Automatic Test Generation for Physical Systems

Carl Hildebrandt[1] and Melony Bennis[1]

***Abstract*— Autonomous robotic systems are beginning to appear everywhere and must be able to execute safety-critical functions that mitigate risks to human life correctly. It is essential to properly simulate their environments, and reproduce as many system faults as possible. However, the isolation of faults can be difficult as they may arise from a variety of sources rooted in either hardware, software or both. We intend to explore hardware-generated changes which exploit faults in software. We plan to devise a set of criteria that distinguish the differences spawned within a system as a result of such changes in hardware. These criteria include the imposition of constraints on obstacle size, distance, and placement. These constraints will be used to generate coordinates to be fed into a virtual autonomous robot simulator that allows for the observations of any variations in behavior. These behavioural differences we hope will lead to finding software bugs. These simulations will serve in visualizing when hardware-induced changes can cause collisions for the autonomous robot, and mirror situations where failures could be produced in the real world. We use the Z3 Solver to check the satisfiability of our constraints while simulating the robot in ROS.**

I. Introduction

Autonomous vehicles are rapidly becoming more popular. Companies such as Tesla and Google have made groundbreaking progress in the field; however, even their models have succumb to fatal consequences: Google's self-driving car colliding with a bus in February 2016, and the Tesla Model S crashing while in "Autopilot" mode shortly after in May 2016. This begs the question: why do these collisions occur, and more importantly: how can they be prevented?

It is evident that autonomous technologies must still undergo a great deal of vetting before they can be safely integrated into society. These robotic systems are complex – a sophisticated combination of both hardware and software components – and will be tasked with performing safety-critical functions. Often, testing via simulation or through real-time deployment is considered satisfactory, but with the constant introduction of new hardware and software during system maintenance, the likelihood of incurring system faults increases which highlights a need for a stronger, more thorough form of verification.

This compounded with the amount of presently-deployed robotic systems in safety-critical environments exacerbates the need for the proper identification and isolation of system faults caused by hardware and software changes. Because this course entails understanding and constructing formal specifications for cyber-physical systems, we aim to give an overview of this challenge focusing on the smaller subset of hardware-induced changes, and the behavioral responses caused by them. Additionally, in this paper we hope to address the following questions:

- Is it possible to create test cases that identify differences in robot behavior brought about by hardware changes?
- How do system constraints change when new hardware is introduced to the system?
- Will the identification of these hardware-induced changes lead to a more thorough specification and verification of robotic systems?

Fig. 1. The Clearpath Husky robot which is used as a testbed for our automatic test generation tool.

II. Problem

Robotic applications continuously undergo changes and improvements throughout development, deployment, and maintenance. These changes can and often incur a variety of system faults that not only arise from software but from incompatibilities between hardware and software. Isolating and identifying the source of these faults, and the different behaviors spurred by the inclusion of new hardware within a system can prove to be incredibly difficult. Thus, the focus of this project is to focus on the problem subset: identifying and observing the differences in robotic behaviors with the inclusion of new hardware.

III. Related Work

There are several related work associated with the specification of formal constraints for autonomous robots. Kim et al. proposed a framework which automatically generates constraints based on 3D coordinate interpolation schemes [3] and focuses on the generation of constraints for the creation of a road network based on formalized coverage criteria. Additionally, Kim et al. hopes to minimize the manual effort involved road environment construction by vetting of the overall software system in question. Similarly, our approach intends to define a set of constraints for a given autonomous robot. However, with identification of these constraints our

This work is done in requirement of our Formal Methods class

[1]Computer Science Department at the University of Virginia, USA. ch6wd@virginia.edu,mmb4vu@virginia.edu

