Co-Diagnosing Configuration Failures in Co-Robotic Systems

Adam Taylor, Sebastian Elbaum, and Carrick Detweiler

Abstract— **Robotic systems often have complex configuration spaces that, when poorly set, can cause failures. In this work we take advantage of the close synergy between user and robot in co-robotic systems to better diagnose and overcome configuration failures. We leverage users' understanding of the system to mark failures they observe while the system is in operation. A marked failure indicates that the robot either "did not do something when it should have" or "did something when it should not have". The failure marking is coupled with an automated analysis approach that identifies code predicates involving configuration parameters that may be relevant to each failure type, ranks the parameters according to their potential to be associated with the failure, and suggests adjustments based on the run-time outcome of those predicates. We present the approach, its implementation, and a preliminary study on a configurable unmanned air system. The results show how the approach can successfully help diagnose and adjust faulty configuration parameters in co-robotic systems.**

I. Introduction

The software engineering community has recognized the difficulties for developers to validate large configuration spaces and for users to select the proper configurations for their systems to operate as expected. Wrong configurations options have been identified to be one of the main causes of failures in a variety of systems, and a large number of testing, analysis and diagnosis approaches have been developed to solve the issues they cause (Xu et al. provide a comprehensive survey on the area [1]).

Popular robotic systems are not exempt from these configuration challenges and are amenable to some of the proposed solutions. Their configuration spaces can be large and complex, in part, to enable users to tweak the systems to fit many potential usage scenarios. Consider for example Baxter, a humanoid robot that collaborates with workers and has 236 code locations reading in configuration parameters [2]. Or the Arducopter Drone with 622 configuration parameters each containing multiple valid selections or a large range of selectable values [3]. Or the Robot Operating System (ROS) [4] whose's navigation framework [5] includes over 220 listed parameters that can be used to configure the framework for a specific robot. These three examples illustrate the widespread use of large configuration spaces in robotic systems. For some of these systems, the solution space is similar to that already explored by the software engineering community. For example, practices such as the reduction of the configuration space or the elimination of hard to set parameters could help [6]. Similarly, automatic techniques could assist in exposing the configurations available [7], in providing scalable testing coverage of the configurations [8], [9], or in performing root-cause analysis through dynamic profiling [10], [11].

Our work, however, focuses on co-robotic systems where robots closely cooperate with people to complete mundane, dangerous, precise or expensive tasks. These kinds of robots are said to have the greatest potential to impact society [12], [13], integrating the operator's sensing, actuation, and domain expertise with the robot's own. In this work we explore how this symbiosis between humans and robots can be leveraged to diagnose improperly set configurations and to collaborate to fix them and optimize them.

We enable users to mark a co-robotic system failures on the fly as either *Type I*: the system did not do what it was supposed to, or as *Type II*: the system did something when it should not have. Then, we apply an approach rooted on program analysis techniques that identifies code predicates involving configuration parameters that might be relevant to each failure type. Last, the approach can help users adjust configuration parameters based on how the code predicate evaluation changed, and the type of failure observed.

Our work offers the following contributions:

- The insight that co-robotic systems introduce new failure diagnosing opportunities to be carried by users in addition to developers
- A novel approach that integrates user feedback and automated code analysis techniques to diagnose potentially problematic configuration parameters and to recommend how to adjust them
- A preliminary study illustrating the potential of the approach; it helped users co-diagnose the problematic configuration parameter in 7 of 8 trials. The study also identifies the need for more sophisticated failures types and discriminating analyses.

II. Motivation

Consider the co-robotic aerial water sampler in Figure 1 [14]. The UAV aims to enable water scientists to more quickly collect routine samples from a body of water without deploying a boat or a larger, slower robotic system. This system can autonomously fly to a selected location, descend to insert the sampling tube into the

University of Nebraska, Lincoln, NE, USA {ataylor, elbaum, carrick}@cse.unl.edu. This work was supported in part by NSF-1218265 and NRI-USDA-2013-67021-20947. We thank John-Paul Ore for his feedback on the manuscript.

