

July 2009

Remote Procedure Call Generator for Distributed Hardware

Galia A. Traub

Worcester Polytechnic Institute

Follow this and additional works at: <https://digitalcommons.wpi.edu/mqp-all>

Repository Citation

Traub, G. A. (2009). *Remote Procedure Call Generator for Distributed Hardware*. Retrieved from <https://digitalcommons.wpi.edu/mqp-all/2902>

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

Remote Procedure Call Code Generator Framework for a
Distributed Hardware Architecture

Major Qualifying Project Report:

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science by

Galia Traub

Date: July 25, 2009

Approved:

Professor Michael Ciaraldi, Major Advisor

Abstract

The purpose of this project was to create an extensible RPC generator framework optimized for distributed embedded hardware devices such as sensors, actuators and controllers. This paper describes the research and implementation of such a system

Acknowledgements

I first would like to thank Professor Ciaraldi for advising this project and being available to answer questions at odd hours.

I would like to thank Professor Pollice for providing me his resources on code generation. They offered a great deal of insight into the topic and helped to answer many questions.

I would also like to thank Neuron Robotics and its developers for allowing me to use their hardware, as well as the support I received from them in troubleshooting the devices.

Table of Contents

Abstract.....	i
Table of Figures.....	iv
1.Introduction.....	1
2.Problem Statement.....	3
3.Background.....	4
3.1 Understanding Robotics Development.....	4
3.2Distributed Computing.....	6
3.3 Code Generation.....	7
4.4 Finding a Niche.....	9
4.Methodology.....	10
4.1 Tools.....	10
4.2 In the Beginning.....	10
4.3 The Great Refactoring.....	10
4.4 The Module Description Language.....	11
4.5 The Parser.....	13
4.6 The Generator.....	13
4.7 Change Management.....	14
5 Results and Analysis.....	14
5.1 Future Work.....	17
Appendixes	18
Appendix A: Project Requirements.....	18
Appendix B : Table Describing the Final MDL.....	19
Appendix C : XML Schema Document.....	20
Appendix D: A Sample MDL.....	21
Appendix E: Code Generated From a Sample MDL.....	22
Appendix F: Using Generated Code to Communicate with a Module.....	25
Appendix G: Full UML Diagram of the Generator.....	27
Appendix H: A Tutorial in using the Generator.....	28
References	29

Table of Figures

Figure 1 : A YellowJacket Module.....	1
Figure 2: A survey of robotics platforms.....	5
Figure 3: Diagram of a Remote Procedure Call.....	6
Figure 4: A Survey of Remote Procedure Call Generators.....	6
Figure 5: Comparing the Old Architecture with the New.....	11
Figure 6: Structure of a WASP Packet.....	11
Figure 7: Description of a Module.....	12
Figure 8: DataTypes.....	12
Figure 9: Original MDL specification.....	13
Figure 10: Workflow of the System.....	15
Figure 11: Lines Generated.....	16
Figure 12: Overview of Generated Methods.....	17

1. Introduction

An ongoing project started by a small Worcester-based company, Neuron Robotics¹, aims to simplify the robotics development process by creating a set of sensor, actuator and controller modules (called Yellow Jacket modules) that can communicate with each other via a simple custom protocol named WASP (WASP Sensor Actuator Protocol). Figure 1 shows a rendering of one of the modules. For each module the developers created, they also had to write a library to make it simple for the end programmer to work with the modules. The modules communicate with each other using remote procedure calls.



Figure 1 : A YellowJacket Module

Although the project's focus changed a few times, I knew I wanted to do something with the YellowJacket modules because I was interested in both robotics and development frameworks. This led me to investigate remote procedure call (RPC) generators specialized for a distributed embedded hardware environment such as a YellowJacket network. Much to my surprise, I discovered that there was not such a generator and so, for my Major Qualifying Project, I decided to create one.

I researched trends in robotics development, remote procedure calls generators and code generation techniques. Using what I learned, I developed a set of requirements for a useful generator. Then I implemented a framework to make it simple to expand the generator's datatypes and the languages. Using the framework, I generated Java code that crafts packets that work with YellowJacket modules. My examples are specifically for YellowJacket, but the Java code that generator produces could work with any networked embedded hardware device. Future work could also expand the generator to work in any language.