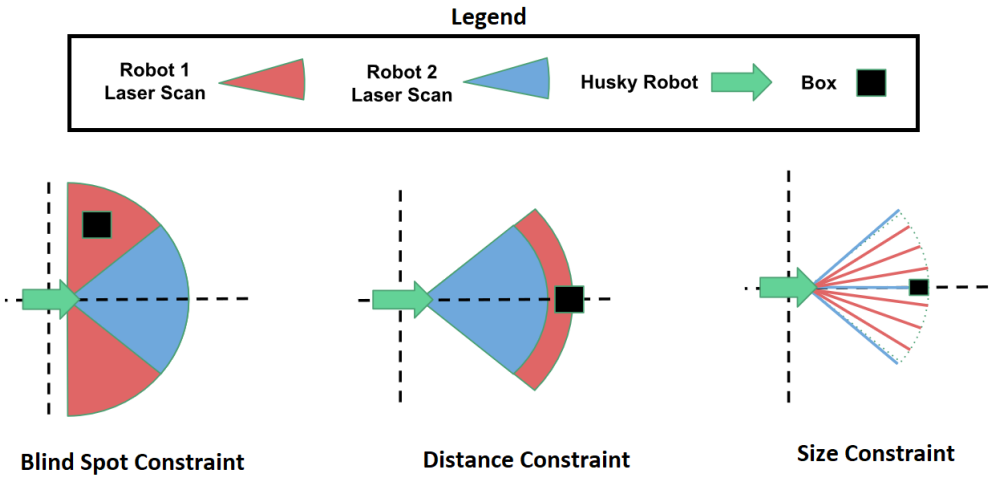

Fig. 2. The three different laser configurations and an example of a block placement which would satisfy the constraints we provided.

aim is to isolate changes sustained from changes in hardware. This means that while their approach intends to test the integrity of a software system via the navigation of an automatically generated road, our work aims to generate constraints which evaluate and observe the hardware changes navigating on predefined path.

The work by Finucane et al. seeks to implement and and test robot controllers at a high level, and use Linear Temporal Logic to write high-level reactive task specifications, which are then be used to drive a simulated or a real robot [2]. Finucane et al. explains that "the more diverse the array of available sensors, actuators, and motion controllers, the more interesting the specifications that can be written" [include citation here], which in part describes the experiment we aim to perform, and the component of their future work seek to explore. The inclusion of different hardware yields a performance change and ultimately different behaviors and possible errors in the robotic system. Although, Finucane et al. describes an approach that is predominantly geared toward the design of an expressive logical specification, they acknowledge that it is important that all components be modular, in order to perform more thorough research on any single component in isolation. Using this methodology lead to the uncovering of underlying system faults, an idea we seek to explore.

Our work was motivated by improving the overall testing domain and testing process, while proving the correctness of a program via the uncovering of bugs and other system faults. Kim et al. gauges the performance of their constraints through the ability or failure of their model to navigate on the generated road, whereas we gauge the performance of our system through the ability or failure of our system to detect obstacles when equipped with new hardware.

IV. APPROACH

A. Overview

The goal of this project was to have a robot perform different behaviors in the same environment given two different laser scanners. To do this the laser scanner was first modeled using a set of attributes listed in Section . Constraints were generated which placed the block in positions that maximized the possibility of different behaviors occurring. We maximized this possibility by positioning the blocks such that they would be visible to one laser scanner but not the other.

Before we began we selected a differential robot which met our needs as described in Section IV-B. Even though our tests were only performed on one robot, our technique is general enough to be applied to any robot that has sensors which can be modeled as a set of attributes. The environment consisted of a single block whose position and size was variable. Section IV-C describes how the block position was selected. This includes how we maximized the possibility of different robotic behaviors occurring. Section IV-D describes how we simulate both robots in the environment.

B. Robot Selection

We decided to use a differential drive vehicle as opposed to an ariel vehicle. The differential drive vehicle was to allow simplification of the simulation and constraints from a three-dimensional problem into a two-dimensional problem. We decided to use the Robotic Operating System (ROS) as it is a very popular middleware in the robotic community. ROS is a collection of software frameworks for robot software development[7]. Three popular differential drive robots are using ROS are the Turtlebot3[10], the iRobot Roomba[9], and a Clearpath Husky robot[1]. Each of these robots has open source simulations available. We selected the Clearpath Husky robot due to it being aimed at industrial use and for its ability to easily switch and support different hardware configurations.

The Clearpath Husky robot had multiple sensors. We selected the laser scanner as a proof of concept. Laser scanners are used to determine if any obstacles are in front of the robot. They do this by rotating a laser beam at set intervals in front of the robot and use this to measure the

distance between the robot and the obstacle. We can model the laser scanner as a set of attributes:

- Sample Size
- Update Rate
- Minimum Angle
- Maximum Angle
- Maximum Range

These attributes were chosen as they are what is used in the Gazebo simulations laser scanner model. These attributes are commonly found on the datasheet of many laser scanners. That meant when physical laser scanner were changed on the robot the new attributes would be easy to find and implement in the simulation.