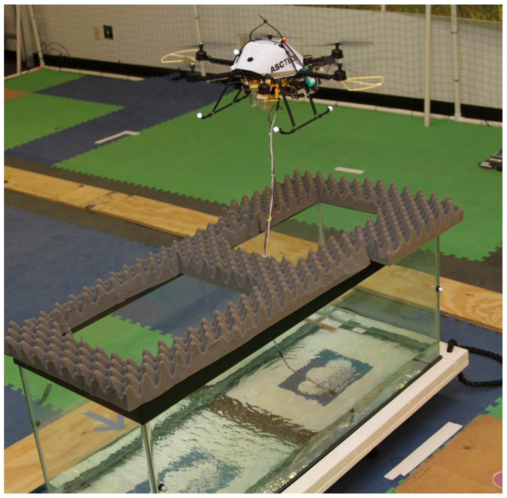


Fig. 1: UAV water sampler operating in the lab, obtaining a water sample from a fish tank.

water, collect water samples, and return the samples back to the launch location. The system relies on ROS, a popular middleware for robotic systems, and includes around 11,000 lines of C++ and Python code.

This system contains a configuration space with 286 parameters that must be set up correctly to run properly. For example, the system has a parameter that determines the allowable distance error when flying to a target waypoint. If the allowable error is too small and there is an external force, such as wind, the UAV will not reach the desired location and might appear to be stuck to a human observer (*Type I* failure). If the allowable error is instead too large, the UAV may prematurely consider that it has reached a waypoint and obtain a sample at the wrong location (*Type II* failure). Another example is the safety parameter to set the minimum altitude at which the UAV can fly before aborting the mission. If the value is too small the UAV may crash into the water. If the value is too large, the UAV may abort a mission prematurely as it wrongly deemed to be operating dangerously close to the water, even when it could continue to operate safely.

When a system presents such failures in the field, it is extremely difficult for even an expert to diagnose and adjust such space of parameters to overcome the problem. As we shall see, the approach we propose aims to assist in handling such failures, which in these scenarios would be attributable to the allowable waypoint error (needs to be higher) and to the minimum safe altitude over water (needs to be lower), respectively.

Our approach is based on three key assumptions illustrated through this example. First, we assume that configuration spaces for co-robots are complex and users often set them incorrectly. We illustrated some of these challenges in the configuration of the aerial water sampler but, as stated in the related work, the problems are more general and manifest extensively in practice.

Second, humans in co-robotic systems have an intrinsic understanding of what to expect from their robots and can effectively recognize instances of some failures. For the aerial water sampler, the operator tends to perform the same flying operations over water repeatedly and is able to diagnose behaviors that differ from those typically observed. We argue this is part of the essence of the synergy in co-robotic systems.

Third, many failures in co-robotic systems are associated with uncommon but observable changes in the system execution that are driven by the choices made by the human when configuring the robot. Again, in the case of aerial water sampler, the configuration space is large enough and subtle enough that can lead to erroneous behavior, and that behavior can be recognized by the user as a rare occurrence.

Although simplified, this section illustrates some of the challenges in diagnosing configuration failures in co-robotic systems. Some of those failures could be overcome through alternative designs, methodologies, and toolsets, but there seem to be an opportunity for an orthogonal approach that can leverage the users' abilities to recognize anomalies and adjust the configuration parameters. We explore such approach next.

III. Approach

We now describe the approach and some key implementation details.

A. Overview

Figure 2 provides an overview of the approach which contains a static analysis phase (that focuses on the system source code and deployment files structure) and a runtime analysis phase (that focuses on the system as it executes). As shown in the figure, the first phase is responsible for identifying threshold predicates, that is, predicates in the source code of the target co-robotic system, S, that compare an expression against a configured parameter that acts as a threshold.