During the course of the project, the focus and requirements changed frequently. My methodology explains the reasoning behind design decisions and briefly discusses the coding practices and tools that I used. It also describes the changes that I needed to adapt to throughout

¹ <http://www.neuronrobotics.com>

the project and how I dealt with them. The results section focuses on the final version of the code. It describes how the final product works and how to use and expand the generator. It also touches on what the generator does *not* do and describes potential future work to expand and improve the generator.

2. Problem Statement

The purpose of this project was to investigate various methodologies for implementing a remote procedure call (RPC) generator specialized for a distributed embedded hardware environment, and then implement such a generator. Examples of such environments include robotic applications, weather sensing systems and automobiles. In these environments, the embedded hardware devices (modules) can be networked in any variety of ways from serial to Zigbee to TCP/IP. The generator should not dictate what the networking mechanism is because each environment has its own unique needs. Additionally, it is important that the description of the modules used to generate the RPC code must be simple to write so that it has a low learning curve.

This project has three deliverables:

- 1) A well specified text format to describe a module – This is simple to write and describes the full functionality of the module. From now on, this will be referred to as the MDL or Module Description Language.
- 2) A Code Generator Framework – This is the set of code that takes in an MDL file and produces working code. The framework can be extended to produce any language but the initial implementation is in Java.
- 3) An example of how to use the generated code - This uses the generated code to interact with YellowJacket modules over TCP/IP. It is meant as a proof of concept.

3. Background

To understand the problem I needed to approach it from three angles. First and foremost to outline the requirements for such software I needed to understand how robotic systems are developed and the aspects of development tools that are important to people who create robots. There are several robotics platforms that use distributed hardware which will be discussed later on. The next thing I researched was distributed computing and remote procedure calls. Distributed computing is a complex field and there are many challenges in creating a system that can allow disparate and varied computing devices to interact properly. Remote procedure calls are often created via some sort of code generation. The last topic I researched was code generation techniques. By looking at these all together I created a set of functional requirements for the software that I would later develop.

3.1 Understanding Robotics Development

A frequent approach to developing robotics system on a small to mid-size scale is to use a single central controller. For example, a common starter robot, or basebot, will have wheels to move it around and perhaps a few sensors so that it can react to the environment rather than move by dead reckoning where the robot would attempt to blindly move in the environment. The basebot's sensors send information to a central controller which uses the information to determine what the wheels should do. The same thing occurs in industrial robots. Each individual robot has one central computer, and those computers have no or minimal interaction.

Centrally controlled robots work well in the case of the basebot where the complexity (number of sensors and actuators) is low. They are also effective in situations where there is little uncertainty. Industrial robots can be complex, but because the tasks that they are given are fully articulated and the environment in which they work is controlled and predictable, the centrally controlled architecture works well.

This approach fails when there is uncertainty in the environment or in how exactly to accomplish the given task. There is a high demand for robots that can work in uncertain environments with applications in defense and security, elder care, automation of household tasks, customized manufacturing, and interactive entertainment. Distributed control allows the system to give a more immediate, localized response.

Additionally when the developers wish to add or change some of the sensors or actuators, they might need to update not only the physical system (such as wiring) but also the code in the centralized controller. For example, if a developer wanted to upgrade the drive train of the basebot to use holonomic wheels rather traditional wheels, the developer would need to make several changes to the existing code base. If the system were organized into subsystems that can understand higher level commands, then it would be an easier task. The developer could

continue to use some higher level command like “go over there” and let the subsystem determine the best method for accomplishing the task. Across the engineering fields this is a proven approach to deal with the issue of change.

For example, in software engineering, code is grouped into classes and methods and into libraries and procedures. The components work together through pre-determined interfaces. This makes it easy to change the inner functionality of a component without disrupting the entire system.

