UML Model Based Design of the Claw Car Robot

Andre Layne[1], **Adria Mason**[1], **Yujian Fu**[1], **Mezemir Wagaw**[2]
[1]Department of Electrical Engineering & Computer Science,
[2]Department of Food & Science,
Alabama A&M University, Normal, AL, United States

Abstract - *Robots are intricate systems and applied in many aspects of today's society. It is highly desirable to design and develop robust robotics systems. This paper aims at developing an autonomous robotic system using object-oriented software development (OOSD) methodology – UML – to ensure the quality of the system. Unified Modeling Language (UML), a typical OOSD method, is a standard visualization language for object-oriented system development that has been widely used in the design of safety critical and mission critical systems (e.g. aeronautic systems, missile defense systems, etc). In this paper, the UML class diagram is used to represent the static structure and relations of objects. The dynamic behavior is modeled in the state machine diagram. A case study is performed in the LEGO NXT tool kit, a low cost highly integrated educational robot setting, in Java programming language. In this case study, a Claw Car Robot is designed and assembled with the function of continuously moving forward and stop on the color definition. The robot can close the claw and clasp the object in its path once an object is detected. The LEGO NXT tool kit includes multiple sensors and supported by several platforms including Java and C++. This LEGO NXT tool kit is very convenient for the design and implementation of robotics systems, and has been widely adopted in institutions for educational and initial research purposes.*

Keywords: UML; Autonomous; NXT; Object-Oriented Programming Language

1 Introduction

The robot population is doubling every few years. According to IEEE Spectrum [6], the world population of robots had reached over 4.5 million at the end of 2006 and 8.5 million at the end of 2008. The size and growth of these numbers show that robots contribute to a very important role in our society today. These robots use various types of integrated technologies to achieve specific goals in various types of environment. Therefore, reliability and quality of the robotics system is becoming more and more important. However, there is not enough research on the building of reliable robot systems. In this research work, we developed the UML model of the robotics system, and then implemented in Java, based on the assembled LEGO NXT tool kit.

A typical feature of robotic systems is multiple interfaces and multiple objects. To provide flexible functionality, a robot usually integrates with several different types of sensors to get the data from its environment as well as multiple devices for different purposes such as arms, claws and some other tools.

All these integrated sensors and devices apply their own way to read and handle information, which is defined by various APIs. Therefore, designing and the programming of the multiple interfaces, and also integrating them to the system smoothly, are challenges for the robotics design. Any miscommunication between controller and sensors or devices may cause unexpected results and huge losses. It is key to build and develop reliable and quality software for the robotics systems.

Object-oriented software development (OOSD) methodology has been widely used in the design of safety critical and mission critical systems (e.g. aeronautic systems, missile defense systems, etc). Unified Modeling Language (UML) [9, 10, 11], a typical OOSD method, is a standard visualized language for object-oriented system development. In this paper, we proposed a UML 2.0 model that includes the class diagram and state machine diagram on the robotics systems to specify the system design requirements, and use Object Constraint Language (OCL) to define the desired properties. Therefore, the object-oriented design model of a robotics system includes three components in general – static structure, dynamic behavior, and property specification. This paper presented the UML based model of the robotics system that is represented by the above three components.

This work is implemented in an assembled claw car using LEGO NXT tool kit and implemented with the Java programming language. The Mindstorm NXT brick uses a 32-bit ARM processor as its main processor, with 256 kilobytes of flash memory available for program storage and 64 kilobytes of RAM for data storage during program execution. To acquire data from the input sensors, another processor is included that has 4 kilobytes of flash memory and 512 bytes of RAM. Two motors can be synchronized as a drive unit. To give the robot the ability to "see," the ultrasonic sensor, which is accurate to 3 centimeters and can measure up to 255 centimeters, and the light sensor, which can distinguish between light and dark, can be attached to the brick. Finally, the two touch sensors give the ability for a robot to determine if it has been pressed, released, or bumped, and react accordingly [12].

This paper is organized as follows. Section 2 introduces the background knowledge used in this work. Section 3 presents the hardware assembly and functions for the LEGO robot. Section 4 shows the UML model of the robotics systems. Section 5 presented the Java implemented with the LeJOS package. Section 6 discusses the results and the conclusion.

