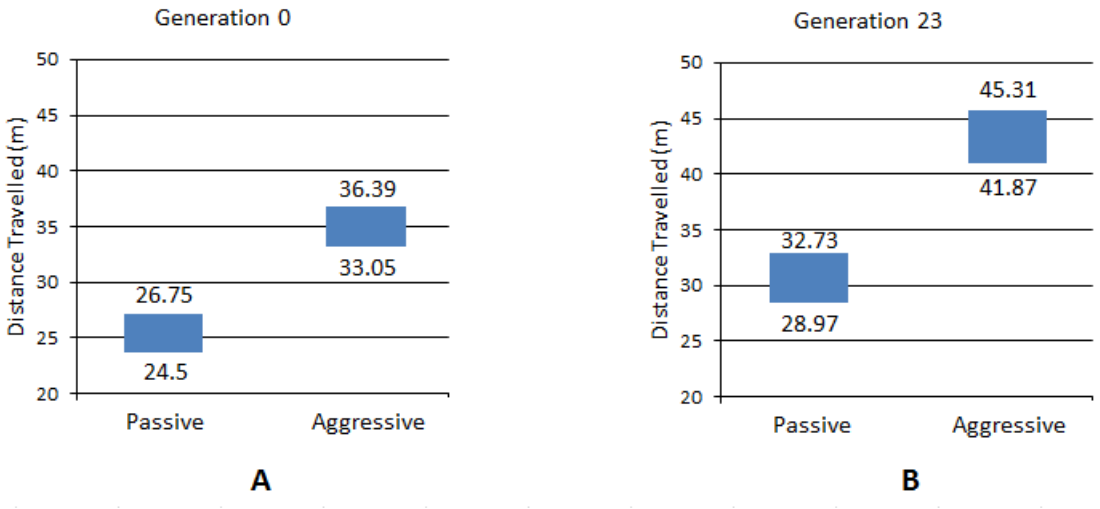

Figure 5.10: Variation of the distance travelled (m) for Hypothesis #1 (A) Generation 0 and (B) Generation 23. The boxes shown in the graph show the range of values (i.e. variation) present for each generation.

Figure 5.10 shows a plot of the variation we encountered when repeating the Hypothesis #1 flight paths. For generation 0, the passive UAV configuration showed a range of 2.25 meters, with a maximum distance of 26.75 meters and a minimum distance of 24.50 meters. The average distance travelled was 25.93 meters with a standard deviation of 0.90 meters. The aggressive UAV configuration for generation 0 showed a range of 3.91 meters, with a maximum distance of 36.39 meters and a minimum distance of 32.48 meters. The average distance travelled was 32.24 meters with a standard deviation of 1.59 meters.

For generation 23 (the last generation considered), the passive UAV configuration showed a range of 3.76 meters, with a maximum distance of 32.73 meters and a minimum distance of 28.97 meters. The average distance travelled was 31.06 meters with a standard deviation of 1.46 meters. The aggressive configuration for generation 23 showed a range of 3.44 meters, with a maximum distance of 45.31 meters and a

minimum distance of 41.87 meters. The average distance travelled was 43.25 meters with a standard deviation of 1.43 meters.

Due to the cost of validating these flight paths (including set up time, and the time required to monitor the trials), and because of the low standard deviation reported, we refrained from further checking.

5.2.5.3 Physical Validation Results

Below, the results of the physical validation are displayed for each hypothesis. Each table shows the results obtained during physical validation, as well as those obtained using NimSim.

For each of the Tables listed below, the "UAV" column displays the source of the results, either from the physical UAV or from NimSim. The "Generation" column displays results for both the randomly generated flight path from generation 0 as well as the maximally fit flight path from the generation when execution was terminated. The "Configuration" column displays the UAV configuration, i.e. aggressive or passive. The "Distance" or "Time" column (depending on the Hypothesis) displays the distance travelled by the UAV, or the time the UAV required to execute the flight path.

The "Difference" column displays the difference between the aggressive and passive configuration for the respective generation. The "% Difference" column displays the percent difference between the aggressive and passive configuration for the respective generation. Finally, the "Change" row, shown below the results for both the physical UAV and NimSim, displays the change in percent difference between generation 0 and the generation when execution was terminated. This value is annotated with either a (+) or a (-) to indicate if the change in percent difference increased or decreased, respectively.

Because our simulator is simplistic, we do not expect the values to match, however we are interested in evaluating if the following three trends (T_1, T_2, and T_3) hold between the NimSim UAV and the physical UAV. These trends are:

1. T_1 : For both NimSim and the physical UAV, the change in difference between the first and last generation will move in the same direction. For example, if for NimSim we observe an increase in the difference in distance travelled between the first and last generation, then so too should we observe an increase in the difference in distance travelled between the first and last generation for the physical UAV. The same trend should apply if we observe a decrease in the difference.

2. T_2 : For both NimSim and the physical UAV, the change in percent difference between the first and last generation will move in the same direction. For example, if for NimSim we observe an increase in percent difference between the first and last generation, then so too should we observe an increase in percent difference between the first and last generation for the physical UAV. The same trend should apply if we observe a decrease in the change in percent difference.

3. T_3 : For both NimSim and the physical UAV, the change of each configuration style between the first and last generation should move in the same direction. For example, if for NimSim we observe the distance travelled by the passive configuration increase between the first and last generation, then so too should be observe an increase in the distance travelled by the passive configuration between the first and last generation for the physical UAV. The same applies to the aggressive configuration. The same trend should apply if we observe a decrease in the distance travelled.

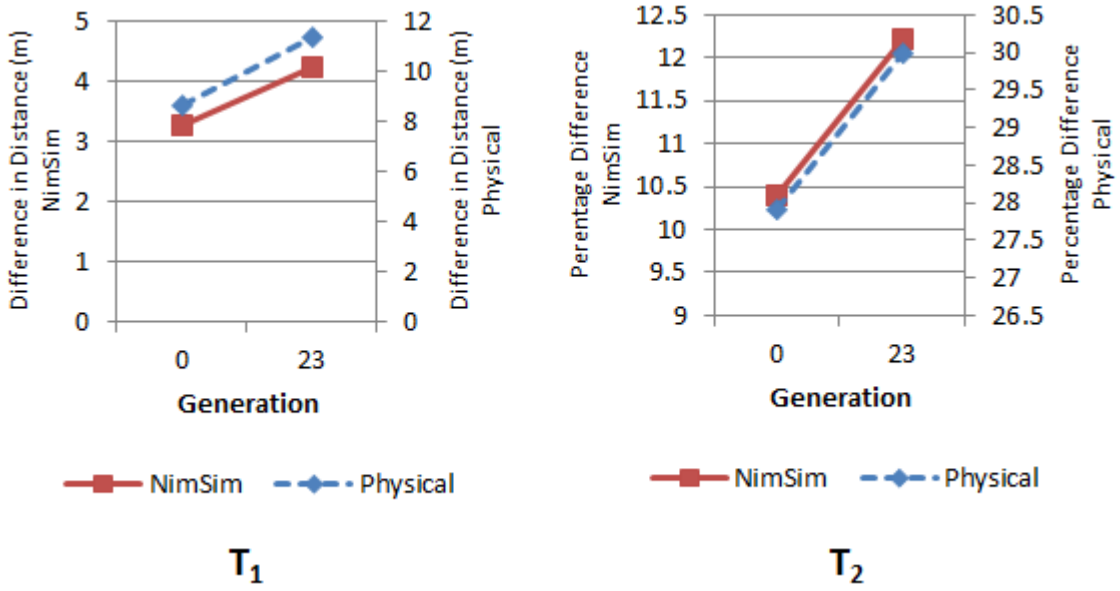

Figure 5.11: Depiction of trends T_1 and T_2 for Hypothesis #1. T_1 shows the change in difference (m) between generation 0 and 23 for both NimSim and the physical UAV, and T_2 shows the change in percent difference between generation 0 and 23 both NimSim and the physical UAV. Two y-axis scales are shown: NimSim on the left and the physical UAV on the right.