Some of the current robotics platforms adhere to the model of subsystems and distributed functionality while others stick to the single controller model. Any robotics platform will consist of a certain set of features. Each platform works with a custom set of sensors actuators and controllers that can communicate via some protocol. Examples of this include WASP, VEX, Willow Garage², CAN Bus (a control system in modern cars) and Lego Mindstorms. Figure 2 shows a survey of the platforms. While each platform boils down to basically the same thing, each has its own implementation

System Name	Primary use	Distributed or Centralized	Vendor
Yellow Jacket	Teaching, hobbyist, rapid prototyping, research	Distributed	Neuron Robotics
VEX	Teaching, hobbyist, rapid prototyping	Centralized	Innovation 1st
Stage/Player/Gazebo	rapid prototyping, research	Distributed	Willow Garage, originally USC
CAN Bus	Time-critical applications	Distributed (ring)	Many. Originally developed by the Society of Automotive Engineers
Lego Mindstorms	Teaching, hobbyist	Centralized	Lego

Figure 2: A survey of robotics platforms

While there are clear advantages to using a distributed architecture, it also introduces the challenge of managing a distributed network of sensors and actuators. Moving from a single powerful controller to several smaller ones means each processor in the network could have less processing power. The decrease in processing power with the increase in overall complexity of a distributed network requires careful thought and a thorough understanding of how distributed systems work. To understand the challenges I investigated issues and solutions to work with a distributed computing environment.

² <http://www.willowgarage.com/>

3.2 Distributed Computing

Distributed computing ranges from loosely coupled heterogeneous systems working in parallel to compute a complex problem³ to a simple request-response mechanism of a client and server. Despite the variations, either system would need to deal with communication over a network that might have unpredictable failures and latencies. In order to have the systems communicate efficiently a technology called Remote Procedure Calls or RPC was developed.

RPC is a mechanism for creating distributed applications. A client can make calls to a server that is independent of location and transport mechanism. In fact, the end programmer doesn't need to know or understand networking protocols or issues to consider when making a remote call. See Figure 3: Diagram of a Remote Procedure Call for more information. What is important is that from the user's point of view the call is made locally. In this figure, some process on Machine A will make a call to some process on Machine B. A is suspended while it waits for a response. Machine B handles the call and returns. Machine A now continues to execute.

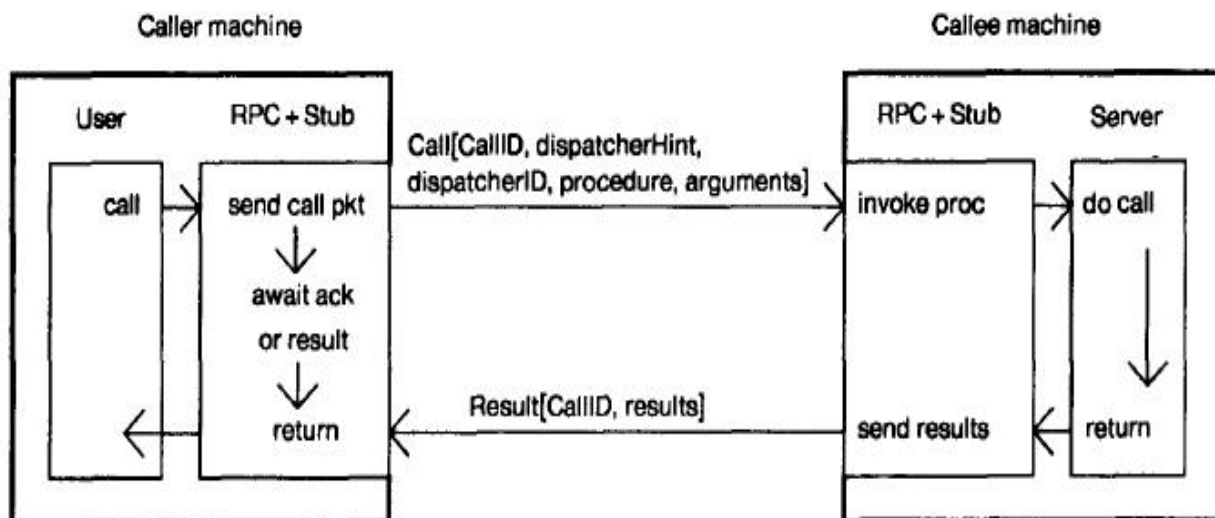


Figure 3: Diagram of a Remote Procedure Call

The concept of remote procedure calls has been around since at least 1967. Since then, various RCP systems have been developed. Figure 4 shows various RPC systems.

RPC System	language	transport layer
RPCgen	C	udp or tcp/ip
Java RMI	java	udp or tcp/ip
SOAP	any	http or smtp
XML-RPC	any	http

Figure 4: A Survey of Remote Procedure Call Generators

³ Folding at Home (<http://folding.stanford.edu/>) is one of my favorite distributed computing applications.