Consider the code snippet:

```
...
minAlt = get_param(...) //configuration call
...
if (UAVcurrAlt > minAlt) then //threshold predicate
    ...
    reachSafeAlt.publish(...) //action exposing call
...
```

Using predefined code patterns and program analysis techniques, the approach identifies the configuration parameter $minAlt$ set through the function call $get\_param(...)$, the exposing action $reachSafeAlt()$ that produces an event that is observable by the user, and the predicate $if(UAVcurrAlt > minAlt)$ that evaluates the configuration parameter $minAlt$ to determine whether to take the action $reachSafeAlt()$.

After identification, the approach instruments the target predicates by inserting additional snippets of code that expose the values of the predicate expressions during the execution of the system. So, for the previous code snippet, every time the predicate $if(UAVcurrAlt >$

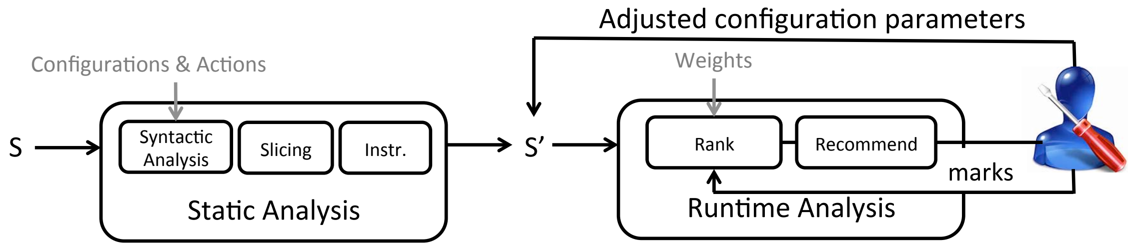


Fig. 2: Overview of Approach

minAlt) is executed, the added instrumentation would log the values for $UAVcurrAlt$ and the predicate outcome (*True* or *False*). This instrumented version of the system is S'.

At runtime, a user observing the system can indicate when it is behaving unexpectedly as per *Type I* or *Type II*. The runtime analysis then calculates a score for each threshold based on factors such as the predicate temporal proximity to the marking, whether their evaluation changed, and the frequency of those changes. The thresholds are then ranked by their scores, which constitute proxies for the likelihood of a threshold to contribute to the failure. So, let's suppose a water scientist using the aerial water sampler notices that the vehicle "stopped the water sampling process prematurely" and marks it as a *Type II* failure. There are more than 15 configuration parameters directly associated with the sampling stage that could be adjusted in the configuration files. Our approach will first identify the configuration parameters that were part of an executed instrumented predicate (the ones that were not executed could not have led to the failure). Then, for a *Type II* failure, the approach will rank $minAlt$ as the top culprit, as it was the parameter involved in a predicate evaluation change (predicate $UAVcurrAlt > minAlt$ flipped from *True* to *False*) just before the user marking, and because such switch is a rare one (most of the times $UAVcurrAlt$ was greater than $minAlt$).

If possible, the approach also makes recommendations for parameters adjustments. For our sample scenario, given the failure type, the approach would recommend decreasing the value of the configuration parameter $minAlt$ to stop the predicate from flipping (ensuring the call indicating that a safe altitude was reached is made) since that led to the inaction deemed as a failure. The expectation is then for the human operator to incrementally change the value of $minAlt$ in S', working with the system to explore and test the effect of the changes.

B. Static Analysis for Identifying Threshold Predicates

This initial phase analyzes system code and produces an instrumented version of the system that can collect information on the threshold predicates (those that operate on configuration parameters).[1] It relies on three basic analysis techniques.

First, a syntactic search identifies two types of function calls: *configuration* and *action exposing*. Configuration calls are the ones that retrieve configuration parameters. These calls are often standardized in robotic systems as they employ common middleware to retrieve configuration parameters. In the case of ROS, for example, these calls take the form of $[RosCoreType].get\_param(\dots)$. Similarly, exposing calls are those that externalize the behavior of a component. In ROS (python), a common form of externalization occurs through message publishing or service calls (e.g., $[PublisherType].publish(\dots)$). This phase of the approach can be extended to include call patterns that encode other configurations and actions (such of those used by other robotic middlewares and libraries).