UAV	Generation	Configuration	Distance (m)	Difference	% Difference
Physical	0	Aggressive	35.22	8.65	27.9
		Passive	26.57		
	23	Aggressive	44.1	11.37	30.0
		Passive	32.73		
				Change	(+) 2.1
NimSim	0	Aggressive	33.0	3.26	10.4
		Passive	29.74		
	23	Aggressive	36.79	4.23	12.22
		Passive	32.56		
				Change	(+) 1.8

Table 5.2: Maximizing the Difference of Distance (Hypothesis #1)

Hypothesis #1 Validation. Table 5.2 shows the results of checking the Hypothesis #1 flight paths with a physical UAV. We observe each of the three trends occurring.

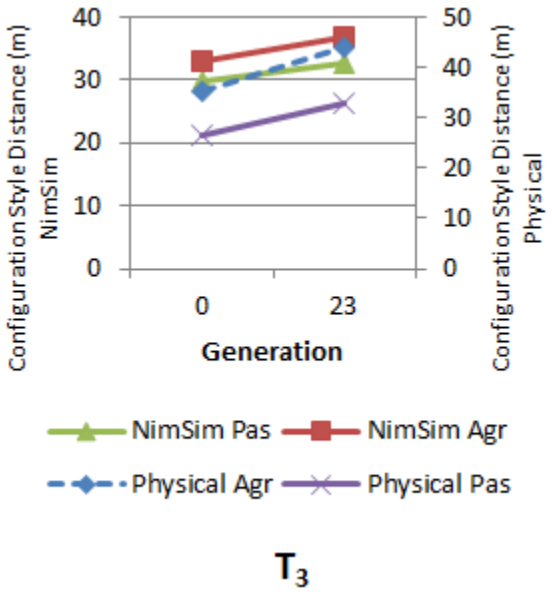

Figure 5.12: Depiction of trend T_3 for Hypothesis #1. T_3 shows the change in difference (m) between generation 0 and 23 for each configuration style (aggressive and passive) of both NimSim and the physical UAV. Two y-axis scales are shown: NimSim on the left and the physical UAV on the right.

Figures 5.11 and 5.11 show a graphical depiction of these trends between generation 0 and 23. The change in percent difference for the physical UAV moves in the positive direction, changing from 27.9% to 30.0% between generation 0 and generation 23. Similarly, the change in percent difference for NimSim moves in the positive direction, changing from 10.4% to 12.2% between generation 0 and generation 23. The change of the difference in distance travelled for physical UAV moves in the positive direction, changing from 8.65 meters to 11.37 meters between generation 0 and generation 23. Similarly, the change of the difference in distance travelled for NimSim moves in the positive direction, changing from 3.26 meters to 4.37 meters between generation 0 and generation 23. The change for each configuration style for the physical UAV moves in the positive direction between generation 0 and generation 23. Between generation 0 and generation 23, the aggressive configuration changed from 35.22 meters

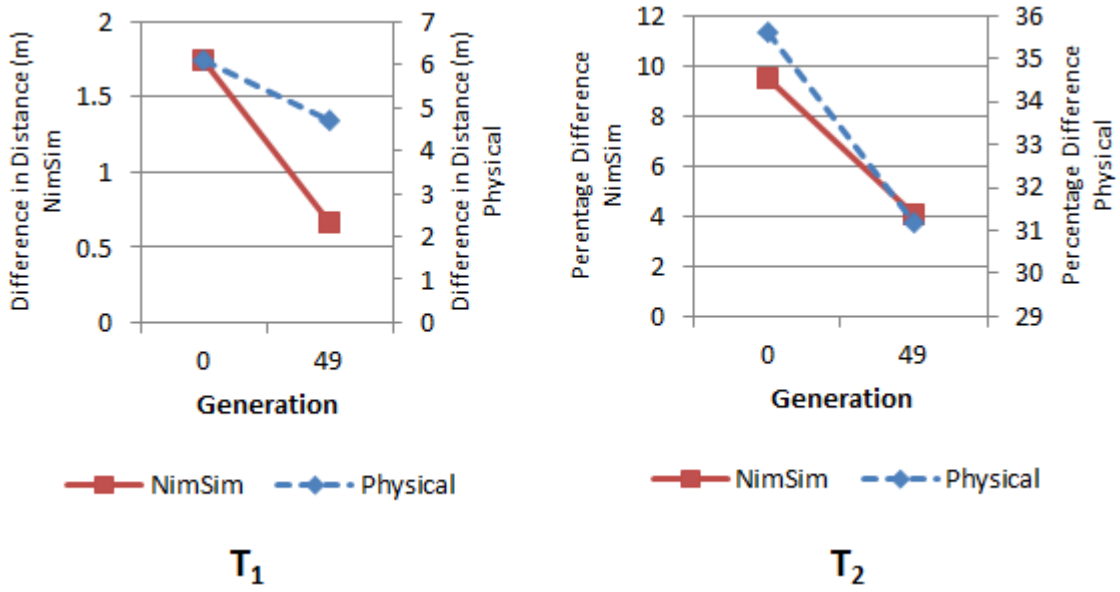

Figure 5.13: Depiction of trends T_1 and T_2 for Hypothesis #2. T_1 shows the change in difference (m) between generation 0 and 49 for both NimSim and the physical UAV, and T_2 shows the change in percent difference between generation 0 and 49 both NimSim and the physical UAV. Two y-axis scales are shown: NimSim on the left and the physical UAV on the right.

to 44.1 meters, whereas the passive configuration changed from 26.57 meters to 32.73 meters. Similarly, the change for each configuration style for the NimSim moves in the positive direction between generation 0 and generation 23. Between generation 0 and generation 23, the aggressive configuration changed from 33.0 meters to 36.7 meters, whereas the passive configuration changed from 29.74 meters to 32.56 meters. Although we observe each of the three trends occurring, as shown in Table 5.2, the actual value of the respective measurements differ when comparing the physical UAV and NimSim.

Hypothesis #2 Validation. Table 5.3 shows the results of checking the Hypothesis #2 flight paths with a physical UAV. We observe each of the three trends occurring. Figures 5.13 and 5.13 show a graphical depiction of these trends between generation 0

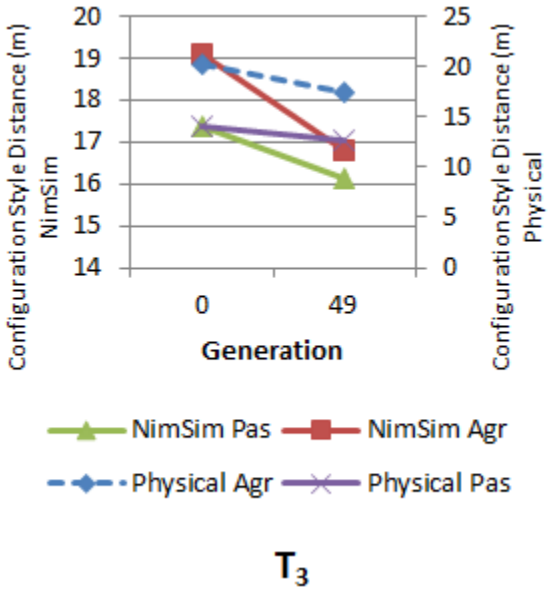

Figure 5.14: Depiction of trend T_3 for Hypothesis #2. T_3 shows the change in difference (m) between generation 0 and 49 for each configuration style (aggressive and passive) of both NimSim and the physical UAV. Two y-axis scales are shown: NimSim on the left and the physical UAV on the right.