There has been some research done using XML-RPC to facilitate communication between robotic modules, however XML-RPC only works over the http protocol. In the case of robotics systems, the ideal RPC mechanism can work over any data transport protocol because some embedded systems cannot handle more complex protocols such as TCP/IP. Additionally because the processors that would handle the RPC might be less powerful in an embedded system, the less that is needed to process, the better. The WASP system made by Neuron Robotics simplifies the amount of work each processor needs to do by having every sensor, actuator and controller be compliant with a well defined interface. In a system of truly heterogeneous machines, each might internally represent data differently. There needs to be some sort of common representation between them to pass data and requests. The server or calling systems must translate the request into a common data representation and the receiving system must then translate what it receives into its own internal data representation. This translation is called marshalling. With WASP, the homogeneity simplifies the marshalling process thus lowering the processing load.

The other major issue with remote procedure calls is that remote calls might fail due to unpredictable network issues. The client making the call needs to deal with these failures, but it might not know whether the remote procedure even happened. If the call is idempotent (if for some reason it is called more than once, there will be no additional effects) then it is easier to deal with failure. This is because the call can simply be made again. A call that sets or requests information about the module's state could be idempotent. An example of an idempotent call would be if a servo received a command to go to position 27. Once the servo is at position 27, being asked to go there again would have no effect.

Components in an RPC system need to know how to communicate with each other. There is typically an Interface Description Language (IDL) to describe the inputs and output of a call. The IDL file is run through a utility to create the methods that do the actually work. Some popular tools to do this are RPCGen, Java RMI and XML-RPC. Figure 4 describes the various RPC generators. Each one works for a specific set of languages and transport layers. None of these can work for every transport layer. This shows a niche for a new RPC utility for a distributed network of modules which have interfaces defined at a higher level, independent of the data transport layer. The generator needs to use some description language as input and generate code to handle remote procedure calls. I will refer to the description language as MDL or Module Description Language.

3.3 Code Generation

The tools that translate an Interface Description Language to working code use some method of code generation. There are several benefits to code generation. Aside from the obvious time-saving benefits, there are the added benefits of consistency and the ability to easily quickly make changes across a code base. When pieces of code are similar but are hand-written and a change needs to be made, there are unnecessary delays because each piece needs to be updated individually.

Before you even start coding the generator, you need to know what kind of program you are going to generate. There are three distinct styles of programming that a generator might create, although the end result will realistically be a hybrid of the three. The three types are Object Oriented -driven style, code-driven style and a table-driven style. An Object Oriented driven style concentrates on creating the program architecture that meets the performance issues and design goals. The main disadvantage of this style is that it is harder to create and less efficient than code-driven style. Code-driven style is considered the most natural way to create a program. The code is generated and data is filled in where needed. The last style is table-driven. It employs a clean separation of data and code. All the data from the MDL document is stored in a table (or another data structure) and looked up when it is needed rather than having the data embedded in the code.

To generate a program a parser reads some sort of input. (For this project, it would be the MDL.) When a programmer needs to create a MDL file to generate code, he should have to specify only as much detail as needed. Having default values and optional fields makes this possible. A common generator input is XML. XML works well as an input file because it is simple to create and read and simple to parse. There are several tools to parse XML.

There are three methods to go from XML input to generated code. The first and simplest simply uses the Document Object Model (DOM) tree created by an XML parser to generate code. The next approach is to use the DOM parser but then analyzes the input to perform error checking and translate the DOM tree data structures into more useable data structures. The newly created custom data structures are then used to generate the code. The final approach uses the technology called SAX or Simple API for XML. SAX creates custom XML parsers. With SAX, data structures are created and processed during the parsing stage rather than afterward. When an XML document is large or streaming, this is very beneficial because it allows any unimportant or unknown portions of the document to be ignored. If the XML document is small, then using SAX is unnecessary.

Once the input is parsed, code can be generated. A generator typically ends up consisting of a series of print statements with flow control statements for repetition and conditional code. This can become almost unreadable with complicated strings full of escaped characters and intermingled with variables.