2 Background

In the Object-Oriented Software Development (OOSD) approach, the system is viewed as a collection of multiple objects and with various interfaces definitions. The functionality of the system is achieved by the interaction and communication among these objects through messages. The Unified Modeling Language – UML – is developed on the above principles and widely used in the complex embedded system and large scale software intensive system development. In this paper, we presented a UML based model driven architecture for the robotics design that includes three components – static structure (class diagram), dynamic structure (state machine diagram) and property specification (OCL). Therefore, we simply introduce each component in the UML syntax.

UML defines twelve types of diagrams which fall into three categories [9, 10]: (i) *Structural Diagrams* which include the Class Diagram, Object Diagram, Component Diagram, and Deployment Diagram, focus on the static organization of instance of the system; (ii) *Behavior Diagrams* which include the Use Case Diagram, Activity Diagram, Communication Diagram, and State Machine Diagram, focus on the functions and collaborations among instances; and (iii) *Model Management Diagrams* which include Packages, Components, and Subsystems, focus on the packaging and setting of diagrams. UML has been used across a wide variety of domains, from computational to physical, making it suitable for specifying systems independently of whether the implementation is accomplished via software or hardware. Since UML was initially introduced in the software domain, most commercial tools based on UML descriptions have the ability of generating software code, such as Java and C++.

However, no such tools are commercially available that can design and synthesize UML models into a model for robotics system model directly, thus imposing a limitation for the usage of UML in robotics system design. Additionally, it is also observed that assuring correct functional behavior is the dominating factor of a successful hardware design. It shows that up to 80% of the overall circuit design costs are due to verification tasks. Assurance of quality of robotics is a key issue now.

To complement the UML diagrammatic notation, the Object Constraint Language (OCL) [14, 15] can be used to express constraints and specify the effect of operations in a declarative way. In each predicate of OCL, the logical statements must be satisfied by all valid instances of the system that are represented by constraints.

The OCL [15, 14, 13] is a textual, declarative language based on first-order logic and set theory. In addition to expressing constraints on class diagrams, OCL can also be used to specify the effect of the execution of an operation, using pre and post conditions. A pre condition is an OCL statement that has to evaluate to true before the execution of an operation, while a post condition is a statement that has to evaluate to true when the operation terminates.

3 Lego Robot & Functionalities

The LEGO Mindstorm tool kit is composed of five external sensors and three motors except for many other pieces that give the physical design and construction of the robot flexibility. In this section, we introduce the assembly and functionalities of the LEGO robot claw car design as well as the challenge issues during the development.

3.1 Functionalities

The Claw Car Robot needs to move forward on the color - white. However, once the color - black is detected, the robot will close the claw and clasp the object in its path. The robot will also slow down once black is detected and the claw is closed. As the Claw Car Robot passes over white again, the robot continues in this same state until the color - black is detected again. Once black is detected again, the robot will open the claw and release the object. In addition, almost simultaneously, the Claw Car Robot will end its program, as designed. In this work, each sensor has its own API. The NXT needs to be able to work with the sensors properly to realize a stable and reliable system. All these factors point to a strong need to maximize software and system development productivity through the use of embedded system platforms, reuse, and synthesis methods driven from system-level models.

3.2 Hardware Development

In the robotics community, most robots manipulate objects using what is called an End of Arm Tool (EOAT) [5]. The most common type of EOAT is the robotic gripper. These grippers come in various shapes and sizes. There are two different categories of robotic grippers which are friction and encompassing robotic grippers. Friction robotic grippers are used to hold objects by using force only. Encompassing robotic grippers surround objects to grasp them and do not use much force at all. To make a decision on which type of gripper we would use, we had to take into consideration the material we would build the object with as well as the type and size of the objects this robot would manipulate. Since our Robot has been built with light-weight materials, it will be used to pick up light-weight and flexible objects. An encompassing style gripper shown in figure 1 would be the best type to use.

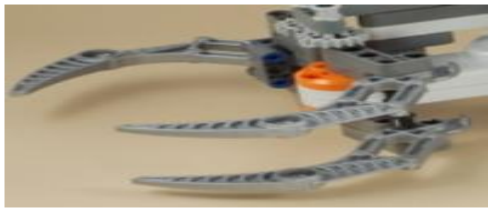

Figure 1. A robotic gripper from NXTPrograms.com

3.3 Challenge Issues

It is not hard to build a mobile robot with the above expected functionalities using the LEGO NXT Mindstorm toolkit. The challenge issue is how to build a reliable robot to satisfy the expected properties and maintain the stable behaviors as required. For instance, a light sensor is used to detect light values given off from the surface beneath the robot. One problem is how to make sure the claw car can detect specific light values efficiently, and manipulate objects in its environment according to the light values detected. We needed an approach that can efficiently respond to light values of the terrain beneath the robot, and perform object manipulation based on these values. In sum, the key issue is how to develop a robotics system with respect to the user requirements, minimize risk, maintain correct behavior, and improve quality.