UAV	Generation	Configuration	Distance (m)	Difference	% Difference
Physical	0	Aggressive	20.2	6.1	35.6
		Passive	14.1		
	49	Aggressive	17.4	4.7	31.2
		Passive	12.7		
				Change	(-) 4.33
NimSim	0	Aggressive	19.1	1.74	9.54
		Passive	17.36		
	49	Aggressive	16.81	0.67	4.07
		Passive	16.14		
				Change	(-) 5.47

Table 5.3: Minimizing the Difference of Distance (Hypothesis #2)

and 49. The change in percent difference for the physical UAV moves in the negative direction, changing from 35.6% to 31.2% between generation 0 and generation 49. Similarly, the change in percent difference for NimSim moves in the negative direction, changing from 9.54% to 4.07% between generation 0 and generation 49. The change of the difference in distance travelled for physical UAV moves in the negative direction, changing from 6.1 meters to 4.7 meters between generation 0 and generation 49. Similarly, the change of the difference in distance travelled for NimSim moves in the negative direction, changing from 1.74 meters to 0.67 meters between generation 0 and generation 49. The change for each configuration style for the physical UAV moves in the negative direction between generation 0 and generation 49. Between generation 0 and generation 49, the aggressive configuration changed from 20.2 meters to 17.4 meters, whereas the passive configuration changed from 14.1 meters to 12.7 meters. Similarly, the change for each configuration style for the NimSim moves in the negative direction between generation 0 and generation 49. Between generation 0 and generation 49, the aggressive configuration changed from 19.1 meters to 16.81 meters, whereas the passive configuration changed from 17.36 meters to 16.14 meters. Although we observe each of the three trends occurring, as shown in Table 5.3, the actual value of the respective measurements differ when comparing the physical UAV and NimSim.

Hypothesis #3 Validation. Table 5.4 shows the results of checking the Hypothesis #3 flight paths with a physical UAV. We observe each of the three trends occurring. Figures 5.15 and 5.15 show a graphical depiction of these trends between generation 0 and 21. The change in percent difference for the physical UAV moves in the positive direction, changing from 7.0% to 14.14% between generation 0 and generation 21.

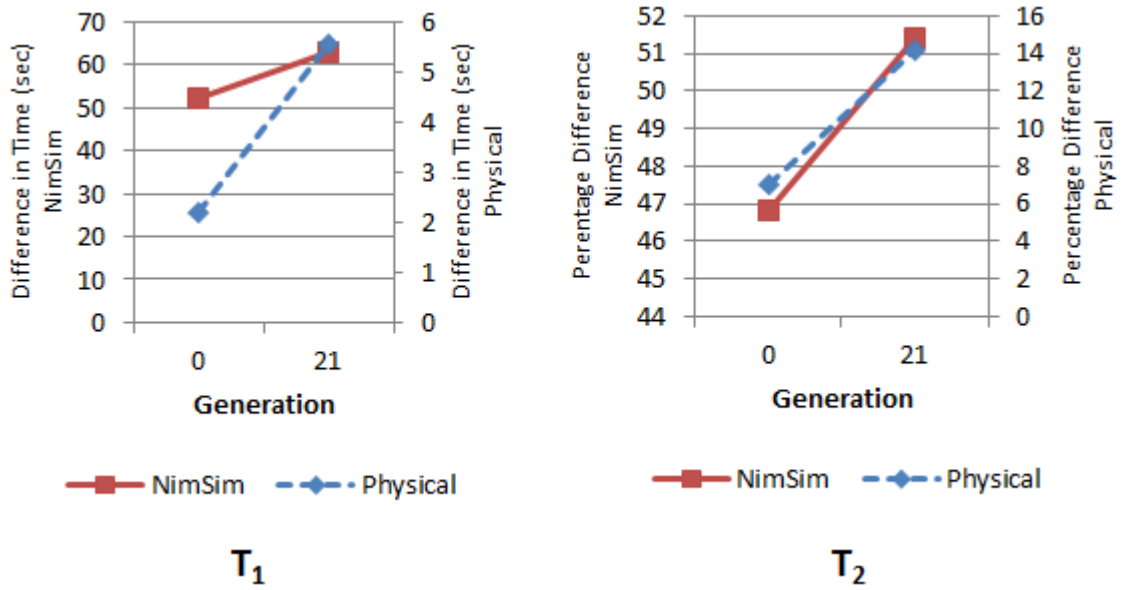

Figure 5.15: Depiction of trends T_1 and T_2 for Hypothesis #3. T_1 shows the change in time (sec) between generation 0 and 21 for both NimSim and the physical UAV, and T_2 shows the change in percent difference between generation 0 and 21 both NimSim and the physical UAV. Two y-axis scales are shown: NimSim on the left and the physical UAV on the right.

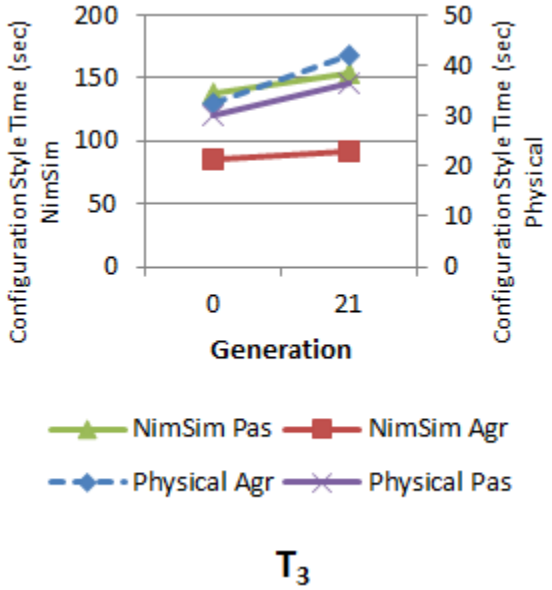

Figure 5.16: Depiction of trend T_3 for Hypothesis #3. T_3 shows the change in time (sec) between generation 0 and 21 for each configuration style (aggressive and passive) of both NimSim and the physical UAV. Two y-axis scales are shown: NimSim on the left and the physical UAV on the right.

UAV	Generation	Configuration	Time (sec)	Difference	% Difference
Physical	0	Aggressive	32.4	2.2	7.0
		Passive	30.2		
	21	Aggressive	41.94	5.54	14.14
		Passive	36.4		
				Change	(+) 7.14
NimSim	0	Aggressive	85	52	46.8
		Passive	137		
	21	Aggressive	91	63	51.4
		Passive	154		
				Change	(+) 4.62

Table 5.4: Maximizing the Difference of Time (Hypothesis #3)

Similarly, the change in percent difference for NimSim moves in the positive direction, changing from 46.8% to 51.4% between generation 0 and generation 21. The change of the difference in seconds required for physical UAV moves in the positive direction, changing from 2.2 seconds to 5.54 seconds between generation 0 and generation 21. Similarly, the change of the difference in seconds required for NimSim moves in the positive direction, changing from 52 seconds to 63 seconds between generation 0 and generation 21. The change for each configuration style for the physical UAV moves in the positive direction between generation 0 and generation 21. Between generation 0 and generation 21, the aggressive configuration changed from 32.4 seconds to 41.94 seconds, whereas the passive configuration changed from 30.2 seconds to 36.4 seconds. Similarly, the change for each configuration style for the NimSim moves in the positive direction between generation 0 and generation 21. Between generation 0 and generation 21, the aggressive configuration changed from 85 seconds to 91 seconds, whereas the passive configuration changed from 137 seconds to 154 seconds. Although we observe each of the three trends occurring, as shown in Table 5.4, the actual value of the respective measurements differ when comparing the physical UAV and NimSim.