To improve readability, several templating systems have been developed. Examples of templating systems are Django⁴, Java Server Pages⁵ and Extensible Stylesheet Language Transformations.⁶ The idea of templating stems from ideas found in the model-view controller pattern of separating presentation, logic and data. A template describes the layout of the code that is going to be generated. It uses special logic statements for conditionals and repetition and

⁴ www.djangoproject.com

⁵ <http://java.sun.com/products/jsp/>

⁶ <http://www.w3.org/TR/xslt>

allows for variables to be filled in. The source template and the data (In this case the MDL file) are run through a template engine which produces the final document. The issue with templates is that they are language specific. Templates can work for more than one language but is a complex task to construct the template and the template engine to be generalized over several languages.

4.4 Finding a Niche

There is a need in the robotics field for a remote procedure call generator that allows compact values (e.g. no null terminator in strings, integers that are 3 bytes long, etc) to be sent. The packet generation and parsing would be lightweight to not draw on the processor too much. It would need to work with varied data transport mechanisms. A utility that generates the data portion of the packet and allows the programmer to choose how to send it would be useful. The utility would need a short, simple way to describe the structure of the data packet that is going to be sent. The input for the generator should be as barebones as possible. This removes some functionality, but it also keeps learning curves low. While the generator needs to be barebones, it should be designed to that it can be expanded as needs change. The full list of requirements is in Appendix A: Project Requirements.

4. Methodology

4.1 Tools

The basic good practices of software engineering were used throughout the project. For my integrated development environment, I selected Eclipse because of its robustness and because I have used it for several other large-scale projects. For code version control I used WPI's SourceForge and linked it to Eclipse with a plugin called Subversive. After completing a section of code, I tested it using j-unit test cases and made sure my code coverage at sufficient using a tool called EclEmma. I consulted with the developers at Neuron Robotics to make sure that what I was creating would be practical for a robotics developer's needs and modified my code based on their feedback.

4.2 In the Beginning

The project had a lot of false starts and dead-end paths during development. While these did not produce any useful results, they were useful as learning experiences. I started out the project thinking I could make, not a code generator, but an Eclipse plug in to display and control modules currently on the network. This was a good idea; however, in order to do that, I needed to be able to communicate with the modules. At this point in the project there was some code to send commands, but only for a few of the modules. After looking at the code for the modules, I noticed that there was a lot of similarity between code for modules and wondered if there might be some way to automate the production of the code for future modules. Without too much thought, I dived straight in, excited to make a code generator, but without doing any background research. Because of this impulsive decision, my original code base was (while functioning) a mess of functions thrown together without any OO principles or code generation techniques.

The code generation aspect of the project alone was daunting and the code was already getting out of hand, so I narrowed the scope so that I would do less, but do it better. From there, I abandoned my research into intuitive GUIs and Eclipse plug-ins and went full force in researching the proper techniques for code generation. After realizing just how bad the code was, I decided to do a master refactoring and do things the right way.

4.3 The Great Refactoring

My background research into code generation revealed good and bad things about what I had done so far. Nearly all of my code could be reused in the refactor, it would need to be restructured. Figure 5 shows a simplified package diagram of the code base before and after the refactor. With both sets of code, the program parses the MDL transform it and then generates code. However in the original code base the steps are intermingled across several classes and all

in one package. To refactor, I created a package for each of the steps and structured the code such that there were no cycles in package dependencies.

The second thing I did was make the code less rigid. In the original codebase, if a developer would want to make the code generate another language, he would need to make changes to the existing code as well as write his own new set of code. I took anything that could change based on which language being generated and put it into interfaces. For each language, the developer can implement the interfaces. For example to make the Java generator, I implemented the generation interface with a class called ModuleJavaGen. This refactor follows the Object Oriented Open/Close principle which states that the code should be open to extension but closed to modification. With each of the generation steps in their own package, each can be examined separately.