We decided that we would model three different laser scanners. For each of the laser scanners, we would change a single attribute of the laser scanner. This was done so that we could easily isolate the cause of behavioral difference noticed in the robot during a simulation. We generated three sets of constraints which could be used to for each of the isolated changes to the laser scanner. The constraints would solve either the position or size of the block such that the block would only be seen by one of the laser scanners. The details of the different configurations are represented in Table IV-B

Test Name	**Blind Spot Constraint**		**Distance Constraint**		**Size Constraint**	
Robot	**1**	**2**	**1**	**2**	**1**	**2**
Sample Size	720	720	720	720	16	32
Update Rate	50	50	50	50	50	50
Minimum Angle	-1	-1.57	-1.57	-1.57	-1.57	-1.57
Maximum Angle	1	-1.57	1.57	1.57	1.57	1.57
Maximum Range	10	10	5	10	10	10

C. Constraints Generation

The constraints were implemented in Z3. Z3 is a high-performance theorem prover developed by Microsoft Research[5]. We only implemented 3 constraints to maximize the likely hood of different behaviors between robot 1 and 2 for the same environment. There is however no reason multiple constraints could not be applied to the same test, and that more constraints could not be generated for more advanced tests. Each of the constraints are described in the subsequent sections.

1) Blind Spot Constraint: In the blind spot constraint, we assumed that the laser scanners maximum and minimum range was changed. This meant that the field of view of robot 1 and 2 changed. This can be seen graphically by the leftmost figure in Figure 2. In the figure, we noticed that the blue region is a subset of the red region. This represents a robot having a smaller field of view. We wanted to place blocks between the maximum angle of the red area and the maximum angle of the blue area. This is represented by Equations 1.

$$\begin{aligned} \text{Block Angle} &> \text{Max\_Angle}_{\text{robot1}} \\ \text{Block Angle} &< \text{Max\_Angle}_{\text{robot2}} \end{aligned} \tag{1}$$

Next, we want the blocks to be placed such that it is inside the range of sensors beam length. This is represented by Equation 2.

$$0 \leq x \leq \text{Beam Length} \tag{2}$$

Once these constraints were solved by Z3, we could simply use the Block Angle and x value to calculate the position of the y value using the equations of a straight line as shown in Equation 3. Note that Equation 3 assumes that the robot is placed at the origin.

$$y = tan(\text{Block Angle}) \times x \tag{3}$$

2) Distance Constraint: The distance constraint is used to generate tests when the beam length of a robot has changed. This is graphically represented in the center figure of Figure 2. In this figure, the red area has a larger beam length and thus can detect obstacles further away from it. By placing a block in this region the robot with the larger beam length would possibly start to avoid the block before the robot with the shorter beam length. We did this by first generating a distance for the block using the constraints in Equation 4.

$$\begin{aligned} x &< \text{Beam Length}_{\text{robot1}} \\ x &> \text{Beam Length}_{\text{robot2}} \end{aligned} \tag{4}$$

Finally, we want the block to be placed inside the field of view for both robots. This is represented using Equation 5 below.

$$\begin{aligned} \text{Block Angle} &> \text{Min\_Angle}_{\text{robot1}} \\ \text{Block Angle} &< \text{Max\_Angle}_{\text{robot2}} \end{aligned} \tag{5}$$

Using this we could then again use the equation of a straight line described in Equation 3 to calculated a y position for the block.