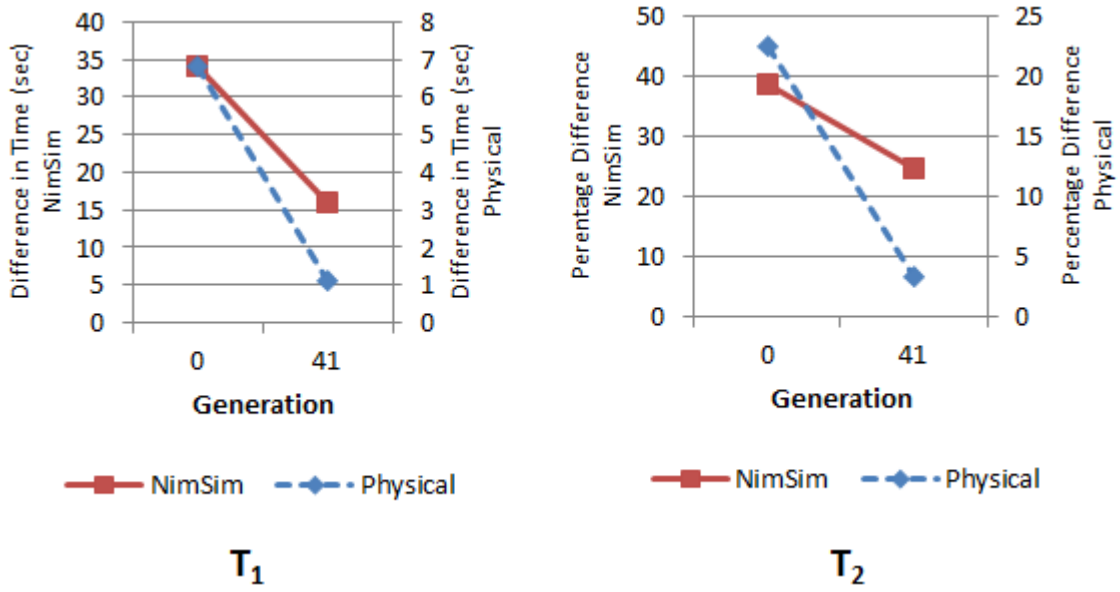

Figure 5.17: Depiction of trends T_1 and T_2 for Hypothesis #4. T_1 shows the change in time (sec) between generation 0 and 41 for both NimSim and the physical UAV, and T_2 shows the change in percent difference between generation 0 and 41 both NimSim and the physical UAV. Two y-axis scales are shown: NimSim on the left and the physical UAV on the right.

UAV	Generation	Configuration	Time (sec)	Difference	% Difference
Physical	0	Aggressive	26.9	6.8	22.4
		Passive	33.7		
	41	Aggressive	26.3	1.1	3.36
		Passive	27.2		
				Change	(-) 19.04
NimSim	0	Aggressive	71	34	38.6
		Passive	105		
	41	Aggressive	57	16	24.6
		Passive	73		
				Change	(-) 14.0

Table 5.5: Minimizing the Difference of Time (Hypothesis #4)

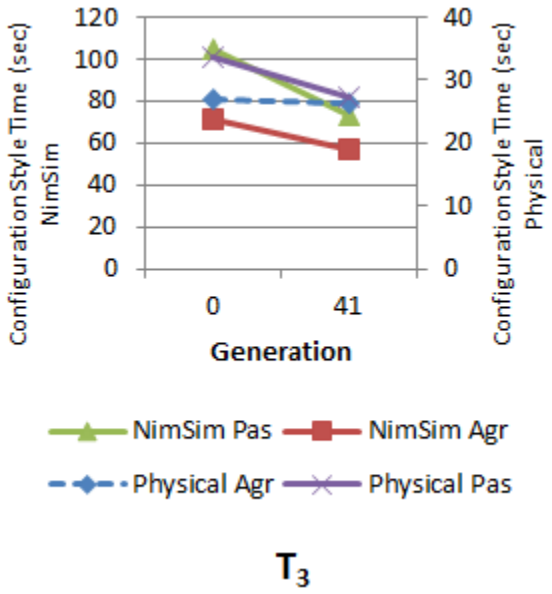

Figure 5.18: Depiction of trend T_3 for Hypothesis #4. T_3 shows the change in time (sec) between generation 0 and 41 for each configuration style (aggressive and passive) of both NimSim and the physical UAV. Two y-axis scales are shown: NimSim on the left and the physical UAV on the right.

Hypothesis #4 Validation. Table 5.5 shows the results of checking the Hypothesis #4 flight paths with a physical UAV. We observe each of the three trends occurring. Figures 5.17 and 5.17 show a graphical depiction of these trends between generation 0 and 41. The change in percent difference for the physical UAV moves in the negative direction, changing from 22.4% to 3.36% between generation 0 and generation 41. Similarly, the change in percent difference for NimSim moves in the negative direction, changing from 38.6% to 24.6% between generation 0 and generation 41. The change of the difference in seconds required for physical UAV moves in the negative direction, changing from 6.8 seconds to 1.1 seconds between generation 0 and generation 41. Similarly, the change of the difference in seconds required for NimSim moves in the negative direction, changing from 34 seconds to 16 seconds between generation 0 and generation 41. The change for each configuration style for the

physical UAV moves in the negative direction between generation 0 and generation 41. Between generation 0 and generation 41, the aggressive configuration changed from 26.9 seconds to 26.3 seconds, whereas the passive configuration changed from 33.7 seconds to 27.2 seconds. Similarly, the change for each configuration style for the NimSim moves in the negative direction between generation 0 and generation 41. Between generation 0 and generation 41, the aggressive configuration changed from 71 seconds to 57 seconds, whereas the passive configuration changed from 105 seconds to 73 seconds. Although we observe each of the three trends occurring, as shown in Table 5.5, the actual value of the respective measurements differ when comparing the physical UAV and NimSim.

5.2.6 Discussion

It is clear that, while we have observed results that follow the three trends presented in Section 5.2.5.3, the values and differences themselves differ. We believe the cause of these differences are the basic simulation capabilities of NimSim, as well as the differences between the way in which a UAV is controlled in NimSim and the physical world (as discussed in Section 5.2.1.1).

To further explore the difference between NimSim and and the physical UAV, we present a side-by-side comparison of the flight paths from Hypothesis #3, for both UAV configurations in generation 0 and generation 21. Figure 5.19 shows the flight paths of the UAV is its respective configurations for generation 0. Figure 5.20 shows the flight paths of the UAV in its respective configurations for generation 21. For simplicity, these flight paths are shown from an "above-facing" XY plane perspective. In addition, each of these Figures shows the staring location of the UAV (denoted as a circle), and the terminal location of the UAV (denoted as a triangle).