The unreliability issue comes from two aspects in the LEGO robot design from our study. One is from the imprecision of the sensors. The LEGO kit is used for the educational purpose and many sensors do not have high precision to reflect the required value. For instance, the responded value for white color and light yellow color can vary between 49 to 52, depending on the lighting condition. Another major issue is from the control software design, which is the one this research focuses on. For instance, even though we give enough space for the white value to be changed, the system may still do not maintain stable status due to internal or external stimuli. For instance, the light sensor may stop working. If there is no other way to fix this sensor, the robot will fail the duty. To solve that problem, we included an alternative light sensor and require the system to continue working with the same behavior, if one of these two sensors fails.

3.4 Redundancy Design

We needed a redundant sensor that will be able to perform the work needed if the default sensor was down. To mitigate this concern we added a secondary, or back up sensor that will become active if the default or primary sensor isn't available. This back up sensor is working during the robot running time. Which means the back up sensor is detecting color, responding to the NXT brick, and maintains the same duty as the default sensor. Once the default light sensor is unplugged, the back up sensor can automatically resume duty without any interruption of the system. Figure 2 displays the image of the robot with redundant sensors. In Figure 2(a), the robot is driving with both sensors working. In Figure 2(b), the default sensor is unplugged and the back up sensor is taking control after the primary sensor isn't functioning.

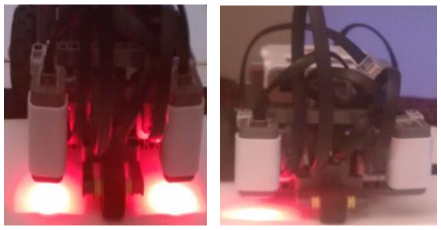
(a) (b)

Figure 2. View of default & secondary light

& System Design

Initially proposed as a unifying notation for object-oriented design, UML has added a semantic underpinning that makes it possible to build platform independent descriptions that can be used by designers and architects to make informed decisions about hardware/software tradeoffs. We present UML based architecture for the robotics system design to ensure the quality of the robot. The architecture is composed of three components (Figure 3.) – static structure (represented by class diagram), dynamic structure (represented by communication diagram or state machine diagram), and system properties (specified by OCL).

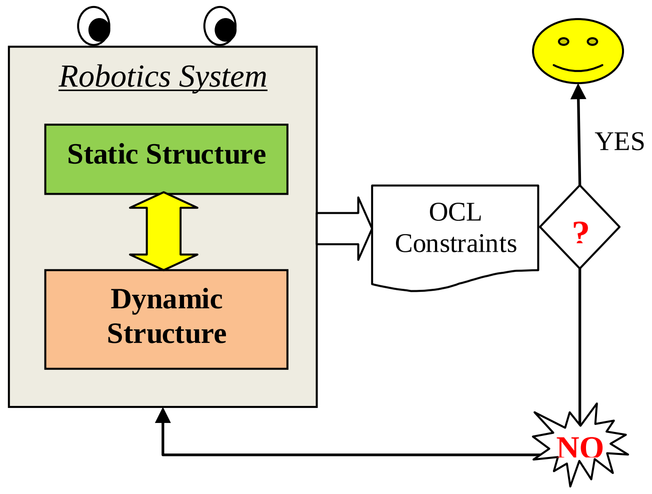

Figure 3. UML based robotics system architecture

The OCL specification can describe all desired constraints on the static structure and dynamic structure. From Figure 3, we can see that is the OCL properties are not satisfied, we can go back to the model architecture and find out what is the problem. After the properties and constraints are ensured, then the system can be implemented based on the model. The verification of OCL properties can be done by software testing tools or model checking techniques.

In the following, we use the LEGO claw car robot to illustrate the UML architecture model.

