Fall 12-4-2015

On Problematic Robotic Thresholds

Adam K. Taylor
University of Nebraska-Lincoln, aktaylor08@gmail.com

ON PROBLEMATIC ROBOTIC THRESHOLDS

by

Adam K. Taylor

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 Sebastian Elbaum

Lincoln, Nebraska

December, 2015

ON PROBLEMATIC ROBOTIC THRESHOLDS

Adam K. Taylor, M. S.

University of Nebraska, 2015

Adviser: Sebastian Elbaum

Large configuration spaces present difficulties for developers validating large software systems and for users selecting the proper configuration to achieve the desired runtime behavior. Robot systems face the same challenges as they may have hundreds of configurable parameters. Our work focuses on co-robotic systems, those in which robots and humans work closely together to augment each other's capabilities. We aim to leverage the user's knowledge about a system to help determine configuration errors. To accomplish this, users mark runtime failures while observing the system in operation. A marked error indicates the robot "did something when it should not have" or was "not doing something when it should have." In this thesis we have developed an approach that identifies predicates involving configuration parameters that may be relevant to each error type and can suggest adjustments based on the outcome of those predicates. In this work we present the following 1) A method to statically analyze Python and C++ code to identify threshold predicate comparisons (predicates with values initialized by configuration parameters that have an effect on specific execution patterns). 2) A characterization of the configuration space of popular robot systems. 3) A recommendation approach for configuration adjustment that combines user input with program analysis. 4) Three case studies assessing the approach and characterizing threshold predicate comparisons present in a running system.

DEDICATION

To Sarah Maresh, my wife. To Ken and Jayne, my parents. And to the rest of my family: Hannah, Wylee, Myles, Ayslee, Floyd, Kay, Bailey, and Corey.

ACKNOWLEDGMENTS

Deep thanks and gratitude to Dr. Sebastian Elbaum and Dr. Carrick Detweiler for all of their suggestions and ideas for the project. A thanks to Dr. Matthew Dwyer for taking the time serve on my committee and provide valuable feedback. Also a thanks to everyone in the NIMBUS Lab for their feedback, help, support, and suggestions during this endeavor.

GRANT INFORMATION

This work was partially supported by the Air Force Office for Scientific Research, Award #FA9550-10-1-0406 and National Robotics Initiative USDA-NIFA Award #2013-67021-20947

Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The software engineering community has recognized the difficulties for developers validating large configuration spaces and for users selecting the proper configurations for their systems to operate as expected. Configurations options have been identified to be one of the main causes of errors in both commercial and open source systems [1], and a number of approaches have been developed to solve the issues they cause as evidenced in a recent survey [2].

Popular robotic systems are not exempt from these configuration challenges. Their configuration spaces can be large and complex, in part, to enable users to tweak the systems to fit many potential usage scenarios. For example, the Arducopter Drone has 622 configuration parameters each containing multiple valid selections or a large range of selectable values [3]. A popular humanoid robot, Baxter, created by Rethink Robotics [4], [5] aims for workers and robots to collaborate to automate tasks in the workplace. In the available source code for Baxter [6], 236 locations are easily identified to read configuration parameters into the system. The large configuration spaces are also present in frameworks used by robot systems. The Robot Operating System (ROS) [7] is a widely used robotic "operating system" containing many frameworks that enable

robotic functionality. For example the navigation framework [8] includes over 220 listed parameters that can be used to configure the framework to work on a specific robot. These three examples show the widespread use of large configuration spaces in robotic systems. For some of these systems, the solution space seems similar to that already explored by the software engineering community.

Our work, however, focuses on a particular type of robotic system called co-robots, which closely cooperate with people to augment the robot's capabilities. We see them assisting patients to regain mobility, doctors to perform medical procedures, drivers to safely operate their vehicles, service personal to clean surfaces, scientists to collect data, and farmers to treat their crops. These kinds of robots are said to have the greatest potential to impact society [9], integrating the operator's sensing, actuation, and domain expertise with the robot's own.

As a concrete example, consider the aerial water sampler developed by the NIMBUS Lab [10]. The sampler aims to allow water scientists to quickly collect samples from a body of water without the need to deploy a boat or a larger, slower robotic system. This ability will increase the ease with which scientists collect routine data in disconnected bodies of water and increase their ability to collect samples quickly after events that may bring about changes in the environment. However, such a system still contains a large configuration space that must be set up correctly to run properly. For example, the system has a parameter that determines the allowable error when flying to a point to begin sampling. If the allowable error is too small and there is an external event, such as the wind, the UAV will not reach the desired location. If the value is too large the UAV may prematurely consider that it has reached the target location. In addition to basic navigation configuration, the system has a number of safety configuration parameters. One safety parameter is the height at which the UAV can fly before aborting the mission. If the value is too high the UAV may abort sampling even when it could continue to safely

operate. If the value is too low the UAV may crash into the water when it should abort the mission. Other configuration parameters set up the proper communication channels and may not directly affect the runtime performance. The source code controlling the robot system contains 353 locations that read configuration parameters. These 353 locations in the source code read values from 286 uniquely named parameters to initialize values. This large configuration space can be daunting to a user especially one which does not have experience configuring a robot system.

In this work we explore how this symbiosis of human and robot can be leveraged to detect when configurations are set improperly, and to collaborate to fix them and optimize them. More specifically, we enable users to mark system errors on the fly as either type I) the system did not do what it supposed to do, or as type II) the system did something unexpected. In the context of the water sampler, a type I error would be the system flying to a point to collect a sample, but not continuing to sample and instead hovering trying to reach a sample location. An example type II error would be the system aborting a sampling mission when operating normally because it was deemed too close to the water.

With these pieces of error information, we have developed an approach that identifies predicates involving configuration parameters that may be relevant to each error type, and that can suggest adjustments based on the outcome of those predicates. More precisely, the approach can determine how to adjust configuration parameters based on whether a predicate relying on a configuration parameter has been flipped or not, leading to the invocation or the lack of an invocation to a system action corresponding to type II and type I error respectively.

Returning to our aerial water sampler, let's suppose our approach would indicate a type II error where the drone did not go on to collect the sample when it should. Let's assume that this was caused because the allowable delta to reach a waypoint was

configured to be too small (the vehicle was close to the target waypoint, but not within the bounds so it keeps hovering around it trying to reach it). Underlying this scenario, there is a predicate in the code that evaluates whether the $current_{loc} = target_{loc} \pm delta$. The predicate never evaluated as true, the UAV did not reach the target location. Given the type of error and the fact that the predicate evaluation did not change, our approach would suggest for $delta$ to be increased in order for the predicate to evaluate to true and the system behavior to change.

Our work offers the following contributions:

- A recommendation approach for threshold adjustment that combines user input with static and dynamic analysis.
- A characterization of the configuration space of popular robotic systems.
- Three case studies assessing the approach and characterizing the characteristics of threshold predicate comparisons on a running system.

This work builds on much of the work in software engineering that has attempted to help users and programmers identify configuration errors in extremely large configuration spaces and extends this work to robot systems. We believe it is one of the first works to examine these large configuration spaces in robot systems. In robotics we have contributed a method that allows users of co-robotic systems to diagnose and solve configuration problems. This work aims to give the users of the many co-robotic systems that have appeared and are appearing another tool in diagnosing problems with the robot system.

We found that threshold predicate comparisons are a common occurrence across a number of open source systems. They appeared in 25 out of 52 of the systems we examined. When present each system had a mean of 21 threshold predicate comparisons that loaded values from 11 unique configuration options. In the runtime experiments

we found that threshold predicate comparisons appear very frequently during execution. However, we found that many of the threshold predicate comparisons identified during static analysis do not appear during runtime. We also found that flops are a rare occurrence. In the studies users had no problem identifying and marking runtime errors, but struggled to mark the correct error type. When users did mark the correct errors our system did a reasonable job identifying the problematic configuration parameter.

There are many avenues of future research that this work can be expanded into. First, more can be done to help users correctly identify the correct type of error. This can include methods to help make transition of the robots state more obvious to the users. In addition work can be done to make the system less dependent on the user identifying the error type. This may be possible by examining the characteristics of systems when the errors are occurring or developing a scoring system that produces comparable results between the two types of errors. We also want to retool how the score is calculated and examine the different methods of determining ranking of the problematic thresholds.

We also want to incorporate other source of setup parameters beside values loaded from the ROS parameter server. These other sources can include integer constants and header constants within the source code. The work can also be expanded from the individual unit of computation to the full robot system. Finally, more can be done to characterize how a threshold behaves in the system and if we can track different groupings of threshold predicate comparisons during execution to determine how the system is performing.

The rest of this thesis will summarize the static analysis used to identify threshold predicate comparisons and go in depth into the runtime approach of identifying the problematic threshold when a user identifies the error. We also will describe the validation we performed on the approach. After describing the approach we will focus on examining how common configurations are across a number of open source robot systems written

in ROS. Finally, we examine the approach and how it performs on three different robot systems. In these examinations we determine how threshold predicate comparisons play a factor in the execution of the system, how users mark errors in the system, and how well the approach can identify problematic thresholds in the three systems.

Chapter 2

Related Work

The software engineering discipline has recognized the challenges associated with the configuration of software systems, especially highly configurable systems. As early as 1985 operator actions, configuration, and maintenance were seen as the main sources of errors in fault tolerance systems and the need for tools to help users configure systems was identified [11]. The problems are also prevalent in all aspects of internet systems from the routers for internet traffic [12], to the setup and configuration of websites providing services [13], [14], and even on the databases serving the persistent data for the sites [15]. Configuration errors have been shown to cause over one third of the errors and repair time for Hadoop clusters [16] and were found to be the cause of 14% of the bugs in cloud computing systems [17]. Further adding to the problems encountered with configurations is the fact many different methods are used to modify configuration parameters and often the state of configuration is not readily available to the user or the developer [18].

Configuration problems extend beyond runtime options and also involve the selection of features included in product lines. Work has been done to identify the problems and offer solutions for configuring such systems [19]. These systems have a problem of configuration space explosion and recent work has examined how to reduce the size of

the space that must be dealt with for testing [20].

More recently, studies have aimed to determine and characterize misconfigurations on commercial and open source software systems [1]. In this work, Yin et al. found that 70-85% of the problems are caused from setting incorrect parameters including illegal values, inconsistent values, and external changes invalidating configuration options. Yin et al. identified that misconfigurations lead to system unavailability or performance degradation 16.1% to 47.3% of the time. They also found that only 7.2% to 15.5% of the problems provide messages to pinpoint the error. A study by Xu et al. found that a vast number of the choices (83% -94%) are never set by users [21] and when they are set only a small portion of possible values are used [21]. Xu et al. were able to successfully remove around half of the configuration space of a system [21]. Behrang et al. found that there are many inconsistencies between the different methods to change configuration options within a system and that the methods at some level may not have an effect on the actual execution of the system [22]. They studied 10 years of Mozilla Core and Firefox and found 40 configuration options that no longer had an effect on the system execution as the software line evolved out of the 2,000 provided.

A large body of work has identified the many problems that configuration of systems causes for users, but what approaches has the community taken to solve the issues? A recent survey by Xu and Zhou examines the enormous effort to address and fix the problems caused by the configuration options in software systems [2]. Kephart and Chess identify the many challenges faced in the configuration and maintenance of extremely large and complex systems [23]. They identified the need to have systems automatically configure themselves. The rest of this chapter will highlight individual work on configuration problems and also examine co-robotic systems.

2.1 Testing Systems with Large Configuration Spaces

One line of research in testing is examining how to exercise a space of configurations for more faults to be discovered. The idea of combinatorially selecting test inputs to find faults has been around for quite some time [24]. The authors present a way to generate tests that cover n-way combinations of test input parameters. This work focuses on all test inputs instead of just the configuration spaces. Richard Kuhn shows that often all faults in a system can be triggered by a combination of n or fewer parameters [25]. Srikanth et al. examine a similar approach, they considered the setup cost of tests while choosing which configurations to use in a test plan [26]. Yoon et al. examined how the numerous different components that must be configured in present systems can be tested and shown to be compatible [27]. One of the biggest threads of work in this area is the use of covering arrays [28]. Using these methods the authors showed that they did not have to test 50-99% of the configurations in the systems they tested. Further work in this line examined how to handle constraints on the configuration space [29] and [30] and what kind of errors are masked when certain selected configurations fail [31]. Once the configurations have been selected for testing, the work of Qu et al. examines which test cases should be used to exercise the selected configurations [32]. Another interesting approach to the problem is to only test configuration options that have been selected by a user in the wild [33]. One final combinatorial testing work examines how configuration values behave in the presence of a regression test [34].

Other testing work includes Reisner et al., which used symbolic evaluation to determine how the grouping of different configuration options affect line, basic block, edge and conditional coverage in different tests suites [35]. The work of Nanda et al. map configuration files to test cases, and when a configuration option is changed the test cases that are affected by the configuration option are highlighted to be reexamined [36]. An-

other approach to testing is determining how well a system can handle misconfigurations. *Conferr* is a tool to benchmark systems on the handling of configurations errors. The authors created a model of configuration errors, used the model to inject errors into real system's configuration files, and reported how well the systems handled configuration problems [37].

Our work does not focus on testing. The work on testing highlights how errors may arise only when certain configuration options are chosen. In robot systems the configuration options, environmental conditions, hardware, and source code all have an effect on the execution. The use of static analysis techniques is also prevalent in the testing work. It is used to determine what parts of code depend on configuration parameters, a key part of our method along with dynamic analysis.

2.2 Compile and Build Time Configuration

Work on the configuration problems also extends to compile time configuration options. Two studies by Nadi et al., [38] and [39], used static analysis of preprocessor directives to extract preprocessor constrains and produce a single propositional formula representing all enforced constraints in highly configurable programs. Nadi et al. studied what effects they had on the system. They found that the dependencies enforce correct builds, enforce the correct features, and their tool can improve the use of configuration options. The tools did a good job at extracting models, 93% of the models ensured a build without errors. However, when their system missed dependencies within the configuration, expert knowledge was required. *TypeChef* [40] is a tool created to perform the checking of #typedef constraints in C code. *Rachet* [27] was created to model the configuration space of a system and create test procedures to prevent build errors on all directly dependent configuration options.

Some software systems offer specific languages that define configuration constrains and offer tools to use these languages. When a constraint is violated *Range Fixes*, [41] and [42], provide a tool that shows which options and values will resolve a configuration constraint violation. The tool uses SMT solvers to solve constraints that have already been identified specifically for larger software systems such as the Linux kernel.

While our work does not directly focus on compile time configuration options. However, much like these works we aim to improve the user experience in interacting with configuration options. Many of these approaches also use a form of static analysis to extract configuration parameters from the system in question, but do not use any form of dynamic analysis as our system does.

2.3 Helping Find, Diagnose, and Fix Configuration Errors

The work in this section aims to assist users, system administrators, or developers in dealing with the many issues that arise from large configuration spaces in systems. The work is divided into two sections, those that work to only find the problematic configuration option after the error has occurred and those that aim to try to fix the problem or prevent it from happening. Our approach falls into the identification and fixing the problem categories; we aim to find the problematic parameter only after the user has identified the problem and offer simple suggestions on how to improve the system performance after it has been identified.

2.3.1 Finding The Problem

Using *Chronus*, the user creates a probe to test the system [43]. This probe answers if the system is operating correctly. The system searches throughout the saved state in the system to find the exact configuration change that caused the error to appear [43].

Whitaker et al. accomplish this by implementing system that stores the history of the system's disk writes. The system can then restore itself to different time periods and use the probe to find the exact time the system stopped functioning. Once the transition is found, the states can be compared to find the change that caused the error. *AutoBash*, [44], takes it one step farther, it provides a set of tools that allows speculative execution and the tracking of cause and effect throughout the computer system.

The approaches of Zhang and Ernst [45], Attariyan and Flinn [46], and Attariyan et al. [47] instrument a system and use a variety of methods to monitor the execution system in an attempt to link the problem back to the configuration option that is the cause of the error. To help solve configuration problems as the system is upgraded Zhang and Ernst instrument the code and examine data in a similar manner to our approach [48]. Their approach instruments the program's statements and predicates. The user provides an example test case that demonstrates the problem after the upgrade. Their approach examines the differences in the traces to determine which configuration options affect the areas where execution traces do not match. The approaches of Rabkin and Katz and Lillack et al. both aim to create a mapping from source code locations to preference values using static analysis [49] and [50].

PrefFinder aims to allow users to use plain text queries to discover which configuration option they should change [51]. A user provides a plain text query and the tool tries to find the preference that will cause the desired change in behavior to occur. The tool extracts preferences from the system using manuals, documentation, or a query against the API of the preference system. Once extracted, the tool splits preferences into meaningful words and parses them for code related words. The system also substitutes for more common words and ensures that only the bases of the words are used. The words in the query are matched with extracted words from the system's preferences and ranked to try and provide the best match. This approach could work very well on robot

systems, as the hope is that all of the parameters are aptly named for their functionality, however more research is needed to determine if that is the case.

Other "text" based approaches have also been explored. Xia et al. try to prevent developers from confusing configuration bug reports with normal bug reports [52]. They use the history of previous reports and language processing to make the determination if an incoming bug report is a configuration related bug. Processing the classified old bug data and performing text mining on the classified and new bug report accomplish the designation. Zhang and Ernst aim to inform developers when their system has error messages that do not expose the root cause of a problem, the configuration error [53]. They first mutate the configuration space by replacing each value in the configuration space. Next, the approach executes the system's tests and collects any failure information. Finally, the approach examines the diagnostic message and uses natural language processing to determine if it adequately matches the description of the parameter that caused the failure in the user manual

The lines of work in this subsection are the closest to our research. Much of the work attempts to find the configuration setting(s) that caused the discovered issue. In addition some of the approaches use static analysis to trace where configuration options affect source code. However, our approach is unique as it focuses on co-robotic systems that have additional sources of error and have a user directly monitoring the system.

2.3.2 Fixing or Preventing the Problem

One line of work examines how to prevent an error introduced in configuration from affecting the system before the configuration option has been verified. Oliveira et al. provide a prototype that first tests configuration changes before deploying them systemwide [15]. With *Barricade* [54] the system itself monitors changes that a user is making

to the configuration options. *Barricade* can prevent mistakes from propagating through the system. They tested such *Barricade* on a webservice with promising results. Another system that observes the system, monitors changes, and prevents and works around issues is *REACT* [55]. This tool creates guards against known configuration problems and dynamically attempts to solve problems within the system through reconfiguration. Our work does not try to prevent errors; instead we want a user to identify the error immediately after it occurs. After identification we will provide suggestions.

A common theme to prevent errors is using the information from known good configurations. *Confseer* [56] takes a snapshot of the system and checks that against a repository of known solutions for configuration problems built from information provided by software vendors. The works of Ramachandran et al. [57] and Zheng et al. [58] examine which parts of the systems depend on one another. The work of Zheng et al. provides templates and other methods to ensure that components within the system are still communicating as expected [58]. The same idea is also present in *EnCore* [59], which examines how configuration settings interact with the environment and how different components in a system interact. Kiciman and Wang created a tool that mines the windows registry to find constraints from known good system states [60].

PeerPressure [61] uses samples from other machines to diagnose problems on a machine that is not performing as it should be. It works on the idea that "the golden state is in the mass" [61]. This means that in a large group of machines one will have the correct configuration and it can be found by mining all of the machines for configuration values. The *Grid Monitoring System* [62] collects log files about system service locally on a networked grid of computers. Then, when requested, an outlier detection algorithm works to find any machines that are possibly misconfigured. Our approach does not leverage knowledge from other systems; however, using the knowledge about configuration parameters in known good systems may be an avenue worth exploring. Parameters that

differ from "normal" distributions may offer another clue that a given parameter is the cause of the issue.

The automatic tuning of parameters is a common theme among work in large configuration spaces. *VCONFG* [63] is a project that aims to use reinforcement learning to tune virtual machines. The work of Cooray et al. aims to proactively keep system reliability high using a learning approach [64]. It aims to reconfigure and tune the system before the reliability suffers. Bu et al. aim to configure web systems using reinforcment learning [65]. Their work presents a tool that is able to change both workload and virtual machine configurations. One type of system that has received a large amount of effort to enable auto tuning is databases. Many methods have been developed and tested to tune the configuration parameters of databases [66], [67], [68], and [69]. Our approach differs from these approaches, as it is not an attempt to automatically tune configuration parameters; it is trying to identify the problematic thresholds and offer suggestions for fixing them.

2.4 Co-robotics

Co-robotics [9] is the ability of robotic systems to be able to safely co-exist and work with humans on mundane, dangerous, precise or expensive tasks and it has been identified as an important research topic [70]. Robots in a co-robotic system will need to work closely with their human counterparts using each other's strengths in the planning and completion of a task [9]. To accomplish this the humans may integrate their sensing, actuation, and domain expertise with that of the robot. For a co-robotic system to gain widespread adoption the system must be inexpensive, easy to use, and widely available [9]. In addition to technical issues there will be ethical, economical, and societal issues [9]. Co-robotic systems will be able to accomplish many tasks including water sampling [10], crop measurements [71], service robots [72], weed control robots [73],

and many other usages. However, many challenges still need to be examined. Hayes and Scassellati [74] examine the challenges that need to be addressed during interaction between humans and robots. This includes how humans and robots can recognize their intents, how to assume roles, how to swap tasks, and how the robot can evaluate itself. In addition to the challenge of cooperative actions between robots and humans, the underlying problems facing roboticists are still present. Included in these challenges is the configuration of the highly customizable robotic systems. This work aims to begin to address this problem by leveraging the knowledge of the human operator present in the operation of co-robotic systems to understand when an error is occurring and help determine the likely cause of a configuration problem in the robot system.

One approach to the identified problems is to carefully control parts of the design or implementation to use specific processes. For example Woodman et al. developed an end-to-end design process to create systems that safely examine robot systems [75]. The key portion of their approach is the safety policies. These policies are created throughout the design process and are enforced by the system at runtime. Another approach by Sattar and Dudek worked on the user communication and command portion of these systems [76] They developed methods for the robot system to reason about the uncertainty of the task as well as the cost of performing the task. Using this the robot would make a determination on whether or not to perform a task or ask for clarification on the task from the user. There has been much work in monitoring execution in robot systems to identify faults [77]. The survey by Pettersson broke methods of runtime execution down into one of three categories: analytical, data-driven, and knowledge-based. Currently, our approach is data driven. We do not have a model of the underlying system except for the assumption that flops in our data are a significant event. The communications between units of execution is highlighted as an important feature of robot systems in a paper by Zaman et al. [78]. In this paper they extended the normal ROS diagnostics tools to create

a model based diagnostic approach. The work of Mendoza et al. worked to build a model based technique that can determine when motion is being impeded without any special sensors [79]. Golombek et al. also examined how a model of the robot system's data can be used to determine when the system is in an error state [80].

Chapter 3

Approach

This chapter details the approach we developed to identify and suggest solutions to problematic thresholds in robot systems. The section provides an overview of the approach as a whole and details the static analysis and runtime portions of the approach. The first section defines the terms used throughout this work. The second section highlights the assumptions of our approach. The third section provides a high level overview of the approach. The fourth section provides an overview of the method to determine the predicates that should be instrumented in the code and provides examples. The final section describes the runtime portion of the approach. The runtime implementation includes methods to mark problems, identifying the threshold that caused the problem, and reporting results to the user of the system.

3.1 Definitions

This section contains definitions for the terms used in this chapter and the rest of the thesis. It is split into terms specific to the approach, terms used in the context of ROS, and LLVM terms used in the C++ description of the approach.

3.1.1 Approach Terms

- Control Dependence : "A statement s is control dependent on the branch condition c of a conditional branch statement if the control structure of the program indicates that c potentially decides, via the branches it controls whether s is executed or not." [81].
- Data Dependence : "A statement s is data flow dependent on statement $s\prime$ if data potentially propagates from $s\prime$ to s via a sequence of variable assignments." [81].
- Program Dependency : A Control Dependence or a data Dependence.
- Reaching Definition : A definition d of some variable v reaches operation i if and only if i reads the value of v and there exists a path from d to i that does not define v [82].
- Branching Statement : Statement in which an expression is evaluated to determine the control flow of a program.
- Predicate : The statement that is evaluated in a branching statement to determine control flow of a program.
- Configuration Variable : Variable identified to have its value loaded from a source of system configuration.
- Exposing Statement: A function call that causes or may cause a change in behavior of the robot system outside of the scope of the current program.
- Program Slice : "Parts of a program that (potentially) affect the values computed at some point of interest" [83].
- Slicing Criterion : Point of interest used to compute a program slice. Defined in [83]

- Threshold : A configuration variable or value used in a predicate.
- Threshold Predicate Comparison : A predicate containing a threshold in part of the expression.
- Flop : Any instance where the value of a predicate changes from the previously held value.
- Type I Error : An error caused by a predicate flopping too early. The robot will change state before it should.
- Type II Error : An error caused by a predicate not flopping. The robot should be changing state, but it is not.
- UUID : Universally Unique Identifier. A unique identifier defined as defined in [84]

3.1.2 Robot Operating System

- ROS Robot Operating System. An "operating system" widely used in robotics. Offers communication protocols, frameworks, and other tools for robotic system development [7].
- Node : A process in ROS. Independent computational unit that handles a particular task or subset of tasks in the robot system as a whole. Communicates over known channels to other nodes to accomplish desired behavior.
- Parameter Server : The main method of passing configuration values to nodes in ROS.
- Topic : A named communication channel in ROS. Has a unique name and a defined message type. Can send and receive messages from many locations including multiple nodes. Provide a method of asynchronous communication.

- Publish : The act of sending a message over a topic.
- Service : Similar to a Topic, but a message is sent and the execution waits until a response is received. Provides a method to guarantee that a message is received and handled. A synchronous form of communication.
- Service Call : A statement which executes a call to a ROS Service.
- Message : Defined data objects that are sent over Topics and Services.
- Package : A collection of nodes and messages in ROS.
- Catkin : The Build system used to compile ROS packages.

3.1.3 LLVM Terms

- LLVM : Low Level Virtual Machine (LLVM) - framework to allow lifelong program analysis and transformation [85].
- Intermediate Representation (IR) : Representation of program during compilation. Mostly independent of machine and programming language.
- Clang : C and C++ frontend for LLVM.
- Module : Top level container for an LLVM IR. Contains functions, global variables, symbol tables, and any other needed information for the module.
- Basic Block : A single entry single exit portion of code in LLVM's IR . Contain a list of instructions.
- Instruction : The smallest unit in the LLVM IR. Performs actions such as reading from memory, terminating a block, or performing mathematic operation.
- Pass : In LLVM passes perform the transformation and analysis of the IR.

3.2 Assumptions

Our approach makes a few key assumptions about robot systems, threshold predicate comparisons, the runtime characteristics of them, and the ability of users to mark errors timely and correctly. These assumptions are addressed throughout the remainder of the thesis as data becomes available to addresses their validity.

Our first assumption is that the software systems controlling these robots have large and complex configuration spaces. As with other software systems the multitude of configuration options leads to configuration errors as users set incorrect values. We assume that these configuration errors manifest themselves as Type I and Type II errors. We also assume that the configuration parameters are each only used in a small number of predicates. This allows the tracing of problematic predicates to the configuration value. It also allows the changing of the configuration value without a large impact on the system.

We also assume that threshold predicate comparisons are very common during the execution of the system. Without many threshold predicate comparisons occurring, we could not try to determine the problematic configuration parameter. However, we assume that a threshold predicate comparison flopping is a rare and significant event. If flops occurred very frequently the ability to find the threshold predicate comparison that is most likely to flop or most recently flopped would not be helpful in identifying configuration errors.

Finally, we assume that the system has a state machine that is controlled by the threshold predicate comparisons. We also assume that the logic controlled by threshold predicate comparisons is contained within a computational unit and our approach can examine the computational units on an individual basis. We also assume that when the state machine changes an observable change will occur in the system and that the user

will be able to easily identify the change in the system's behavior. This allows the user to identify errors that are caused by the threshold predicate comparisons within the state machine. We also assume that the users will be able to mark the errors quickly and accurately when they occur to allow our system to help them determine the configuration option that caused the error. We also assume that since the users will be very familiar with the system, they will be able to easily identify the systems incorrect behavior.

3.3 Overall Approach

Figure 3.1 provides a high level overview of the approach. The approach contains two distinct parts: a static analysis and instrumentation component and a runtime component. The static analysis and instrumentation portion is responsible for identifying threshold predicate comparisons within the source code of a robot system. After identification, the approach instruments the predicates to expose the values in the comparisons during the execution of the system. The runtime portion of the approach uses the values from the instrumented portion of the code. During runtime, a user watches the system during normal operation and marks any abnormalities in the execution of the system. The runtime analysis calculates a score for each of the instrumented threshold predicate comparisons and provides a suggestion to the user on which configuration parameter could be the source of the problem.

Consider as an example a robot that delivers items between rooms in an office building. As part of the system there is a safety node that prevents the robot from traveling too quickly in the x direction due to instability. If the robot does change positions too quickly it will immediately stop until reset. The safety node has a configurable parameter that allows the users of the system to set the safety speed threshold.

The source code for the safety node is shown in Listing 3.1. The node contains three

methods, all of which serve a specific purpose in relation to how ROS programs are built. The *__init__* method sets up the class and creates the data structures necessary for the system to communicate with other nodes. The calls to the *rospy.Subscriber* and *rospy.Publisher* set up the communication channels. The *rospy.get_param* reads in the configuration parameter used to help control execution within the node and stores the value in the *self.threshold* class variable. The *t1callback* method receives messages for the node and makes comparisons to the configuration parameter's value. The *safety_thread* method is the main method of the node. It checks to see if the threshold has been exceeded, if so it publishes the value 1 to the publisher setup when the node was created.

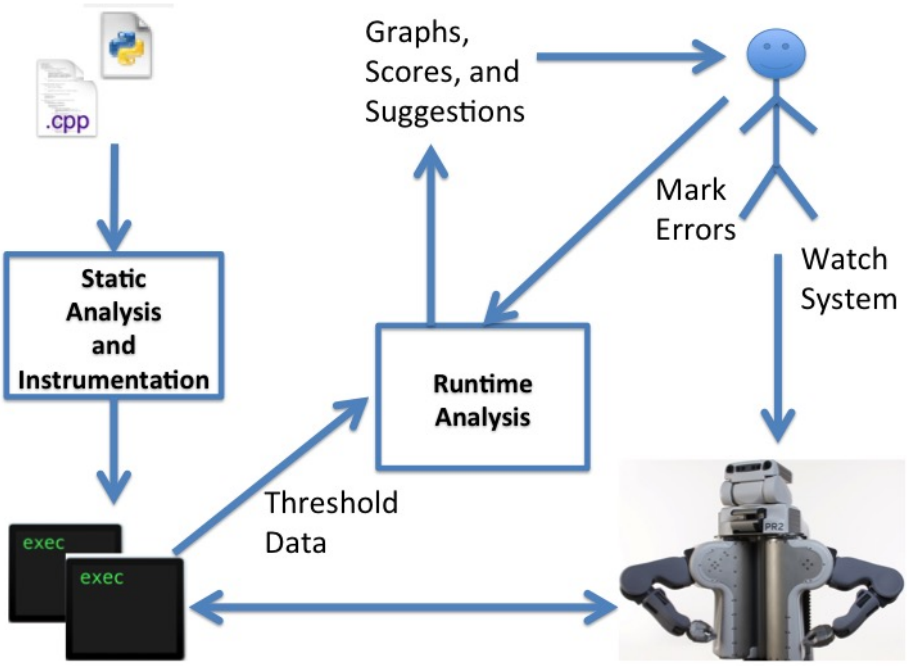


Figure 3.1: Overview of the whole analysis

Now consider the user notices that the robot stops too frequently, and they are not sure of the cause of the error. It may be because the robot is getting too close to people, the navigation system is misconfigured, a battery is too low, or any error within the system. Using our approach, the user would be able to run the robot system with all of the threshold predicate comparisons in the source code instrumented. The process of instrumenting and running the robot is fully integrated with the normal build and

launching procedures of the robot system. The threshold predicate comparison on line 22 is found and instrumented during this process along with many others in the system's source code.

The user observes the robot in operation and marks errors using the runtime tools we have created. The analysis then computes scores for all identified threshold predicate comparisons in the system. The threshold predicate comparison on line 22 in Listing 3.1 will be identified as the likely cause of the problem because it flopped recently. This error is a type I error, the configuration parameter was too low, causing the value of the predicate to change and the robot to stop. With this information the user can take corrective action for the problem. This is just a quick example of how the approach we developed can be used to identify and fix configuration errors that cause the robot to misbehave.

3.4 Static Analysis and Instrumentation

The first portion of our analysis involves the identification and instrumentation of threshold predicate comparisons within the source code of a robot system. This enables the runtime portion of our work to monitor the system and suggest solutions when the user indicates a runtime problem. This section will outline what is required for the analysis and instrumentation and provide an overview of the process.

The analysis identifies two sets of features in the source code and finds predicates that contain both of the features. The first set of features is configuration variables. These can include many different type of variables, but in our initial work configuration variables are all variables that have values assigned to them by the configuration options in the ROS system. Configuration variables are found using a syntactic search through the source code. The second set of features is predicate statements that cause data or control flow

decisions for exposing statements. These are found performing a programming slicing routine from exposing statements in source code. After these operations are performed, locations that contain both of the features are found and instrumented. The code is then transformed into the executable version.

ROS systems are broken into logical units or ”nodes.” These nodes control individual tasks within the system and communicate using topics and services. We assume that these individual nodes contain state machines that control the communication over these topics and manifest themselves as outward visible behavior. Because of this our approach only examines individual nodes and does not make any assumptions or perform analysis on the whole system.

3.4.1 Overview

As an example throughout this section consider the code from Listing 3.1. On line 22 the system contains a threshold predicate comparison. The program initializes the class variable *self.thresh* using ROSs API to read values from configuration files on line 8. On line 22 *self.thresh* is used in an *if* statement that directly determines whether or not the class variable *self.thresh_met* is set to True. In turn this *self.thresh_met* variable is used on line 15 to determine if a message is published to a ROS topic to change the behavior of the system. This short example will be used throughout the rest of the overview to show the methods used to find relationships and dependencies like this in general and then specifically in Python and C++ implementations of ROS nodes.

Listing 3.1: Sample Python node containing one threshold predicate comparison

```
1  import rospy
2  from std_msgs.msg import Int32
3  class ExampleNode:
4
5      def __init__(self):
6          self.old_msg = None
7          # Loading of configuration variable
8          self.thresh= rospy.get_param('theshold', 2)
9          rospy.Subscriber('pose_topic', Int32, self.t1callback)
10         self.stop_pub = rospy.Publisher('emergency_stop', Int32)
11         self.thresh_met = False
12
13     def safety_thread(self):
14         while not rospy.is_shutdown():
15             if self.thresh_met: # Decision relies on value set on line 23
16                 self.stop_pub.publish(1) # Exposing statement
17                 self.thresh_met = False
18
19     def t1callback(self, msg):
20         diff = msg.x - self.old_msg.x
21         self.old_msg = msg
22         if diff < self.thresh: # threshold predicate comparision
23             self.thresh_met = True # Dependent on comparison
24
25 if __name__ == '__main__':
26     ExampleNode().safety_thread()
```

The analysis and instrumentation consists of 5 steps as shown in Figure 3.2.

1. Identify "configuration variables" within the source code. These are variables that have been read from configuration sources. These can include variables that have constant values or are read from a specific configuration source. In our experiments we focus on configuration parameters read using the specific API methods ROS provides.

2. Identify "exposing statements." These are statements that communicate between individual computational units. In our experiments these are statements that publish messages or call a service in the ROS system. However, this could be expanded to include any function call that is well defined.