3) Size Constraint: The size constraint was used to generate tests for laser scanners with different sample rates. Different sample rates introduce a unique problem for laser scanners in which a thin obstacle might not be detected because the lasers' beams never intersect with the obstacle. This is represented in the rightmost figure in Figure 2. In the figure, the red beams represent a laser scanner with a higher sample rate. The blue beams are from a laser with a lower sample rate and thus have fewer beams for the same area. If a small block is placed between the beams of one of the laser scanners and not for the other, one laser scanner will detect the block while the other will not. Firstly we want the block to be placed inside the range of both beam lengths. This is done using Equation 6.

$$x = \text{Beam Length}$$
$$y = 0 \tag{6}$$

Next, we want to calculate the range of our laser scanner. This represents the full arch of the field of view of the laser scanner. Calculation of the range is done using Equation 7.

$$\text{Range} = |\text{Max\_Angle}| + |\text{Min\_Angle}| \tag{7}$$

Then we want to calculate the number of sectors. A sector is an area between beams. This can be done using Equation 8.

$$\text{Sectors} = max(\text{Number beams}) - 1 \tag{8}$$

To place a block inside these sectors and to calculate the size of the block we need to know how large an angle each sector is. This is done by diving the range by the number of sectors using Equation 9.

$$\text{Sector Angle} = \frac{\text{Range}}{\text{Sectors}} \tag{9}$$

Finally, we can calculate the maximum size in the y-direction by converting the sector to two right angle triangles and then using the Tangent. This is shown in Equation 10.

$$\text{Sector Size} = tan(\text{Sector Angle} \div 2) \times x \tag{10}$$

We can then use this information to generate a constraint for the size of the block. The constraint is shown in Equation 11.

$$0 < \text{Size} \leq \text{Sector Size} \tag{11}$$

D. Simulation Environment

In order to test to see if the two robots would behave differently in the same environment given different hardware configurations we needed the following:

- Source code of the navigational planner, and robot controllers.
- Source code of the laser scanner configuration.
- Simulator capable of accurately simulating the robot.
- The ability to test for collisions in the simulated environment.
- The ability to alter the environment of the simulator.

The Clearpath Husky robot has open source code for its navigation planner and robot controller. They are using a modification of ROS's navigation stack [4]. This gave us the benefit that the results found on this robot would apply in some degree to all robots using the ROS navigation stack. We wrote a custom node to generate goal positions for the robot. This allowed us to repetitively generate the same precise goal location for each of the robot.

We needed to change the laser scanner configuration easily. The Clearpath Husky robot uses a ROS driver for the SICK LMS1xx line of LIDARs [6]. The source code contains a URDF model file which specifies all the laser scanner attributes. Changing these allows us to change the robots laser scanner.

Pseudocode 1: The software overview

```
for Each Laser Configuration do
    UpdateLaserConfiguration();
    (x,y,size) = GenerateBlockPosition();
    UpdateWorldFile(x,y,size);
    LaunchGazebo(WorldFile, RobotFile);
    LaunchSensorVisualization();
    LaunchNavigationSoftware();
    LaunchGoalSetter();
    while While Robot is Moving do
        c = GetCollisions();
        if c[1] is robot and c[2] is block then
            ReportCollision;
        end
        if c[1] is block and c[2] is robot then
            ReportCollision;
        end
    end
end
```

For the simulator, we used Gazebo a robotic simulator used by ROS [8]. Gazebo has the ability to place objects manually using the GUI interface. However for precise placement of blocks and the ability to run multiple tests we needed to automate this process. To do this we generated a world file which could be loaded by Gazebo. The world file had a single block whose position and size could change precisely using variable floating point values. Gazebo also generated collision data. A custom node was also written to parse this information and detect any collisions between the robot and block. Pseduocode of our approach is listed in Pseduocode 1:

V. Results

The results are broken up into three different constraints. This allows us to isolate the changes and more easily identify the cause of any anomalies in the robots behavior.

A. Blind Spot Test

The first test we ran was for the blind spot constraint. For this, the robots field of view changed. This can easily be seen by the scanner visualizer in Figure 3 and 4.

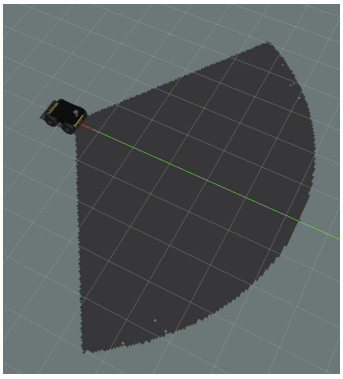

Fig. 3. Laser scanner data and trajectory visualized using RViz for a robot with a maximum and minimum angle of 1 rad.