4.1 Class Diagram

According to Michael Blaha and James Rumbaugh [1], a "class diagram provides a graphic notation for modeling classes and their relationships, thereby describing possible objects. Class diagrams are useful both for abstract modeling and for designing actual programs.” The class diagram of this LEGO claw car robot is shown in Figure 4.

The Claw Car Robot is a robot that drives forward, and is composed of aggregate parts; motors, a light sensor, and a backup light sensor. To represent the above function, the Class diagram shows this relationship as a model with each one of the real world objects represented appropriately. The FinalClawCar Class is the container class for the remaining classes. Therefore, Figure 3 shows the FinalClawCar Class with an aggregation relationship to the other classes.

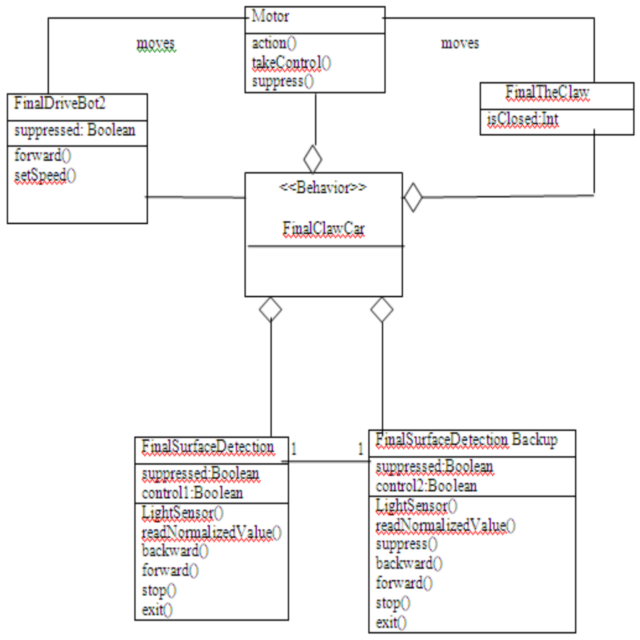

Figure 4. Class Diagram of the Claw Car Robot

4.2 State Machine Diagram

The dynamic behaviors are represented by the State Diagram, which includes state, transition and events. A state is an abstraction of the values and links of an object. An event is an occurrence at a point in time. Events represent points in time. States represent intervals of time. [1]. Before implementing our actual code, it was necessary to first provide a full description of what each object would do in response to different events.

The possible states, transitions, and events of the Claw Car Robot are shown in Figure 5. The black dot represents the entry into the diagram. In this case, the object is originally in the idle state until the Enter button is pressed on the NXT Brick, and the robot begins to drive forward. While the robot is in the ‘Moving Forward’ state, it responds to two possible events. In this diagram, either the robot detects black for the first time, which in this case, it moves into a claw close state. If the color black is detected a second time the claw opens, releasing any obtained object. The challenge issue here is the claw cannot close if there is no object in hand. However, how to detect the claw holding the object is a challenge. Therefore, we reduce the question to a simple case – a) first, detect the object; b) once the object is detected, the car moves and pick up the object; and the claw closes. From the state machine diagram in Figure 5, you can see the car releasing the object without detecting if there is object holding. But, the robot will check the line color before releasing the object (Figure 5).