3. Identify predicates on which the exposing statements have a chain of data dependencies and control flow dependencies leading from the exposing statement to the predicate.

4. Identify the "threshold predicate comparisons." These are predicates identified in step 3 which contain a "configuration variable" that was found in step 1. Any predicate found to contain a threshold predicate comparison is marked for instrumentation.

5. Instrument identified threshold predicate comparisons to report runtime values to the dynamic portion of the analysis. Additionally, output statistics about the identified threshold predicate comparisons, and the source of the thresholds, the number of "steps" from the comparison to the exposing statement, the line number, and the file name.

3.4.2 Configuration Variables

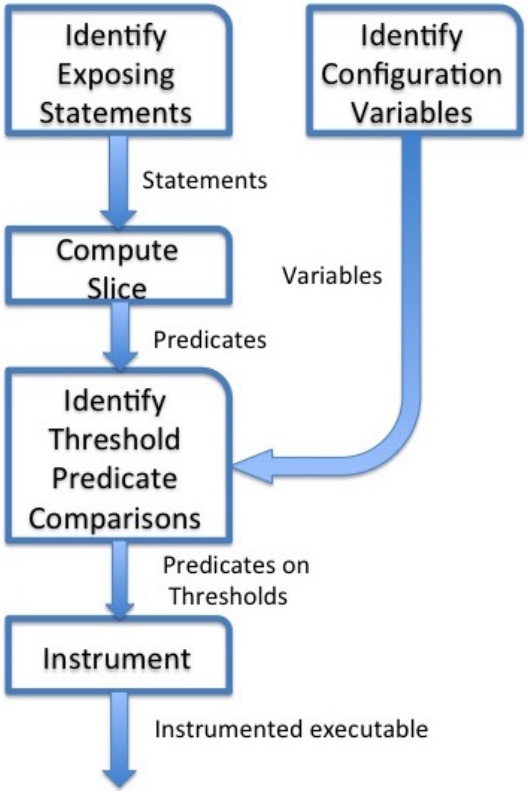


Figure 3.2: Overview of the analysis and instrumentation process

The first step in our analysis determines the configuration variables. There are many possible configuration variable sources. However, in this work we limit the sources to expressions that read values from the ROS parameter server. Identified configuration variables can be global, class, or local variables. Algorithm 1 displays the algorithm for finding the set of candidate variables in the source code used in our experiments. This step is a syntactic analysis that examines every statement in the source code. If the statement is an assignment statement, the expression on the right hand side of the statement is examined. If the expression contains a function call the function name is examined. If it matches the name of the methods that ROS provides the variable is saved as a configuration variable for later portions of the analysis. The rest of this section describes the possible sources of configuration variables.

Algorithm 1 Algorithm for identifying configuration variables.

```
1: procedure CONFIGURATION IDENTIFY(Program P)
2:     for Each statement in P do
3:         if statement is assignment then
4:             Examine right hand expression
5:             if Right hand contains ROS parameter reads then
6:                 Store variable as "configuration variable"
```

The most important set of configuration variables are those that are assigned values

from expressions which use functions to read values from the ROS parameter server. Any user of the robot system can changes these values. The values in these variables may not always maintain a constant value through execution, because of the possibility of dynamic reconfiguration, but they are among the most common way to configure robot systems. In addition to being important, these variables are also easy to find since ROS has specific calls to fetch variable values from the parameter server. In Listing 3.1 one variable will be identified as a configuration variable, on line 8, *self.threshold* is assigned a value from an expression identified as a configuration source.

Even though we only use configuration variables that obtained values from the parameter server in the later experiments, there are other types of expressions and variables that can be considered configuration variables. Some of these variables are from expressions that appear directly in source code and can only be modified through direct modification of the source code of a system. In addition to being important to system configuration, identification of variables of this type may help a developer identify variables that should be configurable through other means if they are important to the system running properly.

The of set configuration variables in source code include literals that appear within a predicate. The set of source code variables will also include variables that are guaranteed to maintain a constant value during execution. These include variables that only have one assignment statement and those that are assigned constants in a header file. These two groups of variables will have the same value throughout the code and are good candidates for threshold predicate comparisons because they represent are constant values used to make branching decisions within the code. They also are not too difficult to find as we can perform simple constant analysis to identify them.

Finally, some sources of configuration variables that may warrant exploration in the future are configuration variables that arise from special features of ROS systems. For

example, messages that are published with a constant value and will therefore always provide a constant value to any location receiving these values, or ROS message files with fields with defined constant values. These can be referenced from within nodes and can be added to the candidate pool of configuration variables. If either of these values appear in predicates they could easily be considered thresholds, but they are not included in our current work.

3.4.3 Exposing Statements

The next step in our analysis is the identification of "Exposing Statements." An Exposing Statement is a statement that communicates external data from a node to other nodes on the robot system. This statement may change the behavior of the robot system by the shared data changing the computation on other nodes. Most often this is achieved by communicating with a different process performing a different computational task for the robot. In our work we defined exposing statements as statements that cause a message to be published to ROS topic or call a ROS service. However, this analysis is designed to allow other types of statements to be considered an "exposing statement" for purposes of the analysis.

Algorithm 2 displays the method to find exposing statements in a given program. To find exposing statements in our analysis every statement in the source code is examined. The determination is made if the statement is one of the specified "exposing statement." If it is an "exposing statement" it is saved for later use in our analysis. In Listing 3.1 only one "exposing statement" is present. The analysis method knows that on line 16 the call *publish(1)* on the variable *self.stop_pub* is an exposing statement. This location is then stored for later usage. More details on how each method determines exposing statements can be found in the Python and C++ sections.

Algorithm 2 Algorithm for identifying exposing statements

```
1: procedure CONFIGURATION IDENTIFY(Program P)
2:     for Each statement in P do
3:         if statement is function call then
4:             Examine function call
5:             if If name matches function and is the correct type then
6:                 Store statement as "exposing statement"
```

3.4.4 Finding Dependent Predicates

This portion of the approach works to identify all predicates on which exposing statements have a chain of control and data dependencies from the exposing statement to the predicate. We determine these predicates by computing a static slice using the exposing statement as the criterion for the slice. During the computation of the slice we keep track of the "distance" from the exposing statement to the predicate containing the configuration variable. The distance is the minimum number of intermediate control flow and data flow dependencies visited along a path during slice creation between the predicate and the exposing statement. During the slicing operation any predicate encountered is saved for later user in the analysis as a control and data dependent predicate.

The slices are computed for the whole compilation unit. They cross data and control flow out of the local function scope in three cases. The first case is class variables, which are a common way to share information throughout the execution of robotic code. Wherever the value of a class variable appears in an expression, all assignment statements that store values in the class variable are considered data dependencies. The other non-local propagation methods involve the propagation of control flow and data flow across functions calls. When the slice computation reaches a point in the function where there are no more local control flow dependencies, it allows propagation of control flow dependencies to the sites where the function is called. For example consider Listing 3.2; if

the slice computation is examining line 8, the statement is not inside any other control flow constructs. At this point, the computation will add line 3 as a control flow dependency. Finally, when a function is called within a predicate, the return statements are marked for examination to determine if any predicates affect the value returned to that predicate. For example consider Listing 3.3; if the slice computation is examining line eight, the return statement at line 5 will be added to the round of statements to examine. The three cases were chosen due to their common usage within robotic control code. These three types of data and control flow often have a direct impact on which values are sent to exposing statements and at what point in execution they occur.

Listing 3.2: An example demonstrating control flow across functions

```
1 def foo(a, b):
2     if a < 42:
3         bar(b)
4     else:
5         # do work
6
7 def bar(value):
8     x = value * 2
9     publish(x)
```

Listing 3.3: An example demonstrating control flow across functions

```
1 def foo(a, b):
2     val = False
3     if a < 42:
4         val = True
5     return val
6
7 def bar(value):
8     if foo(a, b):
9         publish()
```

Algorithm 3 displays the procedure to compute the slice from an exposing statement and find the predicates that affect the behavior as per the exposing statement. At a high level the procedure iterates backwards through the control and data flow dependencies of the exposing statement until there are no more dependencies to visit or the maximum distance from the exposing statement has been exceeded. The visited statements constitute the slice and any predicate visited during the slice computation is saved as a control and data dependence to the exposing statement. The main work is done iterating through statements at the current distance and adding the control and data flow to the set of statements to be visited at the next distance unless the statement has already been visited. Once all statements at the current distance have been visited, the distance is incremented, the next statements are set to the current iteration, and the process continues until the maximum distance or the set of statements is empty.

Lines 2 through 6 set up the data structures for the procedure. The *predicates* variable is a set of all predicates visited during the execution of the procedure and the distance value at discovery. The *distance* is a count of the number of statements visited through control and data dependencies from the exposing statements. The *current* variable is the

Algorithm 3 Algorithm to produce slice for exposing statement and determine predicates with data and control dependence on an exposing statement

```
1:  procedure PREDICATE IDENTIFY(Program P, Exposing Statement E, maximum distance
    maxD)
2:      predicates ← ∅
3:      distance ← 0
4:      current ← {E}
5:      next ← ∅
6:      visited ← ∅
7:      while current ≠ ∅ and distance < maxD do
8:          for each statement s ∈ current do
9:              visited.add(s)
10:             if checkPredicate(s) then
11:                 predicates.add((s, distance))
12:             dependencies ← ∅
13:             dependencies.add(getControlDependencies(s))
14:             dependencies.add(getDataDependencies(s))
15:             for each dependency d ∈ dependencies do
16:                 if d ∉ current and d ∉ visited then
17:                     next.add(d)
18:         current ← next
19:         next ← ∅
20:         distance ← distance + 1
        return predicates
```

set of statements that are to be visited at the current distance. The *next* variable is the set of statements discovered at the current distance that need to be examined at the next distance. The *visited* variable keeps track of all the statements that have been visited by the procedure and on completion contains the slice with the criterion of the exposing statement.

After setup, the remainder of the procedure handles the iteration through control and data dependencies. Line 7 determines if the procedure should terminate after each iteration at the current distance. The procedure terminates if no more data or control dependencies were discovered or if the maximum distance has been exceeded. If neither of these cases arise, the procedure iterates through the set of statements found at the

previous distance and adds their control and data flow statements to the set of statements to be visited at the next distance. This continues until there is no more statements to visit or the maximum distance is exceeded.

During the iterative process, for each statement encountered, the procedure inserts the statement into the visited queue at line 9. If the statement is a predicate, the instruction and distance are saved for use in the later stages of the analysis. This is handled on lines 10 and 11 of the procedure. The next steps in the procedure (lines 12-14) determine the data and control flow dependencies for current statement. Each of the dependencies are added to the set of statements to examine at the next distance if the statement has not been visited or they are not in line to be visited in at the current distance. This logic is handled on lines 15-17. Finally, once all statements at the current distance have been visited, the procedure increments the distance, and sets the next round of statements to be the set to visit at the current distance and clears the next set of statements.

The method to determine which control flow dependencies to add for the current statement is straightforward. Algorithm 4 displays the method to determine which control flow dependencies to visit in the next iteration. First, the procedure determines which if statement and which loop statement contain the current statement. The call to *getif()* determines the innermost if that the statement is contained in. If more than one if statement contains the call will return the *if* statement dominated by all other *if* statements. If it is in both types of statements, the inner branching statement, the one dominated by the other statement is chosen. If the current statement is only in one type of branch, that branch is chosen. Finally, if it is not inside of any branching statements, all statements within the compilation unit that call the function are returned as candidates for the next iteration.

The next step in the procedure is determining all of the data dependencies for the current statement. Algorithm 5 displays the method of determining these dependencies.

Algorithm 4 Method to determine control flow dependencies to add for a statement

```
1:  procedure getFlowDependencies(Program P, Statement S)
2:      ifPred = getIf(S)
3:      loop = getLoop(S)
4:      if ifPred ≠ null and loop ≠ null then
5:          if dominates(ifPred, loop) then return {loop}
6:          elsereturn {ifPred}
7:      if ifPred ≠ null then return {ifPred}
8:      if loop ≠ null then return {loop}
9:      toReturn = {}
10:     for each calling location, L of function containing S do
11:         toReturn.add(L)
        return toReturn
```

As a first step the procedure needs access to the reaching definitions for the current function. The first set of data dependencies added are all of the reaching definitions for each variable in the statement being examined. Each statement that assigns to a variable uses within the current statement that is not changed before reaching is added. The second set of data dependencies that are identified involve class variables. If a variable used in the statement is a class variable, this procedure will add each location in the source code which writes to the class variable as a data dependency to be returned.

Algorithm 5 Method to determine data flow dependencies to add for a statement

```
1:  procedure getDataDependencies(Program P, Statement S)
2:      toReturn = {}
3:      for each variable, v ∈ S do
4:          for each function scoped reaching defintion, r of v do
5:              toReturn.add(r)
6:          if v is class or global variable then
7:              for each statement w, which writes to v do
8:                  toReturn.add(w)
        return toReturn
```

On lines 15 through 17 Algorithm 3 examines all of the control and data flow dependencies. If the statement has not been visited and is not in the set of statements to visit

at the current distance the statement is added to the set of statements to visit during the next distance iteration. If it has previously been visited or is queued to be visited at the current distance the statement is discarded. Finally, lines 18, 19, and 20 set up the procedure for the next distance's iteration before returning to line 7 to continue the procedure.

The procedure will terminate because it can only visit each instruction in the compilation unit once. It keeps track of visited instructions and will not attempt to revisit any instruction after it has been examined. This ensures that the backwards analysis will stop computation after visiting every statement in the compilation unit in the worst case.

3.4.4.1 Impact of Assumptions

We assume that all functions can be executed in any order due to the multi-threaded nature and callbacks of robot system design. This allows us to mark any location a class variable is written to as a data dependency of that class variable. We assume that functions may be called from outside of the compilation unit and we cannot determine if the source of the data in a function call is a configuration parameter, as they may be from an external source. This means that some threshold predicate comparisons may not be identified, as they are dependent on control and data flow between different compilation units. However, we can still consider any function call sites as control flow dependencies. We can also consider functions called directly in predicates as dependencies, because we know that the return value may have a direct effect on the result of the comparison.

3.4.4.2 Correctness

The procedure will correctly mark all predicates that are control and data dependencies from a given exposing statement that are less than the maximum distance. It iteratively visits statements in the reverse control and data flow dependency chains until it has

visited all identifiable dependencies in the compilation unit. As long as the creation of the control and data flow dependencies is correct the procedure will correctly visit them all. This includes the marking of locations within data where class variables are set. It will not correctly handle any modifications to the variables that come from outside of the compilation unit. This implies that if any part of the source code changes data from outside of the compilation unit (e.g. another part of the code has a reference to and changes variables of the analyzed node) it will not be analyzed and any potential threshold predicate comparisons will not be captured in the analysis.

3.4.4.3 Efficiency

The procedure has a runtime of $O(I)$ where I is the number of instructions in the compilation unit. Using E as the number of exposing statements in the source code, the runtime to find all of the slices from exposing statements is $O(EI)$. The runtime does not include the cost of computing the control and data dependencies as well as dominance values as these can be computed once and saved during other stages of compilation. The algorithm will only visit each instruction within the compilation unit at most once. There may be some inefficiency in the creation and querying of the required data flow and control flow structures, but as the size of the compilation unit grows the amount work to be down by the backward analysis should also grow at the same rate.

3.4.4.4 Example

This subsection examines how the procedure works in the example shown in Listing 3.1. The control and data flow dependencies for the example code are shown in Figure 3.3. In the graph the data dependencies are shown with dashed lines and control dependencies are shown with solid lines. Class variables and their relations are included in the graph (e.g. the value of *self.thresh_met* on line 15 depends on line 23). Also included are the

control flow dependencies on function calls (e.g. line 6-11 depend on the call to *__init__* on line 26). In the figure red nodes indicate an exposing statement and a green node indicates a threshold predicate comparison.

Table 3.1 displays the values present in the data structures in Algorithm 3 after each distance's iteration. For initialization line 16 is placed in the current iteration set, next is set to an empty set, and visited is set to an empty set. During the first iteration only line 15 is added as a control flow dependency because the exposing statement is contained in the *if* statement's body. Line 10 is also added as a data dependency as the publisher is created on this line. During the next iteration many interesting things happen. First, the predicate on line 15 is added to the set of predicates with a distance of 1. The statements on lines 11, 17, and 23 are all added as data dependencies because of the class variable used in the predicate on line 15. The while statement on line 14 is added as a control dependencies. During the next iteration the while statement's predicate (line 14) is added to the list of predicates with a distance of 2. The *if* statement on line 22 is added to the next iteration due to the statement on line 23 being inside of the statement. During the iteration at distance 3 the predicate on 22 is added to the list of predicates. All of the reaching definitions for *diff* are next to be

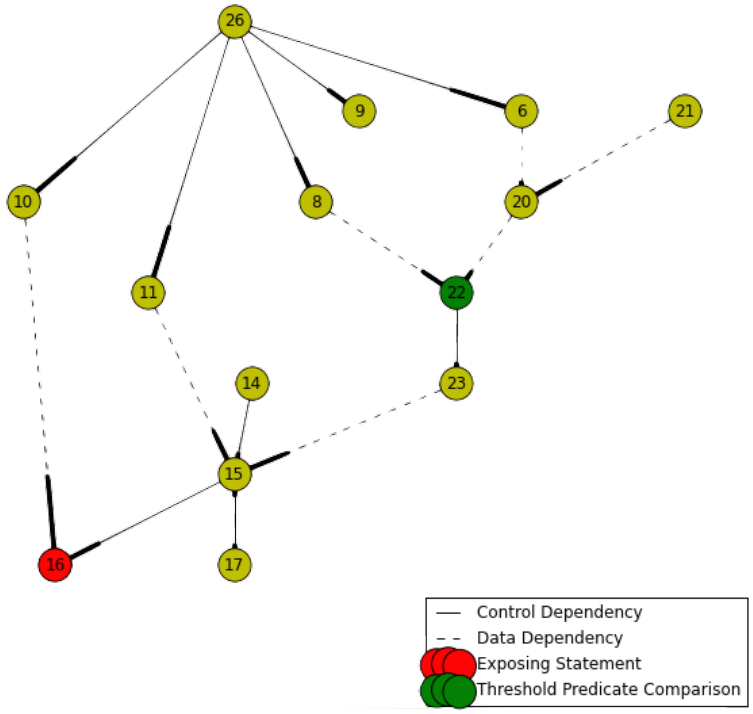


Figure 3.3: Control and Data dependencies in Listing 3.1

added. In this case only line 20 is added to the next iteration. In line 21 *self.thresh* is a class variable, because of this the statement on line 8 is added to the list to be examined during the next iteration. The final statement added at this distance is the control flow dependency of line 26, the if statement on line 25. At distance 4 the if statement on line 25 is added to the list of predicates at a distance of 4. Two data dependencies are added at this distance from the statement on line 20. They are both from the class variable *old_msg*, the variable is written to on line 6 and line 21. Finally, no dependencies are added for line 8 because the control flow dependencies have already been examined. On the next iteration (distance 5) no new control or data dependencies are discovered. At this point the iteration will stop and the procedure will exit.

The slice for the exposing statement in this example contains nearly the entire example program. This is because most of the functionality deals with the publish call and is to be expected. The predicates on line 15, 14, and 25 all provide a control or data dependency for line 16 and are identified by the procedure. This example gives an idea of how the method works during iteration backwards through all of the control and data flow dependencies.

3.4.5 Threshold Predicate Comparisons

Now that we have the predicates from program slices on the exposing statements and the configuration variables, we can identify the threshold predicate comparisons in the source code. This step identifies every location that should be instrumented to report values during the runtime analysis of configuration problems. To find threshold predicate comparisons, we examine every predicate identified in the program slices with respect to the exposing statements. We examine the values used in the expression of the predicate. If a value used in the predicate is a variable identified as a configuration variable or has

Table 3.1: Values present in Algorithm 3 after each distance on example shown in listing 3.1

Distance	current	next	visited	data dependencies	control dependencies	predicates and distance
Start	{16}	{}	{}	{}	{}	
0	{16}	{15, 10}	{16}	{10}	{15}	{}
1	{15, 10}	{11, 17, 23, 14, 26}	{16, 10, 15}	{11, 17, 23}	{14, 26}	{(15, 1)}
2	{11, 17, 23, 14}	{22}	{16, 10, 15, 11, 17, 23, 14, 26}	{}	{22}	{(15, 1), (14, 2)}
3	{26, 22}	{20,8, 25}	{16, 10, 15, 11, 17, 23, 14, 26, 22}	{20}	{25, }	{(15, 1), (14, 2), (22, 3)}
4	{20, 8, 25}	{21, 6}	{16, 10, 15, 11, 17, 23, 14, 26, 22, 20, 8, 25}	{21, 6, }	{}	{(15, 1), (14, 2), (22, 3), (25, 4)}
5	{21, 6}	{}	{16, 10, 15, 11, 17, 23, 14, 26, 22, 20, 8, 25, 21, 6}	{}	{}	{(15, 1), (14, 2), (22, 3), (25, 4)}
Done	{}	{}	{16, 10, 15, 11, 17, 23, 14, 26, 22, 20, 8, 25, 21, 6}	{}	{}	{(15, 1), (14, 2), (22, 3), (25, 4)}

a direct data dependency to one of the values without expressions that will modify the value a threshold predicate comparison has been found. For example any threshold that is modified by a different variable in a statement before appearing in the predicate it will not be used. We chose to exclude these types of configuration options because the error may be caused by the variable or expression that modifies the input parameter. For example the *self.thresh* will not be included in Listing 3.4 because the value of threshold is modified in an expression on line 6 before reaching the predicate. Support can be added

for constant valued arithmetic to be allowed on configuration variables, but currently these operations are not allowed. Our final step in the approach instruments the identified threshold predicate comparisons.

Listing 3.4: A configuration parameter that will not be included in threshold predicate comparisons

```
1 def __init__(self):
2     self.pub = rospy.publisher('topic', Int32)
3     self.thresh = rospy.get_param('threshold1')
4
5 def bar(value, coeff):
6     comparator = coeff * self.thresh
7     if comparator <   value:
8         self.pub.publish(42)
```

3.4.6 Instrumentation of Predicates on Thresholds

In the instrumentation step we achieve two tasks. The first task is creating an executable that reports a number of values from each predicate threshold comparison identified in the previous step. These values include the value of the threshold, the value of the comparator, the execution time, the result of the predicate, and the value of the overall boolean statement. The second task is creating basic information about all of the predicate threshold comparisons encountered. This information does not change during execution and is saved once at analysis time.

The static information contains the distance computed during backward analysis, file name, line number, and source of threshold in the threshold predicate comparison. In our implementation the source is the parameter name that was read to instantiate the value

of the threshold in the comparison. This may also include the numerical literal, header file, or assignment statement in some of the other uses. Information on the number of parameters read into the file and the number of exposing statements within the file is also included. A unique key in the static information identifies each threshold. This allows the static information to be paired with dynamic information produced during runtime. For example in Listing 3.1 the static information discovered by the approach would be similar to that found in Listing 3.5.

Listing 3.5: Static information for threshold predicate comparison in Listing 3.1

```
{
 "/home/ataylor/example_node/example_node.py:22:0": {
  "publishes": 1,
  "distance": 3,
  "file": "/home/ataylor/example_node/example_node.py",
  "param_reads": 1,
  "source": "threshold",
  "lineno": 22,
  "key": "/home/ataylor/example_node/example_node.py:22:0",
  "type": "parameter"
 }
}
```

Dynamic information published by our process include the unique key that matches with the key provided in the static information, current time, the predicate result, the threshold value, and comparison value for the identified threshold predicate comparisons. Having these values allow us to track the number of times the threshold changed values and trends in the values used in the predicate. The instrumented code creates a message each time it is executed. The message is created by either replacing expression in the

branching statement with a function call or inserting code to publish the value just before the branch is taken. More details about the methods to publish these values can be found in the respective implementation section. These values can either be saved to disk or processed by the runtime analysis portion in real time.

3.4.7 Requirements

The C++ analysis and instrumentation requires a version of LLVM and clang that have been customized to include the compiler passes that perform the analysis. The custom versions of LLVM and clang are available online [86]. Additionally, since nodes are often implemented across multiple files, the analysis must run after all files are present to access information that may be present in multiple files. This requires a linker that can support link time optimizations. This is accomplished by using the gold linker [87]. Also to make the build process integrate fully with ROS and allow link time optimizations to be run without any issue an installation of ROS is required. More details about the C++ implementation can be found in Appendix A.

To perform the Python identification, a valid Python 2.7 installation is required along with the source code of the robot system. The analysis works primarily on the Python standard library abstract syntax tree. It works primarily on classes contained in each individual Python file. In a few instances the analysis examines outside the scope of the class to perform analysis on common cases in which is it warranted. These exceptions and other details will be described in Appendix B.

3.5 Runtime Analysis

This section overviews the portion of our approach to identify problematic robotic thresholds while the robot system is operating. This portion of the approach has four

main requirements.

1. Collect and organize all threshold predicate comparison information from instrumented nodes running on the ROS system.

2. Allow the user to mark type I and type II errors and record these errors for analysis. type I errors indicate which threshold "flopped" when it should not have and type II errors indicate that a threshold is close to "flopping", but cannot quite reach the value needed.

3. Calculate scores for all instrumented threshold predicate comparisons at the time of a marked error. These scores represent estimates on which threshold is the cause of the type I or type II error.

4. Present the results to the user to enable the user to make informed decisions about which configuration parameters need to be adjusted.

3.5.1 Overview

Figure 3.4 displays the high level design of our runtime tools. The system containing instrumented threshold predicate comparisons will report the values and results each time any predicate is executed. This information flows to the data compilation and storage portion of our approach. The compilation and storage portion is responsible for translating data from the instrumented thresholds into a format that can be used by the rest of the approach. It also must store the data in a format that can be easily queried to get all of the recent values for each of the individual threshold predicate comparison locations. This portion of the system is composed of two ROS nodes to handle the translation and compilation. The data compilation and storage portion of the system also keeps track of the marks of type I and type II errors the user makes during

observation and operation of the system. This allows other portions of the approach to query as needed for the information required to compute scores and display information for the user. The GUI tool is responsible for displaying information about the current threshold values in the system and scores, suggestions and graphs for any marked type I and type II errors in the system. It reads information from the compilation and storage portion of the system and also makes requests to the computation tools to get scores for marks to display the results. The computation tools are responsible for computing scores for type I and type II error marks for all of the threshold predicate comparisons in the system and offering suggestions to raise and lower the thresholds. The runtime tools are designed to be used as the robot system is operating or on a recorded system execution trace. The remainder of this subsection will highlight the features of each of the parts of the approach.

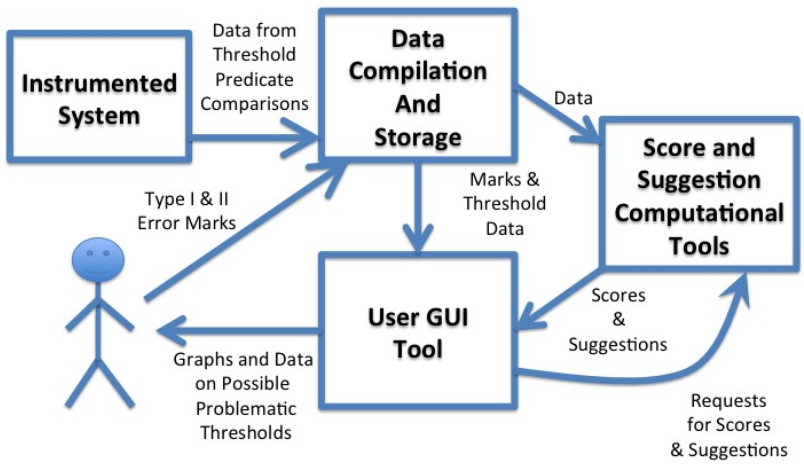


Figure 3.4: Components of the runtime approach

The distributed nature of ROS means that different nodes will be producing data from instrumented predicates simultaneously. This data must be compiled and organized to enable usage by the computational and GUI portions of the approach. The Python and C++ instrumented code are in different formats. This is because the Python code publishes to a specified topic while the C++ code

outputs all information to the ROS system log. This part of the approach transforms the different data formats of and extracts and prepares it for use in the other portions of the approach. The two different sources of data are parsed and time indexed. The data must be stored in a format that can be easily queried and grouped by threshold predicate comparison location.

The data compilation and storage portion of the approach was designed to allow users to be able to mark errors in a variety of ways such as using a GUI, a keyboard, or a controller. The data compilation and storage portion of our approach listens for user marks, and the marks can be sent using any tool as long as the compilation receives the mark in the correct format.

The score and suggestions computational tools are responsible for computing and returning scores and suggestions for each predicate threshold location. These calculations require the data for each of the instrumented locations stored by the approach as well as the type and time of the error marking. This section also requires the static information for each threshold location provided by the static analysis portion of the approach. From this information it produces scores and suggestions for each instrumented location in the source code. The scores indicate which threshold predicate comparison location was the most likely source of the type I or type II error. The tools will provide a suggestion to either "raise" or "lower" the potential problematic parameter to alleviate the problem. The functions that produce the scores for each type of error provide customizable parameters. More details on the parameters and the calculations of the scores and suggestions can be found in Subsection 3.5.5.

Finally, the GUI tool of the approach allows the user of the robotic system to view information about the parameters currently loaded into the system and view the scores and suggestions provided by the rest of the approach. The tool allows the user of the system to view the marks, scores, and suggestions in real time while operating the system

or on recorded data. This tool provides two views. The first view contains information on all of the thresholds currently in the system and what their last known values are. The second view contains information about the threshold predicate comparisons at the specific points that were marked by the user. In this view the user can see the ranking scores for the error as well as graphs of the values used in the comparison through time. The objective is that these graphs will give the user a better idea of what was going on in the time period before the error in the threshold predicate comparisons that the system suggested for them to change.

3.5.2 Implementation Details

All of the runtime tools are written in Python and contain approximately 1500 lines of code. In order to run the tools wxPython version 3.0.2.0 [88], the pandas Python data framework version 0.15.2+ [89], and matplotlib version 1.4.3 [90] are required.

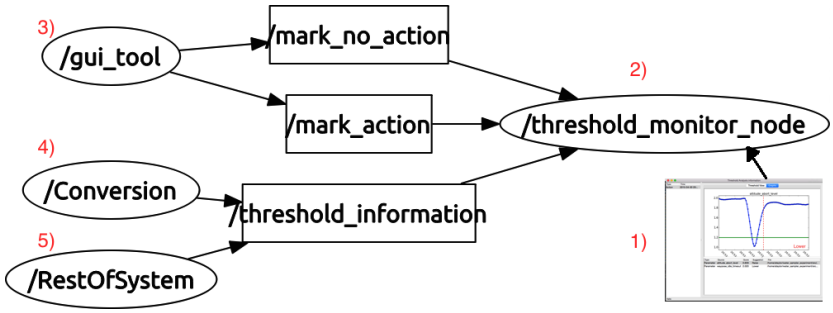


Figure 3.5: Rosgraph of the system during threshold monitoring

The tools are spread out across two ROS nodes, the GUI code, and a Python file containing the utilities to calculate the scores and suggestions. A graph of the ROS system is contained in Figure 3.5

Two nodes handle the translation and aggregation of the data produced by the instrumented threshold predicate comparison locations. In the figure the *threshold_monitor_node* (2) handles the aggregation and the *Conversion* node (4) handles the translation of C++ data. The aggregation node is setup to allow the user to use the recorded system traces

for analysis after the system has ran. This is possible because ROS provides tools to easily store, retrieve, and playback runtime data. The GUI (1) holds references to the aggregation node and the computational tools. The users provide marks using the *gui_tool* node (3), which provides the ability to mark using the mouse.

3.5.3 Data Compilation and Storage

The data from instrumented threshold comparisons is coming from many different locations in two different formats on two separate ROS topics. The data must be standardized and indexed for usage by the GUI and the computation tools. Two ROS nodes are in charge of this task in the runtime portion of our approach. The first node filters the data produced by C++ nodes from the ROS logging stream and transforms it into the format expected by the system. This is known as the *cpp_conversion node*. The second node performs the parsing, compilation and storing of the data. This is the *threshold_analysis* node.

The C++ implementation of instrumentation sends the data from threshold predicate comparisons to the ROS console logging stream. The data must be extracted from the logging stream before it can be processed and compiled along with the data from the instrumented Python code. The *cpp_conversion* node subscribes to the logging stream and transforms the data into the same format as the Python messages. The data is published on the same topic as the Python data to allow the processing occur in one location within the threshold_analysis node. The messages in the logging stream from instrumented C++ locations contain the prefix "threshold_information." This allows the quick identification of the messages from other normal logging messages. Once identified, the message's prefix is removed and the time the message was created is prepended to the front of the string along with a comma. This brings the message in to conform to the expected format

of the *threshold_analysis* node. The modified message is republished onto the correct topic to be processed and stored.

The *threshold_analysis* node is responsible for collecting all of the data from instrumented threshold predicate comparisons. It subscribes to the "threshold_information" topic and listens for string messages on the topic. Upon message arrival the node parses and stores the message data. The node expects messages in the form of a comma separated string. An example message with annotations can be found in Figure 3.6. The first entry in the message is the creation time stamp. This time stamp is converted into a Python datetime object so pandas can easily index the data in the message. The next entry in the message is the unique key that identifies which instrumented threshold predicate comparison created the message. This unique identifier matches one in the static yaml files created during the instrumentation portion of the approach. The next entry contains the overall result of the predicate calculation as a boolean. The remainder of the entries are key value pairs separated by a colon. Each of these key value pairs are parsed and stored. These key value pairs include the value of the threshold, the value of the changing variable in the comparison, and the value of the result of the single predicate in the source code. At the time of message receipt the node calculates if the predicate and full boolean statement "flopped." The predicate flopped if the value has changed since the last received comparison. The floppiness of the comparison as well as the time to the last time it flopped is stored along with the other values from the message.

Figure 3.6: An annotated message from an instrumented threshold predicate comparison

The parsed data is stored as a dictionary of lists until requested by the computational

tools or the GUI for use. Each field in the message from a predicate is stored in its own list. This allows for the cheap addition of data. Once the data is requested by another portion of the approach a pandas Dataframe object is created. The Dataframe object is a 2D tabular data structure that is indexed by time. Each of the fields in the message is a column in the Dataframe. The object also allows quick and easy grouping of the data by the unique ID of the threshold predicate comparison that is the source of the data row. Once the request is processed the node caches the Dataframe and the dictionary is cleared of all of the data in the dataframe. Upon the next request the new data is appended to the cached data frame, the frame is returned to the requesting source, and the full frame is cached.

3.5.4 Recording Marks of Type I and Type II Errors

The system listens for user indications of type I and type I errors on two ROS topics. There is one topic for each error type. Each of the topics expects a message that contains the ROS system timestamp of when the user indicated an error. This allows any node in the system to signal to the approach that an error has occurred. For our experiments we implemented a node that allows marking errors using the keyboard. We also implemented a GUI tool that allows the marking of errors. This GUI tool is shown in Figure 3.7. In addition, creating a node that listens to controller inputs or any form of user feedback is as simple as writing a small node that publishes to the correct topics. The *threshold_analysis* node consumes any marks on the topic and they are stored until the

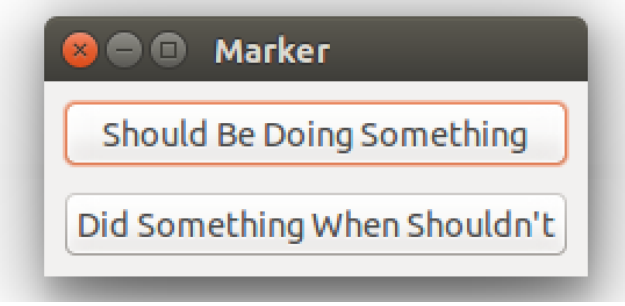


Figure 3.7: GUI tool to mark type I and type II errors during runtime

GUI tool or the computational tools request them.