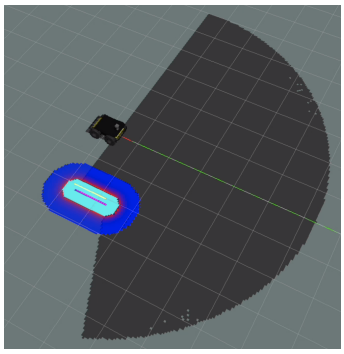

Fig. 4. Laser scanner data and trajectory visualized using RViz for a robot with a maximum and minimum angle of $\pi/2$ rad.

The first thing to note is that our constraints placed the block exactly where we expected it, in the region one laser scanner could see it while the other could not. Figure 3 has no detected obstacles while Figure 4 can detect an obstacle.

The goal was to observe different behaviors to identify bugs. However, in this test, both trajectories are exactly the same. That is due to the obstacle not being an obstruction either robot, even robot 2 which can see it (Figure 4). This, however, is still useful information as it means that the software correctly handles nonobstructive obstacles.

B. Distance Test

For the distance constraint, the robots were simulated having different laser scanner beam lengths. The block was placed just outside of the range of one of the robots but inside the range of the other robot as seen in Figure 5. This again confirmed the accuracy and correctness of our constraints.

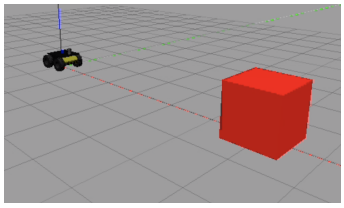

Fig. 5. The block placement in the simulator for the distance constraint.

The first robot had a reduced laser scanner range of $5m$. We, however, noticed unusual behavior of the robots detection algorithm. The robot detected a continuous obstacle in front of it. This can be seen in the laser scanner and trajectory visualization in Figure 6. This is definitely a bug as it can be seen that no such obstacle exists in Figure 5. This was further confirmed when the robot with a beam length of $10m$ correctly identified the block and generated an appropriate trajectory as seen in Figure 7.

The propose of this paper was only test generation and thus due to the project scope, the actual cause was not fully investigated. The bug however was isolated to two possible causes. The bug could be that a hardcoded threshold for obstacle detection was used in the obstacle detection class of the Clearpath Husky robot. We confirmed that it was a hardcoded value as any laser scanner less than $6m$ would cause obstacles to be detected. This would mean that any

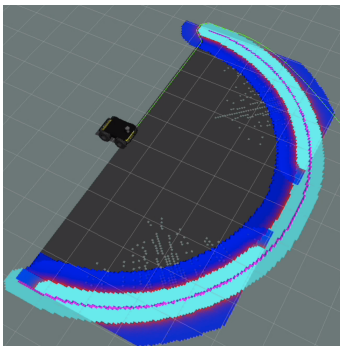

Fig. 6. Laser scanner data and trajectory visualized using RViz for a robot with a maximum beam length of $5m$

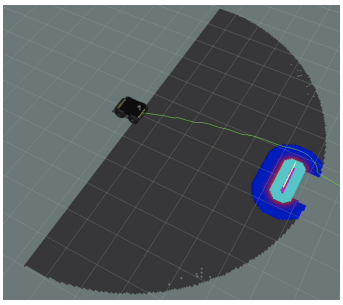

Fig. 7. Laser scanner data and trajectory visualized using RViz for a robot with a maximum beam length of $10m$

laser scanner with a range lower than that hardcoded value was detected as an obstacle. The second cause of the bug could be in the ROS laser scan driver. There could be an error in which changing the parameters in the configuration file do not fully propagate throughout the code leaving certain detection thresholds unchanged.

C. Size Tests

The final test was placing a block such that it was in between samples from the laser scanner. You will notice that the first robot is unable to detect the block. This is seen in Figure 8. The second robot was able to detect the obstacle as seen in Figure 10.

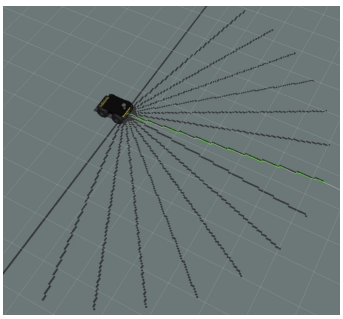

Fig. 8. Laser scanner data and trajectory visualized using RViz for a robot with 16 samples.