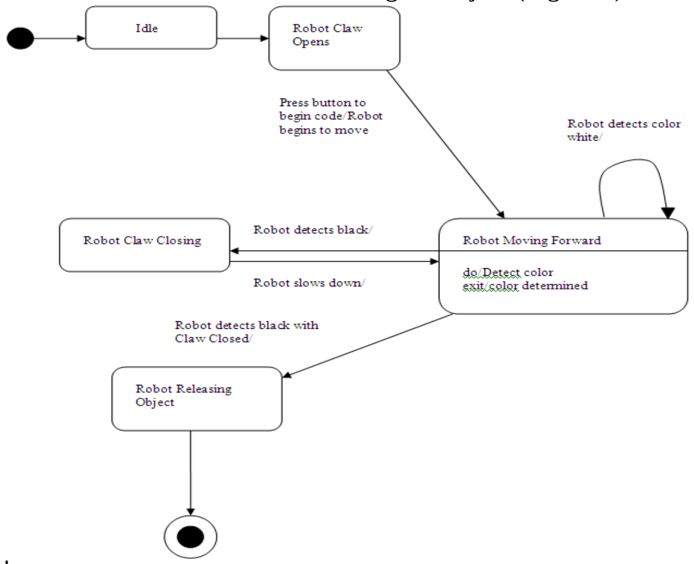

Figure 5. State machine diagram of the Claw Car Robot

4.3 Quality Of Robot – Object Constraint Language

UML provides a formal language to specify and express the constraints within a system, named the Object Constraint Language (OCL) [8]. A constraint restricts the values that elements can assume. We utilized the OCL 2.0 version to specify the constraints within this system. Several constraints are defined for the system on classes. For example, for the class Motor, we define an invariant as if the speed is larger than zero, then the robot is moving (shown in formula (b)). As discussed before, the claw open and close checking is critical to grab the object. Therefore, we have defined the invariant for the claw close checking (shown in formula (f)).

Context: Motor
INV: Motor.forward *implies* Motor.isMoving (a)

INV: Motor.setSpeed()>0 *implies* Motor.isMoving (b)

Pre: Motor.stop() (c)
Post: Motor.isMoving = false (d)

Context: LightSensor
INV: LightSensor.getLightValue() >330 *implies* LightSensor.isWhite (e)

Context: FinalTheClaw

INV: FinalTheClaw.isClosed= = 0 *implies* FinalTheClaw.isOpen (f)

5 Robot Implementation In Java

This robot is implemented in Java, a typical object-oriented programming language (OOPL). There are several reasons to choose Java as the choice coding language.

First, Java is considered a pure OOPL with typical OO features, including encapsulation, inheritance, polymorphism, besides objects and classes. In Java, object communication is defined by response to classes on messages. When a certain method is executed in response to a message to another object, this method can generate new information that can be a message. Therefore, Java is a perfect implementation language of UML model. Secondly, currently, there is not enough LEGO NXT projects that are implemented in Java. Most of LEGO projects are used for the motivation of high school students and the demonstration of robotics. Therefore, the implementation is mainly reduced to the simple diagram based langauge such as LabView. However, there is no space for the students to explore the design issues and quality assurance of the software control systems. The last reason is LEGO Mindstorm Kit provides free platform for Java applications named LeJOS [16].

Most robotics implementation uses Behavior Programming (BP), which is supported by the LeJOS package. The important aspect of BP is sequential ordering of the concurrent behaviors issued from multiple objects of a robot. In other words, BP uses sequence to implement concurrence. The Behavior interface is located in a package called Robotics.Subsumption. The Subsumption package also provides a class that handles the Behavior objects called an arbitrator. The Behavior interface provide three methods that allow the code to work in a more logical manner than writing a large amount of if-else statements, which LejosSourceForge.com calls “spaghetti code”. Instead of our code being tangled with all if-else statements, we were able to take a behavioral approach and write the application so that each method and function performed would work in a logical sequence.

The Behavior interface uses three methods to provide a seamless, behavioral interaction between multiple objects. The methods are; takeControl(), suppress(), and action(). These three methods are described in further detail below:

suppress() - The suppress method returns true whenever a specific object doesn’t want to take control, or when the object’s takeControl() method is false.

takeControl() - An object’s takeControl() method returns a Boolean value whenever it reaches a condition where it can return a true value. When this occurs, the object’s action() function will perform some action according to its priority level. The suppress method is turned false, and all other Behavior objects should remain suppressed.

action() - when an object’s suppress() method returns a false value and the takeControl condition returns true, the code that is within the action() method is performed.

The implementation of the robot is based on the UML model and maintains the constraints defined in the OCL. In this section, we illustrate the Java implementation on the LEGO NXT Mindstorm tool kit on each class. All the classes are defined in Robotic.Subsumption package and Behavior Interface and discussed in the following.

The FinalDriveBot2 Class (Figure 6) is responsible for controlling how and when the claw car robot drives. Since the robot should always want to drive, we initially set the suppress and takeControl functions to false. Although the FinalDriveBot2’s suppress() method is set to false, if any higher level priority object’s takeControl() method returns true, the FinalDriveBot2’s suppress() method will set to the true value. Once all other object’s takeControl() method returns false, FinalDriveBot2 will resume.