Second, an interprocedural backward code slice is computed from every exposing action call to every predicate reachable in the compilation unit. The code slice then contains all statements, potentially spanning across multiple functions, that could affect each exposing action. The code predicates included in each slice and the slice size (distance to the exposing action statement) are saved for later usage.

Third, every code predicate in each slice that has a data dependency to a configuration variable (variables assigned a value by a configuration call) is instrumented. The instrumentation allows the runtime capture of several pieces of information such as the values of the variables in a predicate and its outcome, which are consumed by the runtime analysis.

C. Runtime Analysis to Rank Culprits and Recommend Fixes

The runtime analysis portion of the approach consumes the threshold predicate data produced by the instrumented system (generated by the previous phase) and, when the user marks a failure, it triggers the computation of scores on each threshold predicate, their prioritization, and a recommendation.[2]

When the user perceives that the system is misbehaving, she marks a failure by clicking on one of two interface *Action* buttons labeled as *Type I* and *Type II*. The marking type and time is recorded, and it triggers the scoring process which assigns blame to each predicate threshold based on the collected

[1]An implementation of this analysis for ROS is available online at `https://github.com/aktaylor08/thresholdanalysis`.

[2]The runtime implementation in Python is also available at `https://github.com/aktaylor08/thresholdanalysis`.

evidence. The evidence includes whether the predicate has flipped recently (its current evaluation changed from its previous evaluation), its flipping rate (how often does it change), and its temporal or spatial distance [3] to the exposing action statement (smaller distance is likely to have a greater influence on the predicate). As defined, lower scores mean higher likelihood of being associated with a marked failure, but their computation varies slightly between failure types.

Type I failures are caused when a robot fails to make progress, and can often be attributed to a configuration parameter that acts as a threshold being set too high or low. For scoring, we favor the predicates that are the closest to flipping but have not yet done so. We are interested in the *gap* from the current values to the required values to flip a predicate. We normalize the flipping gaps to values between 0 and 1 (dividing by their min/max values found at runtime) to make them comparable across predicates that operate at different scales. Then we compute the $score = gap^{\alpha} * n^{\beta} * d^{\gamma}$, where α, β, and γ are weights that can be set externally to tune the scoring based on the importance of the *gap* size, the number n of previous flips, and the importance of the distance d to the exposing statement. We use values of $\alpha = 1.0$, $\beta = 1.0$, and $\gamma = 0.0$. These values were chosen through trial and error, and produced good results on the experimental data. We chose to exclude the distance ($\gamma = 0$) because the system includes C++ and Python code where distance in the code means different things even when dealing with similar predicate evaluations.

For *Type II* failures in which the system did something when it should not, we favor blaming threshold predicates that flipped recently, that is, predicates whose outcome changed leading to the execution of an exposing action statement. More specifically we use $score = t^{\alpha} * n^{\beta} * d^{\gamma}$ where t is the time since the flipping occurred. In the studies we conducted we use values of $\alpha = 1.0, \beta = 1.0$, and $\gamma = 0.0$ for these parameters.

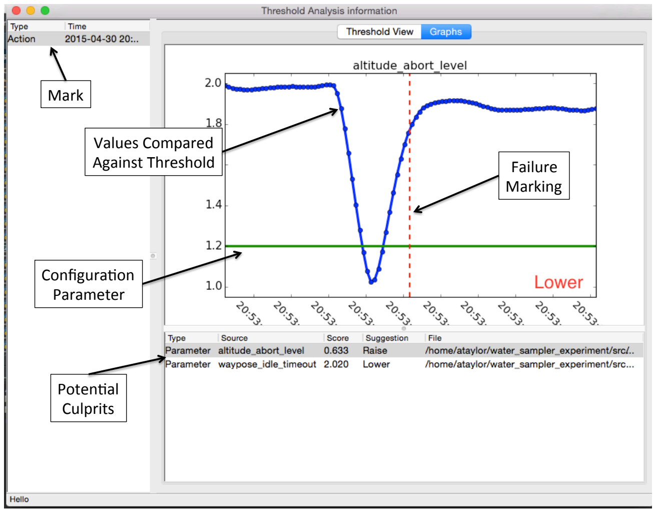
The approach also generates recommendations for predicates performing inequality comparisons on order-sets (int and float types), according to the type of failure as shown in Algorithm 1. For *Type I* failures, if flipping has not occurred, that is, the predicate outcome has not changed, and the value compared against the threshold was higher, then the recommendation is to "raise" the configuration parameter to facilitate the flipping in the future. If the threshold was higher than the value, then the recommendation is to "lower" the threshold to facilitate the flipping. Similarly, for *Type II* failure marking, if the outcome of the predicate was flipped, then we want to "raise" the threshold to make the flipping harder in the future and maybe lead to a restoration of the system state before the flip occurred.