3.5.5 Score Calculation

The GUI or any other tool can request the calculation of scores for type I errors and type II errors and request threshold suggestions at any point in time. All of these utilities will return a score value for each predicate on threshold that has been reported to and stored by the *threshold_analysis* node. To compute the scores, the functions need the time of the error mark, the dataframe from the *threshold_analysis* node, the static information produced during instrumentation, and values for parameters used in the calculation of the scores. While computing scores, the threshold data is grouped by unique predicate ID and a score is calculated for each group. For both type I and type II errors, a lower score produced indicates that that threshold predicate comparison was the likely cause of the type I or type II error. A dictionary containing each ID and the score associated with the ID is returned by the functions. The suggestion function requires the dataframe, the time of marked error, and the type of the error to return a "raise" or "lower" suggestion for the threshold value. It again computes one for each threshold predicate comparison in the data and returns them as a dictionary.

Type I errors are caused by the predicate changing from one boolean value to the other at the incorrect time (flopping). Therefore, to find the most likely cause of the type I errors we favor predicates that flopped most recently before the marking. To Equation 3.1 displays the calculation of Type I scores in full detail. accommodate thresholds that flop often from overwhelming the scores, we enable an adjustable term that includes the number of previous flops. We also enable the distance to the exposing statement to be included in the score calculation since we may want to provide more weight to flops that occur much closer to the exposing statement compared to those at a larger distance. α,

β, and γ are all configurable parameters that allow tuning of the scoring. t is the time since the last flop of the threshold predicate comparison, n is the total number of flops of that comparison, and d is the distance to the exposing statement. The values are raised to scaling factors and multiplied together to allow tweaking of the values. If we do not want to include part of the equation we can raise the value to 0.0 to set that part of the equation to 1.0 and it will have no effect on the score value. In the experiments we used values of $\alpha = 1.0, \beta = 1.0$, and $\gamma = 0.0$ for these parameters. These values were chosen through trial and error, and produced good results on the experimental data. In this setup the time since the last flop and the number of previous flops both carry weight in the calculations of the type I scores. We chose to exclude the distance because the distances between C++ and Python implementations would lead to the analysis being heavier in favor of the Python analysis. Values of 1.0 for the other two allowed them to have a balanced effect on the produced score.

$$score = t^{\alpha} * n^{\beta} * d^{\gamma} \tag{3.1}$$

Type II errors are caused when a threshold is preventing the robot system from progressing in its operation. For scoring, we favor the predicates that are the closest to flopping. These are the ones with the smallest normalized distance to flop, since the values closest to the threshold are the ones most likely to flop. First, the computation selects all the data from instrumented threshold predicate comparisons within the past x number of seconds. In our experiments we used 12 seconds, after experimentation on how long users waited to mark errors and trial and error on which values produced the best results. The full equation for calculating the type II scores is shown in 3.4. The computation scales all data in that time range to be between 0 and 1 using equation 3.2 where max is the maximum value during all the execution for the threshold and value

in the comparator, *min* is the minimum value during all execution for threshold and comparator, and v is the value of the value in the predicate. The distance from flopping is calculated using Equation 3.3. t and c are the scaled values from Equation 3.2 of the threshold and value being compared to the threshold respectively. We need to use the absolute value to ensure that positive and negative results from the difference to not cancel each other out. The mean is needed to standardize a different number of messages in the time period between threshold predicate comparisons. As with the type I errors, we also take into account the previous number of flops (n) in the time period and the distance (d) to the exposing statement from the static analysis approach. The full equation is shown in equation 3.4. Again α, β, and γ are configurable parameters. In Equation 3.4 *distance* is the value computed in equation 3.3. For our experiments we used values of $\alpha = 1.0$, $\beta = 1.0$, and $\gamma = 0.0$ respectively. Again, the values are raised to scaling factors and multiplied together to allow tweaking of the values. If we do not want to include part of the equation we can raise the value to 0.0 to set that part of the equation to 1.0 and it will have no effect on the score value. Again, these values were determined using trial and error on the experimental data. These values again take into account how close the predicate is to flopping and the number of previous flops. We chose a value for previous flops to punish nodes that have recently flopped. If the node has already flopped there is a good chance that it is not the cause of the problem. We would expect the system's behavior to change if the threshold that flopped resolved the issue. We did not include the distance because the difference between the distance values of the Python and C++ implementations. The C++ analysis works at a lower level and therefore has on average higher distance values.

$$value = \frac{(v - min)}{max - min} \tag{3.2}$$

$$distance = mean(abs(t - c)) \tag{3.3}$$

$$score = distance^{\alpha} * n^{\beta} * d^{\gamma} \tag{3.4}$$

As an example, consider a parameter with a value of 0.5. In the time range the parameter was compared against values of 1.0, 0.7, 0.6, and 0.55. The maximum value during all of the system's runtime is 4.0 and the minimum value is 0.0. After scaling, the parameter has a value of 0.125 and the comparison values are 0.25, 0.175, 0.15, and 0.1375. These scaled values are ready to be used in Equation 3.3. After subtraction the values are 0.125, 0.05, 0.025, and 0.0125. The final result from Equation 3.3 is 0.069 by taking the square of the mean of the subtraction squared. Finally, the distance value is ready for Equation 3.4 to get the final score for the type II error.

The algorithms for calculating the suggestions are shown in Algorithms 6 and 7. If the error was identified as a type I error, we determine the time of the last flop and look at the values in the predicate before the flop. If the values of the comparator are greater than the threshold the threshold should be "lowered" because the values are higher than the threshold and should have remained higher to prevent the flop. If the values were less than threshold we return that the threshold should be "raised", because the values should have remained below the threshold. Type II errors are similar, except we examine values from the current time. If the value of the comparator is greater than the threshold the threshold should be "raised." The threshold should be raised because the values above the threshold need to be met by the threshold to cause the flop. If the value is less than the comparator, we state that the threshold should be "lowered." This logic is implemented in the get_suggestion function.

Algorithm 6 Algorithm to calculate suggestions for type I errors

```
1: procedure TYPE I SUGGESTION
2:     Find the last "flop"
3:     Subtract the last n comparisons before the flop and sum them to find the result r.
4:     if r > 0 then
5:         The parameter should be "lowered."
6:     else
7:         The parameter should be "raised."
```

Algorithm 7 Algorithm to calculate suggestions for type II errors

```
1: procedure TYPE II SUGGESTION
2:     Subtract the last n comparisons before the mark and sum them to find the result r.
3:     if r > 0 then
4:         The parameter should be "raised."
5:     else
6:         The parameter should be "lowered."
```

3.5.6 GUI Tool

To tie all of the components together and provide the user with a tool to view and analyze thresholds, we implemented a GUI that allows the viewing of marks, threshold information, and suggestions on what to change in the configuration. The tool is created to support a user of the system that may not know the full implementation details of the system. It is designed to help the user identify and correct configuration errors dealing with parameters in the system. Experts trying to use the system or tune its parameters can also use it

The tool can be seen in Figure 3.8. On startup the tool ingests the static files created during instrumentation. It also starts the threshold_analysis node to gather data from instrumented threshold predicate comparisons or load a saved system trace. The node also listens for user marks of type I and type II errors on the correct topics. The GUI tool requests and receives any new marks on an adjustable timer with a default value of 1 second. Upon receiving a mark, the scores for that type of error are calculated for

every threshold predicate comparison that has data the threshold_analysis node's store. These results are presented for the user to make determinations on which configuration parameters need to be changed.

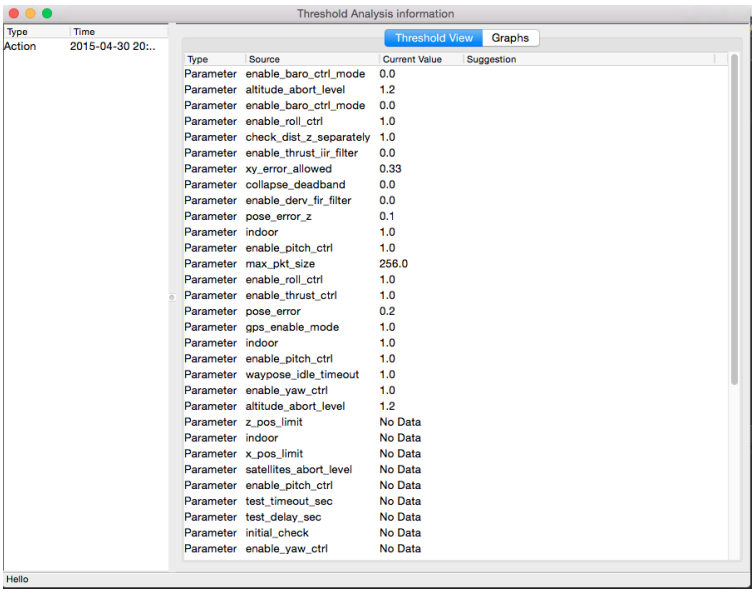


Figure 3.8: The GUI for runtime analysis.

The tool contains two main views: the overview and the graph view. On startup the tool displays the overview. In this mode every user mark is displayed on the left side of the screen as they happen. On the right of the window is a table containing all seen threshold predicate comparisons. It contains information on the source code location, the parameter used to populate the threshold, and the most recent value of the threshold. When a user mark is selected on the left the GUI grabs data from the threshold_analysis and calculates the score values for each threshold predicate comparison. The comparisons that return the best scores are highlighted in the right. An example of this view with a selected mark and highlighted results is shown in Figure 3.9. This view is useful for gaining an overall view of all of the parameters and their values loaded into the system. It also provides a quick way to view the threshold predicate comparisons the approach has marked as the possible problem for a given user mark.

The graph view displays information about individual comparisons once the user has selected a type I or type II error mark. This view is shown in Figure 3.10. Once the user has selected a mark the tool calculates scores for all threshold predicate comparisons collected

by the threshold_analysis node. The results of the score calculations are displayed on the lower half of the right side of the tool. They are sorted with the lowest scores at the top to show which locations are the most likely to have caused the error. Also displayed on the bottom are the source code location, the score, the source of the threshold, and the suggestion. The graph contains the threshold, the value being compared to the threshold, and the user mark. The threshold appears as a solid green line. The comparison value appears as the varying blue line. The time of the user mark appears as the dashed red line. The configuration parameter that loaded the value appears as the title of the graph.

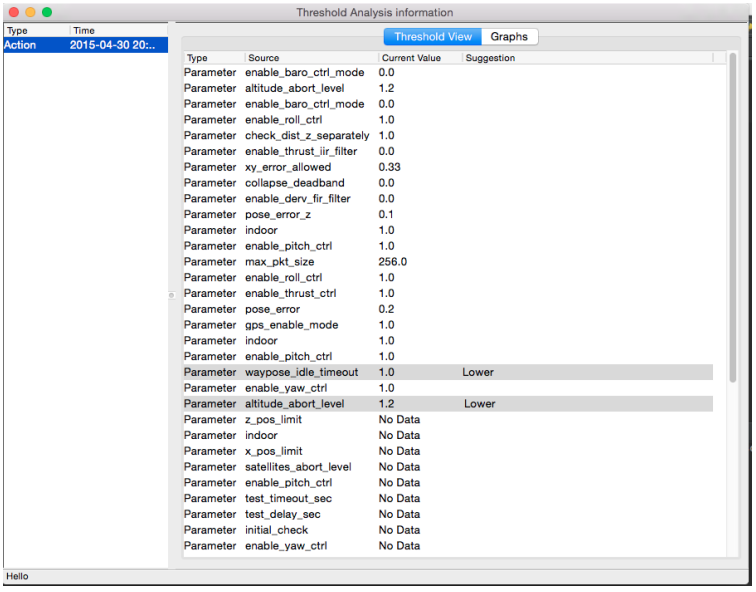


Figure 3.9: The GUI with a selected user error mark and highlighted threshold predicate comparisons

Selecting an entry from the bottom will cause the top view to display a graph of the predicates values through time. In the graph the threshold will appear as a solid horizontal line, the comparison values as a time varying line, and the user mark will show as a vertical line. Also displayed on the graph is the suggestion on whether to raise or lower the configuration value.

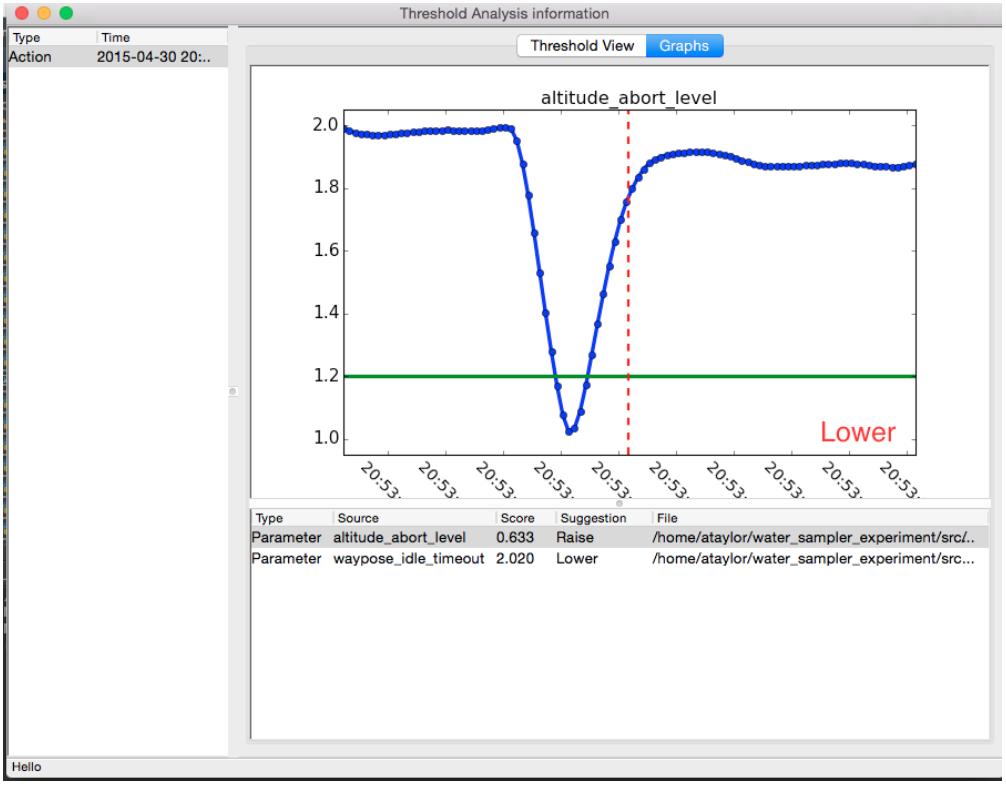


Figure 3.10: The view containing the graph and predicate threshold scores

Chapter 4

Validation

In this chapter we perform two studies to validate that the approach behaves as expected and can correctly identify threshold predicate comparisons. We perform the analysis on 35 nodes that we designed as unit tests to confirm that the approach correctly identifies threshold predicate comparisons. These nodes are written in both Python and C++. The second validation study examines more in depth on how the analysis performs on 20 nodes. We determine if each predicate in the source code was correctly identified by the analysis. Ten of the nodes chosen were written in Python and the other ten nodes were written in C++. These nodes were selected from the open source code repositories examined in Chapter 5. The approach passed both the test nodes and the manual analysis.

4.1 Test Nodes

To validate that the static analysis can correctly identify threshold predicate comparisons we created 25 test nodes that contain threshold predicate comparisons. These nodes vary in the type of dependencies, type of exposing statements, and distance between the threshold and the exposing statement. In addition there are some nodes in which the

exposing statement is separated from threshold by a number of functional calls. We also created 10 additional test nodes that do not contain threshold predicate comparisons, but have features that may be perceived as threshold predicate comparisons. We implemented each of the test nodes in both C++ and Python and tested them in both languages.

More information and the results of the test cases can be found in Table 4.1. X's indicate that the feature is present for testing. In the table a data dependency means that the threshold predicate comparison affects the data in the exposing statement. A control dependency means that the execution of the exposing statement's execution depends on the threshold predicate comparison. Function call means that there is one or more function call between the threshold predicate comparison and the exposing statement. Class variables indicates that a dependency flows through a class variable. LOC indicates the lines of code in the C++ implementation of the node. Finally, distance is the number of steps in the dependencies between the threshold predicate comparison and the exposing statement. Expected Threshold Predicate Comparisons shows the number of threshold predicate comparisons that we expect the approach to find in the source code. Each of the tests produced the expected results.

4.2 Hand Inspection of Selected Nodes

In this section we perform a manual analysis of 20 ROS nodes to determine if the approach correctly identifies threshold predicate comparisons within the source code. We selected 10 Python and 10 C++ nodes to perform this analysis on from the source code repositories examined in Chapter 5.

To perform the analysis we manually:

1. Located all predicates in a node's source code.

Table 4.1: Information on test cases

Test	Data Dependency	Control Dependency	Function Call	Class Variables	LOC	Distance	Expected Threshold Predicate Comparisons	C++	Python
P1		x			41	0	1	1	1
P2		x			45	1	2	2	2
P3		x			48	1	1	1	1
P4	x				45	1	1	1	1
P5	x				49	1	1	1	1
P6	x				43	1	1	1	1
P7		x	x		45	2	1	1	1
P8		x	x		46	2	1	1	1
P9		x		x	61	1	1	1	1
P10	x			x	63	1	1	1	1
P11	x	x	x	x	70	2	2	2	2
P12	x	x		x	73	2	1	1	1
P13		x	x		50	2	2	2	2
P14		x	x		53	3	1	1	1
P15	x		x		51	3	1	1	1
P16	x	x	x		50	3	1	1	1
P17	x	x			53	3	1	1	1
P18		x			56	4	1	1	1
P19	x	x	x	x	72	4	2	2	2
P20	x	x	x	x	66	4	2	2	2
P21		x	x		61	4	1	1	1
P22	x	x	x	x	65	5	2	2	2
P23	x		x		48	3	1	1	1
P24	x		x	x	69	3	1	1	1
P25	x		x	x	66	4	1	1	1
N1	x				44	0	0	0	0
N2		x			48	0	0	0	0
N3	x	x		x	55	0	0	0	0
N4					47	0	0	0	0
N5	x				50	0	0	0	0
N6					46	0	0	0	0
N7				x	54	0	0	0	0
N8			x	x	49	0	0	0	0
N9		x			54	0	0	0	0
N10	x	x		x	59	0	0	0	0

2. Determined if there is an exposing statement with a data and control dependence on the predicate.

3. Ran the analysis and determined if every predicate is correctly marked.

4. Recorded any discrepancies between the manual and the automated analysis.

4.2.1 Results

Table 4.2 shows the results from the manual analysis of the selected nodes. The analysis performed very well at identifying threshold predicate comparisons. There was one case for the analysis of a Python node that we considered a true negative (we missed its detection). Listing 4.1 displays the missed threshold predicate comparison. On line 6 the *self.RANGE_MINIMUM* is used in one of the statement's comparisons to determine if the distance is within a minimum value. The boolean value is used on line 11 in the predicate to help determine if a message should be published. Currently, the Python implementation does not support comparisons outside of the branching statement. The C++ implementation will find threshold predicate comparisons that follow this pattern. We could add support for this type of analysis if it appears to be widespread in Python source code. All of the other Python nodes did not have any errors in the identification of threshold predicate comparisons. There were no errors in the identification of threshold predicate comparisons in the C++ nodes examined.

Listing 4.1: The threshold predicate comparison missed during the hand analysis

```
for laser_distance in self.scan.ranges:
    (base_distance, base_angle) = self.laser_to_base(laser_distance,
            laser_angle)
    angle_diff = abs(base_angle - target_angle)
    will_get_closer = angle_diff < (np.pi / 2.0)
    is_too_close = (base_distance < self.RANGE_MINIMUM) and (laser_distance
            > self.scan.range_min)
    if will_get_closer:
        self.speed_multiplier = min(self.speed_multiplier,
                (base_distance - self.RANGE_MINIMUM) / self.SLOWDOWN_RANGE)
    if is_too_close and will_get_closer:
        blocked = True
        block_reason = (base_distance, base_angle, angle_diff)
        break
```

In the manual analysis a number of numerical constants appeared in the predicates. These offer opportunities to highlight values that could possibly be parameterized to make the system more configurable to the end user. Additionally, there were values in source code can be shown to be constant that were present in predicate comparisons. This shows that identifying constant valued variables within the source code that are used in predicates can highlight values that are candidates for being parameterized.

Finally, the C++ analysis encountered a larger number of predicates compared to the Python analysis. The first number in the predicate column is the number of predicates including outside files linked into the node. This highlights the low level nature of the C++ analysis. It works on LLVM bitcode and therefore there are more predicates involved. The second number is the number of predicates only in the source files directly involved in the nodes source code.

Table 4.2: Results of the manual analysis of 20 ROS nodes

Node Name	Type	Predicates	Hand TPC	Analysis TPC	False Positives	True Negatives
safey_node.py	Python	30	4	4	0	0
move_trans.py	Python	13	11	11	0	0
jsk_teleopjoy.py	Python	31	4	4	0	0
basic_commands.py	Python	16	0	0	0	0
pr2_move_base.py	Python	18	0	0	0	0
calibrate.py	Python	28	2	2	0	0
joint_trajectroy_file_playback.py	Python	25	0	0	0	0
base_controller.py	Python	7	3	3	0	0
sr_friction_compensation.py	Python	19	9	9	0	0
move_base_straight.py	Python	18	**6**	**5**	0	**1**
virtual_cage	C++	280/26	7	7	0	0
move_base	C++	3390/578	11	11	0	0
cob_teleop_v2	C++	2383/380	13	13	0	0
sac_inc_ground_removal_node	C++	1726/128	3	3	0	0
gripper_controller	C++	1524/130	6	6	0	0
kalman_filter	C++	397/37	0	0	0	0
hector_mapping	C++	2868/257	12	12	0	0
smile_detector	C++	446/192	10	10	0	0
crop_surveying	C++	331/13	1	1	0	0
nao_path_finder	C++	2445/263	17	17	0	0

4.3 Conclusions

In this chapter we performed two studies to validate that our approach behaves as expected. First, we analyzed 35 Python and 35 C++ nodes specifically created to confirm that the approach can identify a number of threshold predicate comparison patterns. Twenty-five of the nodes of each type contain threshold predicate comparisons that have varying control and data flow dependencies. An additional 20 nodes, 10 C++ and 10 Python, contain no threshold predicate comparisons, but have a structure that is somewhat similar. The analysis correctly identified all of the nodes without issue.

After verifying the approach using test nodes, we turned our attention to nodes found in real world robotic systems. We examined 10 C++ and 10 Python nodes from a variety of robot systems and determined if the approach correctly classified each predicate within the code. We found one instance in which the Python approach did not

identify a threshold predicate comparison. This was caused by the predicate containing the comparisons appearing outside the branching statement. This deficiency could be addressed easily. We found no errors in the C++ identification. However, in both approaches there were a number of constant value variables and literals that could provide further sources of configuration options in future work.

Chapter 5

Study of Thresholds in ROS Systems

This chapter examines occurrence threshold predicate comparisons across a multitude of different open source robot system code repositories. We want to answer how common threshold predicate comparisons are in a wide variety of systems. We also want to characterize how many files in a system contain threshold predicate comparisons and how many unique parameters are used to give the thresholds values. We want to know how common they are to show that our analysis can be used on many robot systems. We also want to show that parameters are unique enough that problematic threshold predicate comparison can be traced back to an input parameter than can be easily changed.

The source code was examined at two different levels of granularity The first, the "metapackage" level, contains many related ROS packages that perform some common functionality for a robot system or perform a common robot task. The second grouping aggregates related meapackages packages used to control a full robot system. Information about the statistics gathered for each meta package or group can be found in Table C.1.

5.1 Metapackages

The first three repositories we examined are from the NIMBUS lab. These include the metapackage created to control our Ascending Technology UAVs [91], our water sampling robot [10], and our crop surveying robot [71]. The other source of repositories for analysis s ROS metapackages listed on the main ROS wiki at http://www.ros.org/browse/list.php. We did not analyze all of the repositories on the list; we filtered them using two passes to get to the final list of repositories we analyzed. To get through the first round of selection the metapackage needed to be: 1) a core functionality to a robot system as judged by us, such as navigation, or be a metapackage directly involved in the operation of a robot, 2) contain source code and not just configuration files and 3) be compatible with ROS Indigo.

The second filtering pass constrained the time allowed for the analysis of each metapackage. Each metapackage was required to compile in 10 minutes and 25 minutes using our modified compiler analysis. Additionally, if there were configuration errors when trying to compile the metapackage we allowed 15 minutes to resolve any issues or the metapackage was discarded.

After filtering, 101 ROS metapackages were used in our final analysis. A complete list of the surveyed metapackages and information about their size and makeup is shown in Table C.2. There is a total of 769,264 lines of code in the repositories analyzed with an average of 7616 lines of code per metapackage. The median lines of code for all repositories is 2637. One project contained 224,018 lines of code and one metapacakge contained only 28 lines of code. On average each metapackage contained 44 files. The median file count is 19. The range of files is from of 1 to 606 files in one metapackage. This selection of repositories gives us a large variety of code sizes and also a large variety of robot systems including ground robots, aerial vehicles, and functionality such as

navigation and control. After combining metpackages there are a total of 52 systems to examine. The mean number of lines of code is 14768.8 per system. The mean number of files per system is 85 the median is 42.5 and the minimum and maximum remain the same.

5.2 Results

Statistics from the analysis can be found in Table 5.1. In total 538 threshold predicate comparisons were found in all of the examined repositories. Of the threshold predicate comparisons found, 437 were found in C++ and header files and 83 were found in Python files. On average each package contained 5.3 threshold predicate comparisons. Of the 101 repositories examined 36 of the 101 (35.6%) contained threshold predicate comparisons. In the 36 repositories the average occurrence of threshold predicate comparisons rose to 14.9 per metapackage. This provides evidence that thresholds are used in robot systems, and that if they are present there are on average more than 14 locations in code in which a threshold controls the data or occurrence of communication between ROS nodes. This works out to around 0.69 threshold predicate comparisons per 1000 lines of code when all of the examined code is included. When including only metapackages that contain threshold predicate comparisons this value is 2.85 threshold predicate comparisons per 1000 lines of code.

On average 0.9 files per package contained threshold predicate comparisons. Examining only the 36 repositories with threshold predicate comparisons present they appeared in an average of 2.4 files. In repositories with thresholds there were an average of 42.9 files. When examining each of the 36 systems the average percentage of files in the metapackage that contained threshold predicate comparisons was 9.3%. This shows that thresholds are usually confined to a few files in the metapackage and these are usually in

the higher-level decision making files. In the repositories with predicates on thresholds, 7.7 out of the 14.9 threshold predicate comparisons had a unique source of threshold setup. This means that each threshold only is used on average in 2 locations. Also, if a threshold predicate comparison is highlighted as a possible error source it will only be used in one other location on average. This shows that being able to identify the threshold predicate comparison that caused the problem has a very high chance of exposing the setup parameter that caused the problem.

Information on the analysis at the system level can be found in Table 5.2. We found the same number of predicates on thresholds and the average number per system rose to 10.3. Out of the 51 systems we analyzed 25 (48.0%) have threshold comparisons present in the source code. These system all contain robot systems which have a tasteful higher level functionality. All of the systems that do not contain any thresholds are from single metapackage sources. All of the systems that were spread across more than one metapackage contain thresholds. However, single metapackage systems make up 80% of the systems examined.

When only considering systems with threshold predicate comparisons, the mean number of threshold predicate comparisons per system is 21.5. The thresholds in the predicates come from an average of 11.1 unique sources. These counts do not include threshold predicate comparisons that are present in the base ROS functionality that many of these systems use to accomplish tasks. As an example, the controls and navigation stack can add up to 31 unique thresholds to any robot using those two ROS provided tools. So these statistics may under approximate the true number of threshold predicate comparisons that will be present in the source code of an operating robot system.

In the systems with threshold predicate comparisons, an average of 3.48 files contained thresholds, but there were an average of 95.24 files per system. On a system-by-system basis the average percentage of files in a system with predicates on thresholds is only

6.5%. This again shows the concentration of threshold predicate comparisons in a few files on a robot system.

On average the repositories took 39.24 seconds to compile without running the link time analysis. When running our analysis the average compile time rose to 111.57 seconds. This resulted in a time factor of 2.6 across all of the repositories. The minimum time factor was 1.8 and the maximum was 19.8. This means that the analysis will take on average over twice as long to compile with a few repositories taking much longer to compile. Part of the slowdown may result from other link time optimizations being run at the same time as the analysis as part of llvm. Further work can be done to completely isolate the analysis and speed up the runtime.

5.3 Discussion

Predicates on thresholds are present across a number of ROS code repositories and in almost half of the systems we examined. When they are present in a system there are often many threshold predicate comparisons present in the source code. This shows that thresholds setup by parameters are often used to determine if an exposing statement executes and what data is present in the statement at execution. In addition often systems use some of the provided ROS base functionality while using their robot. This base ROS functionality also adds additional threshold predicate comparisons as evidenced by the control and navigation libraries that could possibly add 55 threshold predicate comparisons to any system using them.

The small percentage of files with thresholds shows that they are often concentrated in a few nodes in a package. Also the low ratio of threshold predicate comparisons to unique threshold sources shows that often a parameter is used to set a value that is only used in one or two locations to determine the behavior of a node across a system. This

uniqueness provides an opportunity to highlight which source of threshold should be changed to fix an identified predicate on threshold without the need to worry about affecting a whole range of other locations or decision points.

The analysis does add to the time required to compile the nodes to run any robot system. However, the average compile time only increased by a factor of 2 to 3. In some rare cases the compile time increased by 19 times. However some of the additional compile time may be a result of the other link time optimizations being run by llvm. It may be possible to remove these optimizations and only incur the cost of the analysis and instrumentation alone.

Table 5.1: Results of the analysis on repositories.

Name	Time	Time Factor	Threshold Predicate Comparisons	C++	Python	Unique	Files
airbotix ros package	18.5	2.0	2.0	0.0	2.0	2.0	2.0
app manager	8.2	2.0	0.0	0.0	0.0	0.0	0.0
apriltags tracking	35.6	2.1	0.0	0.0	0.0	0.0	0.0
ar tracking	79.8	2.3	0.0	0.0	0.0	0.0	0.0
arm nav	16.0	2.0	0.0	0.0	0.0	0.0	0.0
asctec base	129.5	2.3	49.0	35.0	14.0	38.0	10.0
asctec mav pacakge	61.8	5.9	0.0	0.0	0.0	0.0	0.0
baxter robot	29.5	2.0	0.0	0.0	0.0	0.0	0.0
bwi from texas	241.5	1.9	0.0	0.0	0.0	0.0	0.0
calibration	61.9	2.6	0.0	0.0	0.0	0.0	0.0
calvin ros stack	35.1	2.2	0.0	0.0	0.0	0.0	0.0
careobot contro	71.3	2.6	17.0	17.0	0.0	9.0	4.0
careobot evnironment perception	0.7	2.0	0.0	0.0	0.0	0.0	0.0
careobot manipulation	55.3	3.5	0.0	0.0	0.0	0.0	0.0
careobot navigation perception	24.2	3.3	11.0	11.0	0.0	8.0	1.0
careobot perception	50.2	2.2	0.0	0.0	0.0	0.0	0.0
cob command tools	35.2	3.0	142.0	142.0	0.0	28.0	2.0
cob common	27.2	2.0	0.0	0.0	0.0	0.0	0.0
cob driver	221.0	2.6	9.0	9.0	0.0	7.0	4.0

Continued on next page

Table 5.1 – continued from previous page

Name	Time	Time Factor	Threshold Predicate Comparisons	C++	Python	Unique	Files
cob external	21.9	2.0	0.0	0.0	0.0	0.0	0.0
cob robots	15.3	2.0	0.0	0.0	0.0	0.0	0.0
control toolbox	8.1	2.5	0.0	0.0	0.0	0.0	0.0
crazyflie ros stack	17.4	2.0	1.0	0.0	1.0	1.0	1.0
crop surveying	132.5	2.3	3.0	3.0	0.0	3.0	3.0
func maninulators	44.8	2.1	0.0	0.0	0.0	0.0	0.0
graft	22.9	2.6	0.0	0.0	0.0	0.0	0.0
grizzly robot	26.0	2.0	0.0	0.0	0.0	0.0	0.0
hector arm	19.8	2.2	0.0	0.0	0.0	0.0	0.0
hector diagnostics	60.2	2.3	15.0	15.0	0.0	13.0	2.0
hector navigation	58.3	2.3	15.0	15.0	0.0	13.0	2.0
hector slam	49.7	2.6	12.0	12.0	0.0	11.0	2.0
hector turtlebot	17.5	2.1	0.0	0.0	0.0	0.0	0.0
icart mini	34.1	2.5	0.0	0.0	0.0	0.0	0.0
jaco robot arm	38.3	4.4	10.0	10.0	0.0	3.0	2.0
jsk control	61.1	2.0	4.0	0.0	4.0	1.0	1.0
jsk smart apps	13.9	2.0	0.0	0.0	0.0	0.0	0.0
jsk travis	2.3	2.0	0.0	0.0	0.0	0.0	0.0
kobuki	52.5	2.3	0.0	0.0	0.0	0.0	0.0
kobuki soft	22.9	2.1	1.0	1.0	0.0	1.0	1.0
mav ros	102.9	5.8	7.0	7.0	0.0	4.0	4.0
maxwell	14.9	2.0	0.0	0.0	0.0	0.0	0.0
motoman	53.7	2.5	0.0	0.0	0.0	0.0	0.0
nao camera	3.4	2.0	0.0	0.0	0.0	0.0	0.0
nao extras	24.2	3.7	26.0	26.0	0.0	17.0	2.0

Continued on next page

Table 5.1 – continued from previous page

Name	Time	Time Factor	Threshold Predicate Comparisons	C++	Python	Unique	Files
nao interaction	18.8	2.0	0.0	0.0	0.0	0.0	0.0
nao robot repo	42.3	2.0	0.0	0.0	0.0	0.0	0.0
nao ros	20.6	2.2	0.0	0.0	0.0	0.0	0.0
nao sensors	6.9	2.0	0.0	0.0	0.0	0.0	0.0
nao virtual	14.3	2.0	0.0	0.0	0.0	0.0	0.0
nao viz	14.6	2.0	0.0	0.0	0.0	0.0	0.0
naopi bridge	42.5	2.0	0.0	0.0	0.0	0.0	0.0
nav2 platform	19.5	2.1	0.0	0.0	0.0	0.0	0.0
navigation stack	172.6	2.8	45.0	45.0	0.0	25.0	8.0
neo robot	39.1	2.4	3.0	3.0	0.0	3.0	3.0
next stage	16.6	2.0	0.0	0.0	0.0	0.0	0.0
novatel spann	17.8	2.0	2.0	0.0	2.0	2.0	1.0
ocs library	68.3	2.7	3.0	3.0	0.0	3.0	2.0
p2 os robot	31.7	3.1	18.0	0.0	0.0	14.0	2.0
people tracking ros	65.3	3.0	24.0	24.0	0.0	8.0	3.0
pepper robot for stuff	14.9	2.0	0.0	0.0	0.0	0.0	0.0
pr futre	0.7	2.0	0.0	0.0	0.0	0.0	0.0
pr2 common	20.9	2.0	0.0	0.0	0.0	0.0	0.0
pr2 colibraiton	33.5	2.4	0.0	0.0	0.0	0.0	0.0
pr2 common actions	33.3	2.7	0.0	0.0	0.0	0.0	0.0
pr2 delivery	3.6	2.0	0.0	0.0	0.0	0.0	0.0
pr2 doors	14.1	2.0	0.0	0.0	0.0	0.0	0.0
pr2 kinematics	49.4	1.9	0.0	0.0	0.0	0.0	0.0

Continued on next page

Table 5.1 – continued from previous page

Name	Time	Time Factor	Threshold Predicate Comparisons	C++	Python	Unique	Files
pr2 navigation	69.5	2.2	7.0	7.0	0.0	4.0	3.0
pr2 pbd	0.6	2.0	0.0	0.0	0.0	0.0	0.0
pr2 precise trajectory	13.9	2.0	3.0	0.0	3.0	2.0	1.0
pr2 self test	38.7	2.3	1.0	1.0	0.0	1.0	1.0
pr2 surrogate	11.6	2.7	3.0	3.0	0.0	2.0	2.0
pr2 apps	33.2	19.8	27.0	27.0	0.0	13.0	2.0
rail ceiling	11.8	2.5	1.0	1.0	0.0	1.0	1.0
rail pick and place library	110.4	7.8	0.0	0.0	0.0	0.0	0.0
rail segmentation	18.9	2.6	0.0	0.0	0.0	0.0	0.0
realtime tools	5.2	2.1	0.0	0.0	0.0	0.0	0.0
robitician ric	28.6	2.1	0.0	0.0	0.0	0.0	0.0
robot rescue	32.5	2.7	0.0	0.0	0.0	0.0	0.0
ros concert	109.6	2.1	0.0	0.0	0.0	0.0	0.0
ros control	42.4	2.2	0.0	0.0	0.0	0.0	0.0
ros controllers	47.2	2.7	10.0	10.0	0.0	6.0	3.0
ros create driver	19.1	2.0	15.0	0.0	15.0	6.0	1.0
ros darwin	15.3	2.0	0.0	0.0	0.0	0.0	0.0
ros descartes	55.7	2.2	0.0	0.0	0.0	0.0	0.0
ros filter library	9.0	2.5	0.0	0.0	0.0	0.0	0.0
ros universial robot	27.1	2.1	0.0	0.0	0.0	0.0	0.0
rqt pr2 dashboard	2.4	2.0	0.0	0.0	0.0	0.0	0.0
segbot	28.1	2.1	0.0	0.0	0.0	0.0	0.0
segbot apps	30.5	3.2	0.0	0.0	0.0	0.0	0.0

Continued on next page

Table 5.1 – continued from previous page

Name	Time	Time Factor	Threshold Predicate Comparisons	C++	Python	Unique	Files
shcunk modular	68.5	2.9	9.0	9.0	0.0	3.0	1.0
sr demo	0.6	2.0	9.0	0.0	9.0	3.0	1.0
sr manipulation	0.7	2.0	0.0	0.0	0.0	0.0	0.0
sr utils	17.4	2.0	0.0	0.0	0.0	0.0	0.0
turtlebot	18.2	2.1	0.0	0.0	0.0	0.0	0.0
turtlebot apps	73.8	2.3	5.0	1.0	4.0	4.0	3.0
turtlebot arm	48.5	4.1	0.0	0.0	0.0	0.0	0.0
turtlebot create	19.0	2.0	15.0	0.0	15.0	6.0	1.0
turtlebot interactions	18.7	2.0	0.0	0.0	0.0	0.0	0.0
uos tools	27.4	2.2	5.0	0.0	5.0	4.0	1.0
water sampler	26.4	2.1	9.0	0.0	9.0	9.0	4.0
mean	39.2	2.6	5.3	4.3	0.8	2.8	0.9
median	27.2	2.1	0.0	0.0	0.0	0.0	0.0
std	41.0	1.9	16.2	15.8	2.9	6.2	1.6
min	0.6	1.9	0.0	0.0	0.0	0.0	0.0
max	241.5	19.8	142.0	142.0	15.0	38.0	10.0
sum	3963.2	263.8	538.0	437.0	83.0	278.0	87.0
only threshold mean	53.4	3.1	14.9	12.1	2.3	7.7	2.4
only threshold median	38.5	2.4	9.0	3.0	0.0	4.0	2.0
only threshold std	47.3	3.0	24.5	24.8	4.4	8.3	1.9
only threshold min	0.6	2.0	1.0	0.0	0.0	1.0	1.0

Continued on next page

Table 5.1 – continued from previous page								
Name		Time	Time Factor	Threshold Predicate Comparisons	C++	Python	Unique	Files
only max	**threshold**	221.0	19.8	142.0	142.0	15.0	38.0	10.0
only sum	**threshold**	1922.9	111.1	538.0	437.0	83.0	278.0	87.0

Table 5.2: Results of analysis when repositories grouped as robot systems.