The FinalSurfaceDetection Class (Figure 7) utilizes the LightSensor Class. Through a method called readNormalizedValue, the FinalSurfaceDetection Class will detect light values reflected from the surface beneath the Robot. This class is set to take control if it detects the color black or a light value above 330. Once this class takes control, it will check the value of a static variable named FinalTheClaw.isClosed. If isClosed is set to 0, the robots claw will close and change the value of FinalTheClaw.isClosed to 1. If black is detected and the value of FinalTheClaw.isClosed is 1, then the claw will open. The logic behind this is to provide the robot with a flag to indicate whether the claw is in an opened or closed state.

The FnalSurfaceDetectectionBackup Class (Figure 8) provides a 2nd LightSensor object for the default sensor. The code is very similar to the FinalSurfaceDetection Class since it will be checking for most of the same conditions. This will allow redundancy between the two sensors. The design is set up so that if one sensor stops working, the other sensor will resume the work without any downtime. The takeControl() method for this class is set so that it will only takeControl if black is detected and sensor1 is returning 0(indicating sensor 1 is not functioning). Since there is a static variable set up in the FinalTheClaw Class, to represent the state of the claw, the second sensor will know the current state of the claw, and therefore will be able to make a logical decision on the next state of the claw.

The FinalDriveBot2 class consists of only one static class variable. This variable is isOpen(). The isOpen variable is used by the FinalSurfaceDetection and the FinalSurfaceDetectionBackup class to determine whether the claw is in an open, or closed state.

```
lejos.nxt.Motor;
import lejos.robotics.subsumption.Behavior;
public class FinalDriveBot2 implements Behavior{
private boolean suppressed = false;
  public boolean takeControl() {return true;}
  public void suppress() {suppressed = true;}
  public void action() {
    suppressed = false;
    Motor.A.forward();
    Motor.C.forward();
    while(!suppressed) Thread.yield();
                 Motor.A.setSpeed(100);
    Motor.C.setSpeed(100);  } }
```

```
public class FinalSurfaceDetection implements  Behavior {
   LightSensor light = new LightSensor
              (SensorPort.S1);  //default  light Sensor object
    boolean suppressed = false;
   public boolean takeControl(){
      // this function will take control
                           //if a light value is returned less than 330
      //but not equal to 0
      boolean control1=false;
      if(light.readNormalizedValue() < 330
                         && light.readNormalizedValue()> 200){
         control1=true;}
         return control1;   }
   public void action() {
      suppressed = false;
      if(  FinalTheClaw.isClosed == 0 ){
      Motor.B.backward();// close claw
      FinalTheClaw.isClosed = 1; //flag
                         //used to indicate if and object has been
                         //grabbed
      while(Motor.B.isMoving() )
          Thread.yield();   }
      else { Motor.B.forward();//open claw
      Motor.A.stop();
      Motor.C.stop();
      System.exit(0);// exit program
      while(Motor.B.isMoving() )
         Thread.yield();
         }
      }
```

Figure 7. The code of class FinalSurfaceDetection

```
public class FinalSurfaceDetectionBackup implements
Behavior{
   LightSensor light1 = new
                         LightSensor(SensorPort.S1);
             LightSensor light2 = new
   LightSensor(SensorPort.S3);
   boolean suppressed = false;
   public boolean takeControl(){
                         boolean control2=false;
                         if(light2.readNormalizedValue()< 330 &&
                         light1.readNormalizedValue() ==0)
                         { control2 = true;}
       return control2;    }
   public void action() {
      suppressed = false;
      if( FinalTheClaw.isClosed == 0 ){
                         … …
      } else {
      … …
            while(Motor.B.isMoving())
         Thread.yield();
            }     }     }
```

Figure 8. The code of class FinalSurfaceDetectionBackup

5.1 Integrated Technologies

This robot was integrated with different technologies to make it successful in its proposed goal. Light Sensor technology was used to sense and determine light values. The Robotic Motor, or mechanical motor technology, was used to mobilize our robot as well as give the gripper functionality. Lego's NXT brick was used as a processor to perform logical decisions and calculations commanded by the downloaded software application. Each one of these technologies worked together in this robot to perform each required action as efficient and seamless as possible.

5.2 Robot Behavior Validation

The system validation is done by checking if the robot meets all requirements and constraints specified. Initially, there were some problems observed. Some of the issues were caused by the design and some were introduced in the implementation. For instance, the robot could not resume the duty when one sensor was down during execution of the robotics system. This was caused by the off-the-shelf light sensor. The back up light sensor was originally built as an off-the-shelf light sensor. Therefore, it wasn't able to resume functionality after the default sensor was down. After examining the design, we found that it can be solved by fixing a variable declaration. To realize the runtime back up sensor, a static variable is introduced and is updated during the sensor switch. To validate the OCL properties, we use the LCD to display the information that is consistent with the robot moving. During implementation, the display is consistent with the robot movement and we found that all constraints are maintained.

