



PACIFIC NW  
28TH ANNUAL  
SOFTWARE  
QUALITY  
CONFERENCE

OCTOBER 18TH – 19TH, 2010



ACHIEVING  
QUALITY  
IN A COMPLEX  
ENVIRONMENT

*Conference Paper Excerpt  
from the*  
CONFERENCE  
PROCEEDINGS

---

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

# Software Quality Assurance in the Physical World

Kingsum Chow<sup>1</sup>, Ida Chow<sup>2</sup>, Vicki Niu<sup>2</sup>, Ethan Takla<sup>2</sup> and Danny Brillhart<sup>2</sup>

<sup>1</sup>[Kingsum.chow@intel.com](mailto:Kingsum.chow@intel.com), Software and Services Group, Intel Corporation

<sup>2</sup> Lincoln High School, Portland, Oregon.

## Abstract

This paper explores the challenges of testing software on a robot in the physical world. Through the case study of running a LEGO robot with motors and sensors on a FIRST LEGO League competition table, it characterizes the environmental factors, some of which are not controllable. Through the sensors that are available on a robot, it characterizes the uncertainties from the sensor readings and the location of the robot. It then describes how it employs a software testing process to test if the robot can perform missions reliably given the less than perfect environment and sensors.

The contributions of this paper are:

1. Characterization of environmental factors in a complex system such as the physical world.
2. Establishing a method to develop the relationship with imprecise sensor readings and the functionality of the software on the robot.
3. Applying the relationship to improve the software testing process to assure the quality and reliability of the robot.

## Biography

*Kingsum Chow is a principal engineer from the Intel Software and Services Group (SSG). He has been working for Intel since receiving his Ph.D. in Computer Science and Engineering from the University of Washington in 1996. He has published more than 40 technical papers and he was issued 10 patents. For the last 10 years, he has been working on characterizing enterprise Java middleware and application performance and scalability on servers. Outside of work, he has been coaching a FIRST LEGO League (FLL) team since 2006.*

*Ida Chow, Vicki Niu, Ethan Takla and Danny Brillhart stayed together as the FLL team Nanites from 2008-2010 as middle school students from ACCESS Academy and West Sylvan Middle School of Portland, Oregon. They have recently joined Lincoln High School as 9<sup>th</sup> graders.*

*Published at the Pacific Northwest Software Quality Conference 2010.*

# 1 Introduction

Testing in the physical world presents unique challenges. In the physical world, the testing environment is constantly changing which makes for differences in the actions that the system executes. When the environment changes, one must make sure that the desirable outcome is reached in all possible situations. One way to achieve this is to change the testing environment to several extremes to assure that the system performs the same in different environments so that the test plan encompasses all possible scenarios. However, this is not the only way to achieve reliability when testing systems in the physical world. Other methods include testing the system without changing the environment, relying on the unpredictability of nature to provide varied scenarios in which to test. In fact, through the FIRST LEGO League, middle school students are participating in software quality assurance in the physical world.

Dean Kamen founded an organization [1] called "For Inspiration and Recognition of Science and Technology" (FIRST). In his words, FIRST's mission is "To transform our culture by creating a world where science and technology are celebrated and where young people dream of becoming science and technology leaders." The leaders of FIRST believe that in our world engineers and scientists currently have the ability to create positive change and revolutions that may alter the way our world operates. The key objectives of FIRST are to get young people involved in the world of engineering and to inspire young people to realize the importance of engineering and technology.

The FIRST program consists of a community that drives for creativity and success among kids all over the world. Within this program is the FIRST LEGO League (FLL), designed to push for innovative solutions and ideas that stand out. FLL is a competition that judges what a team has accomplished by how unique it is and the impact to the community. This way of judging makes kids strive for perfection and imagination, creating ideas that might have never been thought of by adults. With different themes ranging from nanotechnology to green environments each year, it gives a new topic for children to learn about and share with others. The table missions each year are a variety of different tasks that involve triggering machines, delivering and retrieving items, or a combination of both. When teams run missions on the table, there are stationary objects on the table, some of which are secured and are easier to navigate around, and some loosely placed objects, which if are accidentally moved could cause navigational problems. Additionally, there are also different factors on the table, such as the table surface and lighting conditions that may affect the way the robot performs. Some of the problems that we have encountered on the table include; varying table lengths and widths, changes in light intensity and the direction in which light comes from, different types of wall and mat surfaces, and diverse battery levels.