Name	Time	Time Factor	Threshold Predicate Comparisons	C++	Python	Unique	Files
airbotix ros package	18.5	2.0	2.0	0.0	2.0	2.0	2.0
app manager	8.2	2.0	0.0	0.0	0.0	0.0	0.0
apriltags tracking	35.6	2.1	0.0	0.0	0.0	0.0	0.0
ar tracking	79.8	2.3	0.0	0.0	0.0	0.0	0.0
arm nav	16.0	2.0	0.0	0.0	0.0	0.0	0.0
asctec base	129.5	2.3	49.0	35.0	14.0	38.0	10.0
asctec mav pacakge	61.8	5.9	0.0	0.0	0.0	0.0	0.0
baxter robot	29.5	2.0	0.0	0.0	0.0	0.0	0.0
bwi from texas	241.5	1.9	0.0	0.0	0.0	0.0	0.0
calibration	61.9	2.6	0.0	0.0	0.0	0.0	0.0
calvin ros stack	35.1	2.2	0.0	0.0	0.0	0.0	0.0
care o bot	590.8	2.7	188.0	188.0	0.0	55.0	12.0
controls	102.9	2.5	10.0	10.0	0.0	6.0	3.0

Continued on next page

Table 5.2 – continued from previous page

Name	Time	Time Factor	Threshold Predicate Comparisons	C++	Python	Unique	Files
crazyflie ros stack	17.4	2.0	1.0	0.0	1.0	1.0	1.0
crop surveying	132.5	2.3	3.0	3.0	0.0	3.0	3.0
func maninulators	44.8	2.1	0.0	0.0	0.0	0.0	0.0
graft	22.9	2.6	0.0	0.0	0.0	0.0	0.0
grizzly robot	26.0	2.0	0.0	0.0	0.0	0.0	0.0
hector	205.6	2.4	42.0	42.0	0.0	37.0	6.0
icart mini	34.1	2.5	0.0	0.0	0.0	0.0	0.0
jaco robot arm	38.3	4.4	10.0	10.0	0.0	3.0	2.0
jsk applications	77.3	2.0	4.0	0.0	4.0	1.0	1.0
kobuki	52.5	2.3	0.0	0.0	0.0	0.0	0.0
kobuki soft	22.9	2.1	1.0	1.0	0.0	1.0	1.0
mav ros	102.9	5.8	7.0	7.0	0.0	4.0	4.0
maxwell	14.9	2.0	0.0	0.0	0.0	0.0	0.0
motoman	53.7	2.5	0.0	0.0	0.0	0.0	0.0
nao	187.6	2.3	26.0	26.0	0.0	17.0	2.0
nav2 platform	19.5	2.1	0.0	0.0	0.0	0.0	0.0
navigation stack	172.6	2.8	45.0	45.0	0.0	25.0	8.0
neo robot	39.1	2.4	3.0	3.0	0.0	3.0	3.0
next stage	0.0		0.0	0.0	0.0	0.0	0.0
novatel spann	17.8	2.0	2.0	0.0	2.0	2.0	1.0
ocs library	68.3	2.7	3.0	3.0	0.0	3.0	2.0
p2 os robot	31.7	3.1	18.0	0.0	0.0	14.0	2.0
people tracking ros	65.3	3.0	24.0	24.0	0.0	8.0	3.0
pepper robot for stuff	14.9	2.0	0.0	0.0	0.0	0.0	0.0

Continued on next page

Table 5.2 – continued from previous page

Name	Time	Time Factor	Threshold Predicate Comparisons	C++	Python	Unique	Files
pr2	325.5	4.0	41.0	38.0	3.0	22.0	9.0
rail robots	141.1	6.6	1.0	1.0	0.0	1.0	1.0
robitician ric	28.6	2.1	0.0	0.0	0.0	0.0	0.0
robot rescue	32.5	2.7	0.0	0.0	0.0	0.0	0.0
ros concert	109.6	2.1	0.0	0.0	0.0	0.0	0.0
ros create driver	19.1	2.0	15.0	0.0	15.0	6.0	1.0
ros darwin	15.3	2.0	0.0	0.0	0.0	0.0	0.0
ros descartes	55.7	2.2	0.0	0.0	0.0	0.0	0.0
ros filter library	9.0	2.5	0.0	0.0	0.0	0.0	0.0
ros universial robot	27.1	2.1	0.0	0.0	0.0	0.0	0.0
segbot	58.6	2.7	0.0	0.0	0.0	0.0	0.0
sr robots	18.7	2.0	9.0	0.0	9.0	3.0	1.0
turtlebot	178.2	2.7	20.0	1.0	19.0	10.0	4.0
uos tools	27.4	2.2	5.0	0.0	5.0	4.0	1.0
water sampler	26.4	2.1	9.0	0.0	9.0	9.0	4.0
mean	75.9	2.6	10.3	8.4	1.6	5.3	1.7
median	37.0	2.3	0.0	0.0	0.0	0.0	0.0
std	99.1	1.0	28.1	27.9	4.1	11.2	2.8
min	0.0	1.9	0.0	0.0	0.0	0.0	0.0
max	590.8	6.6	188.0	188.0	19.0	55.0	12.0
sum	3946.5	131.9	538.0	437.0	83.0	278.0	87.0
only threshold mean	110.3	2.8	21.5	17.5	3.3	11.1	3.5
only threshold median	68.3	2.4	9.0	3.0	0.0	4.0	2.0

Continued on next page

Table 5.2 – continued from previous page

Name	Time	Time Factor	Threshold Predicate Comparisons	C++	Python	Unique	Files
only threshold std	127.1	1.2	37.8	38.6	5.5	14.1	3.1
only threshold min	17.4	2.0	1.0	0.0	0.0	1.0	1.0
only threshold max	590.8	6.6	188.0	188.0	19.0	55.0	12.0
only threshold sum	2757.3	70.3	538.0	437.0	83.0	278.0	87.0

Chapter 6

Applying the Proposed Approach

As a final study of our approach we performed several experiments to help us determine the characteristics of predicate threshold comparisons on running systems, the ability of users to identify robotic problems, and the ability of our approach to identify which configuration parameters are the source of type I and type II errors. In this chapter we discuss the results from experiments performed on three separate robot systems that were analyzed and instrumented with the approach presented in this work. The three systems examined include an Unmanned Areal water sampler, a ground robot performing a navigation task, and an Unmanned Areal Vehicle locating and capturing an image of a person.

6.1 Research Questions

For each of the experiments in this chapter we aim to answer the following research questions (RQ).

RQ1: What are the runtime characteristics of the instrumented predicate threshold comparisons?

To answer **RQ1** we examine numerous statistics about the runtime characteristics of the instrumented predicate threshold comparisons. We want to examine the frequency that threshold predicate comparisons occur during execution. If they are not frequent enough then the instrumentation and analysis of their execution would not be useful in diagnosing and offering solutions to runtime problems. We also want to determine if threshold predicate comparisons appear equally throughout the mission or if some portions of the missions have a greater frequency of predicate threshold comparisons. For our approach to be useful it is important that at any time an error is marked there are a number of threshold predicate comparisons to examine to determine if they are responsible for the errors. We also want to determine if individual predicates appear throughout all of the execution or only in small portions of the systems operation. If predicates only appear in small portions, it is an indication that they are more important than thresholds that appear throughout the execution at that point in time. We also want to measure the ratio of predicate threshold comparisons in the runtime execution trace to those identified during the static analysis. If the many parameters that are read are not used during runtime this immediately works to reduce the size of the space the user must search through to fix the configuration error.

RQ2: How common are flops?

RQ2 examines one of the fundamental assumptions of our process. We assume that threshold predicate comparisons rarely "flop" and they maintain the same truth value for large periods of time during execution. When they do change values it signals a change in the robot system. If predicates flop often throughout execution and the robots behavior does not change often then the two are not as connected as we assume. Additionally, if a flop is a common occurrence, the ability to find the

last predicate to flop or the predicate that is about to flop is not as useful to help diagnose problems in the robots configuration space.

RQ3: What errors do users highlight and when do they highlight those errors in a robot system?

RQ3 examines how users identify the errors we introduce as part of our studies to the robot system. This includes how long it takes them to mark error, which errors they mark, and at what frequency they are marked. For our approach to correctly rank the thresholds, we need the user to be able to mark the errors soon after they become noticeable and mark the correct type of error. If too much time passes, the state of the instrumented thresholds may change by a significant amount from when the error occurred. If they do not mark the correct type of error, the system will rank the instrumented predicates using the incorrect routine that may lead to very different results.

RQ4: How well does our analysis highlight errors that arise from misconfigured parameters?

RQ4 examines how well the approach we developed can identify the modified parameters. To determine how well the approach works we define the rank score, *rs*, as seen is Equation 6.1, where *rank* is the numerical position of the threshold in the suggested thresholds sorted in ascending order and *total_thresholds* is the total number of threshold predicate comparisons present in the execution trace. We calculate this score for every threshold predicate comparison present in the system execution trace. A score where the analysis correctly identifies the problematic threshold as the most likely to cause the problem is 1.0 and a score of 0.0 indicates the analysis performed very poorly or the threshold was not present up to that point during execution.

$$rs = 1 - \frac{rank - 1}{total\_thresholds} \tag{6.1}$$

As an example, consider that at some time t the approach has identified four threshold predicate comparisons. The user has marked a type II error and the approach has given them scores of $c_0 = .4$, $c_1 = 1.6$, $c2 = .01$, and $c_3 = .2$. In sorted order, the threshold predicate comparisons are c_2, c_3, c_0, and c_1. This would correspond to a rank score of $c_0 = .5$, $c_1 = .25$, $c2 = 1.0$, and $c3 = .75$.

In our experiments, we will examine how well the analysis identifies the modified threshold using the rank score when the user marks a location. However, because the user will not always correctly identify problems, we will also examine the rank scores produced by the analysis throughout the whole system execution time. Finally, this question will also examine the output of the analysis will produce when an error is introduced that is not from a threshold.

6.2 Setup

This next section will describe the setup of the experiments. This includes the systems used, the participants in the experiments, the instructions given to the participants, the design of the mission, the changes made to the system, and what information was recorded.

6.2.1 Systems

We used three separate robot systems for the studies. The systems used include an Unmanned Aerial water sampler, a ground robot performing a navigation task, and an Unmanned Aerial Vehicle capturing an image of a person. Each of the systems highlights

the different capabilities of robot systems and aims to show the wide applicability of our approach.

The first system we performed experiments on is the water sampler developed by the NIMBUS lab [10]. This system has been developed to autonomously fly to a selected location, descend to insert the sampling tube into the water, collect water samples, and return the samples back to the launch location. The system contains around 11,000 lines of code. 290 variables are populated with values from configuration parameters in the source code. Our analysis identified 58 threshold predicate comparisons in the system.

The second system is an iRobot Create [92] navigating a course using the ROS navigation implementation [8]. The Create can successfully navigate a course containing obstacles and stop in a predefined 1.0 m x 1.0 m square. The system contains around 25,000 lines of code, reads configuration parameters into 201 variables, and contains 50 threshold predicate comparisons.

The final system is a UAV that contains a camera and can location and capture the image of a person after performing a simple search for them. The robot will take off and find the person using facial smile recognition. The system contains around 27,000 lines of code, 252 variables are populated with configuration parameters and the static analysis found 88 threshold predicate comparisons in all of the analyzed source code.

6.2.2 Participants

The participants for the experiment have a familiarity with the robotic systems, but were not familiar with the details or implementation of the systems. Each of the participants has worked on robotic systems, but had not worked on the specific systems used in the experiments. All of the participants are graduate and undergraduate students in the NIMBUS lab.

6.2.3 Process

Before each trial we provided the users with a number of instructions. First, we informed them about the trial and what tasks they needed to perform. Next, we gave them a high level description of the system and planned mission. To give them an impression of what the mission and system are supposed to look like, we demonstrated the mission three times. During the missions we instructed the participants to pay special attention to the sequence and timing of events in the mission.

The users had the ability to mark the two separate types of errors during the missions. We provided instructions on what constituted type I and type II errors. We demonstrated how to mark the two types of errors by selecting a specific key on the keyboard. They were given the chance to test out marking the tools during training.

6.2.4 Treatments

We aimed to create four system treatments for each robot system. Two of the treatments would be performed on configuration parameters that can produce type I and type II errors. The other two treatments introduced an error with a fault in the source code and offered a control treatment with no errors. The control treatment examines how the user marked errors during the course of normal operation. The final treatment lets us explore the type of predicates that are highlighted when the error is not a result of configuration, but is instead a result of a fault in the source code.

To select the configuration parameters to mutate, we examined the identified threshold predicate comparisons and selected ones that would 1) produce the desired type of error and 2) produce an error that would manifest in the system in a way that a user would recognize with minimal training as an error condition. After identification, we tested each of the treatments to ensure that they fit the criteria, and the treatment did not cause

the robot to behave in an unsafe manner. The treatment to source code was also chosen in a similar manner. We examined the source code system and identified locations where we could modify statements to produce a visible but safe change the robots behavior.

The users examined the robot running each treatment where each treatment consisted of a system with one treatment. The treatments were provided in random order and the user did not know which one was being administered. During each mission we recorded the following:

1. System execution trace: the time and values of all ROS messages published by the system.

2. Instrumented threshold predicate comparison data: the values, results, and identifier of all threshold predicate comparisons that are executed during operation in the robot system when executed.

3. User marks: the time and type of error marked by the user.

4. Time synchronized video: video of the robot system in operation synchronized with the other three data sources.

Table 6.1 shows the systems and treatment combination we used, the number of user trials for each, and the number of unique users per treatment. There were 12 different configuration configurations and a total of 94 user trials. Only two user trials were run for the water sampler and navigation code because we only needed to confirm the marking characteristics for clean trials and did not need to examine how the suggestions for the tool perform.

6.3 Water Sampler Experiment

Table 6.1: Experimental Setup

System	Treatment 1 Type I Error	Treatment 2 Type II Error	Treatment 3 Source Code Fault	Treatment 4 Clean	User Trials	Unique Users
Water Sampler	x				9	5
Water Sampler		x			9	5
Water Sampler			x		10	5
Water Sampler				x	2	1
Create Navigation	x				10	5
Create Navigation		x			10	5
Create Navigation			x		10	5
Create Navigation				x	2	1
Image Capture	x				8	4
Image Capture		x			8	4
Image Capture			x		8	4
Image Capture				x	8	4

The first system we examined was an autonomous water sampling UAV developed by the Nimbus Lab [10]. This system combines the mit_asctec and water sampling systems analyzed in Chapter 5. The system contains around 11,000 lines of code in total. The full system source code contains a total of 58 threshold predicate comparisons with 47 unique configuration parameter sources used in the predicates. The threshold predicate comparisons are spread across 15 different files. While in operation, the system communicates over 37 separate ROS topics and uses 18 ROS services. The execution is spread across 31 ROS nodes. These include nodes that control the position of the UAV, track the UAV, run the water sampling sub components, manage the mission, and ensure safety.

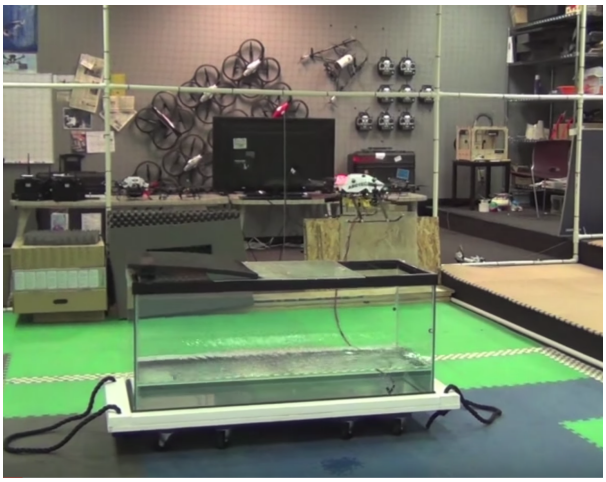


Figure 6.1: The water sampler used during the tests in this section

A successful trial, treatment 4, executes as follows. The UAV begins on the ground with motors off and stationary. The motors turn on and the UAV begins flight, ascending to 2 m. Next, the UAV flies over a fish tank at a height of 2.0 m. Once over the fish tank the UAV decreases altitude to a height of 1.2 m to insert the water sampling mechanism into the water. The system turns on the pump and fills a vial full of water. Once the vial is full the UAV returns to 2.0 m and returns to the takeoff location, lands, and shuts down the motors. Figure 6.1 displays the UAV during operation over the fish tank while sampling.

For the three treatments that cause an error we will be making the following changes:

1. Type I Error: configuration parameter altitude_abort_level. This parameter controls how low the UAV must be before the mission is aborted by the system. This parameter is configured in the main system launch file. When the UAV aborts it stops the current operation and returns to a specified height and hovers until receiving further commands. We will raise the height of the abort level so the UAV will be deemed "too low" during normal operation and abort the sampling.

2. Type II Error: configuration parameter error_xy. This parameter is used to determine if the UAV has reached a target location with an error margin of error_xy. This parameter is configured in the system's main launch file. We will lower the value of the threshold so that the UAV will not continue on to the sampling portion of the mission, generating a type II error because the target location may not be reached.

3. Source Code Error: FlyToObject.py Line 56. we will change the calculation of the x error to be $abs(x_{uav}y_{target})$ instead of $abs(x_{uav}x_{target})$. This will result in an incorrect error term being calculated for the distance of the UAV to the fishtank. The error produced here will cause a behavior similar to changing the error_xy parameter.

6.3.1 Threshold Statistics RQ1

To determine the runtime characteristics of threshold predicate comparisons we gathered data on their execution during all of the trials. The data was then grouped by the parameter that loaded the threshold value into the system. The compiled statistics are found in Table 6.2. The Parameter column is the name of the ROS parameter that populates the values for the threshold predicate comparisons. The Locations column identifies how many individual threshold predicate comparisons use the value read in by the parameter. The Comparisons column contains the total number of times a record from the threshold predicate comparisons appears in the execution traces. The frequency count is the number of comparisons divided by the total number of seconds in the trial. Runtime percentage is the number of seconds the threshold appears in the execution trace divided by the total number of seconds in the execution. It is computed by breaking the trace into second long segments and then determining if a parameter had a threshold predicate comparison present in that time range. Flops in the number of times that a threshold predicate comparison changed values from true to false or vice versa. True % and False % are the percentages that the threshold predicate comparisons had the respected values.

These comparisons took place in 24 (41.4%) out of the 58 predicate locations identified during the static analysis portion of the code. The used predicates represented 17 (41.5%) unique parameters out of the possible 41 found used to populate the thresholds used in predicates. Each of the parameters was used in only one or two predicate locations in the source code.

All of the observed trials had a runtime of around 1250 seconds. In total the instrumented code logged 531,793 executions of the predicates instrumented during the static analysis of the water samplers code. This amounts to around 425 threshold comparisons

Table 6.2: Runtime parameter statistics

	Locations	Comparisons	Frequency	Runtime %	Flops	True %	False %
altitude _abort _level	4	10432	8.35	75	32	8.75	91.25
check _dist _z _separately	2	22884	18.31	11	0	0	100
collapse _deadband	2	37121	29.70	69	0	100	0
enable _baro _ctrl _mode	4	74193	59.36	69	0	100	0
enable _derv _fir _filter	2	36131	28.91	69	0	100	0
enable _pitch _ctrl	3	37125	29.70	69	0	0	100
enable _roll _ctrl	4	37130	29.71	70	0	0	100
enable _thrust _ctrl	2	37124	29.70	69	0	0	100
enable _thrust _iir _filter	2	37117	29.70	69	0	100	0
enable _yaw _ctrl	4	37130	29.71	70	0	0	100
gps _enable _mode	2	37122	29.70	69	0	0	100
indoor	4	10458	8.37	75	0	0	100
max _pkt _size	2	86243	69.01	86	8	100	0
pose _error	2	22893	18.32	11	28	75.46	24.54
pose _error _z	2	5613	4.49	4	34	99.48	0.52
waypose _idle _timeout	2	1201	0.96	86	110	84.93	15.07
xy _error _allowed	4	1876	1.50	38	4	0.21	99.79
mean	2.76	31281.94	25.03	59.35	12.71	45.23	54.77
median	2	37117	29.70	69	0	8.75	91.25
std	0.97	23133.08	18.51	26.20	27.84	48.80	48.80
minimum	2	1201	0.96	4	0	0	0
maximum	4	86243	69.01	86	110	100	100
sum	47	531793	425.50	1009	216	768.83	931.17

per second. The number of comparisons per second for the parameters ranges from less than 1 per second to 69.01 per second for the most frequent.

A graph of the runtime % for each parameter is shown in Figure 6.2. The runtime % of each parameter is defined as the percentage of seconds a comparison using that parameter occurs in the trace compared to the total runtime of the whole trial. Three parameters appear in less than 20% of the total system runtime. One appears in just over 40% of the trial execution. The remaining parameters are compared in over 65% of the seconds that the robot system is running. Two of the parameters are checked over 85% of the time during all of the trials.

The runtime portion of this tool is able to pinpoint which parameters are actually being used during any task the robot system is performing. It can immediately reduce the problem search space from the possible 41 parameters to the 17 that were used during

the operation of the system during the task. This reduces the problem search space by over half before performing any intelligent operations on the data returned from the instrumented predicates. The results also show that that even when a small number of parameters are used, a large number of predicates using the parameters are executed every second.

The bursty nature of thresholds is also apparent from the results of the runtime analysis of the water sampler. Some parameters are only used in small portions of the robot's mission. If an infrequent threshold is present and the user marks an issue, this may be a hint that the infrequent threshold is the one causing the problem and is something that warrants further investigation in future iterations of the runtime analysis.

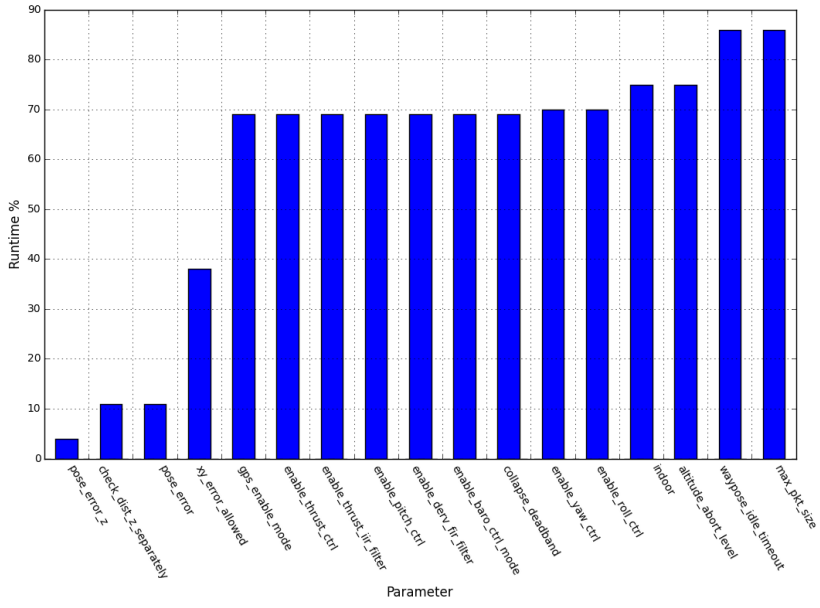


Figure 6.2: Percentage of total seconds in which a parameter is used in a predicate

The final portion of the runtime characteristic section examined the overhead of the instrumentation. During the trials we observed no signs that the system struggled to handle the additional overhead of monitoring the predicates and publishing the messages. The method uses the existing logging stream and one additional topic to reported the values of the predicates. To get an idea of how much additional data the instrumented

code provides and the overhead of the analysis we examined the percentage of overall messages that were produced by the instrumentation. In total 13.1% of the messages in all of the trials were from instrumented threshold predicate comparisons. While this is a large percentage, it is not an unreasonable amount of excess data.

This subsection has shown that threshold predicate comparisons are very common throughout system execution. They appear throughout many parts of the code in the water sampler. It also shows that some thresholds appear throughout the trial execution and others only appear during small sections of the trial. The overhead of the instrumentation is not unreasonable. Finally, it also demonstrates that not all identified thresholds will appear while the robot is in operation.

6.3.2 Flops RQ2

A flop is defined as a predicate in which the result is different from the previous predicate comparison. In the observed trials there are 216 flops. This is only 0.041% of all of the executions of the instrumented predicate threshold comparisons. This provides evidence that the occurrence of a predicate changing values is a somewhat rare occurrence in the water sampler. Only 6 (35.3 %) of the 17 parameters are used in predicates that have both true and false results. This shows that only a few predicates change values so any predicate that does so contains information. If the value of the predicate does change, one result is produced at a much higher rate. The rarity of flops and the high proportion of true or false values compared to the opposite value should enable us to trace type I errors to the problematic parameter easily. If multiple flops were to be present when the error occurs then we can use the frequency of flops to determine which parameter does not change as often and highlight that parameter as the likely source of the problem in future iterations.

Table 6.3: User Marks in all trials

	User 1		User 2		User 3		User 4		Total	
	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
Treatment 1 Trial 1	1	0	1	2	0	0	1	0	3	2
Treatment 1 Trial 2	0	0	1	1	1	1	1	0	3	2
Treatment 1 Trial 3	2	3							2	3
Treatment 1 Trial 4	1	9							1	9
Treatment 2 Trial 1	0	2	0	4	0	1	0	3	0	10
Treatment 2 Trial 2	0	0	0	3	0	2	0	1	0	6
Treatment 2 Trial 3	0	6							0	6
Treatment 2 Trial 4	0	5							0	5
Treatment 3 Trial 1	0	0	0	4	0	1	0	3	0	8
Treatment 3 Trial 2	1	2	0	4	0	1	0	3	1	10
Treatment 3 Trial 3	0	5							0	5
Treatment 3 Trial 4	1	6							1	6
Treatment 4 Trial 1	3	0							3	0
Treatment 4 Trial 2	1	1							1	1
mean	0.71	2.79	0.14	1.29	0.07	0.43	0.14	0.71	1.07	5.21
median	0.50	2	0	0	0	0	0	0	1	5.50
std	0.91	2.94	0.36	1.73	0.27	0.65	0.36	1.27	1.21	3.29
minimum	0	0	0	0	0	0	0	0	0	0
maximum	3	9	1	4	1	2	1	3	3	10
sum	10	39	2	18	1	6	2	10	15	73

6.3.3 User Marks RQ3

Table 6.3 displays the number of type I and type II errors marked by the users during each trial. The users marked a total of 85 errors, or an average of 6.5 errors per trial. The maximum number of errors marked in one trial was 11 and the minimum was 2.

Type II errors were marked at a higher frequency than type I errors. Users marked 58 more type II errors than type I errors. The users also marked type II errors in every trial except one, while type I errors only appeared in eight of the trials. This may be because type II errors are "long" errors; the system takes a long time to resolve the error, if it does at all. Once the robot reaches the error state it is stuck in the state for a longer period of time than the type I error which manifests instantaneously. Another factor leading to the larger proportion of type II errors identified is the fact that treatment 3 mimicked a type II error, so a larger portion was expected.

During trials for treatment 1, the users marked type II errors as well as type I errors. This was done by two of the users. This is likely because of confusion of what the type I error caused. The error causes the UAV to ascend almost immediately after beginning the sampling and returned to a hover. When the robot aborted the mission it did not make progress toward the water sampling. This is similar to the description of a type II error that states that a type II error: "occurs when the robot should be doing something, but does not." The complete definitions and descriptions of the types of errors in the future could be further clarified to assist the user. There is also reason to believe that as users use the system more often or are more familiar with the robot system that they could better distinguish between the two types of errors during runtime.

Table 6.4 displays the confusion matrix for treatments containing the introduced type I and type I parameter errors. In the table it is apparent that users are not perfect at marking error types. They mark almost twice as many type II errors during type I treatments. This shows that they may struggle to mark the correct error type consistently. One promising result is that users did not mark any type I errors during the trials with the type II error introduced.

Table 6.4: Confusion Matrix for Type I and Type I Errors and Treatments

	Type I Marks	Type II Marks
Treatment Type I	9	16
Treatment Type II	0	27

Users did mark errors during the third treatment during an error that was not related to configuration errors but instead a code fault. The error manifested itself exactly as a type II error would, which provides evidence that users will mark non-configuration problems as errors in the system as well. Finally, users did mark errors during the clean treatments; however, the number of error marked during the clean treatments was small and provides evidence that users will not mark normally operating systems that are not

operating as they should.

Table 6.5 displays the average delay in marking an error after the changed threshold has flopped or should have flopped. For type I errors this value is the number of seconds to the first user mark after the modified threshold predicate comparison flopped. For type II errors this is the number of seconds to the users first mark after the threshold exceeded the original parameter value in the functional system. On average it took the user less time to mark the Type I errors (1.5 to 20.8 seconds) than the Type II errors (8.7 to 20.4 seconds). The mean time for type I errors had an outlier of 38 seconds. All of the other values were less than 14 seconds. The delay for type II errors did not contain any outliers. However, it still took a noticeable amount of time to identify and mark the errors for many of the users. The delay for Type II errors should not be an issue. Since the error depends on a threshold not flopping, the threshold should still be close to flopping in the produced scores even after some period of time. If the threshold has flopped, then the error will no longer be present and the user hopefully would not mark the error. The delay may cause an issue in Type I errors. If the system begins to do another activity because of the flop, more threshold predicate comparisons may flop. This may lead to more flops that will be falsely identified as the problematic threshold once the user finally marks the error.

Table 6.5: Delays from error to the first mark by the users.

	User 1	User 2	User 3	User 4	Mean
Treatment 1 Trial 1	38.06	12.40		12.10	20.86
Treatment 1 Trial 2		6.90	14.30	10.96	10.72
Treatment 1 Trial 3	2.30				2.30
Treatment 1 Trial 4	1.55				1.55
Treatment 2 Trial 1	20.70	20.86	20.46	19.62	20.41
Treatment 2 Trial 2		19.48	17.34	27.06	21.29
Treatment 2 Trial 3	11.59				11.59
Treatment 2 Trial 4	8.70				8.70

The results of the users marking errors during runtime of the water sampler trials

demonstrates that users can identify problems in the robot system. It also shows that there is evidence of some confusion between the two types of errors. However, with further usage of the system and better definitions and examples of the error, they may be able to overcome this confusion.

6.3.4 Runtime Analysis Results RQ4

Table 6.6: Average score produced on marked errors and average

	Type I Rank	Type I Score	Type II Rank	Type II Score
Treatment 1 Trial 1	**1.00**	38.30	0.95	0.38
Treatment 1 Trial 2	**1.00**	34.24	0.90	0.52
Treatment 1 Trial 3	0.47	5002.48	**0.95**	0.36
Treatment 1 Trial 4	**1.00**	3.45	0.63	3333.55
Treatment 2 Trial 1	0.00	9999.00	**0.91**	0.08
Treatment 2 Trial 2	0.00	9999.00	**0.95**	0.06
Treatment 2 Trial 3	0.00	9999.00	**0.95**	0.08
Treatment 2 Trial 4	0.00	9999.00	**0.96**	0.08

Table 6.6 displays the raw scores and the ranking of the trials for the treatments with the modified parameters. These scores are the average of the results produced by the system for each user mark during the trials. For each type I or type II mark we computed the corresponding score and rank. The table contains the mean of all values during the trials. For ranking score a value of 1.0 demonstrates that the modified threshold predicate parameter was ranked as the top cause of the error by the analysis. A score of 0.0 is the lowest possible score. The score is the raw score produced by the runtime analysis. A value closer to 0.0 indicates that the threshold predicate comparison is more likely to be the cause of the error. The raw analysis scores have no maximum value.