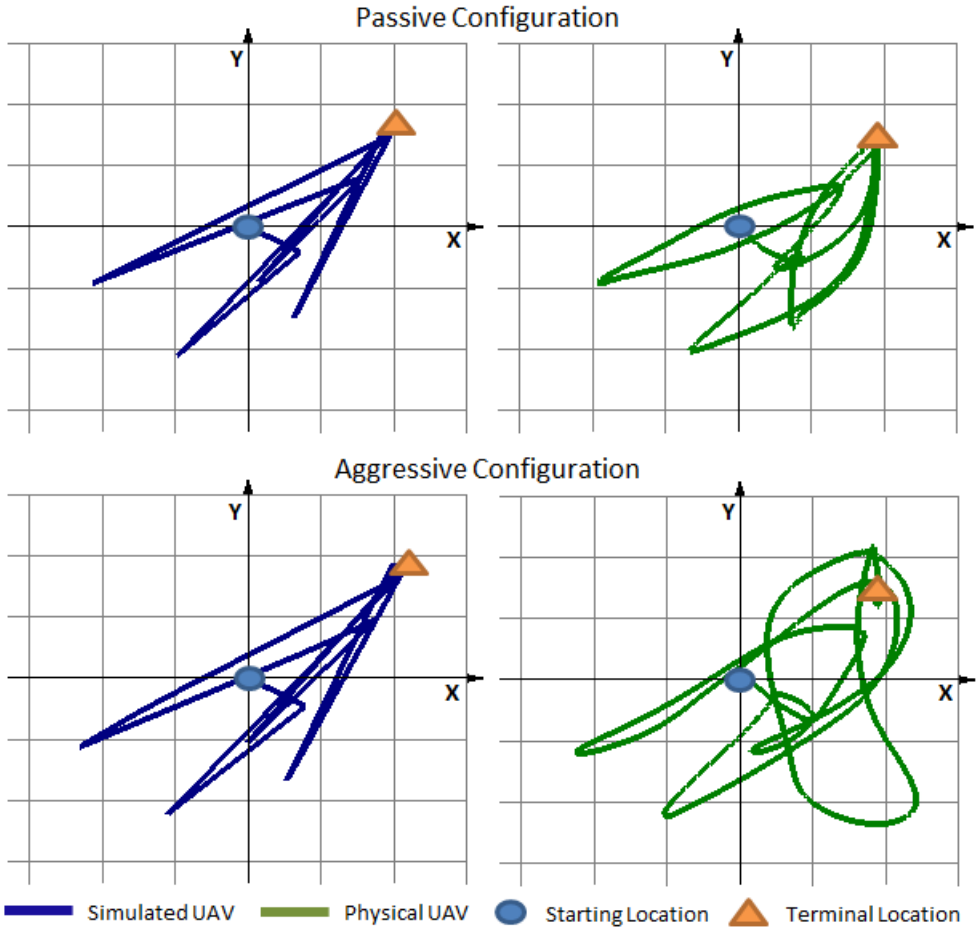

Figure 5.19: Comparison of Hypothesis #3 NimSim flight paths and physical flight paths for each configuration at generation 0. Each grid square represents one square meter.

Because NimSim is not subjected to the same degree of many real-world forces and noise as the physical UAV, its flight paths are precise and rigid (as evidenced by the sharp and corners when the simulated UAV changes direction in an abrupt fashion). The effect of the passive and aggressive configuration can be witnessed by the fact that the simulated UAV in an aggressive configuration will overshoot its target when compared to the simulated UAV in a passive configuration. This can be observed in both Figures 5.19 and 5.20.

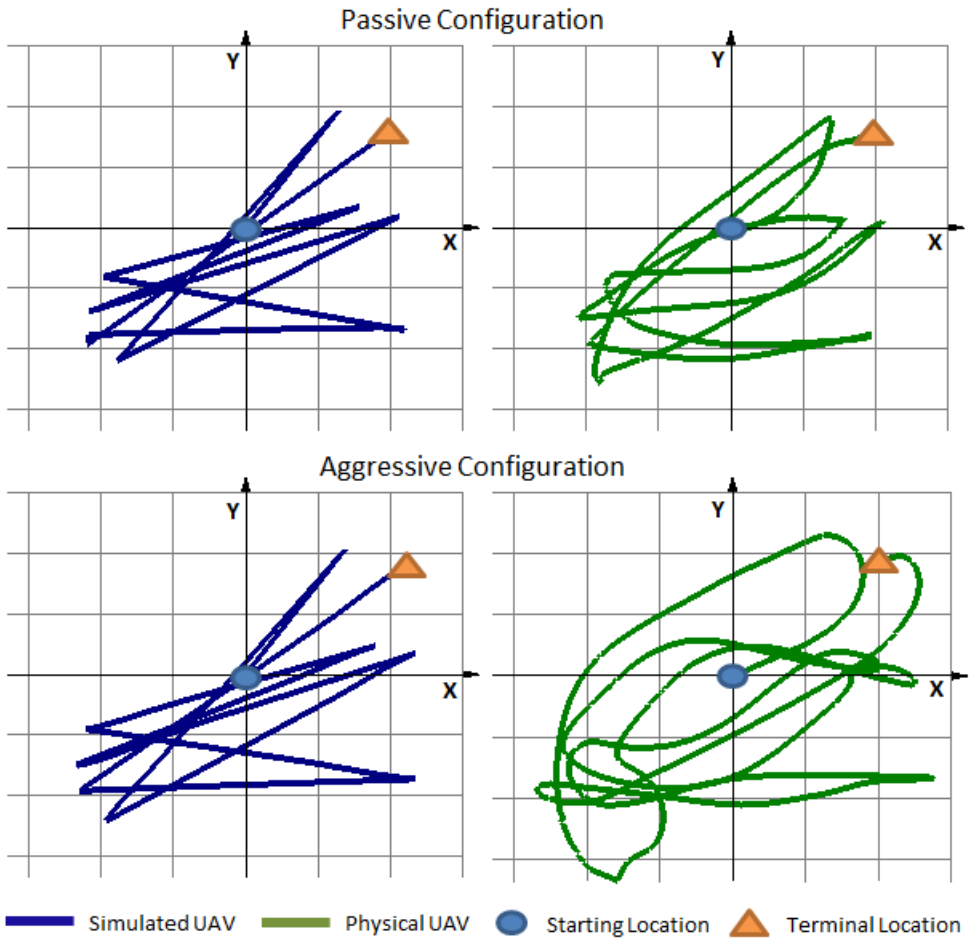

Figure 5.20: Comparison of Hypothesis #3 NimSim flight paths and physical flight paths for each configuration at generation 21. Each grid square represents one square meter.

The effect of real-world forces, noise, and the use of a controller for flying between waypoints can be seen in the physical UAV flight paths in both Figures 5.19 and 5.20. When compared to the simulated UAV flight paths, the physical UAV flight paths appear much less rigid and well defined. While the physical UAV is still executing the same flight path as the simulated UAV, it must constantly correct its trajectory due to real-world forces and noise. In addition, because the physical is leveraging a controller when flying between waypoints, it velocity will change as it gets closer to

(and overshoots) its target. This accounts for the curvature seen in the physical flight paths. This effect is especially evident when the physical UAV is in the aggressive configuration shown in Figure 5.20.

The fact that NimSim does not leverage controllers created a noticeable difference in behavior when compared to a physical UAV that uses these types of controllers. In order to explore this difference further, we set up an additional experiment. We created a simple flight path that is a 1 meter square. Given that this is such a small area with tight stopping and turning distances, we expected to see an exaggerated difference between the physical and simulated paths. When the physical UAV executes this flight path, the aggressive configuration (13.95 seconds) requires more time to execute this basic pattern than the passive configuration (9.7 seconds) because it overshoots each corner, and must compensate for its overshoot before moving on. The passive configuration, on the other hand, overshoots to a smaller degree and requires less compensatory correction, allowing it to navigate the square in less time. The simulated UAV, however, does not attempt to correct for overshoot. The aggressive configuration (19.4 seconds) requires less time to complete the pattern than the passive configuration (35.5 seconds), even though both still overshoot to respective degrees, simply because the aggressive configuration can fly at a greater speed. While this type of simple flight path was not observed in our study, it serves as one basic example of an improvement to NimSim that would allow for more realistic UAV simulation.