FLL robots and table missions are an effective model for software testing. In FLL we use our LEGO Mindstorms [2] technology to build and program a robot. Using math and science, along with critical thinking, we are able to create and utilize innovative solutions to solve difficult problems. When programming the robot [3] we must ensure that the robot can reliably navigate around the table and perform missions regardless of what unexpected changes it faces.

## 2 Methodology

Software quality in the physical world is not just the software, but also the interaction among the software, the hardware and the environment, i.e., Software Quality in the Physical World = Software x Hardware x Environment.

In FLL, a team can fix the hardware design using a set of LEGO pieces, motors and sensors. But the quality and performance of the robot still depends on the environment [4] that is not necessarily under their control. These environment factors include differences in how the table is constructed and the ambient light condition. As some environment condition is not under control, there is variance from run to run even using the same robot and the same software on the same table. To reduce the variance and increase the confidence of a working robot, more runs need to be made on the same table. To increase the assurance for it on other tables, most tables need to be used. The combination of test cases, multiplied by environment variables, quickly gets out of hand. The ultimate question is how to get the most out of testing without making too many runs.

### 2.1 Environmental factors

The environment for the robot can be divided into static and dynamic categories. The static factor is the one that doesn't change within the test environment. For examples, the light sources, the table, the mat and the fixed structures on the mat can be treated as a constant given a test environment. The dynamic factor is the one that changes within a test environment. For examples, the stray light source and the location of loose objects on the mat are treated as variables.

### 2.2 Imprecise sensor readings and the functionality

In planes, there is something known as the flight data recorder. During the duration of the entire flight, this data recorder [5] constantly collects the data values from all of the equipment on the plane. In the case of a malfunction aboard the plane, the data can be used to resolve the problem of what caused the malfunction. This is the method we employ to develop the relationship of imprecise sensor readings and the functionality of the software. The flight data recorder is also known as black box. Black box is looking at how a program might malfunction while not being able to examine the source code for the program. Thus we call our method "black box data analysis". This method can be used to test for the dynamic and static factors that may affect the program, Black box will take data from all the sensors on the robot throughout the running of the program. Reviewing this data can show what varies in different runs, and what factors cause a malfunction.

### 2.3 Software quality assurance in the physical world

In order to apply this to the robot, there is the problem of limited memory in the NXT brick, and to counter this, we cannot collect data points constantly, as is the case in airplanes. As a workaround, the collections of data points only happen at certain checkpoints. Checkpoints are points in the program that are of importance to the success of the program.

### 3 Case Study

In FLL, a team of students are given a table containing a number of physical tasks they must complete. Some of these tasks require the robot to pick up a loop or knock down an obstacle. To do so, the robot must be able to navigate reliably around the table and that is the main focus of this paper. At this point, one can assume that their robot is not fully reliable and that there are a large number of factors causing the unreliability. For example, it may not have a sturdy chassis or uncontrollable environmental factors such as stray light sources could cause the robot's sensor readings to be off. In the interest of reliability, it was found that implementing a black box into the code helped improve the performance in the long run. During our tests, the robot recorded sensor values at pre-defined check points on the table. The checkpoints are generally landmarks, so we know where the data is taken. After the robot achieves a perfect run, we archive the data to compare to when imperfect runs appear. A perfect run is defined as a run where the robot accomplished everything it was programmed to do without error. Once the archive of imperfect and perfect runs has a sufficient amount of data, e.g., more than ten runs, ranges of perfect run data can be created. We can derive other useful statistics such as standard deviation and 95% confidence intervals. Knowing the confidence intervals for the sensors at a given check point, it becomes possible to program the robot to alert its operator, or even correct itself, when sensor values are outside of their normal ranges. This is done purely through an additional piece of code that checks all sensor values at each checkpoint.

In the FLL case study here, we first describe our experience in testing line following programs and then follow by testing the robot for the entire challenge, combining line following and other missions.