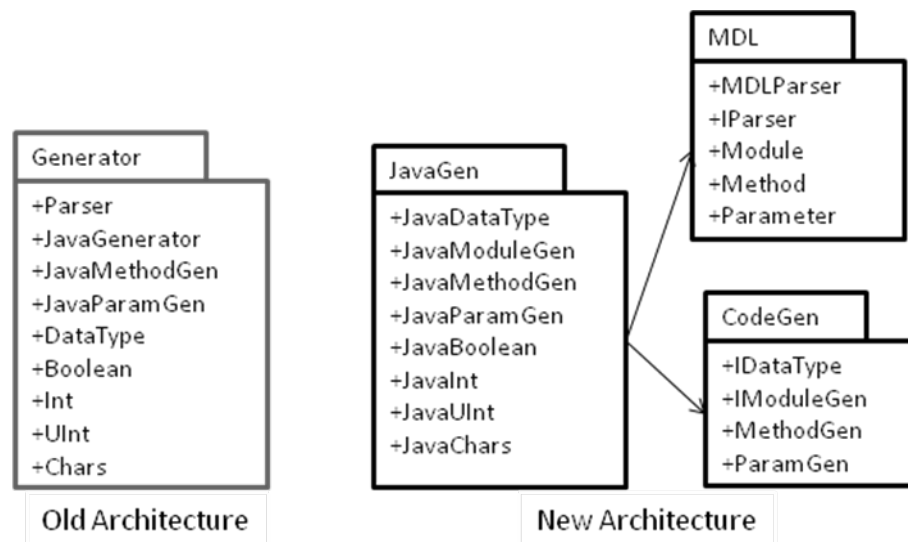


Figure 5: Comparing the Old Architecture with the New

4.4 The Module Description Language

To design software that generates packets, I needed to understand the types of packets that I would be sending. I am using WASP as my concrete example so I analyzed the similarities & differences between the Yellow Jacket modules and the types of WASP packets that they send. Figure 6 shows the basic structure of a WASP packet.



Figure 6: Structure of a WASP Packet

With the WASP system the byte length of a variable is specified and known beforehand. For example, a method to turn a servo a number of degrees is represented as a signed integer from -180 to +180 (actually -32768 to +32767, but the modulo of 180 is used). It only needs two bytes to represent that range, so this variable is a signed integer that is two bytes long. However,

another variable might be an unsigned integer that is one byte long. Every module knows the length to expect for a variable in an incoming and outgoing packet. This keeps the packets very abbreviated. It also means there was an extra aspect to consider when designing the generator – efficient and accurate conversion into and from abbreviated byte form.

In order to design the MDL I needed to develop terminology to describe a module. Figure 7 names each term, explains what it means and lists the other terms that it contains. This terminology was developed from documentation about the Yellow Jacket modules⁷. One point of interest is that Parameters are of several different types. Those types are discussed in more detail in Figure 8.

Term	Contains	Meaning
Module	Version Number, name, description, methods(s)	A discrete entity that exists in the distributed system. It can have any number of methods
Method	name, description, method number, response flag, data length, parameter(s)	A method describes some action that the module can take. Methods can have any number of parameters
Parameter	Name, data type , byte length, description	The data type and byte length are for encoding the parameter to send
Data Type	None	Describes how data is encoded. Figure 8 expands on the data types

Figure 7: Description of a Module

Data Type	Comments
Unsigned Integer	Plain binary
Signed Integer	2's complement
String	No null terminator needed because length is predetermined, However if the string is shorter than the length expected it is padded with null terminals.
Characters	Not a pre-defined data type. Characters are simply strings with a length of 1
Bytes	An array of bytes
Floating Point	Not supported. Would use IEEE 16, 32 or 64 bit encoding
Array	Not supported
Struct	Not supported

Figure 8: DataTypes

The terminology discussed in Figure 7 was developed with the goal of creating an MDL for generating RPC code. The terminology of a module mapped well into an XML document that will be the MDL. The parent element is a Module which can contain Method elements which in turn contain Parameter elements. Figure 9 lists each XML element and its valid

⁷ <http://dev.neuronrobotics.com>

attributes. Before proceeding any further, I got feedback from the developers at Neuron Robotics via informal questioning. My model of a module was accurate for WASP so I proceeded to write a parser to read it.

Element Type	Attribute Name	Data type	Default Value	Meaning
Module	Id	Integer	Required	Each type of module should have a unique ID
	Name	String	Module	The name of the module. Does not need to be unique
	Description	String	null	Describes the module. Used for comment generation.
Method	ID	Integer	Required	Value must be unique for the module
	Name	String	Method	Name of the method. Does not need to be unique
	Description	String	Null	Describes the method. Used for comment generation.
Parameter	Name	String	Required	Must be unique for method
	Length	Positive Integer	1	The byte length of the parameter.
	Type	Enumerated Type	UInt	One of: UInt, Int, String, Float
	Description	String	Null	Describes the parameter. Used for comment generation. Good for discussing valid values

Figure 9: Original MDL specification

4.5 The Parser

I created three classes used to model the module: Module, Method and Parameter. Once I had the structure of the MDL and the classes to model a module, I needed to make some way to populate a Module with data from the MDL. I used an XML parser that reads a module's MDL, converts it to DOM tree. Using the DOM tree, it can populate a Module object with the data from the MDL. I choose to use the DOM parser not SAX because the MDL is very small and using SAX would cause unneeded overhead.