This test produced two interesting results. The first interesting note was that the robot with 16 laser scanners drove into the obstacle and continued driving over it. The moment the robot intersected with the poll and our code detected a

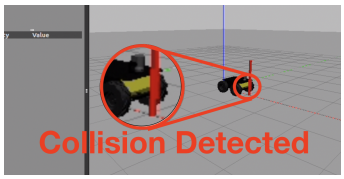

Fig. 9. Collision of robot with 16 samples. The robot here first drove into the pole and continued to accelerate forward.

collision is shown in Figure 9. This is expected as the robot is unable to sense the poll. This is still, however, a useful test as the designer now is shown an example of the robot crashing due to laser scanner changes.

The second interesting result was the robot in Figure 10 with twice as many samples became stuck and was unable to produce a trajectory around the obstacle. The robot drove too close to the block and was unable to turn without colliding with the block. Inspecting the robot specification shows that the robot is capable of backward motion. Thus a suitable resolution to the robot getting stuck would be to reverse enough such that the robot could turn without colliding with the block. The robot however did not do this and remained stuck for the entire test.

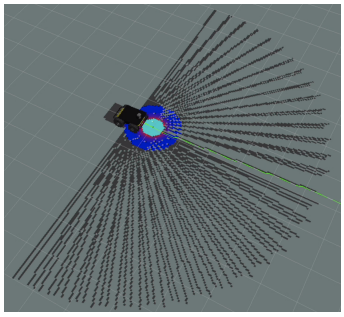

Fig. 10. Laser scanner data and trajectory visualized using RViz for a robot with 32 samples. Note this was taken a second after the robot started driving. That is why it seems as if there are more than 32 samples.

VI. CONCLUSION

The goal of this project was to automatically generate tests for a physical system which had hardware configuration changes. We were able to successfully produce constraints which generated tests leading to changes in robot behavior. Our tests reviled two possible bugs in the code. The first bug was a hardcoded obstacle detection bug, while the other was the inability for the robot to reverse even though the specifications of the robot allowed the robot to reverse.

We believe that tests like this would be beneficial to the robotics community and can be adapted to source code testing as well as the physical tests shown here. For instance generation of test cases highlighting software changes could be achieved through differential symbolic execution of the source code. The path constraints generated by this could then be used to model environments which highlight these changes.

VII. APPENDIX

Our code is available on Github:
`https://github.com/hildebrandt-carl/AutoLaserTestGen`

A video of our project can be found at:
`https://youtu.be/QLC0HHs_QaQ`

REFERENCES

[1] Clearpath Robotics. Husky - Unmanned Ground Vehicle. `https://www.clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/`, 2019. [Online; accessed 27-February-2019].

[2] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. Ltlmop: Experimenting with language, temporal logic and robot control. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1988–1993. IEEE, 2010.

[3] BaekGyu Kim, Akshay Jarandikar, Jonathan Shum, Shinichi Shiraishi, and Masahiro Yamaura. The smt-based automatic road network generation in vehicle simulation environment. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10. IEEE, 2016.

[4] Michael Ferguson. ROS - Navigation. `https://wiki.ros.org/navigation`, 2019. [Online; accessed 30-April-2019].

[5] Microsoft Research. The Z3 Theorem Prover. `https://github.com/Z3Prover/z3`, 2019. [Online; accessed 28-April-2019].

[6] Mike Purvis. ROS - LMS1xx. `https://wiki.ros.org/LMS1xx`, 2019. [Online; accessed 30-April-2019].

[7] Open Source Robotics Foundation. About ROS. `https://www.ros.org/about-ros/`, 2019. [Online; accessed 29-April-2019].

[8] Open Source Robotics Foundation. GAZEBO - Robot simulation made easy. `http://gazebosim.org/`, 2019. [Online; accessed 30-April-2019].

[9] Open Source Robotics Foundation. IRobot Roomba. `https://robots.ros.org/irobot-roomba/`, 2019. [Online; accessed 29-April-2019].

[10] Open Source Robotics Foundation. Turtlebot 3. `https://www.turtlebot.com/`, 2019. [Online; accessed 29-April-2019].