#### 3.1 A simple test case – testing line following programs

To illustrate the problem of testing a program on a robot, we start with a simple program that involves the environment in several ways. The program follows a black line on the mat and tries to follow the line as close as possible and within a reasonable amount of time. Line following is one of the techniques used for robot navigation. An adequate way to test line following is indeed important.

The line following program depends on several environmental factors: the ambient light source, the changes in lighting on the mat, the actual colors and the uniformity of the colors of black and white on the table. The texture of the mat and the flatness of the mat also play a role in affecting the line following activity. While in FLL an icon-based programming language called NXT-G is used, for the sake of simplicity, we converted the NXT-G programs to RobotC to illustrate this case study. RobotC is not used in FLL competitions, but RobotC is really similar to C and we believe most readers of this paper would find it easier to understand RobotC programs.

```

1 #pragma config(Sensor, S1, touchSensor, sensorTouch)?
2 #pragma config(Sensor, S2, rightLightSensor, sensorLightActive)
3 #pragma config(Sensor, S3, leftLightSensor, sensorLightActive)
4 #pragma config(Sensor, S4, ultrasonicSensor, sensorSONAR)
5 #pragma config(Motor, motorA, arm, tmotorNormal, PIDControl, encoder)
6 #pragma config(Motor, motorB, rightMotor, tmotorNormal, PIDControl, encoder)
7 #pragma config(Motor, motorC, leftMotor, tmotorNormal, PIDControl, encoder)
8 /*!!Code automatically generated by 'ROBOTC' configuration wizard !!*/
9
10 task main()
11 {
12     wait1Msec(50); // The program waits 50 milliseconds to initi
13
14     while(true) // Infinite loop
15     {
16         if(SensorValue[rightLightSensor] < 50) // If the Light Sensor reads a value less tha
17         {
18             motor[rightMotor] = 40; // turn left
19             motor[leftMotor] = 5;
20         }
21         else
22         {
23             motor[rightMotor] = 5; // turn right
24             motor[leftMotor] = 40;
25         }
26     }
27 }

```

**Figure 1. A zigzag line following program**

A zigzag line following program (Figure 1) illustrates one of the simplest line following programs written in RobotC and testing even that is not simple. The program defines the four sensors attached in lines 1 to 4 and the 3 motors from line 5 to 7. Without going into the specifics of sensors or motors, it is suffice to say that the sensors generate readings for the robot to act on while the motors control the action the robot is to perform. Note that NXT servo motors also come with rotational sensor readings, i.e., there are actually a total of 7 sensors, 4 are standard alone sensors and 3 come with the motors. The program will check if the light sensor on the right sees something that is too dark (< 50) then it will turn left, or something that is too bright, and then it will turn right. 50 is a reasonable midpoint between bright and dark. The motor speeds (40, and 5) for the turns are arbitrary. In order to test such a program, we would run the program multiple times in many different cases including different starting robot positions to make sure that the program will run under a variety of conditions. However, even if we write down the starting positions of the robot we would like to test, there is still small variation in the way we put the robot down and also almost infinite different kinds of black lines that we need to test with. Yes indeed, there are already too many scenarios to test even for such a simple program.

While the zigzag line following program is sufficient for some simple tasks, it is spending a significant amount of time turning, rather than making progress in going forward. A well known gray line following program solves the deficiency of zigzag line following by introducing a gray region, and within that region, the robot should simply go forward, i.e., turning on both motors at the same speed. Figure 2 illustrates such a program.

```
1 #pragma config(Sensor, S1, touchSensor, sensorTouc?h)
2 #pragma config(Sensor, S2, rightLightSensor, sensorLightActive)
3 #pragma config(Sensor, S3, leftLightSensor, sensorLightActive)
4 #pragma config(Sensor, S4, ultrasonicSensor, sensorSONAR)
5 #pragma config(Motor, motorA, arm, tmotorNormal, PIDControl, encoder)
6 #pragma config(Motor, motorB, rightMotor, tmotorNormal, PIDControl, encoder)
7 #pragma config(Motor, motorC, leftMotor, tmotorNormal, PIDControl, encoder)
8 /*!!Code automatically generated by 'ROBOTC' configuration wizard !!*/
9
10 task main()
11 {
12     wait1Msec(50); // The program waits 50 ms to initialize
13
14     while(true) // Infinite loop
15     {
16         if(SensorValue[rightLightSensor] < 45) // If the Light Sensor value < 45:
17         {
18             motor[rightMotor] = 40; // turn left
19             motor[leftMotor] = 5;
20         }
21         else if(SensorValue[leftLightSensor] > 50) // If the Light Sensor value > 50?
22         {
23             motor[rightMotor] = 5; // turn right
24             motor[leftMotor] = 40;
25         }
26         else // gray region, 45 <= light intensity <= 50
27         {
28             motor[rightMotor] = 20; // go straight
29             motor[leftMotor] = 20;
30         }
31     }
32 }
```