Fortunately, NimSim models the flight of a UAV with enough correctness that we have observed promising results that follow the trends presented in Section 5.2.5.3. It is clear that having a simulator that more closely models the physical behavior of a UAV would allow for our simulated results to be much closer to the physical

world, and perhaps identify flight paths that invalidate their respective hypotheses to a larger degree. We discuss this as future work in Chapter 6.

Chapter 6

Conclusions and Future Work

Identifying divergent behavior between implementations that behave similarly in most cases can help to avoid unexpected surprises when choosing an implementation to use in practice. In this thesis, we have introduced a design and implementation of UDivE, an automated approach to identify this type of divergent behavior. UDivE is capable of treating implementations as black boxes, and generating inputs that are evolved based on the output they produce in order to favor greater divergence. In addition, the most expensive phase of our framework can be parallelized in an effort to significantly reduce its execution cost.

UDivE also incorporates many parameterizable components to enhance its applicability, and the four studies we presented in Chapter 4 and 5 illustrate its potential to uncover unexpected behavior in several diverse domains. Further, the feasibility study presented in Chapter 5 presents a first step in extending UDivE so that it may conjecture about, and interact with, cyber-physical systems and simulations of those systems. Even in the presence of a basic simulator, UDivE was able to produce promising results that trend in the same direction as the results obtained when verifying the behavior with the cyber-physical system itself.

6.1 Future Work

We next present areas of future work that we have identified to enhance the applicability and effectiveness of UDivE. These areas include extensions to UDivE itself, as well as improvements to the process of applying UDivE to cyber-physical systems.

The first area of future work we present is the development of a more diverse set of problem models to be included with UDivE, including enriching the set of problem model components such as chromosome templates, fitness functions, and genetic operators. While a problem model is specific to the problem at hand, and therefore will generally require some degree of customization, many similarities may persist between problems and their respective problem models. Therefore, identifying these similarities and providing additional "pre-packaged" and "off-the-shelf" problem model components would make the application of UDivE to a new problem faster and simpler. This would also enable to UDivE to be applied to a more diverse set of domains.

While UDivE currently provides a rich set of genetic operators (as described in Chapter 3), there is always the potential to support others, especially customized genetic operators that may be required for a domain-specific problem. Allowing customized genetic operators to be easily incorporated into the framework would increase the applicability of UDivE to new problem domains.

The relationship between search space size and runtime has not been analyzed in detail. We believe that, while search space size and runtime do not seem to be strongly correlated, there may be other important factors to consider that would allow for the discovery of other important correlations. The landscape of the search space, and where divergent solutions exist in that space, must be explored in greater detail. Further, understanding how the application of system constraints affect the

landscape of the search space is important. This type of understanding would allow us to better conceptualize how our genetic algorithm traverses its search space and discover new correlations.

The choice to use a genetic algorithm for directed exploration was made for the studies presented in Chapter 4 and 5 due to the large search spaces being explored. A robust population-based heuristic search technique such as a genetic algorithm is beneficial when exploring these types of search spaces. However, other types of heuristic search techniques exist that may be equally or more effective depending on the type of problem. Therefore, following the above discussion of better conceptualizing the search space, an area of future work is understanding when another type of heuristic search technique such as hill climbing could be used with UDivE.

Once UDivE identifies divergent behavior, the question becomes "what is the next step?" Therefore, an area of future work is interacting with domain experts in selected fields to understand how to apply the results UDivE discovers. Not only would this help validate the usefulness of UDivE, but interacting with domain experts would provide feedback that could help tailor UDivE to be more effective in a given domain. This notion applies to the way in which problem models are created (including the chromosome encoding, genetic operators, and constraints chosen), as well as the way in which hypotheses are defined.

Another area of future work is improving user interaction with UDivE and the results it produces. In its current implementation, the framework must be launched from the command line, and all results are written to various files. However, providing a graphical user interface (GUI) that would centralize the launch, configuration, and interaction with the results would make it easier for a user to leverage the framework. This type of GUI could also be extended to provide real-time feedback, such as updating various plots that describe UDivE's progress as it executes. This type of

"centralized control" of UDivE would be especially helpful when UDivE is executing in multiple locations at once (such as a distributed computing environment when UDivE is in parallel mode).

To improve the effectiveness of UDivE when interacting with cyber-physical systems, improvements to UDivE are not the only changes that are necessarily required. Rather, the target systems with which UDivE interacts are a target for improvement. For example, in Chapter 5, UDivE interacts with the NimSim UAV simulator in order to generate flight paths that produce divergent behavior when executed by a UAV in an aggressive and passive configuration. However, the simplistic nature of the simulator, including the method with which it moves the UAV from one waypoint to the next, produces flight paths that differ in key ways from those produced when executed using a physical UAV. Therefore, an area of future work is the improvement of NimSim. An example of such an improvement would be extending NimSim so that it had the ability to simulate more realistic flight paths by leveraging more sophisticated drag, acceleration, and velocity models. Further, incorporating a controller (such as a PID controller) to move the UAV from one waypoint to the next would allow the simulated UAV to travel to target waypoints in a more realistic fashion, including the ability the compensate for target overshoot.

Appendix A

NimSim UAV Simulator

The Nimbus Lab UAV simulator (NimSim) is designed to provide real-time basic quad-rotor UAV simulation functionality. We created NimSim in order to facilitate experiments that require a large number of UAV flights. Rather than attempting to manually collect data, a process that can be prohibitively expensive (as outlined in Chapter 5), NimSim can be used as an alternative.

This appendix describes NimSim's architecture, supported input and produced output, configuration process, and implementation. Each of these are outlined in the following sections.

A.1 NimSim Architecture

NimSim is composed of several decentralized components, each of which are described below. The decentralized nature of NimSim allows for simple integration of additional modules. See Figure A.1 for a depiction of the NimSim architecture.

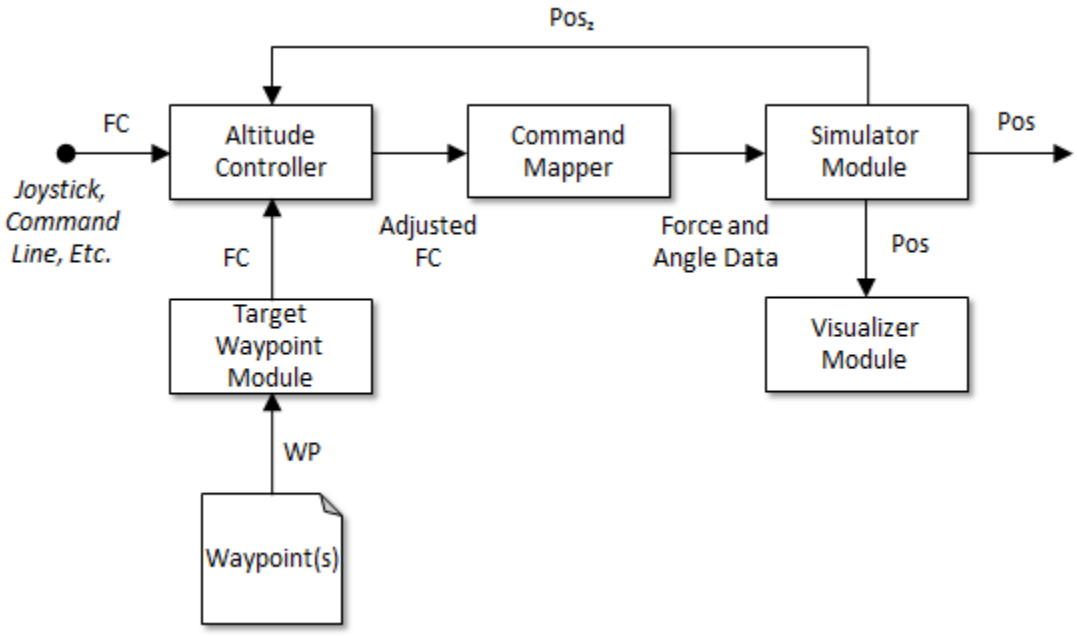

Figure A.1: NimSim Architecture

A.1.1 Target Waypoint Module

The target waypoint module is responsible for directing the simulated UAV from its current position to a target waypoint (as defined in Section A.2.2). It takes as input one or more target waypoints. It then computes the required low-level flight commands (as defined in Section A.2.1) and produced them as output. In the event multiple target waypoints are supplied to the target waypoint module, the simulated UAV will fly to each one sequentially.