The system did a very good job in identifying the problematic threshold in both the first and second treatments to the system. In treatment 1 the system had the modified parameters threshold predicate comparison as the top suggestion except for trial three. During this trial two of the marks came well before the threshold had flopped. In

treatment 2 the modified parameters threshold predicate comparison appears in the top three rank threshold predicate comparisons. In treatment 2 the threshold never "flopped." This means that the score is always the maximum value of 9999 and the rank of 0. This shows the importance of identifying the correct type of error in offering good suggestions. The dependence of the correct results on the user marking the correct error type means that if the user does not mark the correct error, the results the system currently produces will not help them accurately identify the problematic threshold.

6.3.5 Deeper Analysis

More details on individual trials will be presented in the following subsections to help further clarify the research questions. In the following subsections we present graphics for each trial. The graphs on the left of the figures display the UAV's position and location during the trial (The fishtank is at approximately -2.1 m on the x axis during these trials.) The graph on the right of the figure displays the rank scores of the type I and type II errors for all of the threshold predicate comparisons in the trail. In these graphics the red line represents the modified threshold in the trial. The modified threshold also contains red x marks to indicate that it is the modified threshold. All of the other lines are color-coded and each represent a different threshold predicate comparison within the code. If the threshold had a score of 99999 it was given a rank score of 0. More thresholds were present at points in time than appeared within the graphs due to this zeroing of scores. This demonstrates further the ability of the system to limit the number of parameters to examine to determine the root cause of the problem. User marks are indicated in both sides of the figure using vertical lines with x marks on them. In the figures, blue marks represent type I errors and green marks represent type II errors.

For example, if the modified threshold ranks second at the current time during system

execution and there are 10 other thresholds present, there will be 11 different colored lines in the graph. A red line indicating the modified parameter will be second from the top. All of the other threshold predicate comparisons will be color coded and remain the same color throughout even as their ranks change. If the user marks a type I error, there will be a green vertical line at the moment in time the user marked the error.

6.3.5.1 Treatment 1 - Type I Error

Figure 6.3 displays the trials that were examined with treatment 1. In these trials the water sampler began sampling, but aborted and hovered over the fish tank after the abort level was exceeded. In the first two trials there where two user marks that the UAV encountered a type I error. These occurred between 10 and 20 seconds after the UAV aborted and ascended. However, in both cases the predicate that contained the modified threshold value was still at the top of the ranking graph. In the first trial the type II error marks occurred even later than the type I errors. In trial 2 one user marked a type II error after the UAV had already landed. This may be an indication that the mark is not valid. However the other type II error mark came within 5-10 seconds of the UAV changing its behavior. In the other two trials the user marked errors before the error accrued. The marks are understandable in the 4th trial as the UAV spent around 10 seconds at 1.0 m before ascending and continuing on the trial. The cause of this loitering is unknown, but the user was quick to mark it as an error.

An interesting observation in the graphics is that a large number of threshold predicate comparisons appear in the type I ranking graph just before the UAV descends from 2 m to the sampling height. This indicates that there are many "flops" in the system as the UAV changes the current activity from being on the ground, to flight, to then sampling and aborting. Many type II ranks appear at this time indicating that the many parts of the sub control system are beginning to "get close" to flopping to allow changes in the

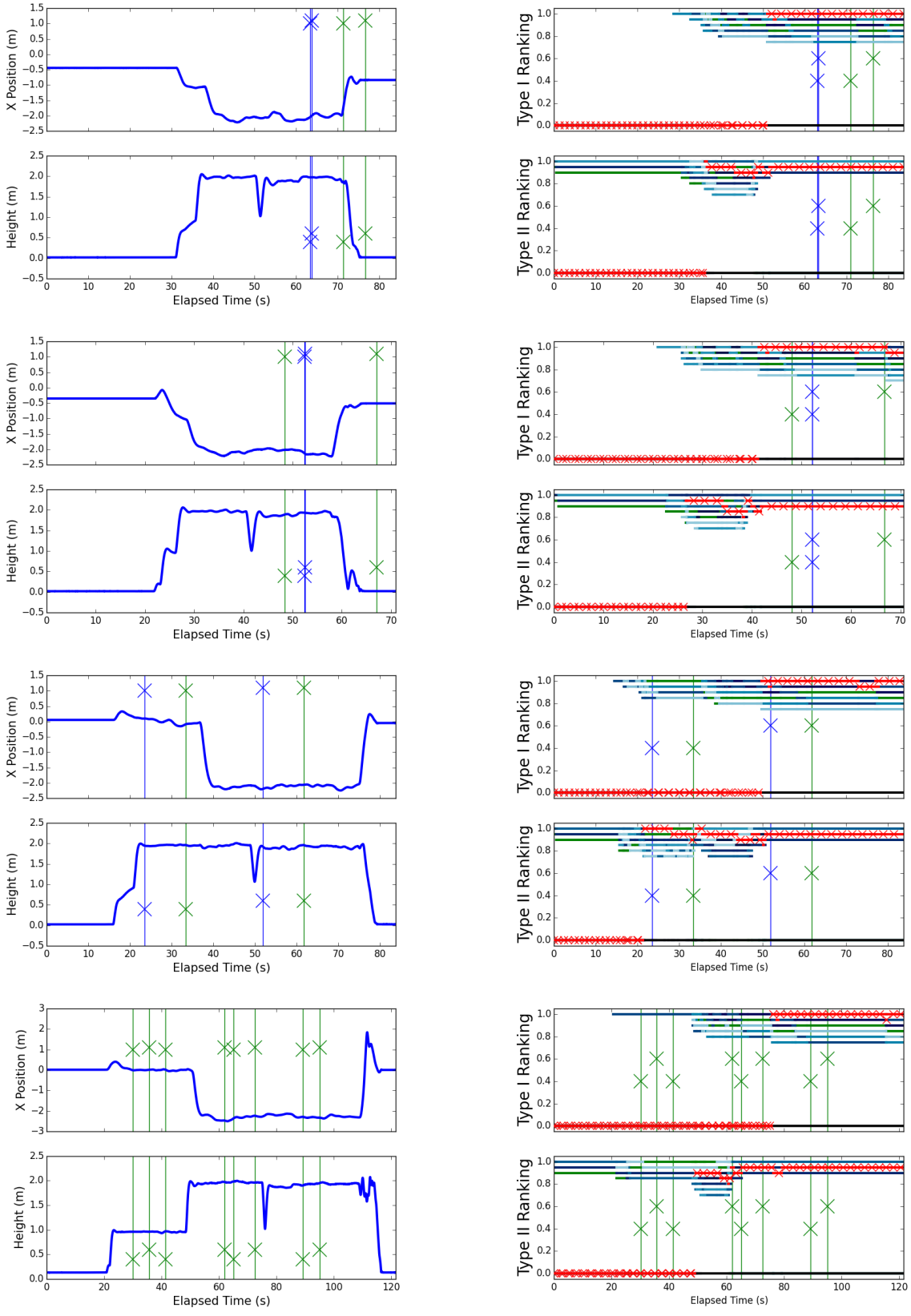
control system of the robot. Many of these thresholds drop from the rank graph after the UAV has leveled off after aborting the trial. This offers evidence that during more active parts of the trail there will be more threshold predicate comparisons active than during relatively stable and non-eventful portions of the trial.

6.3.5.2 Treatment 2 - Type II Error

Figure 6.4 displays the trials that were examined with treatment 2. In all of these trials the UAV never began the decent to collect the water sample. The users all marked type II errors during the missions. All of the users waited on the order of 10 to 20 seconds to begin marking errors after the UAV had reached the area of the fishtank. This provides evidence that users are slower to mark type II errors. The modified threshold was in the second or third positions in the rank graph during the time when the users began marking. Another interesting feature of the rank graphs is that immediately after the UAV got to the target location the modified parameter was near the top in both of the type II error graphs.

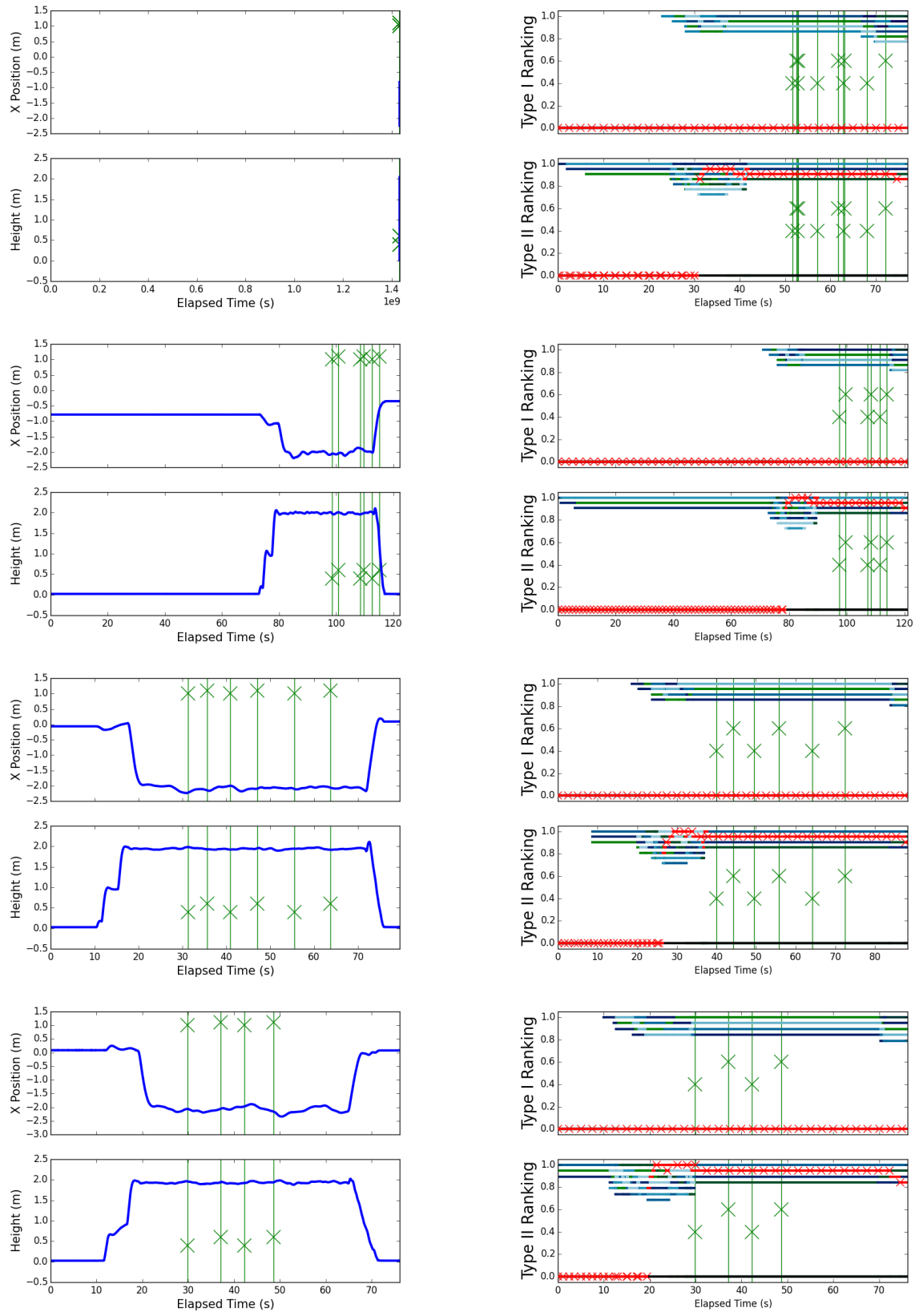
6.3.5.3 Treatment 3 - Source Code Error

Figure 6.5 shows the trials examined with treatment 3. Similar to treatment 2, the users again waited some time before marking any sort of error and only marked type II errors during this trial. Similar to the other treatments, there are many thresholds that flop and are about to flop when the UAV takes off and flies to the target location. Once the UAV is at the target locations however the number of thresholds that are about to "flop" drops off as not as many are involved. All of the ones that have flopped to reach the next portion of the trial have except for the few that are preventing it from continuing. This trial provides evidence that faults in the robot source code may also appear to be type II errors. Users marked type II errors many times during execution. These trials also



(a) (b)

Figure 6.3: The height and X position of the water sampling UAV throughout time and the threshold ranking of the modified parameter throughout time in treatment 1 (Type I Error) trials



(a) (b)

Figure 6.4: The height and X position of the water sampling UAV throughout time and the threshold ranking of the modified parameter throughout time in treatment 2 (Type II Error) trials

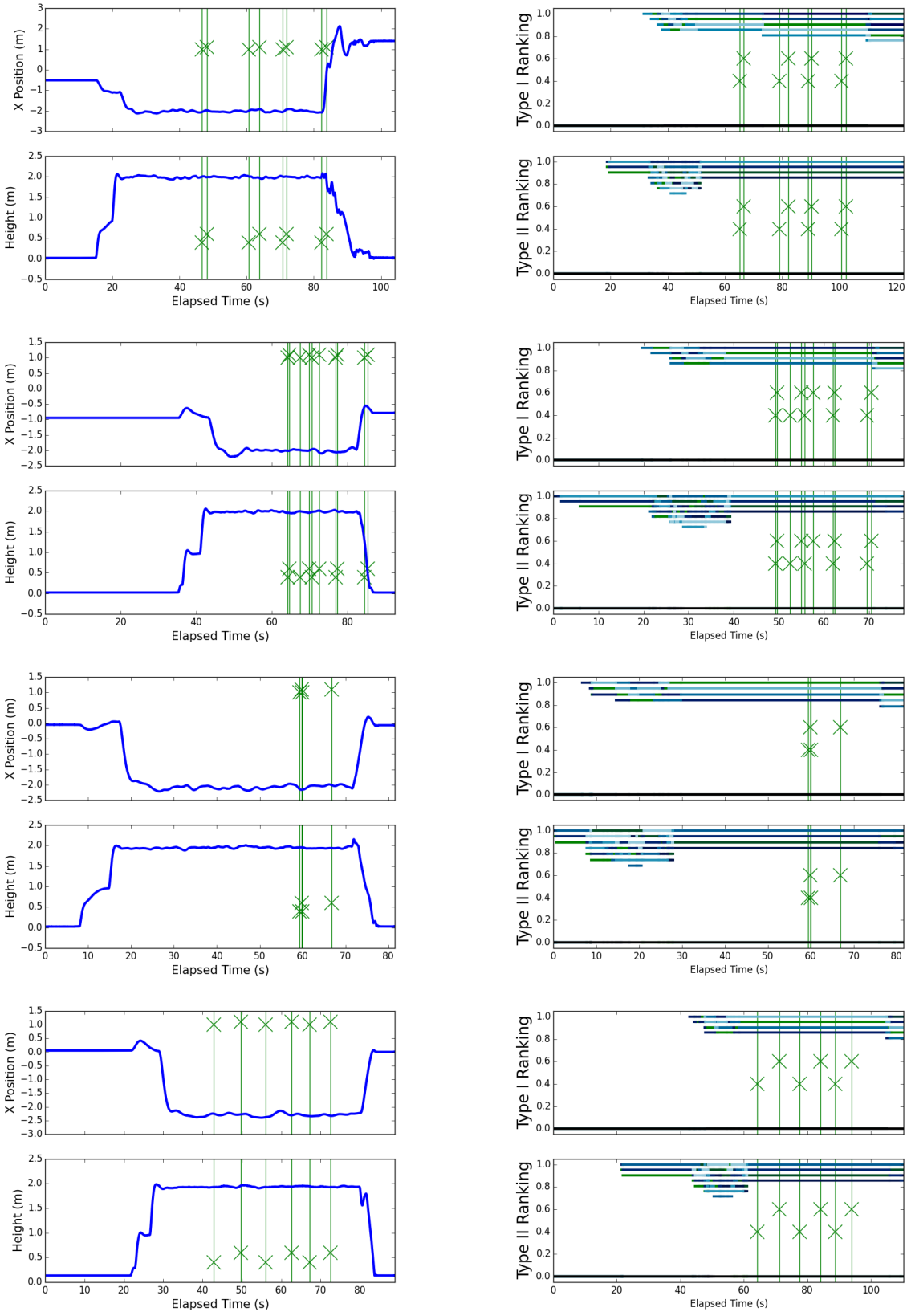
showed that users were likely to wait a period of time before marking any type II errors. This trial also provides evidence that during times of transition within the system there are more thresholds than periods when the system is in a constant state.

6.3.5.4 Treatment 4 - Clean

Figure 6.6 displays the position of the UAV and the ranking scores for the threshold predicate comparisons. As shown in the graphs the UAV successfully completed the trials in both cases. Also, the user marked only a few errors in both of the trials. This demonstrates that users are able to determine when the robot is behaving correctly and will not mark errors as often as cases where there is an error.

6.3.6 Summary

The trials on the water sampler provided many answers to our research questions. First, we found that a number of identified threshold predicate comparisons are identified statically, but do not appear in the execution traces. This allows the filtering of configuration options immediately when searching for an error. In the case of the water sampler the filtering could eliminate 58.5% of the parameters used to configure the system. The trials also provided evidence that threshold predicate comparisons are very common during the execution of the system as on average there were 22.5 threshold predicate comparisons occurring per second in the system. The trials also demonstrate that some thresholds appear in nearly every portion of the execution but other appear only in specific portions of the trial. This allows the possible use of better decisions in determining which threshold is causing the issue. If a problem is marked and a threshold that appears infrequently has occurred recently that threshold may be the source of the problem.



(a) (b)

Figure 6.5: The height and X position of the water sampling UAV throughout time and the threshold ranking of predicates throughout time in treatment 4 (No Error) trials

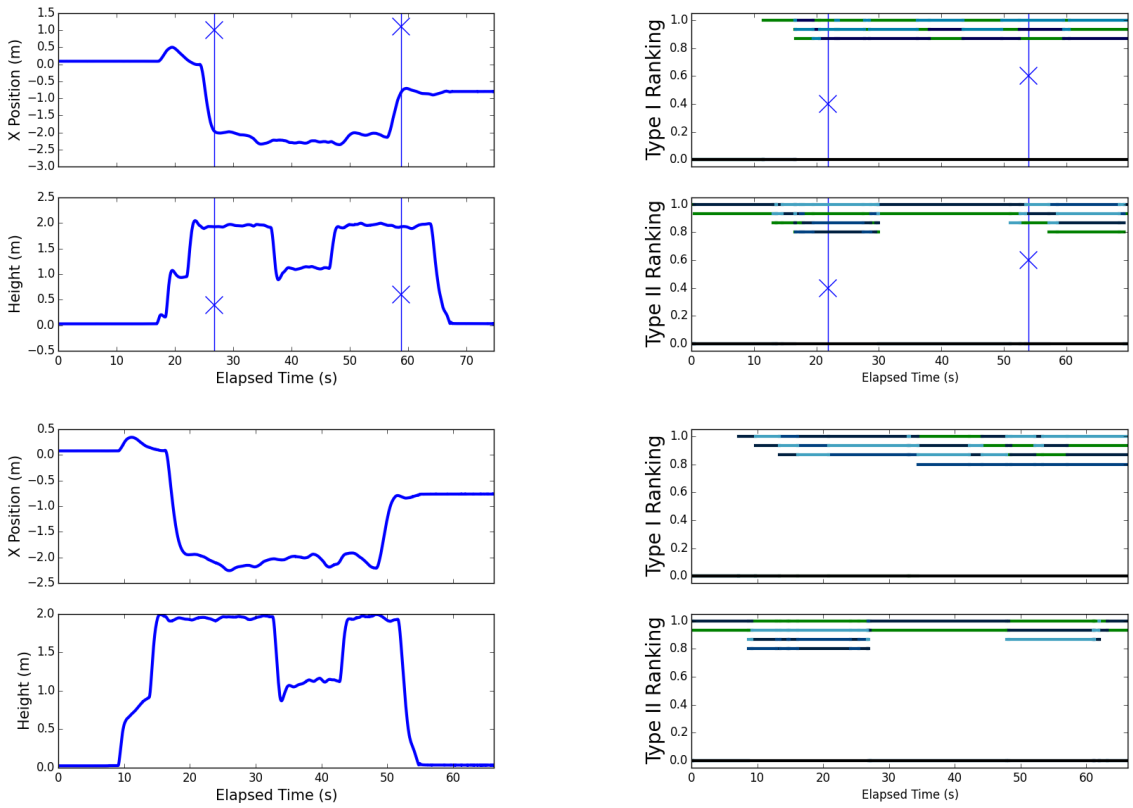


Figure 6.6: The height and X position of the water sampling UAV throughout time and the threshold ranking of predicates throughout time in treatment 4 trials

The trials also provide evidence that "flops" are not common during system execution. Only 0.058% of the results from instrumented threshold predicate comparisons result in a "flop." This offers evidence for our assumption that "flops" are important events that signal a change in the behavior of the robot system. It also signals that being able to identify which predicate flopped or is about to flop may be of use to a user trying to discover configuration errors in the robot system.

Experiments with the water sampler also show that the users are not as accurate in identifying the types of errors that occur during system operation as we would hope. There seemed to be some confusion between type I and type II errors and users marked type II errors more frequently than type I errors in the system. However, as users become more familiar with the system that they are running, we assume that they would become

more familiar with the types of errors present in the system. Additionally, we may need to rethink the types of errors.

Finally, in the trials the runtime analysis did a reasonable job in identifying which threshold was the cause of the issue compared to the other thresholds present in the system. In treatment 1 the changed threshold was the top identified problem threshold for type I errors. In treatment 2 the threshold was in the top three positions during the time when the UAV should have moved on to the next point.

6.4 Navigation Experiment

The second system we used for runtime experiments consisted of an iRobot Create, which is the modified version of the iRobot Roomba for use in research [92] under control of a customized ROS driver navigating in an environment using the ROS navigation packages [8]. In total the system consists of around 25,000 lines of code including the code to operate the mobile robot and the full code of the ROS navigation repository. The full system contains 56 threshold predicate comparisons as found during static analysis. The code running the system is spread across 12 ROS nodes and two different computers. One computer controls the iRobot Create and the laser scanner. The other computer is responsible for the processing of all of the odometry and laser scanner data to localize and plan the path for the robot. After the robot is localized using the odometery and laser scanning

Figure 6.7: The iRobot Create and the course for navigation experiments

data the navigation stack is responsible for planning and overseeing the robot while it traverses the planed path. During traversal, the robot's location is recomputed and changes are made to the planned path. Commands are sent from the navigation computer back to the robot controlling the iRobot Create in the form of velocity commands with a forward velocity and a rotational velocity. These are translated into commands for the left and right wheel of the robot.

Figure 6.7 shows the iRobot Create in the environment. The target location is the marked box on the floor in the upper right of the image. A trial begins with the robot in the location in Figure 6.7. Next, the navigation stack is given a target pose inside of the marked box. The robot makes its way to the box on the planned path while avoiding obstacles. To successfully complete the trial the iRobot Create must be fully inside the box and stationary. This occurs once the localization methods have determined the robot is inside the box and the path planning algorithms recognize that the robot has reached the location.

For the three treatments we will be making the following changes to the system configuration and source code:

1. Type I error: configuration parameter brake_limit. This parameter controls the speed that a command to one of the iRobot Create's wheels exceed for the iRobot Create to actually move. This limit is defined in the launch file that launches the iRobot Create for operation. If the speed is less than the value the robot will not move the wheel. This threshold prevents damage to the motor from trying to move the wheel when friction cannot be overcome. If this threshold is raised to a value that is too high, the robot will stop moving early or will not move at all. Raising this value results in a type I error.

2. Type II error: configuration parameter xy_goal_tolerance. The navigation stack uses

this parameter to determine when the robot has reached the desired location. It is initialized in the launch file that brings up the navigation nodes for the robot. The proper value for this threshold depends on the ability of the robot to localize in the environment and the level of accuracy the user desires in the finalized position. If the value is set to too small the robot will not be able to achieve the goal position will continue to make movements about the final location. If the value is too low the robot will stop in an undesired location. In the treatment set the value too to cause a type II error.

3. Source Code Error: Trajectory_Planner.cpp. When the call is made to plan the robots trajectory the start and end locations are swapped. This will cause the robot to believe that it is already at the end location on the next localization update. It will move briefly, but stop immediately after an update in the localization. This emulates a type I error.

6.4.1 Threshold Statistics RQ1

Runtime characteristics for the threshold predicate comparisons grouped by loading parameter are shown in Table 6.7. The columns in this table have the same definitions as described in section 6.3.1. The traces of the trials contain 25 (44.6.0%) of the 56 threshold predicate comparisons discovered in the static analysis of the source code. This is a slight increase in percentage of present comparisons in comparison to the water sampler. An interesting characteristic of this system is that parameters are used in multiple locations more often than the water sampler. Of the 25 threshold predicate comparisons only 13 parameters supply values for all of the locations. The parameters are used in between 1 and 4 different locations in code and on average are used in 1.9 locations. This shows that not all systems only use parameters in a single threshold predicate comparison.

However, it still provides evidence that a parameter is still only used in a small number of predicates.

The trials had a total runtime of 1132.6 seconds. During this time the system trace contains 134034 values from instrumented threshold predicate comparisons. This corresponds to around 126.3 threshold predicate comparisons per second. The frequency of individual parameters ranged from less than .01 per second up to 23.1 comparisons per second. This is less than the amount of threshold predicate comparisons present during the water sampler. This provides evidence that systems have differing frequency of threshold predicate comparisons. One possible reason is that the water sampler contains more high level functionality and has a multiple stage mission while the navigation experiment only has one high level activity. Difference in coding styles and design patterns may also play a role in how often threshold predicate comparisons appear. A big contributing factor here is the use of nodelets to share data instead of publishing across computational nodes. However, even with the decrease in the number of threshold predicate comparisons there are a relatively high number of comparisons per second.

The runtime percentage distribution is shown in Figure 6.8. The definition and computation of runtime percentage is described in subsection 6.3.1. The distribution shows a similar pattern to that of the water sampler. Five sources of thresholds appear less than 10% of the time. Five appear in about 60% of the trial time and 4 parameters have threshold predicate comparisons that appear in over 90% of the trials. This provides evidence that there are three levels of how often thresholds appear within the runtime of a robot. This may lead to the possibility of grouping thresholds more intelligently when determining which ones cause problems. The appearance of rare occurrence group and common group suggest that those two groups are common across many robotic systems.

The final portion of the runtime characteristic section examined the overhead of the instrumentation. 21.1% of the messages on the system were from instrumented threshold

Table 6.7: Runtime parameter statistics for the navigation trials

	Locations	Comparisons	Frequency	Runtime %	Flops	True %	False %
latch _xy _goal _tolerance	1	757	0.71	5	0	100	0
oscillation _distance	2	12460	11.74	61	125	99.45	0.55
prune _plan	1	11855	11.17	58	0	0	100
recovery _behavior _enabled	1	10	0.01	0	0	0	100
shutdown _costmaps	4	19	0.02	1	0	100	0
tf _broadcast	2	10530	9.92	95	0	0	100
update _min _d	2	20914	19.71	95	493	98.82	1.18
visualize _potential	1	79	0.07	5	0	100	0
xy _goal _tolerance	1	11855	11.17	58	27	93.61	6.39
yaw _goal _tolerance	1	757	0.71	5	4	98.94	1.06
brake _limit	2	10553	9.94	95	94	49.44	50.56
cmd _vel _timeout	1	5279	4.97	95	42	35.20	64.80
max _forward _speed	4	24484	23.07	60	0	0	100
min _forward _speed	4	24482	23.07	60	0	0	100
mean	1.93	9573.86	9.02	49.50	56.07	55.39	44.61
median	1.50	10541.50	9.93	59	0	71.52	28.48
std	1.21	8946.89	8.43	38.75	131.85	47.18	47.18
minimum	1	10	0.01	0	0	0	0
maximum	4	24484	23.07	95	493	100	100
sum	27	134034	126.30	693	785	775.45	624.55

predicate comparisons. This does consume a fair amount of resources, however it did not produce prevent the system from operating normally. We did not observe any slowdown or system problems due to the instrumentation.

This subsection has shown that many of the findings that appear in the water sampling trials also appear in the navigation trials. Threshold predicate comparisons are very common throughout the execution of a mission by the robot system. However, different sources of the thresholds within the comparisons have different rates of occurrence throughout the systems execution. Three different groups of runtime percentage appear with almost no range of thresholds appearing in between. Similar to the water sampling trials, less than half of the threshold predicate comparisons appear in the static analysis do not appear in the system trace. This shows that not all of the code involved in a system is used and also shows that the approach can be used to reduce the problem search space without the requirement to perform other types of analysis on the system or runtime

data.

A few things in the navigation trials did differ from the water sampling trials. First, the overall number and frequency of total threshold predicate comparisons was lower. A variety of factors contribute to this including the design of the system, coding styles, and the number of high level tasks each mission was performing. The other difference is the number of locations that the parameters where used in threshold predicate comparisons. In these trials the average ROS parameter was used in 2 threshold predicate comparisons. This may be attributed to the design of the system. At any rate the mapping of parameters to locations is still relatively small and still allows the pinpointing of a problematic parameter by finding the predicate threshold comparison in which it was used.

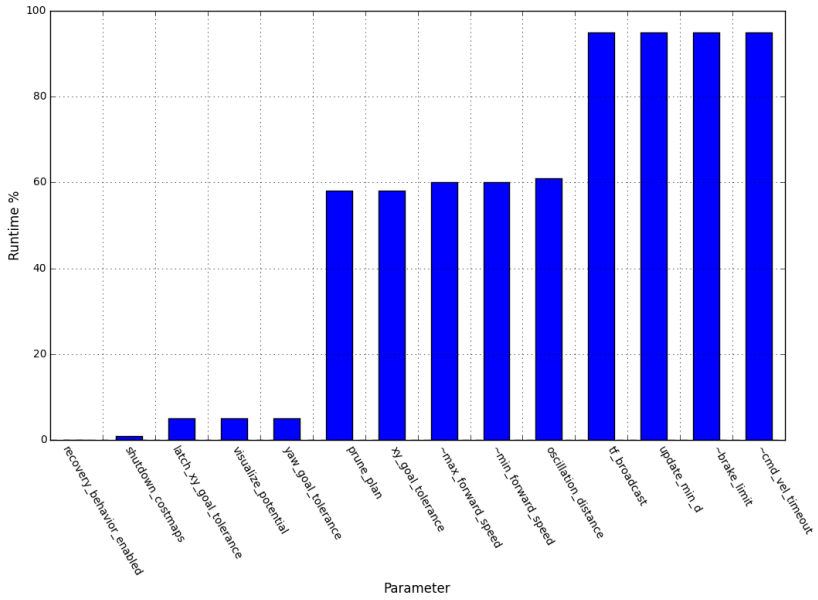


Figure 6.8: Percentage of total seconds in which a parameter is used in a predicate

While the navigation stack contains more lines of code then the water sampling trials, it is spread across fewer nodes. A number of the navigation stack nodes contain nodelets to share information within the process instead of sending it between nodes. This is one reason why there is not an increase in threshold predicate comparisons even though there was an increase in the size of the source code. Additionally, there is only one high

level functionality in this code compared with the multiple sub tasks that are done in the water sampler. This is another possible reason for the non-increase in threshold predicate comparison count. The examination of how to instrument nodelets and the information shared between them is something that needs to be examined in future work.

6.4.2 Flops RQ2

In the six missions there are a total of 815 flops. This is corresponds to 0.57% of the threshold predicate comparisons executed during the trails. This is a very small percentage and provides further evidence that flops are a rare occurrence. This value is 10 times that of the flop percentage of the water sampler. However, around 75% of the flops are caused by one parameter in the system. Factoring this parameter out of the calculation the rest of the system's predicate threshold comparisons flop at a range of about 0.1% This provides evidence that flops do happen more often for some parameters, but when they are more frequent they may be easier to filter out of the results than other threshold predicate comparisons as the flopping may not be as important in their case. Similar to the water sampling trials, 8 (58%) of the parameters are used in predicates that never flop. The results from the navigation trials offer more evidence to that a flop is an important event during the execution of a robot system.

6.4.3 User Marks RQ3

Table 6.8 displays information about user marks for the navigation trials. Users marked a total of 42 type I errors and 58 type II errors. This is nearly double the amount of marks that occurred in the water sampling trials. The mean number of marks per trial is 12.5. The maximum number of marks in one trial is 38 and the minimum number of marks is 9. Every user marked at least one error in every trial.

Table 6.8: User Marks in all trials

	User 1		User 2		User 3		User 4		Total	
	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
Treatment 1 Trial 1	0	2	0	2	1	1	0	3	1	8
Treatment 1 Trial 2	0	1	0	5	2	1	0	9	2	16
Treatment 1 Trial 3	1	7							1	7
Treatment 1 Trial 4	0	7							0	7
Treatment 2 Trial 1	1	0	3	2	1	1	3	0	8	3
Treatment 2 Trial 2	3	0	16	6	0	1	11	1	30	8
Treatment 2 Trial 3	2	7							2	7
Treatment 2 Trial 4	1	13							1	13
Treatment 3 Trial 1	0	1	0	6	0	1	0	4	0	12
Treatment 3 Trial 2	0	2	0	4	0	1	1	4	1	11
Treatment 3 Trial 3	0	14							0	14
Treatment 3 Trial 4	0	5							0	5
Treatment 4 Trial 1	0	1							0	1
Treatment 4 Trial 2	0	3	0							3
mean	0.57	4.50	1.36	1.79	0.29	0.43	1.07	1.50	3.29	8.21
median	0	2.50	0	0	0	0	0	0	1	7.50
std	0.94	4.59	4.29	2.42	0.61	0.51	2.97	2.65	7.97	4.48
minimum	0	0	0	0	0	0	0	0	0	1
maximum	3	14	16	6	2	1	11	9	30	16
sum	8	63	19	25	4	6	15	21	46	115

Almost half of the marks occurred in treatment 2. This is expected as the type II error in treatment 2 causes the robot to spend an extended period of time moving about the final target location. The much larger number of marks shows that users will mark different errors at different rates across different systems. If the robot system continues to struggle to complete the task the users will continue to mark errors. This may also help to identify the true difference between type I and type II errors. The larger amount of marks during a type II error show that the robot is stalled. This may help disambiguate which type of error the user is truly marking if the confusion continues to be a problem.

In the trials users marked less type I errors (42) than type II errors (58). All but one trial had both type I and type II error marks. This is slightly misleading; nearly all of the marked type I errors occurred in the treatment 2 trials. This may be due to confusion on what constitutes each type of error. The definition of the two types of errors is somewhat ambiguous. The users all mentioned that the robot could be considered to

be both ”should be doing something” (stopping at the location) and ”doing something that it shouldn’t” (moving after it has reached the location) at the same time in the trials.

Table 6.9 displays the confusion matrix for treatments 1 and 2. From this matrix, it is very easy to see the confusion between type I and type II errors by the users in these experiments. The users marked nearly 10 times the incorrect type II errors during treatment 1 which introduced a type I error. During treatment 2 the users marked more errors incorrectly, but the disparity was not as large. Overall in the two treatments users marked incorrectly 79 times and correctly only 35 times.