We prototyped an interface with two views to convey

[3]Measured by the number of edges in the dataflow graph.

Algorithm 1 Algorithm to assign blame and make recommendation on a threshold predicate

```
1: procedure BLAMERECOMMEND(type, flip, gap, t, n, d, cmpVal, threshold)
2:     if type == I then                          ▷ Type I: Failed to perform an action
3:         $score = gap^{\alpha} * n^{\beta} * d^{\gamma}$
4:         if ! flip then
5:             adjust($threshold, cmpVal$)
6:     else                          ▷ Type II: Performed action when it should not
7:         $score = t^{\alpha} * n^{\beta} * d^{\gamma}$
8:         if flip then
9:             adjust($threshold, cmpVal$)
10: procedure ADJUST($threshold, cmpVal$)
11:     if $cmpVal > threshold$ then
12:         raise($threshold$)
13:     else
14:         lower($threshold$)
```



Type	Time
Action	2015-04-30 20:..

Type	Source	Score	Suggestion	File
Parameter	altitude_abort_level	0.633	Raise	/home/ataylor/water_sampler_experiment/src/..
Parameter	waypose_idle_timeout	2.020	Lower	/home/ataylor/water_sampler_experiment/src...

Fig. 3: User interface. View after failure marking.

the collected data. The first view contains current information on all thresholds in the system and is meant to be consulted under normal conditions. The second view, shown in Figure 3, is triggered when a marking occurs. In the lower right area it contains the potential culprit parameters. When one is clicked, a graph showing the configuration parameter value (horizontal line), the value against which it is being compared (the curved blue line), and the time of the failure (dashed vertical line) marking is shown in the upper-right area, together a recommendation (in this case to "Lower" the parameter value).

IV. Preliminary Assessment of Approach

In this section we explore the performance of the proposed approach to diagnose and overcome failures that arise from misconfigured parameters.

A. Setup

We studied the autonomous water sampling UAV shown earlier in Figure 1. While in operation, the system deploys 31 execution processes, that communicate through 37 ROS topics and 18 ROS services. The nodes control the position of the UAV, track the UAV, run the water sampling sub components, manage the mission, and ensure safety. We designed a study with a single mission, two treatments (each containing a configuration fault), and five participants that could mark *Type I* or *Type II* failures. The mission took about 45 seconds, began on the ground with motors off, then the motors turn on and the UAV begins flight, ascending to an altitude of 2 m. Next, the UAV flies over a fish tank at an altitude of 2.0 m. Then it descends to 1.2 m to insert the sampling mechanism into the water, turns on the pump, and fills a vial with water. Last, the UAV ascends to 2.0 m and returns to the takeoff location.

The study participants belong to the lab where the system was developed. They had seen the system in operation, but they had not operated it nor were they familiar with its implementation. Each participant was first given a high-level written description of the system and the planned mission, and shown the mission three times so get them further familiarized with the system and provide a common baseline for the study. They were instructed to pay special attention to the sequence and timing of events, and the failure types and marking procedures were explained.