A.1.2 Altitude Controller

The altitude controller serves two purposes. First, it serves as the entry point for a low-level flight command into NimSim. Second, as the name suggests, the altitude controller is responsible for maintaining the simulated UAV's altitude at a set point. This allows the simulated UAV to hover. The altitude control feature may be acti-

vated or deactivated. If deactivated, simulated gravity will affect the vertical position of the simulated UAV without intervention.

The altitude controller has two sources of input. The first is a low-level flight command. The source of flight commands could either come directly from the user (e.g. via a joystick or the command line), or the flight commands could come from the target waypoint module. Both sources of flight commands are of the same format, as defined in Section A.2.1. The second input source is the current altitude of the simulated UAV, supplied by the simulator module of NimSim. This is required if a loop-feedback style controller is being used to control the altitude of the simulated UAV (such as a PID controller).

A.1.3 Command Mapper

The command mapper is responsible for mapping flight commands provided by the user into input data that the rest of the simulator can consume. The command mapper takes as input x, y, and z component values, each of which are in the range $[-1.0, 1.0]$, and maps the values into force and angle values, in the units of newtons and radians, respectively. It then produces as output these force and angle values. The way in which these values are mapped depends on the configuration of the simulator, as outlined in Section A.3.

A.1.4 Simulator Module

The simulator module determines the simulated position of the UAV. The simulator module takes as input force and angle data, computes an updated position estimate of the UAV, and outputs the updated position estimate. The position output is in the format defined in Section A.2.3.

A.1.5 Visualizer Module

The visualizer module provides a graphical view of the simulated UAV in a simulated environment. This component is optional, however if leveraged it provides a simple three-dimensional, interactive window inside of which the simulated UAV will fly. Not only is the position of the simulated UAV shown, but its heading, flight path, and text displaying its position in space (in the form of a three-dimensional point containing an x, y, and z value) is shown as well.

The visualizer module takes as input the current position of the simulated UAV. Internally, it stores all of the provided position data, transforms said data into a format that the graphical user interface can consume, and draws the most recent simulated UAV position (including heading, flight path, and text) in the visualizer window.

A.2 NimSim Input and Output

NimSim supports two types of inputs: low-level flight commands or a target waypoint. Low-level flight commands are provided as input in the event direct control of the simulated UAV is desired. For example, if the user wishes to s simulate the flight of the UAV using a joystick. Joystick commands can be mapped into low-level flight commands that are consumed by NimSim. If a target waypoint is supplied as an input, NimSim will automatically simulate UAV flight to the specified waypoint. Each of these inputs are described in more detail below.

A.2.1 Low-Level Flight Commands

Low-level flight commands can be supplied to NimSim in the event direct control of the simulated UAV is desired. A flight command FC is defined as $FC = (x, y, z, w)$, where x, y, and z describe the components of the three-dimensional direction vector along which the simulated UAV is to fly and w describes the rate of yaw motion that is to be applied in the unit of radians per second.

Each of the values x, y, and z are permitted to take any value in the range $[-1.0, 1.0]$. The value w is permitted to take any value in the range $[0, w_{max}]$, where w_{max} is the maximum yaw rate of the simulated UAV.

As an example, suppose the input $Input_i = (0.5, -0.3, 0.8, 3.14)$ is supplied to NimSim. This input will send the UAV along a direction vector with component values $(0.5, -0.3, 0.8)$. The components of the direction vector describe the percentage of maximum force the UAV is capable of applying (in both the positive and negative direction). This input will also introduce yaw motion at a rate of 3.14 radians per second. NimSim maps these component and yaw values to simulated rates of motion based on the configuration supplied by the user, described in Section A.3.

In addition to low-level flight commands, NimSim supports three operational commands, "take off", "land," and "reset" that encode pre-defined low-level flight commands. The take off operation simulates UAV take off, the land command simulates the UAV landing, and the reset command resets all force and position data within the simulator (this is equivalent to powering down or restarting a physical UAV).

A.2.2 Target Waypoint

NimSim will accept as input a target waypoint $WP = (x, y, z)$, a three-tuple that describes a point in space. The units of each x, y, and z value are provided in millimeters.

Upon initialization, the simulated UAV will begin at point $(0, 0, 0)$. The values provided in a waypoint $WP_i = (x_i, y_i, z_i)$ are relative to this starting position. For example, if the UAV begins at point $(0, 0, 0)$, and the waypoint $WP_1 = (1000, 2000, 1500)$ is supplied, the simulated UAV will fly 1000mm in the x direction, 2000mm in the y direction, and 1500mm in the z direction.

In order to direct the simulated UAV to a target waypoint, NimSim computes a three-dimensional direction vector that points from the simulated UAV's current position to the target waypoint. The components of this vector are used to create low-level flight commands. These flight commands then move the simulated UAV at a constant velocity towards the target waypoint. When the simulated UAV reaches the target waypoint, a zero-valued flight command is applied and simulated drag slows the simulated UAV until it stops.

A.2.3 Output

NimSim will produce as output the simulated position Pos_i of the simulated UAV expressed as a four-dimensional point in space for a given time slice i. Formally, Pos_i is defined as $Pos_i = (x_i, y_i, z_i, w_i)$, where (x_i, y_i, z_i) describes the three-dimensional point in space where the simulated UAV is located at time slice i, and w_i describes the heading of the UAV at time slice i. Positional data reported by NimSim is relative to the simulated UAV's starting location. By default, the UAV originates at $Pos_0 = (0, 0, 0, 0)$ when the simulation begins or is reset.

A.3 NimSim Configuration

In order to effectively simulate a wide array of UAVs, NimSim relies on a set of configuration values that describe the physical properties of the UAV to be simulated

Configuration Type	Description	Units	Default Value
MAX_VZ	Max Vertical Velocity	mm/sec	2000 mm/sec
MAX_YAW_RATE	Max Yaw Rate	$radians/sec$	3.14 $radians/sec$
MAX_EULER	Max Euler Angle	$radians$	0.26 $radians$
MAX_ALTITUDE	Max Altitude	mm	5000 mm
FORCE_OF_GRAVITY	Gravity Force	N	4.0 $4N$
ACCEL_DUE_TO_GRAV	Gravity Acceleration	mm/sec^2	9800 mm/sec^2
MASS_OF_DRONE	UAV Mass	kg	0.408 kg
POS_VZ_GAIN	Ascent Gain	$none$	0.97
NEG_VZ_GAIN	Descent Gain	$none$	0.4
PITCH_GAIN	Pitch Gain	$none$	1.0
ROLL_GAIN	Roll Gain	$none$	1.0
ROTOR_SU_TIME	Rotor Spin-Up	sec	1.0 sec
TAKEOFF_DURATION	Takeoff Duration	sec	1.5 sec
TARGET_TO_ALT	Takeoff Altitude	mm	480 mm

Table A.1: NimSim Configuration Values

(such as the UAV's mass), physics values that describe the UAV's movement (such as acceleration rates), and restrictions on the movement of the UAV (such as the maximum euler angle and yaw rate). NimSim can be configured in two ways, via a configuration file or dynamically at run-time. If no configuration values are supplied at runtime, default values from the configuration file are applied. NimSim includes configuration values that are tuned to simulate a Parrot AR.Drone[1].