**Figure 2. A gray line following program**

The main difference between the two programs is captured in the else part, i.e., lines 26-30. The gray line following program will simply move forward within the light sensor reading range of 45 and 50. Given the two line following programs, how do we test which program is better for the robot? Intuitively most people would agree time and accuracy would be the two most important criteria. We can easily time the robot from a starting point to an end point. But how do we measure the accuracy of following the line? It is not uncommon for some teams to decide to just observe the run and if the robot is seen following the line by several people, then we call it following the line. Our team took a different approach. We decided not to rely on human judgment at the table but instead we log the sensor readings throughout the entire run while following the line. We checked if the sensor readings are close to the midpoint of the line we are following and add up the sum of the differences between the observed sensor readings and the midpoint. In a way, if the robot is moving fast, fewer data points would be collected and the sum of errors may be small even if individual errors might be big. We believe this is a good way to measure accurate and fast results. We are still exploring other better ways.

One short coming of the gray line following program is the need to slow down the forward move so it can detect an error, e.g. making a sharp turn, before it is too late. To overcome that problem, we implemented PID based line following program that greatly improves the accuracy and performance of line following. Still, line following is just a small component of testing robots in FLL. We are going to describe how we test the entire challenge mission next.

### 3.2 A complex test case – testing all programs for the table competition

For the LEGO Mindstorms NXT robots used in the FIRST LEGO League tournaments all across the world, a plethora of problems are encountered throughout testing the programs. Different ways of testing are employed in order to find as many of the flaws that are existent in any robot.

In testing the multiple missions in FLL, we first map out the programs that will be executed. What the programs do, the order they are executed, and how they are implemented are different for all the FLL teams. However, all the teams face the same challenge of testing their programs on their own table and the robot is still expected to perform in the tournament at a table that is the same according to specification but different because of the static and dynamic factors described earlier.

We attempted to test the static environment variables by constructing models to simulate environments that are not exactly the same as our own table. We developed four test environments based on our experience how a robot that works on a table may fail at a different (but built according to specification) table.



Figure 3. Rough Wall Simulation



Figure 5. Shorter Wall Simulation

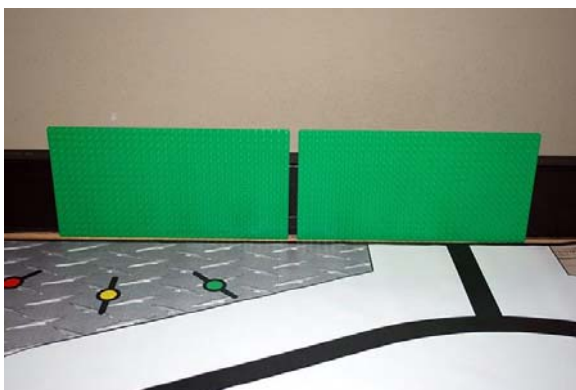


Figure 4. Wall Crack Simulation



Figure 6. Extra Ambient Light Simulation

Figure 3 illustrates changing the environment to test the limits of a robot includes the use of a rough extra wooden board added to already existent wall, in order to see the effects of this surface on the movement of the robot. The rough board also tests how the robot is able to glide along the differently textured barrier. Figure 4 illustrates a test to check how the robot responds to a crack in the wall. The gap between two large green LEGO plates can simulate if a robot that is gliding on a wall

will get stuck in a crack like that. Figure 5 illustrates a test if a table is slightly shorter in length. A shorter table may interfere with sensor readings and may show the limitations of room that might be introduced at the actual tournament runs. Figure 6 illustrates a test with different lighting condition and a shadow is introduced. A bright LED is used in an attempt to drag bad light readings into the sensor as the robot follows the line. In actual tournaments, a strong light can be shining from above, as that allows the tables to be more visually pleasing for the media that is present at tournaments.