6 Conclusion

This paper presented UML based robotics system design architecture on the three components – static structure, behavior structure, and OCL constraints. The approach to building correct and reliable robotics is validated in a LEGO NXT Mindstorm tool kit on a well developed claw car robot. During the study, we have carefully developed the UML robotics architecture model, design, assembly, and implemented the software code in the Java platform. Afterwards, a system validation was conducted to validate the OCL constraints of the robotics system.

From the study on the LEGO NXT tool kit, we can conclude the following: First, the fundament diagrams of UML model with the OCL constraints are suitable for the design and development of reliable robotics systems. Secondly, the UML based robotics architecture can be used for the general robotics system design. Finally, the LEGO NXT tool kit can be used for the fundamental design and implementation research study.

For the future work, we expect two aspects – one is the real time embedded system specification. The other aspect is developing model checking on the OCL constraints. The UML based robotics system architecture can be used to describe the real time system architecture. OCL is suitable for the specification of timing constraints if the class diagram and behavior diagrams include the timing concerns. Secondly, the constraints can be validated in two other ways – a) model checking and b) assertion implementation.

UML robotics architecture model allows the system analyst and developer to focus on front end conceptual issues before implementation. Using this architecture, we are able to develop reliable robotic controller code that performed all specifications and requirements. It is worth to note that LEGO NXT tool kit is a good set for the low cost robot design. However, the imprecision of the accessories causes some problems during actual implementation.

This study shows that combining the use of UML and OCL, the flexibility of the Lego NXT kit, and the robustness of Java LeJOS, we were able to build a robot that met requirements. Although there were specific hardware parts necessary for these abilities to be realized, the use of UML gave us the ability to use a clear, concise method in the software development process to reach each one of the robot's projected goals.

7 Acknowledgements

We would like to extend our thanks for valuable comments from reviewers.

8 References

[1] Blaha, M. and Rumbaugh, J., *Object-Oriented Modeling and Design with UML Second Edition.* Pearson Prentice Hall, 2005

[2] The Association for Computing Machinery, Inc. A list of Implicit Subject Descriptors in ACM CCS (N.D.) Available from: http://dl.acm.org/lookup/CcsNoun.cfm

[3] The Association for Computing Machinery, Inc. Copyright 2011 Retrieved on November 26, 2011 available from: www.acm.org

[4] Dave Parker, LEGO MINDSTORMS NXT! Gives fun projects and building instructions using LEGO MINDSTORMS NXT robotic kit. Copyright 2007-2011, available from www.nxtprograms.com

[5] Robotic Equipment Spotlight, January, 2011. Available from: http://www.robots.com/blog.php?tag=496

[6] Guizzo, Erico IEEE Spectrum Inside Technology, World Robot Population Reaches 8.6 Million Wed, April 14, 2010 Available from http://www.robots.com/blog.php?tag=496

[7] A. Brown, Robot Population Expansion, ASME, February 2009, http://www.asme.org/kb/news---articles/articles/robotics/robot-population-explosion

[8] Document Associated With Object Constraint Language, version 2. http://www.omg.org/spec/OCL/2.0/. May 2006.

[9] Jacobson, I., G. Booch and J. Rumbaugh, 1999. The Unified Software Development Process, Addison-Wesley.

[10] Gomaa, H., 2000. Designing Concurrent, Distributed and Real-Time Applications with UML. Addison-Wesley.

[11] Maciaszec, L., 2001. Requirements analysis and system design, Addison-Wesley.

[12] Bagnall, B., 2007. Maximum Lego NXT: Building Robots with Java Brains. Variant Press.

[13] JosWarmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. The Addison-Wesley object technology series. Addison Wesley, Reading, Mass., 2 edition, 2003.

[14] JosWarmer and Anneke Kleppe. *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley, 1998.

[15] OMG. OCL Version 2.0, 2006. Document id: formal/06-05-01. [cited April 2008]. Available from: URL: http://www.omg.org.

[16] LEGO Team. NXJ API. Available from: http://lejos.sourceforge.net/nxt/nxj/api/index.html.