4.6 The Generator

With an understanding of the packets, I hand wrote code that would produce the desired packets for the servo module. This was where the parameters' abbreviated byte form became a challenge. Java does not have unsigned integers. To deal with this I stored any unsigned integers as signed types that are one size larger than the original unsigned integer size because unsigned integers have one more bit to store value. I then wrote and extensively tested special functions to convert. At the byte level it is important to remember endianness. The JVM does not

specify endianness, however it is reasonable to assume that the data will be coming in as network byte order, which is big endian. I also had to write and test functions to convert to and from java data types to the WASP data types. I tested the whole system and the methods I coded could be used successfully to send commands to the servo.

With the Java servo code as a sample piece of working code that my generator could create, I was able to create a generator that consisted mostly of print statements with control statements for looping and conditional code.

4.7 Change Management

Change is inevitable in a project such as this one. As I was writing code to work with hardware that was still in development, the specification would change. It was in my best interest to make it simple to adapt to changes. The modularized system worked well. A good measure of a system is how much work needs to be done to accommodate one change. For example, the new specification called for a rather drastic change - the ability to add constant values. This isn't just adding a new data type, this fundamentally changes the entire model. I had to make changes in the MDL spec to have constants. The final version of the MDL specification is in Appendix B. Updating the parser to be able to read in the new constant was simple. It took one if-statement. Next I needed to update the Module model to be able to hold parameters that have a constant value. I had to update the constructor and add an instance variable to the Parameter class. I also added helper methods called 'boolean isConstant()' because I anticipated needing it in the generator.

The largest amount of effort was exerted dealing with the generator. Because there are now two types of parameters, those that can vary in value and those with a constant value, I added an if-statement every time I dealt with parameters to make sure they were handled in the proper way. In retrospect, it would have been smarter (from an Object Oriented design point of view) to make parameter an interface that both the Constant and a Variable implemented. This would have eliminated the need for the all of the if-statements and the change would have been quite manageable. Refactoring to this structure will be part of future work.

5 Results and Analysis

The final version of the code meets the original goal of creating an extensible tool that can take a simple description of a module and generate useful, well commented code based on the input. Figure 10 shows the flow of the system. An MDL is put through its parser. The parser creates a model of the module. The diagram also shows an alternative flow. If a programmer designed some other MDL (for example in JSON) and made a parser for it, he could plug it into the system so long as the parser implements the MDL parser interface. Once the model of the module is created, it can be put through any generator that implements the IModuleGen interface. Appendix H shows the steps a programmer would go through to do this.