Table A.1 shows the NimSim configuration values that can be modified by the user. In addition, it gives their units and default values. The first 4 configuration values describe limits imposed on the simulated UAV. That is, maximum vertical velocity, yaw rate, Euler angle, and altitude. The next 5 values are concerned with ascent and descent. They take into account gravity and the simulated UAV's mass. In addition, "gain" values can be supplied that apply to ascent, descend, pitch, and roll. These values fine-tune the movement of the simulated UAV and must be determined

[1] `http://ardrone2.parrot.com/usa/`

empirically. The last 3 configuration values describe the simulated UAV's takeoff behavior, i.e. rotor spin-up time, takeoff duration, and the target takeoff altitude.

There are two more configurations that are not listed in Table A.1. These configurations are responsible for mapping flight commands into force values, both in the vertical and horizontal directions. The configurations are supplied in the form of polynomial that describes an trend line. The trend line maps a flight command component into a force value. These trend lines must be determined empirically for each type of UAV that is to be simulated. The default trend line provided for vertical force mapping is $0.6402x^3 - 2.0 \times 10^{-16}x^2 + 0.4647x$. The default trend line provided for horizontal (i.e. pitch and roll) force mapping is $0.5385x^1 + 0.8675$. For a given flight command component (i.e. "x" value), these trends lines will produce the net force (N) to be applied to the simulated UAV.

A.4 NimSim Implementation

This section describes in the implementation of NimSim. NimSim is implemented in the C++ programming language [35]. Further, NimSim is built with Robotic Operating System (ROS) [29]. ROS is a meta-operating system that provides publisher/subscriber communication functionality (referred to as "topics") between distinct computational units (referred to as "nodes").

Each architectural module described in Section A.1 is composed of one or more ROS nodes. This type of implementation is advantageous because it allows modules to be easily changes and modified with minimal impact on the rest of the simulator. Further, because each module is decoupled and communicates via a publisher/subscriber scheme, if a node stops executing due to a failure, the other nodes in the system will continue to execute without being affected. Another benefit of using ROS for

NimSim is that the UAVs used in the Nimbus Lab are already built and configured to communicate and interact with ROS. Therefore, NimSim can consume and produce messages in the same way as the physical UAVs, lessening the impact of integrating and using NimSim with new projects.

In order to visualize the simulated UAV in a graphical environment (as described in Section A.1.5), RViz is leveraged. RViz is three-dimensional visualization tool for ROS. Because it is built for ROS, it consumes messages from special topics that describe the content to be visualized. Therefore, is it a simple operation to publish messages from NimSim that describe the position and flight path of the simulated UAV that are then used by RViz to produce an interactive three-dimensional visualization.

Bibliography

[1] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '04, pages 2–13, 2004.

[2] Shawn A. Bohner and R. Arnold. Software change impact analysis. *IEEE Computer Society Press*, 1996.

[3] F. Bourgeois, L. Kneip, S. Weiss, and R. Siegwart. Delay and dropout tolerant state estimation for mavs. *International Symposium on Experimental Robotics*, 2010.

[4] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[5] Glen Van Brummelen. *Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry*. Princeton University Press, Princeton, NJ, 2013.

[6] CATG: Java concolic testing tool). website, 2013. https://github.com/ksen007/janala2.

[7] Sheng-Tzong Cheng and Tun-Yu Chang. A cyber physical system model using genetic algorithm for actuators control. In *Consumer Electronics, Communi-*

cations and Networks (CECNet), 2012 2nd International Conference on, pages 2269–2272, 2012.

[8] Ze Cheng, Ying Sun, and Yanli Liu. Path planning based on immune genetic algorithm for uav. In *Electric Information and Control Engineering (ICEICE), 2011 International Conference on*, pages 590–593, 2011.

[9] Paul Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2001.

[10] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *International Conference on Software Engineering (ICSE)*, pages 342–351, May 2005.

[11] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *ESEC/SIGSOFT FSE*, pages 185–194, 2007.

[12] C. Detweiler, B. Griffin, and H. Roehr. Omni-directional hovercraft design as a foundation for mav education. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 786–792, 2012.

[13] Dowek Gilles, Munoz Cesar, and Geser Alfons. Tactical conflict detection and resolution in a 3-d airspace. Technical report, NASA, 2001. NASA/CR-2001-210853 ICASE Report No. 2001-7.

[14] George Hagen, Ricky Butler, and Jeffrey Maddalon. Stratway: A modular approach to strategic conflict resolution. In *Preceedings of 11th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference*, Virgina Beach, VA, September 2011.

[15] Mark Harman. The current state and future of search based software engineering. In *Future of Software Engineering, International Conference on Software Engineering*, pages 342–357, 2007.

[16] Wei Jin, Alessandro Orso, and Tao Xie. Automated behavioral regression testing. In *ICST*, pages 137–146, 2010.

[17] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proceedings of the International Conference on Software Engineering*, ICSE '09, pages 309–319, 2009.

[18] J. K. Kuchar and L. C. Yang. A review of conflict detection and resolution modeling methods. *Transactions on Intelligent Transportation Systems*, 1(4):179–189, December 2000.

[19] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the International Conference on Software Engineering, ICSE*, pages 308–318, 2003.

[20] Phil McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1689–1696, 2009.

[21] Nathan Michael, D. Mellinger, Q. Lindsey, and V. Kumar. The grasp multiple micro-uav testbed. *Robotics Automation Magazine, IEEE*, 17(3):56–65, 2010.

[22] Joseph Moore and R. Tedrake. Magnetic localization for perching uavs on powerlines. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 2700–2707, 2011.

[23] NASA airborne coordinated conflict resolution and detection ACCoRD framework. website, 2012. http://shemesh.larc.nasa.gov/people/cam/ACCoRD/.

[24] C.K. Oh and G.J. Barlow. Autonomous controller design for unmanned aerial vehicles using multi-objective genetic programming. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 1538–1545 Vol.2, 2004.

[25] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings Joint European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 128–137, 2003.

[26] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 226–237, 2008.

[27] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.

[28] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Reicpes, The Art of Scientific Computing, 3rd Edition*. Cambridge University Press, New York, 2007.

[29] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[30] Colin Reeves. *Handbook of Metaheuristics*, chapter Genetic Algorithms. Kluwer Academic Publishers, Boston, 2003.

[31] C.R. Reeves. *Modern Heuristic Search Methods*, chapter Modern Heuristic Techniques. John Wiley & Sons Ltd., West Sussex, England, 1996.

[32] Matthew A. Russell and Gary B. Lamont. A genetic algorithm for unmanned aerial vehicle routing. *GECCO*, pages 1523–1530, June 2005.

[33] S. Shen, N. Michael, and V. Kumar. Autonomous multi-floor indoor navigation with a computationally constrained mav. *Proc. of the IEEE Intl. Conf. on Robot. and Autom*, May 2011.

[34] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[35] Bjarne Stroustrup, editor. *The C++ Programming Language*. Addison-Wesley, 2004.

[36] C. Teuliere, L. Eck, and E. Marchand. Chasing a moving target from a flying uav. *IEEE/RSJ International Conference on Intelligent Robotics and Systems*, 2011.

[37] Fangfang Zhang, Yoon-Chan Jhi, Dinghao Wu, Peng Liu, and Sencun Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 111–121, 2012.