### 3.3 Testing static and dynamic environmental factors

After overcoming testing of a specific challenge like line following, and fixed environmental factors, we realized that the number of test cases had already increased exponentially. It was impractical for us to continue testing the robot in the same way and hope to complete the necessary tests within a couple of months, while debugging and fixing many other issues. We resorted to the method we called “black box data analysis” in section 2 to combine situations in a way we can act on.

In our method of black box data analysis, data are taken from all five sensors at specific check points in the program. They are then uploaded to the computer for analysis. Each data point is categorized into which checkpoint it was collected from, which sensor it comes from, and which run of the program it came from.

**Table 1. Black Box Data Collection for NXT Example**

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run10
A	-49	-49	-35	-19	-6	1	1	-2	-2	-2
B	1424	1427	1426	1424	1425	1423	1424	1416	1417	1416
C	1408	1410	1409	1406	1406	1409	1406	1404	1405	1404
2	41	40	41	41	40	40	40	40	40	40
3	54	56	55	55	54	55	55	54	54	55
A	443	443	457	474	487	493	494	492	492	492
B	2099	2105	2101	2099	2097	2099	2100	2103	2104	2100
C	2167	2173	2168	2169	2163	2172	2168	2177	2174	2168
2	25	24	24	24	24	24	24	25	24	25
3	36	36	36	36	36	38	39	38	36	37
A	476	476	490	436	469	461	462	460	459	461
B	3321	3328	3322	3324	3320	3322	3326	3328	3328	3323
C	3376	3383	3378	3379	3371	3379	3380	3391	3386	3380
2	25	25	25	25	25	25	25	25	25	25
3	62	62	62	62	62	62	62	62	62	62
A	-6	-5	10	-29	-3	-4	-4	-6	-6	-6
B	4857	4867	4865	4857	4891	4867	4867	4870	4870	4867
C	4976	4986	4988	4978	5007	4950	4947	4956	4951	4948
2	26	26	32	26	26	41	35	35	39	39
3	61	61	61	61	58	61	61	61	62	62
A	296	298	312	268	298	292	293	290	291	291
B	3556	3566	3543	3557	3782	3587	3580	3584	3591	3589
C	3558	3567	3552	3558	3778	3538	3533	3541	3538	3536
2	39	40	38	39	25	39	38	39	38	38
3	62	62	62	62	37	62	62	62	62	62
A	-34	-32	-19	-33	-13	-12	-11	-14	-13	-13
B	3138	3147	3125	3135	3358	3167	3160	3164	3170	3168

C	3140	3148	3135	3140	3359	3121	3116	3122	3120	3117
2	24	25	25	25	48	25	26	25	25	25
3	62	62	62	62	46	62	63	63	63	62
A	0	0	16	-21	4	0	0	0	0	0
B	2439	2446	2423	2437	2662	2465	2459	2464	2474	2474
C	2436	2444	2432	2438	2658	2416	2415	2421	2419	2416
2	50	50	50	50	50	50	50	50	50	50
3	63	63	63	63	64	63	63	64	63	64
Blue	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Grey	Y	Y	Y	N	N	Y	N	Y	Y	Y
Red	Y	Y	Y	Y	N	Y	Y	Y	Y	Y
Brown	N	N	N	Y	Y	Y	Y	Y	Y	Y
Beacons	8	8	8	8	8	8	8	8	8	8
Base	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Battery	7.9	7.9	7.9	7.9	7.8	7.8	7.8	7.7	7.7	7.7