Treatment 1. Lower *error_xy* which is the configurable margin of error to determine if the UAV has reached a target waypoint. This parameter is configured in the system's main launch file. As a result of this change the UAV will not continue on to the sampling portion of the mission because the target waypoint is not reached, causing a *Type I* failure. *Treatment 2.* Raise the configuration parameter *altitude_abort_level* which sets the minimal UAV altitude threshold before the mission is aborted. When the UAV aborts, it stops the current operation and returns to a specified altitude and hovers until receiving further commands. As a result of this fault, the UAV will be deemed "too low" during normal operation and abort the sampling before filling the vial raising to a new altitude, leading to a *Type II* failure.

We conducted 16 trials across the treatments, with participants taking each treatment once or twice. During each trial we recorded: 1) a system execution trace of all ROS messages, 2) values and result of each threshold predicate, 3) time and type failure marked by the user, and 4) a video of the robot system synchronized with the other three data sources.

B. Results

The first phase of our analysis identified 58 threshold predicates in the system, spread across 15 files, involving 41 unique configuration parameters (libraries were not included). This static analysis required slightly more than twice the time to compile and link the system.

The instrumented code logged over 400K predicate executions[4]. Out of the 58 instrumented predicates, only 24 (41%) were executed, accounting for 17 out of the 41 (42%) configuration parameters. Flips were

[4]Although the impact of the instrumentation on the system's performance was not obvious to the subjects, the additional code generated over 13% of additional monitoring messages.

TABLE I: Average Rank of the Faulty Configuration Parameter After the Marked Failures.

Treatment (Failure Type)	Trial	Time from Failure to 1st Mark (secs.)	Rank
1	1	20.6	**0.91**
1	2	19.5	**0.95**
1	3	11.6	**0.95**
1	4	8.7	**0.96**
2	1	12.4	**1.00**
2	2	11.0	**1.00**
2	3	2.3	0.47
2	4	1.6	**1.00**

also rare, present in only 0.041% of the threshold predicate executions. Furthermore, only 6 (35%) of the 17 configuration parameters were used in predicates that have both true and false results. This shows that few predicates change their outcome, so any predicate that flips contains information, and if the outcome of the predicate does change, one outcome is produced at a much higher rate. Hence, **there is a strong signal in the execution of the instrumented predicates on thresholds, and in those flipping their outcome.**

Table I displays the ranking of the faulty configuration parameters across all trials and treatments. These rankings are averaged across the execution of the trial after each failure is marked by the user, and can range between 0 and 1. **Our proposed approach identified the problematic configuration threshold as the top culprit on average in 7 out of *8* trials**.

Figure 4 shows what such ranks mean in the context of a Treatment 1 trial. User marks are indicated using vertical lines with 'X' marks on them, where lighter green lines represent marks of *Type I* failures and darker blue marks represent marks of *Type II* failures. Figure 4-bottom shows the aerial water sampler reaching the fish tank area (at -2.1m) x-position after 80 seconds. Within 10 seconds the operator starts repeatedly marking a *Type I* failure because the vehicle keeps hovering, failing to start the sampling process. The top graph shows the ranked configuration parameters, one per colored line, with the red 'x' line representing the threshold that was modified as part of the trial treatment. The approach was able to confine the problem to just seven configurable parameters, and **the faulty configuration parameter, after being ranked at the bottom until the failure is about to occur, ranks among the top three culprits in the proximity of the markings, and it is identified as a main culprit as per its average ranking after the markings started.**

Figure 5 illustrates another trial but for Treatment 2, with the vehicle reaching the fish tank area at a 2.0m altitude after approximately 35 seconds. At 50 seconds it descends to 1.2m to start the water sampling process but it dips below the erroneously configured threshold, canceling the sampling process, going up to 2m, hovering a bit over the fish tank. The users marked two *Type II* failures about 10 seconds after the sampling process was canceled. The graph on the top