These trials provide evidence that the user may not be able to successfully determine which type of error is occurring during operation, but they can determine that an error is occurring. However, the large number of markings for the trials may help to provide evidence when the system is encountering a type II error. Further work needs to be done to help users tell the difference between these two types of errors.

Table 6.9: Confusion Matrix for Type I and Type I Errors and Treatments

	Type I Marks	Type II Marks
Treatment Type I	4	38
Treatment Type II	41	31

As with the water sampler trials, users marked errors during the third treatment. This shows that users will mark errors from code faults as well as configuration errors. While this may not be an issue it may lead to some confusion when changing parameters does not cause a change in the behavior of the robot.

Similar to the water experiment, user marked far fewer errors in the clean trials. The user participating in the clean trials only marked 4 errors, while during the trials with the other treatments, the user marked almost 50 errors. This provides evidence that the users are much less likely to mark errors during normal operation.

Table 6.10 displays the average delay in marking an error after the changed threshold

has flopped or should have flopped. The marking of the type II errors took nearly twice as long as the marking of the type I errors. The delay for the type I errors was still between .76 and 10.4 seconds. Type II errors took between 2.5 and 20.8 seconds. This provides evidence that the users will not always immediately mark an error and it may take some time for them to mark a type II error.

Table 6.10: Delays from error to the first mark by the users.

	User 1	User 2	User 3	User 4	Mean
Treatment 1 Trial 1	8.12	8.62	10.40	8.32	8.86
Treatment 1 Trial 2	10.16	3.82	5.46	4.66	6.02
Treatment 1 Trial 3	0.76				0.76
Treatment 1 Trial 4	1.93				1.93
Treatment 2 Trial 1	11.62	18.94	15.96	17.92	16.11
Treatment 2 Trial 2	10.72	12.14	12.92	20.82	14.15
Treatment 2 Trial 3	2.50				2.50
Treatment 2 Trial 4	10.11				10.11

The results of the users marking errors during runtime of the navigation trials demonstrate that users can identify problems in the robot system. It also provides further evidence that there is confusion between the two types of errors. Some errors will manifest themselves in ways that will be very tough to disambiguate without knowledge of the system's source code. Some of the confusion should disappear with more familiarity with the system, but more needs to be done to help the users correctly identify the type of error occurring in the system. If that is not possible the system needs to be able to determine which analysis to run and provide the best results for a marked error. Further details about the time to mark errors will be examined in the individual trials and finish answering **RQ3**.

6.4.4 Runtime Results

Table 6.11 displays the average scores given to the marked errors by the system in the two treatments with modified parameters. In the first trial the lone mark of a type I error

occurred when the changed parameter ranked as the top scoring threshold predicate comparison. All of the type II marks by the user occurred when the threshold was in the second position on the type II error ranks. In the second trial the threshold was not as high in the type II errors at that point in time. However, it was still within the top 3 of the rankings. The threshold also ranked in the third position on average in the type II errors. This provides evidence that incorrectly marking a type I error as a type II error does not ruin the results in all cases. There is still a likelihood that the problematic threshold will appear in the top of the ranks. This may not be true if the error drastically changes the behavior of the system. The second treatment shows the importance of correctly marking type II errors. The modified threshold is ranked very poorly when type I errors are marked because it has not "flopped" yet. It does score very will in the type II error ranks for those trials.

Table 6.11: Average score produced on marked errors and average ranking

	Type I Rank	Type I Score	Type II Rank	Type II Score
Treatment 1 Trial 1	**1.00**	30.34	0.95	0.11
Treatment 1 Trial 2	**0.95**	34.75	0.91	0.25
Treatment 1 Trial 3	**1.00**	2.07	0.70	0.84
Treatment 1 Trial 4	0.00	9999.00	**0.87**	0.38
Treatment 2 Trial 1	0.57	3759.59	**0.94**	0.06
Treatment 2 Trial 2	0.32	6673.81	**1.00**	0.03
Treatment 2 Trial 3	0.00	9999.90	**0.33**	4286.24
Treatment 2 Trial 4	0.00	9999.90	**0.93**	0.07

6.4.5 Deeper Analysis

More details on the individual trials will be presented in the following subsections grouped by treatment. The results will help to further clarify the research questions. In the following subsections we present graphics for each individual trial. The first graphics contain the X and Y position of the robot as calculated by the navigation stack and the left and right commands to the wheels of the robots. We included the wheel commands

because sometimes location data does not update at a high frequency. These four data points should give a very good picture of what the robot is doing (the target location is at approximately -1.7 m 1.4 m). In treatment 1 and 3 when the robot stops giving commands to the wheels outside of the target location, the type I error occurred. In treatment 2 when the robot continues to give commands at the target location the type II error is occurring.

The other figure displayed for the missions are the rank scores through time. They show the rank scores of the type I and type II errors for all of the threshold predicate comparisons in the mission. In these graphics the red line represents the modified threshold in the treatment. All of the other lines are color coded and each represent a different threshold predicate comparison within the code. User marks are indicated in both sides of the figure using vertical lines with x marks on them. In the figures the blue marks represent type I errors and green marks represent type II errors as marked by the users.

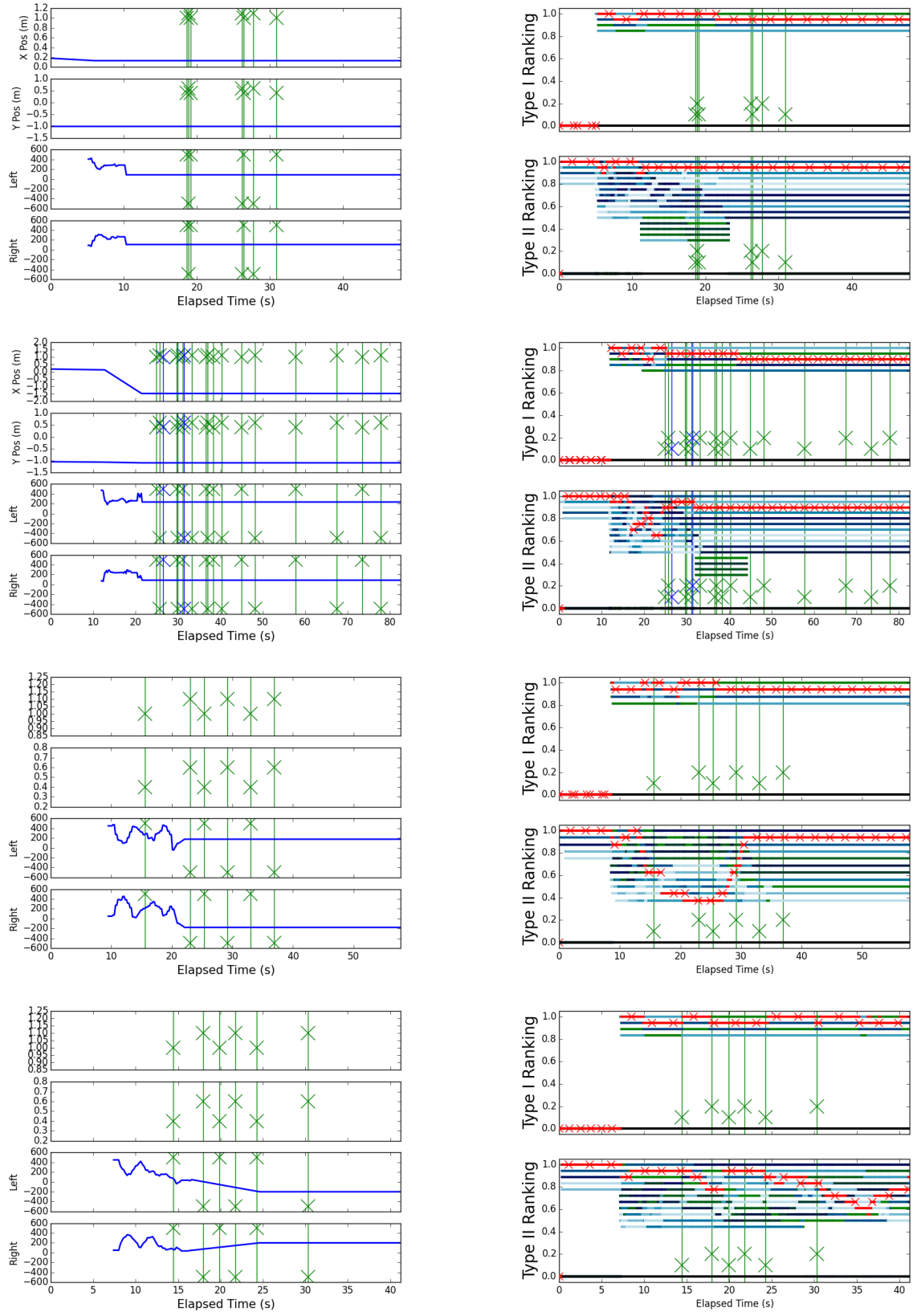
6.4.5.1 Treatment 1 - Type I Error

Figure 6.9 displays the location, wheel commands, and the threshold information for the trials with treatment 1. In these missions the robot began to travel towards the target, but the speed commands to the wheels soon did not meet the threshold value and the robot stopped in place. The forward movement in both missions lasted between 5 and 10 seconds. Once the robot stopped moving the users began marking both type I and type II errors. However, the users marked many more type II errors than type I errors during these missions. In both trials the first marks on errors came within 5-10 seconds of the robot movement stopping. The robot then stayed stationary for a period of time and the users continued to mark type II errors. This is because the robot was supposed to be moving towards the target location and the lack of movement can be seen as a no action type II error. This provides evidence for the ambiguity between the two types of

errors and the importance of helping the user understanding which error leads to which type of problem. The users marked twice as many errors in the second trial then the first. This is interesting as similar missions resulted in a different number of markings.

The figure also displays the rank score for the mutated threshold throughout the mission's execution. The mutated threshold is at the top of the type I rankings immediately after the robot stops because of the misconfigured threshold. In the first grouping of user marks in each trial the threshold was near the top of type I errors. However, not all of the users marked type I errors. Even if the users marked the incorrect error they still would see the threshold predicate comparison in the top of the type II error suggestions. This is expected as the threshold should be very close to flopping again as commands are being sent with values just below the threshold to make the robot drive again.

There are many threshold being compared once the robot begins moving, but perhaps more interesting is the increase in thresholds that are active after the robot stops moving. These may be part of the recovery process in the navigation stack that begins to try and find new paths once the one fails. They also could be related to some other periodic code that is running on the robot system. Once the robot begins moving the number of thresholds remains constant except the additional ones that appear and drop out of the trace. This pattern provides evidence that thresholds are likely to be grouped with specific behaviors in the robot. Some work may be done to group thresholds that occur together in specific robot behaviors with more in depth user feedback on the actions that the robot is performing. For example the user could highlight which parts of the mission are when the robot is traveling, which parts are when it is performing another task such as grasping to get a group of thresholds that appear during specific tasks and which ones "flop" when the task is complete.



(a) (b)

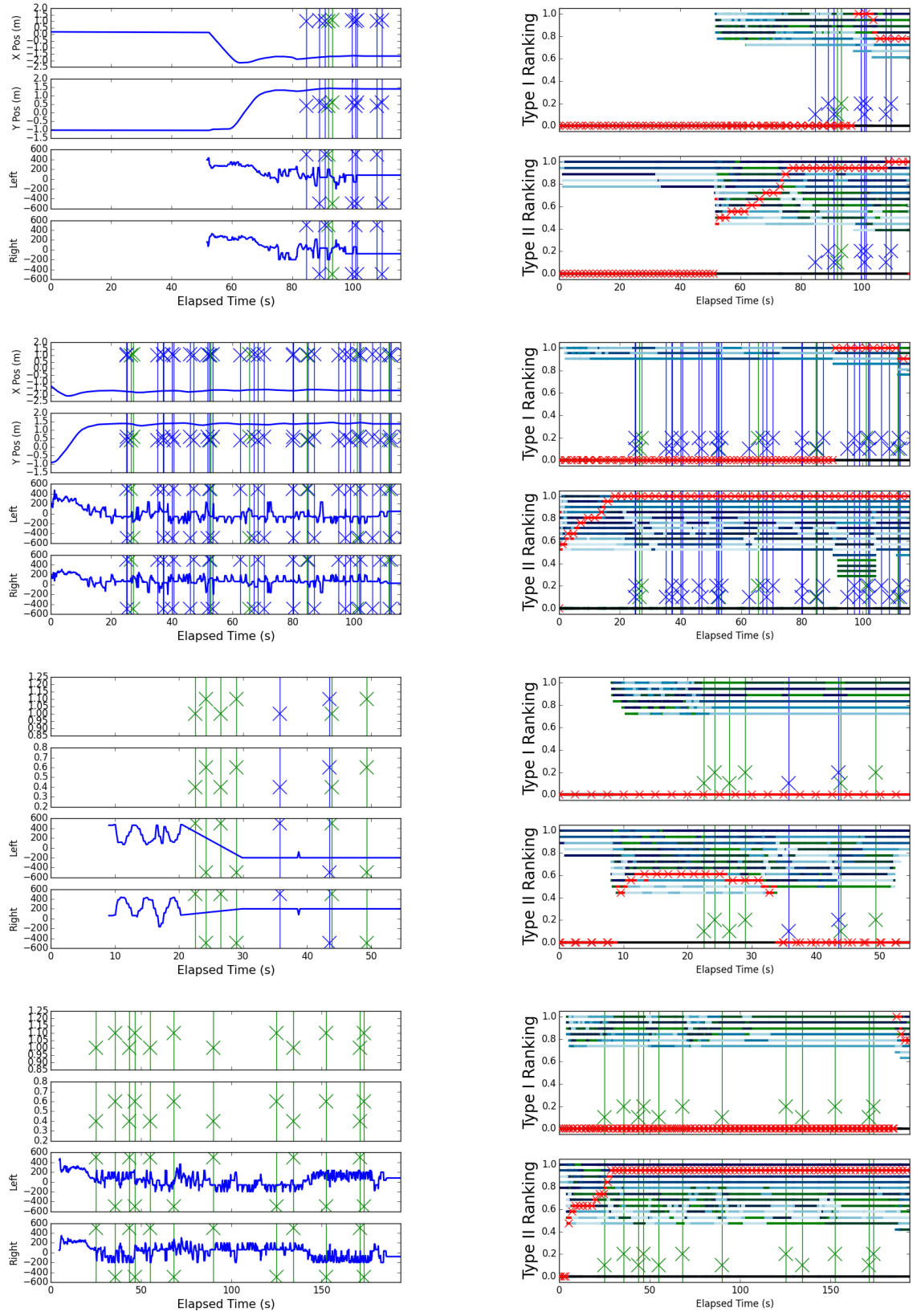
Figure 6.9: The position of the robot, wheel commands, and the ranking score of the modified threshold predicate comparison during treatment 1 (Type I Error)

6.4.5.2 Treatment 2 - Type II Error

Figure 6.10 displays information on the location, commands, and threshold predicate comparison score ranks for the missions involving treatment 2. In these trials the robot reaches the target location around 10-15 seconds after beginning to move. Once it reaches the location the robot moves about the target for a period of 20 to 60 seconds before finally coming to rest within the targets bounds.

Soon after the robot gets to the location and continues to move the users begin to mark errors. The users continue to mark errors until the robot finally stops moving and comes to rest at the target location. It is understandable that they mark many more errors during this mission. The robot spent upwards of a minute making small movements inside of the marked location trying to bring the error within the allowable value set by the parameter. However, as with the other treatment, a large number of the errors were marked as the wrong type. This is problematic because the modified parameter's threshold predicate comparison does not flop until the robot has come to rest. This means that all of the type I error markings will not have the predicate present since it is yet to flop. This highlights the importance disambiguating the types of errors for the user or developing a smarter way to determine which scores to report to the user once they have marked a location.

The ranking graphs for the treatment look exactly as expected. The threshold does not appear in type I errors until it flops after the robot has stopped. In each of type II rank graphs the modified parameter's predicate threshold comparison shows up at the bottom of the rank score at the start of the mission. Through time as the robot gets closer and closer to the goal, the score climbs up the rank graph. Finally, once the user begins to mark issues the mutated parameter is at the top of the score graph.



(a) (b)

Figure 6.10: The position of the robot, wheel commands, and the ranking score of the modified threshold predicate comparison during treatment 2 (Type II Error)

6.4.5.3 Treatment 3 - Source Code Error

The information about trials for treatment 3 is shown in Figure 6.11. As shown the robot did not move far due to the error introduced in the code. Users again waited a period of time before beginning to mark the errors.

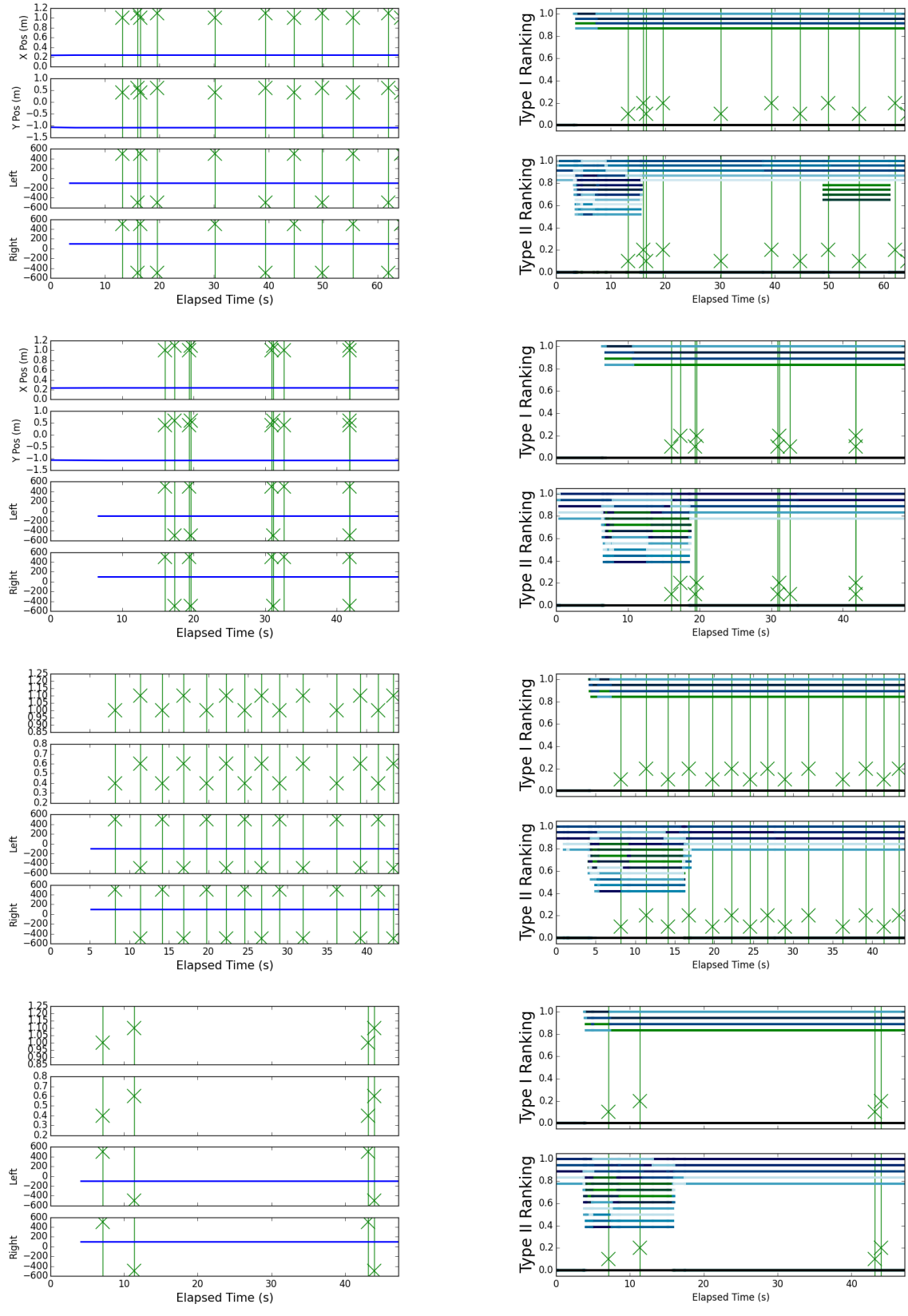
Examining the rank graphs shows that 4 predicates flopped over once the robot started moving and that many more starting to be compared during the initial movement. It also shows that there appear to be two separate groups of thresholds that appear at different times in the trace.

6.4.5.4 Treatment 4 - Clean

Figure 6.12 shows the wheel commands and ranking scores of the robot during the two trials with no change to the system. There was an issue with the amcl data so no position estimates are provided. However, the movement of the robot can be seen through the wheel commands. The robot moves from the starting location to the correct location and stops once it reaches the target. The user marked very few errors during the execution, and this provides evidence that users will not mark errors for the robot system as often when under normal operation. In the ranking graphs you can see the number of flopping thresholds increase as the robot continues on its mission.

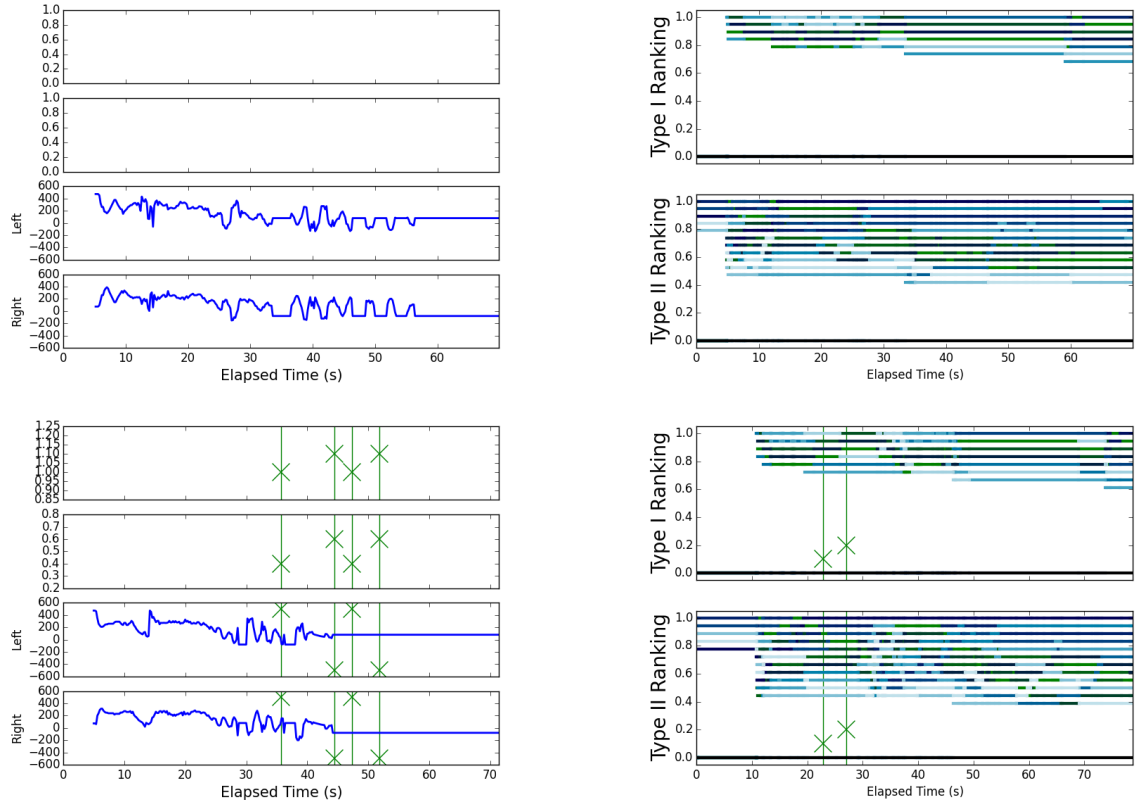
6.4.6 Summary

The navigation trials provided further evidence to support the answers to research questions found in the water sampling trials and helped to further clarify other questions. The trials found that a number of threshold predicate comparisons are found during the static analysis of the system that do not appear in the execution traces of the system. In these trials over half of the locations which contained threshold predicate comparisons in



(a) (b)

Figure 6.11: The position of the robot, wheel commands, and the ranking score of the modified threshold predicate comparison during treatment 3 (Source Code Error)



(a) (b)

Figure 6.12: The position of the robot, wheel commands, and the ranking score of the modified threshold predicate comparison during the treatment 4 (No Error)

the static analysis did not appear in the execution trace. This is similar to the number that where not executed in the water sampling trials. This demonstrates the ability of the approach to eliminate a large number of possible problem parameters before any other analysis is needed.

The trials provided more evidence that the comparisons are common throughout the operation of a robot system. On average there were 126.9 threshold predicate comparisons occurring per second during the trials. This is about half that of the water sampler, but still a large number of threshold predicate comparisons per second. The trials also further supported that different threshold predicate comparisons appear in different portions of the mission. Some appear in every second of the mission while a few only appear in less than 10% of the mission. The bursty nature of some thresholds is seen in the type II

error ranking graphs of the runtime trials. A group of thresholds appear at specific points in the graph, but then drop off quickly after appearing. The different times at which different thresholds appear may provide a way to group thresholds with high-level robot activity. If the user can watch the system during normal operation and group the behavior current it may be possible to match threshold predicate comparisons characteristics with runtime actions.

The trials with the navigation system further support our assumption that "flops" are not common during system execution. Only 0.4% of the threshold predicate comparisons resulted in flops. Additionally 75% of the flops were caused by one parameter. The low number of flops occurring compared to all comparisons still further backs our evidence that a flop is an important occurrence in the execution of a robot system. This backs our approach in determining when a flop occurs or when one is about to occur is very important for diagnosing configuration errors that manifest themselves during runtime.

The navigation treatments and trials offered more insight into how users marked runtime errors in the system. Users did not have any problem marking errors once they occurred. The number of errors marked was double the amount marked in the water sampling trials. This can be attributed to one trial staying in the error state of moving around the target location for over 60 seconds. The users marked 38 errors on this trial alone. Users did however show the same time gap between when the error occurred and when they marked the error. There is a delay of between 5 and 10 seconds in all of the trials. While the delay does not have large impact on type II error calculation, the delay may cause issues with computing results for type I errors.

Similar to the water sampling trials, the users had issues identifying the type of an error that was occurring during execution. The ambiguity between the two error types is clear in these trials. The users provided evidence that the robot stopping early could be interpreted as both a type I error (the robot stopped when it shouldn't) and a type II

error (the robot is not moving when it should be). They also mentioned confusion when the robot was moving about the final location. This provides evidence that selecting the correct error can be hard to choose. The problem may be further compounded when trying to mark the error quickly as the robot is executing.

For the approach to work the user must identify the correct error type so that the correct calculation and results are presented. A number of things can be done to help alleviate the issues. First, we can redefine the description of the error types to help the user better understand how to mark them in the system. This may help a little, but there will still be issues as the errors can be very ambiguous as shown in the navigation trials. Logically it seems that type II errors may be reported multiple times by the user. This means that we may be able to ask the user if the error is related to the previous error, and they say it is there may be a higher likelihood of the error arising from a type II error. Another important ability is the ability of the approach to capture all information and revisit errors at a later point in time. If they do not find the issue they can replay the mission and mark the other type of error to get a different set of suggestions. Finally, it may be best to simply allow the user to mark an error and report both sets of results to the user and allow the to make the decision based on all of the information provided to them. Work could also be done to provide a combined score that represents the most likely problem threshold.

The runtime analysis also did a reasonable job identifying the problematic threshold. In the trials in treatment 1 the modified parameter appeared in the top two or three positions of the rank graph. The system performed outstanding identifying the problem with treatment 2. When the comparisons began the threshold was at the bottom of the rank graphs. As the robot approached the goal, the modified threshold predicate comparison quickly approached the top of the rank graph. The threshold predicate comparison was in the top two suggested threshold predicate comparisons once the users

began to mark the errors. This provides evidence that the system can determine which threshold predicate comparisons are "close" to "flopping" and can also identify the most recently flopped thresholds.

6.5 Image Capture Experiment

6.5.1 Overview

The final system we examined was a UAV equipped with a camera that would search for a smiling person and capture their photo. The system contains around 25,800 lines of code in total. The static analysis identified 61 threshold predicate comparisons. 32 parameters are read and used to populate the thresholds in the predicates. The predicate comparisons are spread across 12 different files. The execution is spread across multiple ROS nodes. These nodes deal with camera image processing, UAV control, and communication. Figure 6.13 contains an image of the UAV and camera used for the trials in this section.

A trial begins with the UAV on the ground and off. The motors start and the UAV takes off and flies to a predetermined point and orientation. The UAV rotates and place and performs image processing on the image stream from the camera located on the UAV. Once the UAV identifies a face it remains at the same orientation attempts to center the face in the image frame. Once the UAV has the person centered in the image it snaps a photo and

Figure 6.13: The UAV and camera used during the Image Capture experiments

lands at a predetermined location. If at any point in time it looses track of the person, the UAV returns to circling in place.

We made the following three treatments to the system:

1. Type I Error: configuration parameter face_detection_limit. This parameter controls how many times the face must be seen in the images before the drone captures an image. This parameter is setup in the main launch file. With too low of a value, a face found in the noise of the image could trigger the image capture stop the mission before the person is found and captured. We will lower this threshold to cause a type I error.

2. Type II Error: configuration parameter min_coverage_limit. This parameter determines how much of the image the identified face must cover before the image is saved. If the limit is not exceeded the UAV will remain in the same position without taking the picture for a long period of time. We will raise the threshold to create a type II error.

3. Source Code Error: smile_detector_node_members.cpp Line 114. We will modify the range comparisons this line to result in the system only continuing to take a photo when the image is over the max_coverage_limit. The error produced here emulates a type II error.

6.5.2 Threshold Statistics RQ1

Information on the runtime characteristics of threshold predicate comparisons grouped by the parameter that loaded the value into the system can be found in Table 6.12. The columns are defined in Subsection 6.3.1. The runtime analysis observed 36 (59.0%) out of the 61 predicate locations identified in the static analysis. The used predicates represented

32 (59.3%) unique parameters out of the possible 41 found used to populate the thresholds used in the instrumented predicates. Each of the parameters was used in between one or three threshold predicate locations in the source code. A higher proportion of threshold predicate comparisons appear in the static analysis and in the system trace compared two the other systems. However, still over 40% of the threshold predicate comparisons found in the static analysis are not found in the system execution trace. This shows that not all of the code involved in a system is used and also shows that the approach can be used to reduce the problem search space without the requirement to perform other types of analysis on the system or runtime data.

Table 6.12: Runtime parameter statistics

	Locations	Comparisons	Frequency	Runtime %	Flops	True %	False %
collapse _deadband	1	17833	28.37	95	0	100	0
enable _baro _ctrl _mode	2	35633	56.69	95	0	100	0
enable _derv _fir _filter	1	17439	27.74	95	0	100	0
enable _pitch _ctrl	1	17831	28.37	95	0	0	100
enable _roll _ctrl	2	17835	28.37	96	0	0	100
enable _thrust _ctrl	1	17833	28.37	95	0	0	100
enable _thrust _iir _filter	1	17833	28.37	95	0	100	0
enable _yaw _ctrl	2	17836	28.38	96	0	0	100
face _detection _threshold	2	6700	10.66	59	23	58.55	41.45
gps _enable _mode	1	17833	28.37	95	0	0	100
max _coverage _limit	8	7188	11.44	29	0	66.44	33.56
max _pkt _size	1	64884	103.22	99	12	99.99	0.01
max _selfie _count	4	16554	26.34	72	673	81.71	18.29
min _coverage _limit	4	4707	7.49	29	57	47.27	52.73
min _displacement	1	6183	9.84	99	81	63.72	36.28
min _land _height	1	132	0.21	2	5	96.21	3.79
min _z _displacement	1	2243	3.57	41	8	7.09	92.91
selfie _control _rate	1	5	0.01	0	0	100	0
waypose _idle _timeout	1	617	0.98	98	18	90.28	9.72
mean	1.89	15111.53	24.04	72.89	46.16	58.49	41.51
median	1	17439	27.74	95	0	66.44	33.56
std	1.76	15106.26	24.03	34.87	153.36	43	43
minimum	1	5	0.01	0	0	0	0
maximum	8	64884	103.22	99	673	100	100
sum	36	287119	456.78	1385	877	1111.27	788.73

The combined six observed missions had a runtime of around 628 seconds. In total the instrumented code logged 287119 executions of the predicates instrumented during

the static analysis of the system's source code. This amounts to around 456 threshold comparisons per second. The number of comparisons per second for the parameters ranges from less than 0.01 per second to over 103 per second for the most frequent.

The runtime percentage distribution is shown in Figure 6.14. The definition and computation of runtime percentage is described in subsection 6.3.1. The runtime characteristics of this system are slightly different than those of the other two. There are not three very distinct groups as in the other two systems. Two groups are very well defined, the very frequent and the very rare. 12 parameters appear in over 90 % of the runtime. This is a higher proportion of parameters occurring very often compared to the other two systems. The other five parameters appear in between 30 and 80% of the runtime percentage, but do not have as strong of a grouping as the middle frequency parameters of the other two systems. There is still a strong grouping of parameters that occur very often and those that appear very infrequently.

We did not observe any evidence of the system experiencing problems handling the increased overhead of the threshold predicate comparison monitoring. To determine how many additional resources the messages from the threshold predicate comparisons we examined the percentage of messages that came from the instrumentation. In total 16.1% of the messages were from instrumented threshold predicate comparisons. The overhead is not completely unreasonable, but the approach does consume some resources.

This subsection has shown that many of the findings that appear in the two other systems also appear in the image capture trials. Threshold predicate comparisons are very common throughout the execution of a mission by the robot system. However, different sources of the thresholds within the comparisons have different rates of occurrence throughout the systems execution. In this trial the very common and very rare comparisons frequency are again obvious. However there is not a defined grouping in the middle frequencies. Again many of the threshold predicate comparisons that are

identified in the static analysis do not appear in the system execution trace.

The trials for this system differed from the other two systems in a few ways. First, the trials had the highest amount of threshold predicate comparisons in the execution trace and the highest frequency of threshold predicate comparisons. There are around 100 more per second in these trials than the water sampler and over 3.6 times the amount of comparisons compared to the navigation trials. These trials are also similar to the navigation trials, in the number of locations a parameter is used in a threshold predicate comparison. In these trials they were commonly used in 2 or even 3 different locations. However, the mapping of parameters to locations is still relatively small, and still allows the pinpointing of a problematic parameter by finding the threshold predicate comparison in which it was used.

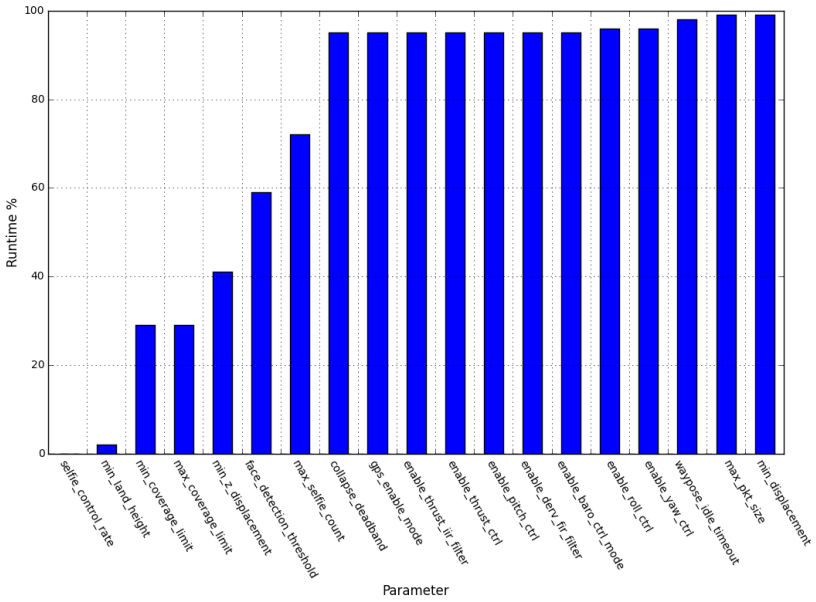


Figure 6.14: Percentage of total seconds in which a parameter is used in a predicate

6.5.3 Flops RQ2

In the 8 trials there are a total of 877 flops. This corresponds to 0.31% of the threshold predicate comparisons in the execution trace. This is a very small percentage and shows

further evidence that flops are a rare occurrence. This value is similar to that of the flop percentage of the navigation system trials. One parameter is responsible for 76.7% of the flops in the system execution trace. This again provides evidence that some parameters do flop more often than other parameters. Similar to the other systems 11 (57%) of the parameters are used in predicates that never change values. The results from the image capture trials offer more evidence to that a flop is an important event during the execution of a robot system.