Table 1 shows how the data is collected. A, B, C, 2, and 3 all represent different types of sensors. A, B and C are rotational sensor readings associated with motors A, B and C. Sensor readings 2 and 3 are left and right sensor readings. Each group of the horizontal rows of “A, B, C, 2 and 3” is a set of data read at a given check point. In the table, 7 check points are taken in a given run. The columns contain different runs of the data. A total of 10 runs were made. Near the bottom of the table, the rows “blue, grey, red, brown, beacons, base” all describe the success conditions for the FLL challenge. A “Y” means that component of the challenge is completed and an “N” means the robot fails to achieve that component of the objectives. The battery section of the table lists the voltage level at the time of the run. We used the data from the successful runs to determine the range of working sensor values for the check points. We then checked if the runs that failed had values outside the confidence intervals of the successful runs. We also implemented a warning sound in the program should the robot predict that it is going to fail based on the confidence intervals. We believe we have only scratched the usefulness of the “black box data analysis” approach and we plan to refine it in our next experiments.

## 4 Discussion

Robot Software [5] consists of the instructions that control a robot's actions and provide information regarding required tasks. When a program is written using this software, the robot is able to execute commands and perform tasks. Programming robots can be a complex and challenging process, and while it has become easier over the years, the lack of cross-platform industry standards has affected the development of software tools for robots compared to other automated control systems.

Software quality assurance in some way is inspecting the products in order to determine whether they meet the requirements and also detecting the defects of the system. In reality, there is no perfect test system that can detect all defects. It just reduces defect risks as much as possible. The ISO 9126-1 software quality model [7] identifies 6 main quality characteristics, namely functionality, reliability, usability, efficiency, maintainability, and portability. In this paper, we focus on functionality,

reliability and efficiency. Hwang [8] describes an alternative method to adapt software quality assurance to robotics.

Nakajima et al [9] describes a software design contest involving the design of software to automatically control a line-trace robot, and conduct running performance tests was held. They found that the quantitative measurement of the structural complexity of the design models bears a strong relationship to qualitative evaluation of the design conducted by judges. However, there is no strong correlation between design model quality evaluated and the final system performance as the software judging might not have taken into account of the algorithm used to handle environmental variables. Our method here does include a testing process that would enable the correlation of the black box data analysis and the outcome of the robot performance.

## 5 Conclusion

This paper describes the authors' experience in testing software quality in the physical world using a case study based on FIRST LEGO League competition. After staying together five years as a team, we have been exploring ways to improve the testing process to find out if the robot is going to perform on a different table in the competition environment. We worked on the characterization of environmental factors in a complex system such as the physical world and establishing a method to develop the relationship with imprecise sensor readings and the functionality of the software on the robot. We applied the relationship to improve the software testing process to assure its quality and reliability.

## Acknowledgments

Moss Drake helped review and improve the quality of this paper.

## References

1. FIRST. <http://usfirst.org/>
2. The LEGO Group, LEGO Mindstorms, <http://mindstorms.lego.com/>
3. Joseph L. Jones, Robot Programming – A Practical Guide to Behavior-Based Robotics, McGraw Hill 2004.
4. Ida Chow, Vicki Niu, Ethan Takla and Danny Brillhart, Designing and Testing Robots for Reliable Performance, FIRST Robotics Conference, April 14, 2010, Atlanta, GA.
5. Flight data recorder. [http://en.wikipedia.org/wiki/Flight\\_data\\_recorder](http://en.wikipedia.org/wiki/Flight_data_recorder)
6. Robot Software. [http://en.wikipedia.org/wiki/Robot\\_software](http://en.wikipedia.org/wiki/Robot_software)
7. ISO 9126 Software Quality Model. <http://www.sqa.net/iso9126.html>
8. Sun-Myung Hwang, Testing Method for Intelligent Robot Software Components Testing, <http://staff.aist.go.jp/t.kotoku/conf/iros2006ws/WS4-6.pdf>
9. Eiji Nakajima, et al, "Experiments on quality evaluation of embedded software in Japan robot software design contest," pp.551-560, 28th International Conference on Software Engineering (ICSE'06), 2006. <http://www.computer.org/portal/web/csdl/doi/10.1145/1134285.1134363>