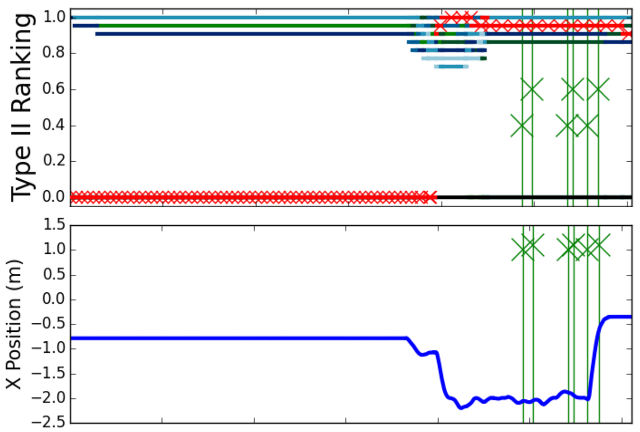


Fig. 4: Ranking of configured threshold (red x line) across Trial 1 under Treatment 1 (Type I Failure)

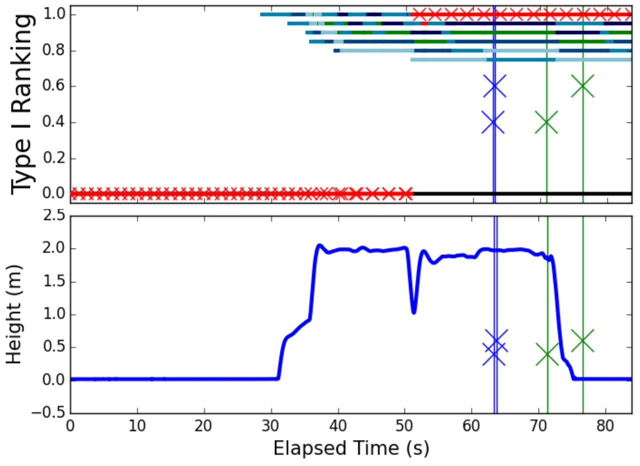


Fig. 5: Ranking of configured threshold (red x line) across Trial 1 under Treatment 2 (Type II Failure)

of Figure 5 shows that **the approach identified the problematic threshold soon after the sampling was canceled and remained as the top culprit until the end of the mission.**

Marking *Type I* failures took more time than *Type II* failures, a median of 20s and 12s respectively, as users had to wait before confirming that the system was not making any progress for the *Type I* failures. The only misdiagnosed case was under Treatment 2 Trial 3, in part because the user also marked failures of *Type I*. As we observed in Figure 5, there are instances of users marking *Type I* failures (instead of just *Type II*) when an unexpected action is followed by a pause as the system gets ready for the next action. Interestingly, we did not find instances where *Type I* failures were marked as *Type II*. Further studies with more users and systems would be needed to understand how to curb this source of noise, whether the failure types need to be revised, and how user experience may reduce affect the markings quality and timeliness.

V. Conclusions

Robotic systems can have complex configuration spaces that, when poorly set, can cause systems to fail. Although significant effort has been dedicated to supporting component developers of such systems, our work is orthogonal and novel in that it takes advantage of the synergy between users and robots in co-robots to better diagnose and solve such configuration problem. We enable users to timely mark system failures. Coupled with an automated analysis to identify how the configuration parameters influenced the underlying code execution flow at the time of the failure, our approach is able to identify culprits, the problematic parameters, and suggest adjustments.

This paper illustrates the potential of the approach, but the study is still preliminary and there is much to explore. We plan to perform more sophisticated and robust failures marking schemes, automatically identify observable external actions, analyze confounding configuration parameters and fault types, and assess our approach more extensively.

References

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

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

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

[4] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote *et al.*, "Ros: an open-source robot operating system," in *Workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.

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

[6] 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 *Foundations of Software Engineering*. ACM, 2015, pp. 307–319.

[7] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *International Conference on Software Engineering*. IEEE, 2011, pp. 131–140.

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

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

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

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

[12] 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.

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

[14] J.-P. Ore, S. Elbaum, A. Burgin, and C. Detweiler, "Autonomous aerial water sampling," *Journal of Field Robotics*, 2015.