6.5.4 User Marks RQ3

Table 6.15 displays information about user marks for the image capture trials. Users marked a total of 39 type I errors and 78 type II errors. This is similar to the amount marked in the navigation system. The mean number of marks per trial across all users is 14.6. The maximum number of marks in one trial is 29 and the minimum number of marks in one trial is 0. One user only marked 2 errors in one trial and did not mark any other errors. All of the other users marked upwards of 20 errors.

As with the other systems type II or type II like errors caused many more marks than the trials involving type I errors. The marks in these trials continue to show that different users will mark different errors at different rates across different systems. If the robot system continues to struggle to complete the task the users will mark many errors. This may also help to identify the true difference between type I and type II errors. The larger number of marks around a type II error show that the robot is stalled if the user marks many times in a row. This may help disambiguate which type of error the user is truly marking.

In the trials users marked fewer type I errors (39) than type II errors (78). These trials provide evidence that the user may not be able to successfully determine which type of

error is occurring during operation, but they can determine that an error is occurring. However, the large number of markings for the trials may help to provide evidence when the system is encountering a type II error. Further work needs to be done to help users tell the difference between these two types of errors.

Table 6.13 displays the confusion matrix for treatments 1 and 2. From this matrix, it is very easy to see the confusion between type I and type II errors by the users in the trials. Again the users marked far more type II errors during the treatment with type I errors. However, in this case it may be because the type II error was not as obvious as other systems. There was a near equal marking of type I and type II errors in the second treatment. These results again show that users struggle to correctly identify type I and type II errors and that something must be done to allow the system to better handle misidentified errors or to allow users to better mark the error types.

Table 6.13: Confusion Matrix for Type I and Type I Errors and Treatments

	Type I Marks	Type II Marks
Treatment Type I	2	15
Treatment Type II	26	25

As with the other two sets of trials users marked errors in the code treatment. This shows that users will mark errors that occur due to code faults as well as configuration errors. While this may not be an issue it may lead to some confusion when changing parameters does not cause a change in the error behavior of the robot.

Finally, the users marked 0 errors in one of the clean trials and only 7 errors in the other clean trial. These are the two smallest mark totals for any trial. This provides evidence that marks are not as common during clean trials as other trials.

Table 6.14 displays the average delay in marking an error after the changed threshold has flopped or should have flopped. In this system there is a delay between 1.7 and 25 seconds before marking the error. Different from the other systems there is not the clear

difference between marking times in the two experiments. There is still a large mean time from the start of the error to marking the issue in the systems. As stated previously, the delay is not an issue during type II errors, but may cause problems after a Type I error has occurred.

Table 6.14: Delays from error to the first mark by the users.

	User 1	User 2	User 3	User 4	Mean
Treatment 1 Trial 1	11.69	25.07	11.45	21.63	17.46
Treatment 1 Trial 2	3.12	3.78		0.96	2.62
Treatment 2 Trial 1	6.99	12.37		12.07	10.47
Treatment 2 Trial 2	1.71	9.97		9.91	7.19

The results of the users marking errors during the image capture trials demonstrate that users can identify problems in the robot system. It also provides further evidence that there is confusion between the two types of errors. However, there continues to be evidence that type II errors will be marked much more frequently in the system in comparison to type I errors even if they are incorrectly marked by the user.

Table 6.15: User marks during the image capture trials

	User 1		User 2		User 3		User 4		Total	
	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
Treatment 1 Trial 1	0	3	0	1	0	2	0	2	0	8
Treatment 1 Trial 2	0	3	0	2	0	0	2	2	2	7
Treatment 2 Trial 1	6	1	3	1	0	0	10	1	19	3
Treatment 2 Trial 2	2	7	3	3	0	0	2	12	7	22
Treatment 3 Trial 1	3	5	1	3	0	0	2	11	6	19
Treatment 3 Trial 2	0	6	0	2	0	0	0	9	0	17
Treatment 4 Trial 1	0	0	0	0	0	0	0	0	0	0
Treatment 4 Trial 2	2	1	1	1	0	0	2	0	5	2
mean	1.62	3.25	1	1.62	0	0.25	2.25	4.62	4.88	9.75
median	1	3	0.50	1.50	0	0	2	2	3.50	7.50
std	2.13	2.55	1.31	1.06	0	0.71	3.28	5.13	6.38	8.45
minimum	0	0	0	0	0	0	0	0	0	0
maximum	6	7	3	3	0	2	10	12	19	22
sum	13	26	8	13	0	2	18	37	39	78

6.5.5 Runtime Analysis Results RQ4

Table 6.16 displays the average scores given to the marked errors by the analysis in the two treatments with modified parameters. No type I error marks occurred in the first trial because the error situation did not arise during the execution of the system. The threshold appeared in the upper part of the ranks in the Type II error reports. In the second trial again the type I marks are occur in parts of the system where the threshold has flopped, but other parts of the system prevented the system from continuing on and displaying the error. These two missions did not perform as well as expected, but the previous trial sets show that the system is able to identify the most recently flopped predicates and can assist users in determining when an error arose.

The second treatment again shows how the approach is able to identify threshold predicate comparisons that are about to "flop". In the first trial the marks occur when the UAV modified parameter is in the top four threshold predicate comparisons. It has a high-ranking score on the error marks. In the second mission the predicate is in the top of the rankings until it drops from comparisons and the mission is aborted after the robot fails to capture the image. Many of the marks came after comparisons to the modified parameter end so the score for these missions is lower than that of the others.

Table 6.16: Average score produced on marked errors and average ranking

	Type I Rank	Type I Score	Type II Rank	Type II Score
Treatment 1 Trial 1	0.00	9999.00	**0.80**	0.74
Treatment 1 Trial 2	**0.98**	36.68	0.88	0.50
Treatment 2 Trial 1	0.20	7895.84	**0.96**	0.52
Treatment 2 Trial 2	0.00	9999.90	**0.15**	8181.87

6.5.6 Trial Summaries

More details on the individual trials will be presented in the following subsections grouped by trial. The results will help to further clarify the research questions. In the

following subsections we present graphics for each individual trial. The first graphics contain the X, Y, Z, and yaw position of the UAV. The UAV should identify the person and capture the image when the yaw is at a value of around 2.2. After capturing the image one can see the UAV landing by the decrease in height.

The other figure displayed for the missions are the rank scores through time. They show the rank scores of the type I and type II errors for all of the threshold predicate comparisons in the mission. In these graphics the red line represents the modified threshold in the trial if there is a modification. All of the other lines are color-coded and each represent a different threshold predicate comparison within the code. If the threshold predicate comparison had a score of 99999 it was given a rank score of 0. More thresholds were present at points in time than appeared within the graphs due to this zeroing of scores.

User marks are indicated in both sides of the figure using vertical lines with x marks on them. In the figures the blue marks represent a user marked a type I error and green marks represent type II errors as marked by the users.

6.5.6.1 Treatment 1 - Type I Error

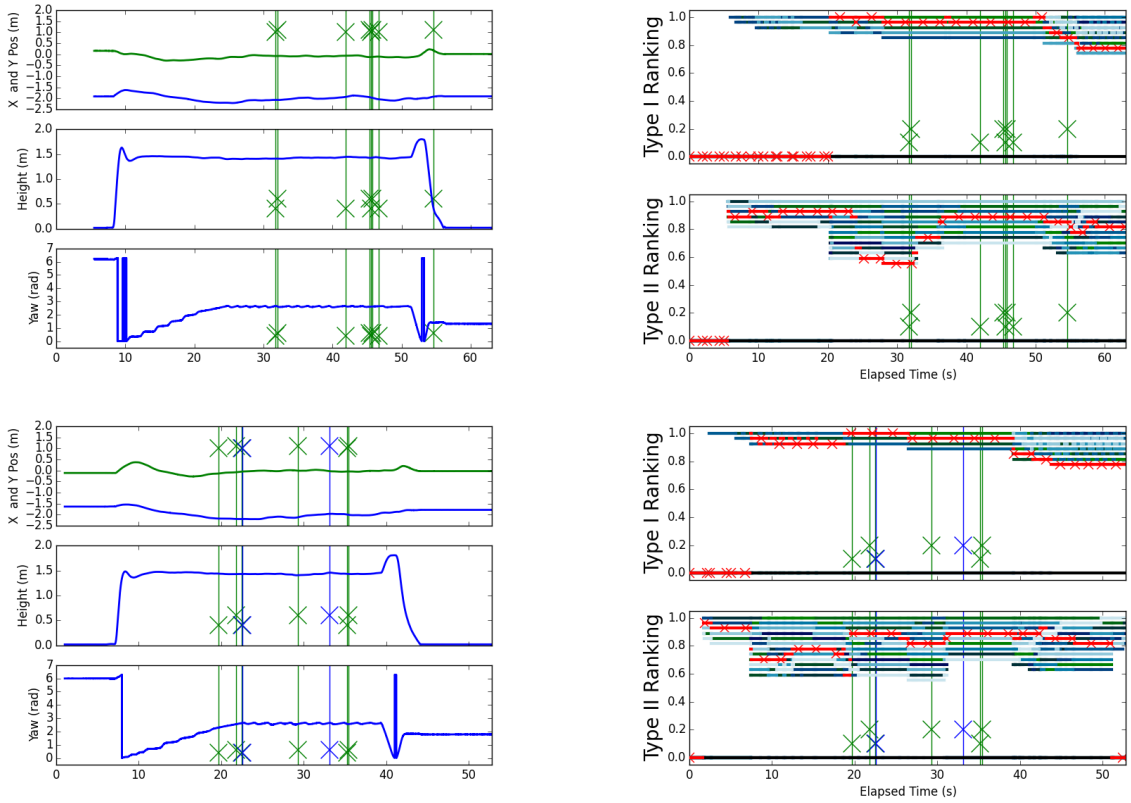
Figure 6.15 displays location and threshold values for the two trials with the first treatment. In both trials the UAV took off and circled and found the target. The flopping of the threshold did not cause an immediate error and the UAV did not change behavior as it did in previous runs with the modified thresholds. However, the users did mark errors after the UAV obtained a yaw pointing at the person and maintained that position. This is obvious in the graphs of the yaw position. The lack of concrete type I errors is not an issue as the previous trial sets have shown that the approach can identify recently flopped thresholds. The results from the first treatment also show that the user will mark errors that are perceived after a short wait of the robot maintaining the same position.

They mark many type II errors after the UAV attains the yaw pointing to the position of the person.

As in previous trial sets both of the trials show a large increase in the number of threshold predicate comparisons present in the graph during what appear to be key portions of the mission. The large increase in the type II graphs appear to correspond to the moment in which person first comes into view of the UAV's camera. This shows that there is a lot of activity in the code while it tries to lock in on the person's location. There is also a second increase in present threshold predicate comparisons when the UAV is landing. There are also more threshold predicate comparisons in the type I graph showing that more have flopped when the mission is completed. These graphics bolster the argument that there are more active threshold predicate comparisons during transitional or active parts of the mission.

6.5.6.2 Treatment 2 - Type II Error

Figure 6.16 displays location and threshold values for the two trials with the second treatment. In the first mission the UAV identified the person and tried to meet the modified minimum coverage threshold and failed. It then rotated twice before it identified the person again and finally captured the image. Users began marking errors immediately after the UAV paused when it identified the person and did not take a picture and land. There is another grouping of markings when the UAV was directly pointed at the person on the second rotation and a final group of markings when the UAV landed. This provides evidence that the users were able to identify that errors were occurring when the UAV should have been capturing the image. However, nearly all of the markings are of type I errors, which provides further evidence that users struggle to correctly identify the underlying type of error that is occurring in the system. The marking of errors while the UAV is landing is also an interesting feature. It's almost as if the users were used to

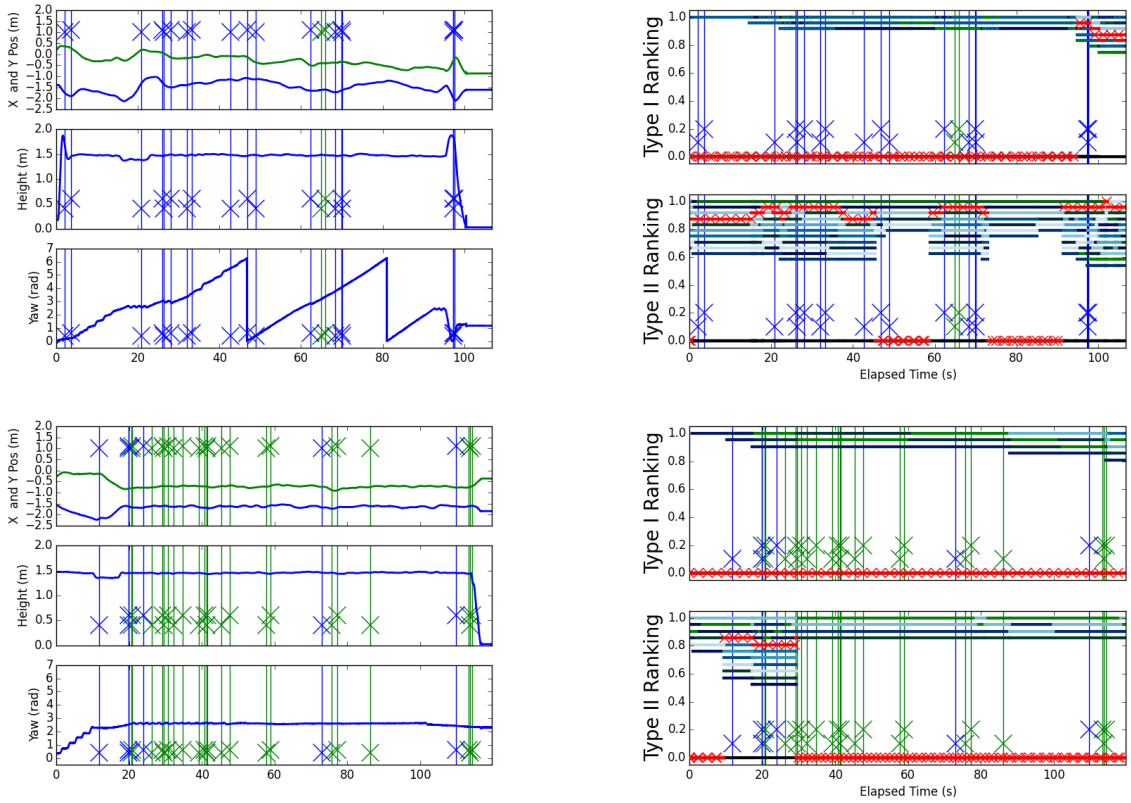


(a) (b)

Figure 6.15: The position of the UAV and the ranking score of the modified threshold predicate comparison during treatment 1 (Type I Error)

the UAV circling at that point and did not expect the mission to complete at that point in time.

In the second trial for this treatment the UAV took off and began searching. Once it identified the user, the UAV tried to meet the minimum threshold by changing X and Y location of the UAV. However, unlike the first trial, the UAV became stuck in the state after it did not meet the threshold after some time. After 2 minutes of flight the mission was aborted. What is most interesting about this mission is that the code stopped executing many of the threshold predicate comparisons after failing to compete the image capture the first time. This may indicate that the robot system entered an error state or that there is an error in the logic of the mission. However, while it was comparing values the modified threshold predicate comparison was in the top 4 of the values in the rank



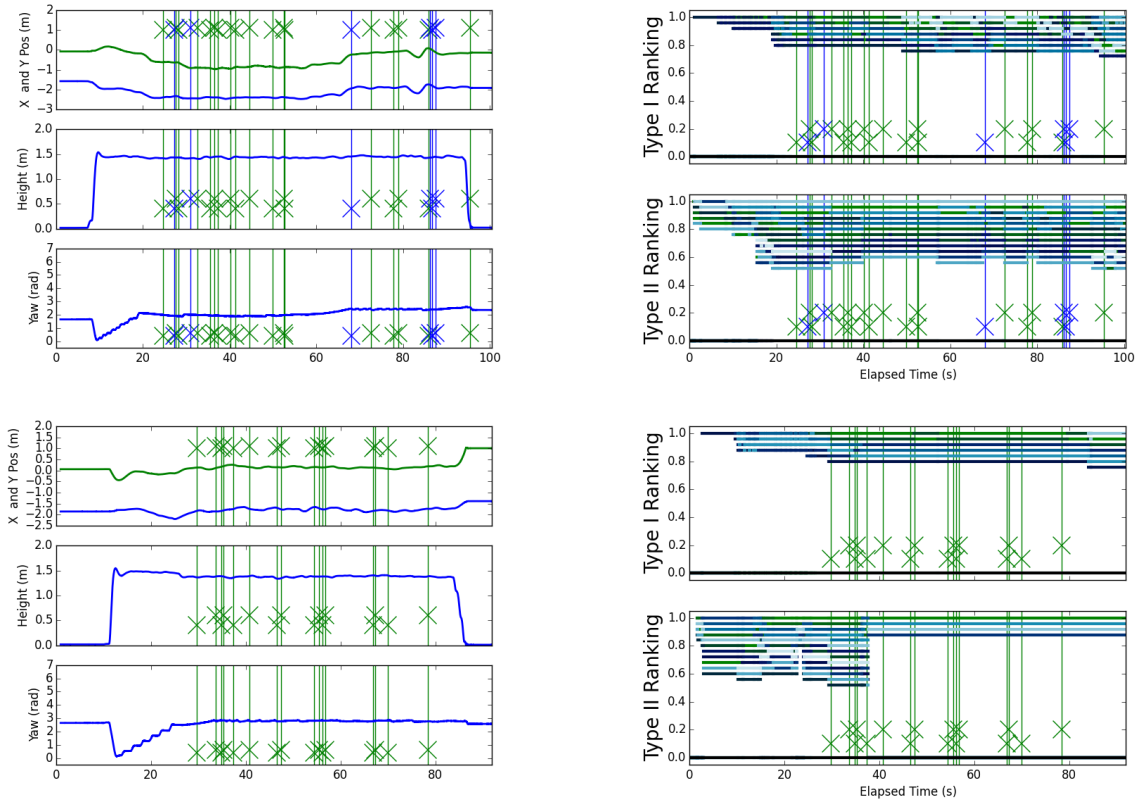
(a) (b)

Figure 6.16: The position of the UAV and the ranking score of the modified threshold predicate comparison during treatment 2 (Type II Error)

graphs. As with previous trials most of the marks came after the robot was "stuck" for 10 or more seconds. The users continued to mark errors until the mission was aborted. Also interesting is the fact that the users marked the opposite error type in this mission. They marked 3 times as many type II errors. This may be because the UAV was stationary while trying to meet the threshold and in the stuck state. This provides evidence that the type of error marked has a lot to do with the motion that the robot system is performing and not the underlying error type.

6.5.6.3 Treatment 3 - Source Code Error

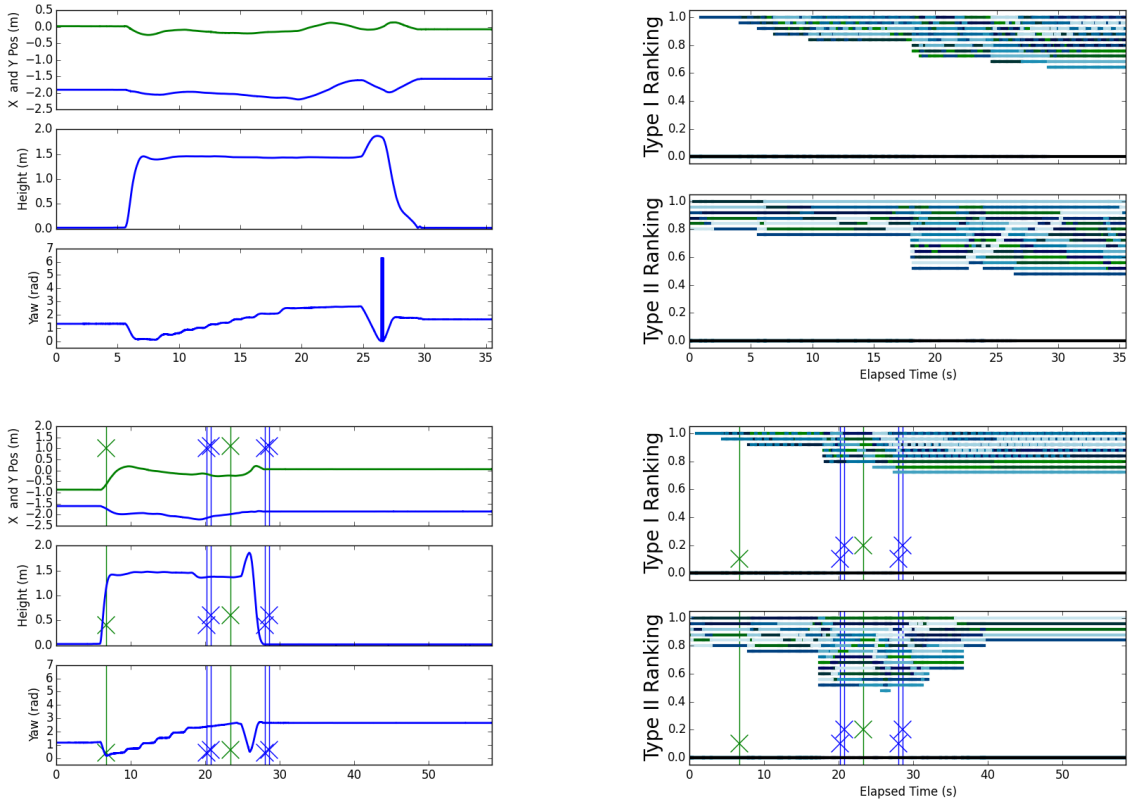
Figure 6.17 displays location and threshold values for the two trials with the source code error treatment. In both cases the UAV began the search, identified the person, and then



(a) (b)

Figure 6.17: The position of the UAV and the ranking score of the threshold predicate comparisons during treatment 3 (Source Code Error)

become stuck in a state that did not continue on the mission. Eventually the mission was aborted after around 100 seconds. The characteristics of user markings during these two missions are very similar to other missions where the robot becomes stuck. After a delay of around, 10 seconds the user begin marking many errors and continue to do so until the mission is aborted. The rank graphs again show something very similar to all of the other trials. The thresholds are active while the UAV is actively trying to search for the person. As with one of the trials in treatment 2 the drone enters a state where it stops comparing values to thresholds after it fails to meet the threshold and capture the image. The results of these trials again show that users will mark errors in the source code as type I and type II errors and that there are active portions of threshold when the system is running.



(a) (b)

Figure 6.18: The position of the UAV and the ranking score of the threshold predicate comparisons during treatment 4 (No Error)

6.5.6.4 Treatment 4 - Clean

Figure 6.18 displays location and threshold values for the two trials without any modification. In both trials the UAV took off, found the person, and captured the image within 30 seconds. In the first trial no marks were recorded. Interestingly in the second trial, marks did get recorded. This is one of the few times that marks appeared before 10 seconds passed after the UAV reached the target location. The exact reasoning of these marks is not known for certain. However, there were the fewest marks of any trial for the image capture trial. This provides evidence that users do not mark correctly operating systems as often as they would work errors in a system with configuration errors, if they mark them at all.

6.5.7 Summary

The image capture trials provided further evidence to support the answers to research questions for the water sampling and navigation trials and helped to further clarify other questions. A consistent finding between all of the trials is that a number of threshold predicate comparisons found in the source code do not appear in the system execution trace. The number of threshold predicate comparisons that did not appear in the execution trace is slightly lower in this instance, but it was still over 40%. This demonstrates the ability of the approach to eliminate a large number of possible problem parameters, before the need of any type of additional analysis.

The image capture trials provide more evidence to show that threshold predicate comparisons are frequent during operation. This experiment had the highest number of predicate comparisons per second at 456. The trials also further supported that different threshold predicate comparisons appear at different times and different rates in different portions of the mission. A large portion of them appear over 90% of the time in image capture experiment and two of them appear less than 10% of the time. The bursty nature of predicate appearing during important parts of the mission can be seen in the type II ranking graphs for all of the trials.

Flops were extremely rare throughout all of the systems execution in the image capture trials. Only 0.31% of the threshold predicate comparisons resulted in flops. This provides further evidence to show that a flop is an important occurrence that can help users identify why the robot system changed behavior or did not change behavior.

As with previous trial sets users did not have problems marking errors when they occurred, but they did have problems marking the correct type of error. Users marked both types of errors in all trials. Hopefully, more familiarity with a system will help the users identify the types of errors more accurately. Otherwise, steps must be taken to help

users identify the correct type of error or allow the system to determine which error type to use for the ranking score calculation. For the two trials on the clean treatment, users marked 0 errors and the fewest number of errors respectively.

The runtime analysis also did a reasonable job identifying the problematic threshold when it was the cause of the error and the correct type of error was marked. The system correctly identifies the most recently flopped threshold. When the threshold has not been met you can see the modified threshold rise towards the top as it gets past the point where it should have flopped. This provides evidence that the system can help to determine which threshold predicate comparisons are close to flopping and can also identify the most recently flopped thresholds.

6.6 Summary

We performed experiments with three robot systems to answer a number of research questions on the runtime characteristics of threshold predicate comparisons, the user identification of runtime errors, and the success of the system on identifying the problematic threshold predicate comparisons.

We found that threshold predicate comparisons occur very frequently during the execution of a robot system. The number of threshold predicate comparisons per second present in the systems ranged between 126 and 456. We found that between 40 and 60 percent of the threshold predicate comparisons identified in the static analysis did not appear in the execution trace. This allows for a simple filtering of possible problematic thresholds. We also found evidence that some parameters are used throughout the whole execution of the system and others only appear in small portions of the execution. This may allow for a more fine-grained approach to determining problematic thresholds in the future.

We also found evidence that "flops" are rare during system execution. Flops were found to happen in between 0.03-0.4% of the time on threshold predicate comparisons in the systems. This provides evidence that our assumption that "flops" are important events that signal a change in the behavior of the robot system is correct. It also provides evidence that being able to identify flops or near flops is of use to a user trying to discover configuration errors in the robot system.

Users did not have a problem identifying when an error occurs and they mark them many times during the course of execution. They also do not mark error free trials as often as other trials as evidenced in the clean trials. The three systems' trials provide evidence that the users are not as accurate in identifying the types of errors that occur during system operation, as we would hope. There is major confusion between type I and type II errors. They often marked the incorrect type during the trials for the opposite type. As users become more familiar with the system that they are running we assume that they would become more familiar with the types of errors present. If they do not improve at marking the correct error type something must be done to allow the system to accommodate incorrect error markings or help the user correctly identify errors.

There is evidence that there is a delay between when the error occurs in the running system and the user first marks an error. Type II errors took around twice as long to mark as Type I errors as well. Type II calculations should not be affected by this delay. The threshold will still be close to flopping and appear in the top of the rankings. Type I errors may be affected if the system continues on with another task and other threshold predicate comparisons flop.

Finally, in the approach did a reasonable job of identifying which thresholds were the causes of marked errors. The system can reasonably identify the most recently "flopped" threshold predicate comparison on type I errors. It also did a reasonable job identifying which thresholds are about to flop when the robot should be continuing on with the

mission. When the users correctly marked the error type the threshold was in the top portion of the rankings.

Chapter 7

Conclusions

The validation and selection of proper configuration on large systems has been recognized as a difficult task in our community. Robotic systems face many of the same issues. The systems we examined in this work showed the many possible configuration options are available across the spectrum of robotic systems. Poorly set configuration options often cause the robot system to behave in a manner that does not complete the desired mission. Co-robot systems offer the ability for the user of the system to integrate their capabilities and with the robots ability. This work aims to use that ability to identify errors and leverage a static analysis and instrumentation technique to determine which configuration parameter may be causing troublesome in the system. We have developed an approach the user to identify problems and offer suggestions on which parameters to change. Our work expands prevous software engineering work on large configuration spaces into co-robotic systems. It appears to be one of the first works examining and instrumenting configuration options within robot systems and aims to help users diagnose and solve configuration problems.

In this thesis, we developed two static analysis tools for Python and C++ ROS nodes. The static analysis tools are able to identify threshold predicate comparisons within

the nodes. A threshold predicate comparison is defined as a predicate in a branching statement on which an exposing statement has a control and or data dependency. This means that the branching statement determines the execution or data present in a call that exposes the execution of part of the robot system to the rest of the system. We also developed methods to take the data produced by instrumented threshold predicate comparisons and offer recommendations on which configuration parameters to change when a user marks and error. We validated the static analysis by manually examining the performance of the analysis on 20 nodes and with suite of 70 test nodes.

After the development and validation of the approach we examined threshold predicate comparisons on over 100 open source robot systems. We found that one third of the packages contained threshold predicate comparisons and on average each of the packages contained 5 threshold predicate comparisons. The comparisons are often present in only a small number of files in each repository.

Finally, we used our analysis on three different robotic systems to determine the runtime characteristics of threshold predicate comparisons, how often the values in a particular predicate comparisons flopped, how users marked errors in a running system, and how well the approach could suggest the correct fix for a threshold comparison. We found that threshold predicate comparisons occur very frequently while the system is in operation, but many of the values do not occur in the system trace. We also found that flops are relatively rare while the system is in operation that indicates that a flop is an important event. Users are able to mark and identify errors, but are not able to easily determine the type of error. This may cause problems for the suggestion and determining of the predicate that is error. There was also a delay between when the error occured in running system code and the users marking their first error. Finally, when the correct error was chosen the system did a reasonable job in highlighting which parameter was the cause of the error.

There are a few key limitations to our work. First, it relies on the system setting up configuration options through the standard interface. If values are defined or read into the system in a different way, they will not be captured by our analysis. Our approach also relies on the system containing a state machine containing the threshold predicate comparisons that has a behavior that is recognizable to a system observer. The system must also be comprised of different nodes communicating to one another, will be no threshold predicate comparisons to identify. If they are not present the user will not be able to mark errors. Finally, the largest limitation of our system is that it depends on the user marking the correct type of error to produce good score estimates.

In the future work can be done to help the user correctly identify the type of error that is occurring or allow the system to work on only the marking of an error and not require the correct error type. We also want to incorporate other source of setup parameters beside values loaded from the ROS parameter server. These other sources can include integer constants and header constants within the source code. Finally, more can be done to characterize how a threshold behaves in the system and if we can track different groupings of threshold predicate comparisons during execution to determine how the system is performing.

Bibliography

[1] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 159–172, ACM, 2011. 1, 2

[2] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 70, 2015. 1, 2

[3] Dronecode, "Arducopter Configuration Parameters." http://copter.ardupilot.com/wiki/configuration/arducopter-parameters/, 2015. [Online; accessed 10-July-2015]. 1

[4] R. Robotics, "Baxter Robot." http://www.rethinkrobotics.com/baxter, 2015. [Online; accessed 10-July-2015]. 1

[5] C. Fitzgerald, "Developing baxter," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, pp. 1–6, IEEE, 2013. 1

[6] R. Robotics, "Baxter Robot Source Repository." https://github.com/RethinkRobotics, 2015. [Online; accessed 10-July-2015]. 1

[7] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, 2009. 1, 3.1.2

[8] "ROS Navigation Stack." http://wiki.ros.org/navigation, 2015. [Online; accessed 10-July-2015]. 1, 6.2.1, 6.4

[9] G. Bekey, R. Ambrose, V. Kumar, A. Sanderson, B. Wilcox, and Y. Zheng, "Wtec panel report on international assessment of research and development in robotics," *A roadmap for US Robotics From Internet to Robotics 2013 Edition*, 2013. 1, 2.4

[10] J.-P. Ore, S. Elbaum, A. Burgin, and C. Detweiler, "Autonomous aerial water sampling," *Journal of Field Robotics*, 2015. 1, 2.4, 5.1, 6.2.1, 6.3

[11] J. Gray, "Why do computers stop and what can be done about it?," in *Symposium on reliability in distributed software and database systems*, pp. 3–12, Los Angeles, CA, USA, 1986. 2

[12] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding bgp misconfiguration," in *ACM SIGCOMM Computer Communication Review*, vol. 32, pp. 3–16, ACM, 2002. 2

[13] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and dealing with operator mistakes in internet services.," in *OSDI*, vol. 4, pp. 61–76, 2004. 2

[14] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?," in *USENIX Symposium on Internet Technologies and Systems*, vol. 67, Seattle, WA, 2003. 2

[15] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and validating database system administration.," in *USENIX Annual Technical Conference, General Track*, pp. 213–228, Boston, MA, 2006. 2, 2.3.2

[16] A. Rabkin and R. H. Katz, "How hadoop clusters break," *Software, IEEE*, vol. 30, no. 4, pp. 88–94, 2013. 2

[17] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, *et al.*, "What bugs live in the cloud?: A study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–14, ACM, 2014. 2

[18] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, "Configurations everywhere: Implications for testing and debugging in practice," in *Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 215–224, ACM, 2014. 2

[19] R. Rabiser, P. Grunbacher, and M. Lehofer, "A qualitative study on user guidance capabilities in product configuration tools," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pp. 110–119, IEEE, 2012. 2

[20] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 131–140, IEEE, 2011. 2

[21] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 307–319, ACM, 2015. 2

[22] F. Behrang, M. B. Cohen, and A. Orso, "Users beware: preference inconsistencies ahead," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 295–306, ACM, 2015. 2

[23] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003. 2

[24] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *Software Engineering, IEEE Transactions on*, vol. 23, no. 7, pp. 437–444, 1997. 2.1

[25] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr, "Software fault interactions and implications for software testing," *Software Engineering, IEEE Transactions on*, vol. 30, no. 6, pp. 418–421, 2004. 2.1

[26] H. Srikanth, M. B. Cohen, and X. Qu, "Reducing field failures in system configurable software: Cost-based prioritization," in *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*, pp. 61–70, IEEE, 2009. 2.1

[27] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter, "Effective and scalable software compatibility testing," in *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 63–74, ACM, 2008. 2.1, 2.2

[28] C. Yilmaz, M. B. Cohen, A. Porter, *et al.*, "Covering arrays for efficient fault characterization in complex configuration spaces," *Software Engineering, IEEE Transactions on*, vol. 32, no. 1, pp. 20–34, 2006. 2.1

[29] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 129–139, ACM, 2007. 2.1

[30] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *Software Engineering, IEEE Transactions on*, vol. 34, no. 5, pp. 633–650, 2008. 2.1

[31] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter, "Feedback driven adaptive combinatorial testing," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 243–253, ACM, 2011. 2.1

[32] X. Qu, M. Acharya, and B. Robinson, "Impact analysis of configuration changes for test case selection," in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pp. 140–149, IEEE, 2011. 2.1

[33] B. Robinson and L. White, "On the testing of user-configurable software systems using firewalls," *Software Testing, Verification and Reliability*, vol. 22, no. 1, pp. 3–31, 2012. 2.1

[34] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 75–86, ACM, 2008. 2.1

[35] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 445–454, ACM, 2010. 2.1

[36] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso, "Regression testing in the presence of non-code changes," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pp. 21–30, IEEE, 2011. 2.1

[37] L. Keller, P. Upadhyaya, and G. Candea, "Conferr: A tool for assessing resilience to human configuration errors," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pp. 157–166, IEEE, 2008. 2.1

[38] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 140–151, ACM, 2014. 2.2

[39] S. Nadi, T. Berger, C. Kastner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Transactions on Software Engineering*, vol. 41, pp. 820–841, 2015. 2.2

[40] A. Kenner, C. Kästner, S. Haase, and T. Leich, "Typechef: toward type checking# ifdef variability in c," in *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pp. 25–32, ACM, 2010. 2.2

[41] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 58–68, IEEE, 2012. 2.2

[42] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, "Range fixes: Interactive error resolution for software configuration," *IEEE Transactions on Software Engineering*, 2014. 2.2

[43] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration debugging as search: Finding the needle in the haystack.," in *OSDI*, vol. 4, pp. 6–6, 2004. 2.3.1

[44] Y.-Y. Su, M. Attariyan, and J. Flinn, "Autobash: improving configuration management with operating system causality analysis," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 237–250, ACM, 2007. 2.3.1

[45] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 312–321, IEEE Press, 2013. 2.3.1

[46] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis.," in *OSDI*, pp. 237–250, 2010. 2.3.1

[47] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software.," in *OSDI*, pp. 307–320, 2012. 2.3.1

[48] S. Zhang and M. D. Ernst, "Which configuration option should i change?," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 152–163, ACM, 2014. 2.3.1

[49] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pp. 193–202, IEEE, 2011. 2.3.1

[50] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 445–456, ACM, 2014. 2.3.1

[51] D. Jin, M. B. Cohen, X. Qu, and B. Robinson, "Preffinder: getting the right preference in configurable software systems," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 151–162, ACM, 2014. 2.3.1

[52] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou, "Automated configuration bug report prediction using text mining," in *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pp. 107–116, IEEE, 2014. 2.3.1

[53] S. Zhang and M. D. Ernst, "Proactive detection of inadequate diagnostic messages for software configuration errors," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 12–23, ACM, 2015. 2.3.1

[54] F. Oliveira, A. Tjang, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Barricade: defending systems against operator mistakes," in *Proceedings of the 5th European conference on Computer systems*, pp. 83–96, ACM, 2010. 2.3.2

[55] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone, "Beyond the rainbow: self-adaptive failure avoidance in configurable systems," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering,(FSE-22), Hong Kong, China*, pp. 377–388, 2014. 2.3.2

[56] R. Potharaju, J. Chan, L. Hu, C. Nita-Rotaru, M. Wang, L. Zhang, and N. Jain, "Confseer: leveraging customer support knowledge bases for automated misconfiguration detection," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1828–1839, 2015. 2.3.2

[57] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury, "Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications," in *Proceedings of the 6th international conference on Autonomic computing*, pp. 169–178, ACM, 2009. 2.3.2

[58] W. Zheng, R. Bianchini, and T. D. Nguyen, "Automatic configuration of internet services," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 219–229, 2007. 2.3.2

[59] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 687–700, 2014. 2.3.2

[60] E. Kiciman and Y.-M. Wang, "Discovering correctness constraints for self-management of system configuration," in *Autonomic Computing, 2004. Proceedings. International Conference on*, pp. 28–35, IEEE, 2004. 2.3.2

[61] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure.," in *OSDI*, vol. 4, pp. 245–257, 2004. 2.3.2

[62] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff, "Mining for misconfigured machines in grid systems," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 687–692, ACM, 2006. 2.3.2

[63] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "Vconf: a reinforcement learning approach to virtual machines auto-configuration," in *Proceedings of the 6th international conference on Autonomic computing*, pp. 137–146, ACM, 2009. 2.3.2

[64] D. Cooray, S. Malek, R. Roshandel, and D. Kilgore, "Resisting reliability degradation through proactive reconfiguration," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 83–92, ACM, 2010. 2.3.2

[65] X. Bu, J. Rao, and C.-Z. Xu, "A reinforcement learning approach to online web systems auto-configuration," in *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pp. 2–11, IEEE, 2009. 2.3.2

[66] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra, "Adaptive self-tuning memory in db2," in *Proceedings of the 32nd international conference on Very large data bases*, pp. 1081–1092, VLDB Endowment, 2006. 2.3.2

[67] B. K. Debnath, D. J. Lilja, and M. F. Mokbel, "Sard: A statistical approach for ranking database tuning parameters," in *Data Engineering Workshop, 2008. ICDEW 2008. IEEE 24th International Conference on*, pp. 11–18, IEEE, 2008. 2.3.2

[68] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. Tung, "A new approach to dynamic self-tuning of database buffers," *ACM Transactions on Storage (TOS)*, vol. 4, no. 1, p. 3, 2008. 2.3.2

[69] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 1246–1257, 2009. 2.3.2

[70] E. Guizzo and T. Deyle, "Robotics trends for 2012," *IEEE Robotics & Automation Magazine*, vol. 19, no. 1, pp. 119–123, 2012. 2.4

[71] D. Anthony, S. Elbaum, A. Lorenz, and C. Detweiler, "On crop height estimation with uavs," in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pp. 4805–4812, IEEE, 2014. 2.4, 5.1

[72] A. Green, H. Hüttenrauch, M. Norman, L. Oestreicher, and K. S. Eklundh, "User centered design for intelligent service robots," in *Robot and Human Interactive Communication, 2000. RO-MAN 2000. Proceedings. 9th IEEE International Workshop on*, pp. 161–166, IEEE, 2000. 2.4

[73] M. Pérez-Ruíz, D. C. Slaughter, F. A. Fathallah, C. J. Gliever, and B. J. Miller, "Co-robotic intra-row weed control system," *Biosystems Engineering*, vol. 126, pp. 45–55, 2014. 2.4

[74] B. Hayes and B. Scassellati, "Challenges in shared-environment human-robot collaboration," *learning*, vol. 8, p. 9, 2013. 2.4

[75] R. Woodman, A. F. Winfield, C. Harper, and M. Fraser, "Building safer robots: Safety driven control," *The International Journal of Robotics Research*, vol. 31, no. 13, pp. 1603–1626, 2012. 2.4

[76] J. Sattar and G. Dudek, "Reducing uncertainty in human-robot interaction: A cost analysis approach," in *Experimental Robotics*, pp. 81–95, Springer, 2014. 2.4

[77] O. Pettersson, "Execution monitoring in robotics: A survey," *Robotics and Autonomous Systems*, vol. 53, no. 2, pp. 73–88, 2005. 2.4

[78] S. Zaman, G. Steinbauer, J. Maurer, P. Lepej, and S. Uran, "An integrated model-based diagnosis and repair architecture for ros-based robot systems," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 482–489, IEEE, 2013. 2.4

[79] J. P. Mendoza, M. Veloso, and R. Simmons, "Motion interference detection in mobile robots," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 370–375, IEEE, 2012. 2.4

[80] R. Golombek, S. Wrede, M. Hanheide, and M. Heckmann, "Learning a probabilistic self-awareness model for robotic systems," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pp. 2745–2750, IEEE, 2010. 2.4

[81] A. Podgurski, L. Clarke, *et al.*, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *Software Engineering, IEEE Transactions on*, vol. 16, no. 9, pp. 965–979, 1990. 3.1.1

[82] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011. 3.1.1, B

[83] F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, no. 3, pp. 121–189, 1995. 3.1.1

[84] P. J. Leach, M. Mealling, and R. Salz, "A universally unique identifier (uuid) urn namespace," 2005. 3.1.1

[85] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pp. 75–86, IEEE, 2004. 3.1.3, A, A.1

[86] A. Taylor, "Custom LLVM and clang tools." https://github.com/aktaylor08, 2015. [Online; accessed 10-July-2015]. 3.4.7

[87] LLVM, "The LLVM Gold Plugin." http://llvm.org/docs/GoldPlugin.html, 2015. [Online; accessed 10-July-2015]. 3.4.7, A, A.2

[88] H. Talbot, "wxpython, a gui toolkit," *Linux journal*, vol. 2000, no. 74es, p. 5, 2000. 3.5.2

[89] W. McKinney, "Data structures for statistical computing in python," *Proceedings of the 9th Python in Science Conference*, pp. 51–56, 2010. 3.5.2

[90] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. 3.5.2

[91] A. Technologies, "Ascending Technologies UAVs." http://www.asctec.de/en/, 2015. [Online; accessed 10-July-2015]. 5.1

[92] J. L. Jones, "Robots at the tipping point: the road to irobot roomba," *Robotics & Automation Magazine, IEEE*, vol. 13, no. 1, pp. 76–78, 2006. 6.2.1, 6.4

[93] C. Lattner, "Llvm and clang: Next generation compiler technology," 2008. A, A.1

[94] O. S. Parsers, "JsonCpp." https://github.com/open-source-parsers/jsoncpp, 2015. [Online; accessed 10-July-2015]. A, A.13

[95] B. G. D. T. Troy Straszheim, Morten Kjaergaard, "Catkin Build System." http://wiki.ros.org/catkin, 2015. [Online; accessed 10-July-2015]. A.3

[96] P. S. Foundation, "Python ast Module." https://docs.python.org/2/library/ast.html, 2015. [Online; accessed 10-July-2015]. B

Appendix A

C++ Considerations

The C++ version of the analysis and instrumentation requires a custom built version of LLVM [85] version 3.7.0 and clang [93] version 3.7.0. More specifically it uses commit *52386ce* of LLVM and commit *ff7b692* of the tools github mirror of the tools. It also requires a version of the gold linker that supports link time plugins to be executed [87]. Each portion of the static analysis and instrumentation is broken into a series of LLVM Module level passes. Each pass works to identify LLVM instructions that carry out specific operations and connect the instructions through analysis of the program dependencies. The separation of the processing allows the slicing algorithm to be expanded easily to account for different sources of configuration variables, different exposing statements, and different instrumentation methods. Any of the main features can be changed by simply adding, removing or extending the passes to incorporate the new desired features. The C++ implementation of the analysis consists of 5850 lines of C++ code and 1300 lines of header files. Around 4000 lines of the C++ code and 900 lines of the header code comes from an open source library [94] used to add JSON functionality to report the static information.

A.1 LLVM and Clang

LLVM [85] and clang [93] offer a very nice set of tools to create analysis and modifications tools for programs written in C++. LLVM implements a SSA intermediate language that allows easy implementation of optimizations and analysis on program dependencies. Clang offers a drop in replacement for the gcc compiler and it serves as a front end for the LLVM system.

A.2 Link Time Analysis

The implementation of a ROS node in C++ is often spread across multiple files and our analysis must examine variables that may be shared across these files. For this reason the analysis and instrumentation must be run after the files have been linked and just before the executable is created. LLVM allows link time optimizations using the gold linker from GNU Binutils [87]. The passes for the analysis are added to the Link Time Optimizations that LLVM performs. After building LLVM link time optimizations can than be enabled by passing the -flto flag to both the compiler and linker when building the ROS nodes.

A.3 Integration with the ROS Catkin Build System

ROS uses the catkin [95] build system to build and install ROS nodes and packages. It makes building, linking and using any ROS system straightforward, but complicates matters when trying to pass special flags or change the compiler. To change the compiler we can pass the "-DCMAKE_CXX_COMPILER=" flag on the command line with the path to the customized clang++ compiler. In addition to changing the compiler we must also enable the link time optimizations of our custom compiler. To do this for each executable in the CMakeLists.txt file we add or create properties using the set_target_properties

command. We add to the "COMPILE_FLAGS" the "-g -flto" to enable debugging symbols and link time optimizations. To the "LINK_FLAGS" we add "-flto" to ensure that ld-gold is used and link time optimizations are performed and our custom passes are ran during compilation. This modification of build files can be automated using the modify_cmakelists.py script created for this work. The script automatically reads and modifies build files for the ROS package to be built using the llvm and clang compilers with the passes created for our approach.

A.4 Passes

The analysis and instrumentation of C++ ROS nodes is split into nine separate LLVM Module passes. Each of these passes performs a different part of the analysis and is able to share data with other passes in the analysis. A quick overview of each pass can be found in Table A.1. Additionally, the analysis makes use of two LLVM analysis passes. The LoopInfoWrapperPass allows the analysis to easily find out if an instruction or basic block is inside of one or more loop constructs. The DominatorTreeWrapperPass allows the analysis to determine information on which BasicBlocks in the program being analyzed dominate other basic blocks. Information about each of the passes implemented can be found in the following sections.

A.5 SimpleCallGraph Pass

This pass creates a mapping from every function defined in the module to all instructions in the module where it is called or invoked. This creates a needed portion of data for the program slice and predicate identification. The pass performs two steps to compute the list of calling locations. The pass creates a list of all functions in the module and

Table A.1: Brief Description of each pass implemented in the analysis.

Name	Description	Lines of Code
SimpleCallGraph Pass	Creates a call graph for usage in slice creation and predicate identification.	57
ClassObjectAccess Pass	Finds all element pointers that are used in the module and determines where they are accessed. Finds class variable access for data flow in slice creation and configuration variable analysis.	104
IfStatements Pass	Finds which *if* statements contain each basic block in the module.	156
ExternCallFinder Pass	Finds all exposing statement in the module.	60
ParamCallFinder Pass	Finds all configuration variables in the module.	75
BackwardPropagate Pass	Implements the procedure to find slices based on the criterion of exposing statements and determine which predicates the exposing statement has a data and control dependency on.	347
ParamUsageFinder Pass	Find the intersection of configuration variables found in ParamCallFinder Pass and predicates identified in the BackwardsPropagate Pass.	198
GatherResults Pass	Gather results to determine which predicates are to be instrumented.	64
InstrumentBranches Pass	Instrument marked predicates to report values at runtime.	398

gives them an empty list of calling locations. Next, the pass examines all instructions and if the instruction is a function call to a method within the module, the instruction is added to the list of calling locations. The determination of what function is called is done by creating a *CallSite* object which provides the *getCalledFunction()* method. This

method returns a pointer to the function being called. This is allows the easy lookup of the function in the map created in the first step. The information in this pass is used in the slice and predicate identification procedure.

A.6 ClassObjectAccess Pass

This class helps to determine the location of where class variables are read from and stored to. LLVM implements classes by creating structures in the Intermediate Representation. The class variables within the structures are accessed using the *GetElementPointerInstruction*. This pass iterates through the Module and when it encounters a GetElementPointerInstruction it determines which element the instruction points to. The instruction is then saved for later usage. The variable the access is using is determined matching both the structure and the indices used in the GetElementPointer operation.

A.7 IfStatement Pass

From our understanding, there is no method similar to LLVM's *LoopInfo.getLoopFor*, which provides the loop which contains the passed *BasicBlock*, for *if* statements. This pass compiles this information about *if* statements for use in the slice creation procedure. It builds a map for each of basic block the immediate conditional predicate that the block is contained within. For every function in the module the pass identifies all conditional predicates in the function. Next, for each predicate, the pass iterates through the control flow graph and marks of the number of times each basic block is visited. It also keeps track of the number of branch statements encountered during iteration. After iteration, all basic blocks that are visited less than the number of branch statements are contained within the branch statement being examined. Each instruction within the branch is then

marked as a child of the branch.

A.8 ExternCallFinder Pass

This pass determines the exposing statements in the compilation unit being analyzed. This pass is a pure syntactic search through the intermediate representation. The pass examines every statement in the module and if it is a function determines if the mangled names match that of the specific ROS calls to publish a message or call a service. Publish calls names begin with "_ZNK3ros9Publisher7publishIN" and services calls begin with "_ZN3ros13ServiceClient4callIN." Any matching function calls or invoke instructions are saved for use for slice criterion.

A.9 ParamCallFinder Pass

This pass determines the parameter variables for use in the identification of the threshold predicate comparison portion of the analysis. It will identify which variables have values loaded from the standard C++ ROS API. Any call to the function "_ZNK3ros10NodeHandle5param" is the reading of a configuration variable from the ROS parameter server. The variable that is used to store the value from this call is stored by this pass for use in the later portion of the analysis.

A.10 BackwardPropagate Pass

The BackwardPropagate Pass is responsible for computing the program slices and predicates that an exposing statement have a data and control dependence on. The pass implements the procedure to compute the slice and find predicates as described in

Algorithm 3. The makes use of the SSA of LLVM to determine data dependencies at the function level. Function level control flow dependencies are determined using data from the IfStatementPass and LLVM's LoopInfoWrapperPass. The pass uses the data structures found in previous structures to determine the program dependencies needed at each statement to run the algorithm. The program dependencies that cross functions are determined from the data in the previously run passes.

A.11 ParamUsageFinder Pass

This pass determines which predicates use configuration variables. The pass loops through all functions in the module and determines which *GetElementPointerInstructions* used to load values match parameter storage calls found by the *ParamCallFinder Pass*. Upon a match, the pass traces the data flow from the load to the branch statements. If it reaches the predicate marked during the *BackwardsAnalysis Pass* without being modified by an operation than we have a location to instrument. The branch is marked for instrumentation and information about the predicate is saved.

A.12 GatherResults Pass

This pass simply gathers results from the passes that mark branch statements for instrumentation. This allows for the expansion of methods to mark branch statements for instrumentation without the need to refactor other parts of the analysis. The pass implements three methods that provide necessary information to the instrumentation portion of the analysis. The *get_results()* function provides a list of all predicate statements in the module that are threshold predicate comparisons. *get_setup(Instruction*)* returns information about the setup of the configuration variable. The *get_distance(Instruction*)* re-

turns the distance from the branch to the identified exposing statements as determined in Algorithm 3. The *get_type(Instruction*)* gets the type of the configuration parameter that is being instrumented (e.g. parameter, source constant, etc.). In the current implementation all of the configuration variables are parameters.

A.13 InstrumentBranches Pass

The InstrumentBranches pass handles the instrumentation and output of static information for each threshold predicate comparison. The instrumentation requires one additional source file that contains the function shown in Listing A.1. This function provides the key, time, comparison results, and values involved in a predicate on threshold. The *ROS_INFO* call will send the string with data to the ROS system with the current time and this information can be accessed by the runtime analysis.

For each marked threshold the pass determines which part of the comparison is the configuration variable. After determining the proper values to report, the pass inserts the function call with runtime values. The threshold and the comparator are converted to doubles. The results of the predicate are transformed into 1 byte integers. In addition, if the information on the comparisons flows across *BasicBlocks* a *PhiStatement* is required to to ensure that the instrumented code conforms to LLVM. The pass also creates a unique UUID so the information reported at runtime can be matched with static information. After setting up all data values the correct call is made to report the values to the outside world and inserted directly before the branch statement

The static information is stored and exported to the file system using the JsonCpp library [94]. This open source library easily allows the pass to save information for later use by the runtime analysis.

Listing A.1: Instrumetaton source file with function to publish data.

```
#include "ros/ros.h"


void log_one(char* key, bool res, double c1, double t1, bool r1){
    std::stringstream ss;
    ss << key << "," << res << ",cmp:" << c1 << ",thresh:" << t1 << ",res:"
    ROS_INFO("threshold_information:%s" ,ss.str().c_str());
}
```

Appendix B

Python Considerations

The Python implementation only requires the source code of the robotic control code to run. It uses the standard Python library Abstract Syntax Tree module [96] to parse, examine, and manipulate the abstract syntax tree of a Python file. In total there are around 3000 lines of Python code in the implementation of the Python tool to find and instrument threshold predicate comparisons. The Python portion of the analysis requires Python 2.7 to run correctly.

The analysis requires the control flow graph and reaching definitions for all of the methods within a Python source file. These two analyses are determined using the algorithm found in [82]. The control flow graph and reaching definitions are only needed at the function scope, because the three exceptions for control flow outside of the function level can be handled during the creation of the program slices.

B.1 Exposing Statements

The Python method to determine exposing statements cannot use a simple syntactic search for the names of methods because the python ast does not covert the common

calls to the underlying function which is used by the publish calls like the C++ version of the approach uses. It requires identifying which variables are created using functions which create ROS publishers and service objects using the calls "rospy.publisher" or "rospy.ServiceProxy" constructors. Once these variables are identified, any location where a function call to the "publish()" or "call()" methods are identified as exposing statements. The algorithm also creates one additional source of exposing statements. When an object is created in source code, its methods are searched to determine if any of the methods contain an exposing statement. If they do contain an exposing statement then any call to the method's method is marked as an exposing statement. This was added because many Python objects are used to encapsulate the functionality of ROS in the code we examined.

B.2 Configuration Variables

To identify configuration variables in the Python implementation the analysis finds all locations where a call of "rospy.get_param()" assign a value to a variable. All class, local, and global variables that are set up in this way are saved as configuration variables. The Python analysis is able to mark class, local, and global variables that contain constant values constants, but those methods are not used in our later experiments.

B.3 Slicing and Predicate Identification

The computation of the slices and identifying predicates follows the procedure identified in Algorithm 3. The procedure makes use of the variable information, control flow graph, and reaching definitions computed for our analysis.

B.4 Threshold Predicate Comparison Identification

The Python implementation follows the methods described in Subsection 3.4.5. The analysis examines every predicate and determines if any of the variables are identified as configuration variables or can be traced back to a configuration variable without a statement to modify the values from the configuration variable. If the predicate meets the criteria, it is marked for instrumentation and information about the source of the threshold, the distance to the exposing statement, and other data output for use during the runtime portion of the analysis.

B.5 Instrumentation

The Python implementation replaces each marked predicate with a function call that will return the value of the original predicate. The function call reports the threshold's identifying key, the result, the comparison, and threshold value out to the rest of the ROS system. This function takes as arguments a lambda function that computes the result of the predicate, a dictionary containing the arguments for that function, keys and values to be reported, and other necessary pieces of data to that need to be processed. Lambda functions are used to prevent the calling of functions that may have side effects more than once. The predicate is replaced with a call to the function and the predicate is substituted in the ast to be that of the lambda function. Upon being called the reporting function evaluates any necessary sub expressions, evaluates the lambda function to get the result of the predicate, and than reports the key, result, time, threshold, and comparisons in the predicate out to the rest of ROS. The reporting is done by using a publisher singleton that publishes comma separated string out to the rest of the ROS system on the "threshold_information" topic which accepts string variables. More information about the

format of runtime messages can be found in section 3.5.3

Appendix C

Ros System Information

Table C.2: Information on the repositories analyzed.

Name	C++ Files	C++ Lines of Code	Header Files	Header Lines of Code	Python Files	Python Lines of Code	Source Files	Total Lines of Code
airbotix ros package	0	0	3	132	20	1747	23	1879
app manager	0	0	16	2977	28	6153	44	9130
apriltags tracking	18	1436	25	733	0	0	43	2169
ar tracking	83	13869	63	3408	0	0	146	17277
arm nav	0	0	3	308	1	25	4	333
asctec base	84	7989	25	2131	14	1018	123	11138
asctec mav pacakge	12	1923	53	2729	0	0	65	4652
baxter robot	0	0	0	0	65	5546	65	5546

Continued on next page

Table C.2 – continued from previous page

Name	C++ Files	C++ Lines of Code	Header Files	Header Lines of Code	Python Files	Python Lines of Code	Source Files	Total Lines of Code
bwi from texas	91	13232	99	4493	30	2600	220	20325
calibration	32	2567	13	477	39	4326	84	7370
calvin ros stack	4	495	1	26	1	28	6	549
careobot contro	29	5492	21	1255	3	305	53	7052
careobot evnironment perception	30	6080	33	2420	14	1004	77	9504
careobot manipulation	10	29853	3	342	24	1277	37	31472
careobot navigation perception	3	622	1	48	0	0	4	670
careobot perception	9	2014	8	520	1	103	18	2637
cob command tools	8	1618	5	477	30	4813	43	6908
cob common	0	0	0	0	1	25	1	25
cob driver	62	18019	61	3543	16	853	139	22415
cob external	5	868	125	14681	0	0	130	15549
cob robots	0	0	0	0	3	101	3	101
control toolbox	8	704	8	238	1	20	17	962

Continued on next page

Table C.2 – continued from previous page

Name	C++ Files	C++ Lines of Code	Header Files	Header Lines of Code	Python Files	Python Lines of Code	Source Files	Total Lines of Code
crazyflie ros stack	0	0	0	0	10	613	10	613
NIMBUS crop surveying	58	4462	33	1665	0	0	91	6127
func maninulators	24	42056	2	221	0	0	26	42277
graft	10	1787	7	270	0	0	17	2057
grizzly robot	9	721	6	279	0	0	15	1000
hector arm	3	156	1	39	1	48	5	243
hector diagnostics	14	2783	7	370	0	0	21	3153
hector navigation	14	2783	7	370	0	0	21	3153
hector slam	14	1676	32	2291	0	0	46	3967
hector turtlebot	2	63	0	0	0	0	2	63
icart mini	7	678	2	54	1	36	10	768
jaco robot arm	8	2379	9	1083	0	0	17	3462
jsk control	9	1209	4	676	54	3802	67	5687
jsk smart apps	3	167	17	223277	9	574	29	224018
jsk travis	0	0	0	0	1	231	1	231
kobuki	15	1719	9	1053	37	2268	61	5040
kobuki soft	4	213	2	84	0	0	6	297

Continued on next page

Table C.2 – continued from previous page

Name	C++ Files	C++ Lines of Code	Header Files	Header Lines of Code	Python Files	Python Lines of Code	Source Files	Total Lines of Code
mav ros	40	6953	15	853	9	503	64	8309
maxwell	0	0	0	0	1	55	1	55
motoman	22	3532	25	1704	1	84	48	5320
nao camera	4	437	2	79	3	241	9	757
nao extras	4	854	1	77	0	0	5	931
nao interaction	0	0	0	0	2	407	2	407
nao robot repo	3	614	1	41	24	2163	28	2818
nao ros	2	87	0	0	21	1999	23	2086
nao sensors	5	465	2	79	7	468	14	1012
nao virtual	0	0	0	0	1	21	1	21
nao viz	0	0	0	0	7	253	7	253
naopi bridge	8	978	3	122	33	10387	44	11487
nav2 platform	3	364	1	34	0	0	4	398
navigation stack	96	16289	89	4090	11	295	196	20674
neo robot	14	2493	12	805	0	0	26	3298
next stage	0	0	0	0	19	1284	19	1284
novatel spann	0	0	0	0	11	674	11	674
ocs library	30	3329	21	1299	18	936	69	5564
people tracking ros	20	3284	19	1707	3	148	42	5139

Continued on next page

Table C.2 – continued from previous page

Name	C++ Files	C++ Lines of Code	Header Files	Header Lines of Code	Python Files	Python Lines of Code	Source Files	Total Lines of Code
pepper robot for stuff	0	0	0	0	4	300	4	300
pr2 os robot	11	7462	6	591	0	0	17	8053
pr2 common	1	84	0	0	1	47	2	131
pr2 futre	1	178	1	42	54	4429	56	4649
pr2 colibraiton	18	1469	6	163	18	1958	42	3590
pr2 common actions	11	1042	2	54	8	250	21	1346
pr2 delivery	0	0	0	0	6	309	6	309
pr2 doors	0	0	0	0	3	91	3	91
pr2 kinematics	8	1788	6	330	0	0	14	2118
pr2 navigation	14	1711	11	1399	1	156	26	3266
pr2 pbd	0	0	0	0	17	5856	17	5856
pr2 precise trajectory	0	0	0	0	12	558	12	558
pr2 self test	9	1756	8	320	10	1230	27	3306
pr2 surrogate	7	145	3	58	0	0	10	203
pr2 apps	7	2992	1	157	7	186	15	3335
rail ceiling	4	731	3	121	0	0	7	852
rail pick and place library	30	3265	23	884	0	0	53	4149

Continued on next page

Table C.2 – continued from previous page

Name	C++ Files	C++ Lines of Code	Header Files	Header Lines of Code	Python Files	Python Lines of Code	Source Files	Total Lines of Code
rail segmentation	4	791	2	132	0	0	6	923
realtime tools	2	72	5	350	1	21	8	443
robitician ric	18	5137	580	49202	8	700	606	55039
robot rescue	12	654	0	0	0	0	12	654
ros concert	16	1444	7	258	431	33911	454	35613
ros control	40	5361	50	2698	13	831	103	8890
ros controllers	33	4176	32	2762	5	370	70	7308
ros create driver	0	0	0	0	15	1151	15	1151
ros darwin	0	0	0	0	4	364	4	364
ros descartes	24	3293	18	1254	0	0	42	4547
ros filter library	12	680	8	1263	0	0	20	1943
ros universial robot	4	1058	3	354	10	1719	17	3131
rqt pr2 dashboard	0	0	0	0	6	303	6	303
segbot	17	3953	18	1351	7	258	42	5562
segbot apps	4	806	1	90	2	104	7	1000
shcunk modular	27	17089	60	5380	3	363	90	22832
sr demo	10	1210	11	571	11	1555	32	3336

Continued on next page

Table C.2 – continued from previous page

Name	C++ Files	C++ Lines of Code	Header Files	Header Lines of Code	Python Files	Python Lines of Code	Source Files	Total Lines of Code
sr manipulation	0	0	0	0	28	2896	28	2896
sr utils	2	212	2	77	0	0	4	289
turtlebot	3	142	0	0	2	17	5	159
turtlebot apps	47	6248	25	1890	16	898	88	9036
turtlebot arm	11	6624	2	246	1	219	14	7089
turtlebot create	0	0	0	0	15	1151	15	1151
turtlebot interactions	2	79	0	0	0	0	2	79
uos tools	8	525	3	139	7	465	18	1129
water sampler	4	762	0	0	20	1217	24	1979
mean	13	2894	18	3522	13	1201	44	7616
median	7	791	3	221	3	241	19	2637
std	20	6031	60	22672	44	3675	81	23524
min	0	0	0	0	0	0	1	21
max	96	42056	580	223277	431	33911	606	224018
sum	1354	292251	1792	355696	1321	121317	4467	769264

Table C.1: The statistics gathered during static analysis of ROS packages.

Data	Description	Table Key
Name	Name of the metapackage	Name
Compilation time	Time the metapackage takes to compile with clang++.	Time
Time Factor	Total compilation time with analysis in reference to original compilation. In other words Compilation Time x Time Factor is total time to compile with analysis	Time Factor
Threshold Predicate Comparisons	A count of the number of predicates in the metapackage which both use a parameter threshold in the predicate and have an "exposing statement" which has a data flow or control flow dependency on the predicate.	Threshold Predicate Comparisons
C++ Threshold Predicate Comparisons	Number of Predicates on Thresholds appearing in C++ code.	C++
Python Threshold Predicate Comparisons	Number Predicates on thresholds appearing in Python code.	Python
Unique Threshold Sources	Number of thresholds loaded from uniquely named parameters.	Unique
Files Containing Threshold Predicate Comparisons	Number of files in the metapackage containing Threshold Predicate Comparisons	Files
C++ Files	Number of C++ files in the metapackage.	-
C++ Lines of Code	Lines of C++ code in the metapackage.	-
Headers	.h header files in the metapackage.	-
Headers Lines of Code	Lines of .h code in the metapackage.	-
Python Files	Python files in the metapackage.	-
Python Lines of Code	Lines of Python code in the metapackage.	-
Total Files	Total Python, C++, and header files in the metapackage.	-
Total Lines of Code	Total Lines of code in the metapackage.	-

Appendix D

User Instructions

This appendex contains the instructions each user was given during the trials for each of the three systems.

D.1 Water Sampler

Monitor my Robot

Name: _____

Date: _____

Goal

I am developing tools to enhance the dependability of robots that interact with people. In particular, I am studying how we can help users collaborate with robots to improve their configuration. A part of that work includes employing user input to determine what may

be wrong with a robot's configuration. And that is where you come in.

Robot and Mission

You will be observing a water sampling UAV that is able to autonomously collect water samples and return them to land. When the mission starts the UAV will be on the ground with motors off in the cage. It will then takeoff, fly to altitude, and approach the fish tank. Once over the fish tank it will lower the sampler into the water and pump one vial full of water. After filling the vial the UAV will return to the target altitude, fly to the takeoff location, land, and shutdown.

Tasks

We will first show you the Robot performing the mission correctly three times. This should give you a better idea of what is expected from the Robot. We will also show you the basic equipment you will be using during the mission.

We will then perform three other trials where the Robot `may` be changed in a way that impacts the mission. During these trials your job is to detect two types of error:

- **Type I Error.** Robot performs an action when it should not have.
- **Type II Error.** Robot should perform an action but it does not.

When you detect **Type I** errors mark them by pressing the ___ key on the keyboard, and when you detect **Type II** errors make them by pressing the ___ key on the keyboard. If you have any additional comments or concerns during or after the study please write them at the bottom of this sheet.

This activity should take 20 minutes. Thank you for your time and assistance.

D.2 Navigation

Monitor my Robot

Name: _____

Date: _____

Goal

I am developing tools to enhance the dependability of robots that interact with people. In particular, I am studying how we can help users collaborate with robots to improve their configuration. A part of that work includes employing user input to determine what may be wrong with a robot's configuration. And that is where you come in.

Robot and Mission

You will be observing a ground robot that is able to autonomously navigate in the environment. When the mission starts it will be at one location in the small sample area. The robot will than navigate to the finish location. Th finish location is marked on the ground. To successfully complete the mission the robot must be completely inside the marked area and stationary.

Tasks

We will first show you the Robot performing the mission correctly three times. This should give you a better idea of what is expected from the Robot. We will also show you the basic equipment you will be using during the mission.

We will then perform three other trials where the Robot may be changed in a way that impacts the mission. During these trials your job is to detect two types of error:

- **Type I Error.** Robot performs an action when it should not have.
- **Type II Error.** Robot should perform an action but it does not.

When you detect **Type I** errors mark them by pressing the ___ key on the keyboard, and when you detect **Type II** errors make them by pressing the ___ key on the keyboard. If you have any additional comments or concerns during or after the study please write them at the bottom of this sheet.

This activity should take 20 minutes. Thank you for your time and assistance.

D.3 Image Capture

Monitor my Robot

Name: _____

Date: _____

Goal

I am developing tools to enhance the dependability of robots that interact with people. In particular, I am studying how we can help users collaborate with robots to improve their configuration. A part of that work includes employing user input to determine what may be wrong with a robot's configuration. And that is where you come in.

Robot and Mission

You will be observing a UAV that is able to autonomously find a person, take their photograph, and return to land. When the mission starts the UAV will be on the ground with motors off. The motors start and the UAV takes off and flies to a predetermined point and orientation. The UAV rotates and place and performs image processing on the image stream from the camera located on the UAV. Once the UAV identifies a face it remains at the same orientation attempts to center the face in the image frame. Once the UAV has the person centered in the image it snaps a photo and lands at a predetermined location. If at any point in time it looses track of the person, the UAV returns to circling in place.

Tasks

We will first show you the Robot performing the mission correctly three times. This should give you a better idea of what is expected from the Robot. We will also show you the basic equipment you will be using during the mission.

We will then perform three other trials where the Robot `may` be changed in a way that impacts the mission. During these trials your job is to detect two types of error:

- **Type I Error.** Robot performs an action when it should not have.
- **Type II Error.** Robot should perform an action but it does not.

When you detect **Type I** errors mark them by pressing the ___ key on the keyboard, and when you detect **Type II** errors make them by pressing the ___ key on the keyboard. If you have any additional comments or concerns during or after the study please write them at the bottom of this sheet.

This activity should take 20 minutes. Thank you for your time and assistance.