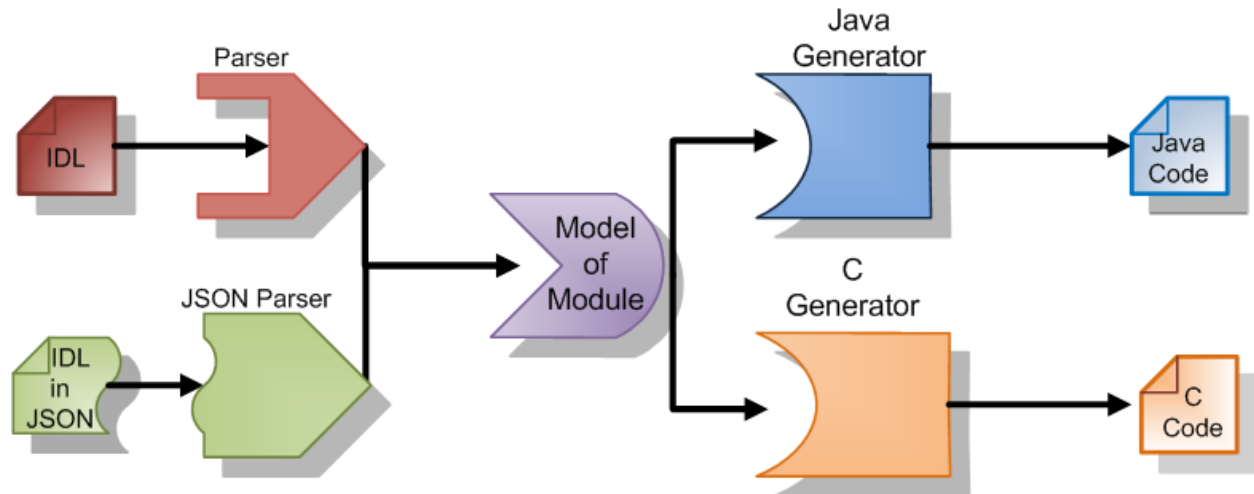


Figure 10: Workflow of the System

Changes to the generator are fairly easy to make. New parsers and new generators can be added without needing to modify the code base. Other changes would require a change to the codebase. However the amount of changes should be minimal. For example, to add a new data type (with a predefined length) the programmer would create a class that would implement the `IDataType` interface and update the `DataType` factory to be able to create an instance of the new `DataType`. This is useful if the programmer wants to create and parse custom-made data structures.

While it is easy to make changes to the generator itself, the generator is not well suited for creating dynamic code. Anything that requires computation at runtime cannot be expressed in the MDL. An example of something that might be computed at run time would be a checksum to ensure packet integrity across the network. While the inability to do such calculations this is limiting, it also makes things simpler for the programmer creating the MDL. Everything in the MDL is important and more pieces are used in several places.

If a change needs to be made to the packet structure, it only needs to be made in the MDL file, not in several places in the code. Additionally, some things that would normally be computed at run can be pre-computed and stored. For example, the length of a packet is a stored constant value, rather than calculated ever time it is needed. This lowers the computation load on the processor.

My sample MDL file of 11 lines produces 137 lines of commented java code. The sample MDL is in Appendix D and the generated java code is in Appendix E. Figure 11 shows how the various elements in MDL can affect the code and comments that are generated from it. This chart should be read by adding up each element. For example, a module with no methods will generate 3 lines of code. A method with no parameters will generate 22 ($0+1+2+4+13+2=22$) lines of code. So, a module with one method that has no parameters would

have 25 (3+22=25) lines of code. The chart indicates that this tool is especially useful when dealing with variables as they are the cause of the bulk of the code.

Element	Code generated	+ lines of code	+ lines of comments
Module	Class of the modules name in the Module Package	3	3
	Method String getName()	3	4
Method	Comment	0	3
	Components Array	1	2
	Constants Array	2	2
	Method byte[] methodName()	4	5
	Method bool isMethodNamePacket(byte[])	13	7
	Length of packet	2	1
Param	Line in components array	1	0
	Comment in byte[] methodName() for conversion	0	1
Param & Variable	Lines in byte[] methodName() for conversion	2	0
	Parser method that returns whatever datatype the parameter is and takes in a datapacket	7	8
Param & Variable & Int	Lines in parser method to parse an int	7	0
Param & Variable & Uint	Lines in parser method to parse a uint	7	0
Param & Variable & Bool	Lines in parser method to parse a bool	4	0
Param & Variable & Bytes	Lines in parser method to parse a byte array	1	0
Param & Variable & Chars	Lines in parser method to parse a char array	1	0

Figure 11: Lines Generated

Right now the generated code has four types of methods. Figure 12 describes the four types of methods. Note that things that are bolded are just placeholder names. These four types of methods should provide all of the functionality that a programmer would want to talk to modules. The methods can be wrapped in whatever protocol the end programmer wants to use. For my implementation I used TCP/IP but it could have just as easily been serial or Zigbee. Appendix F has code to demonstrate using the generated code with TCP/IP sockets.

```

/**
 *@return the name of the module
 */
String getName();

/**
 *@param {any variable parameters}
 *@return the datapacket to send
 */
byte[] methodName(...param list...);

/**
 *If all the constant values for this method match up
 *then, it is considered a packet for this method
 *@param the datapacket to check
 *@return true if the datapacket could be for the method
 */
boolean isMethodNamePacket(byte[] datapacket);

/**
 *parses a packets for a specific value
 *@param datapacket to parse
 *@return the value of of the parameter
 */
Datatype getMethodName_paramName(byte[] datapacket);

```

Figure 12: Overview of Generated Methods

5.1 Future Work

The objects that contain these methods can create and parse packets, but they don't hold any information on state. For future work, these methods could be wrapped in objects that also maintain connection information and state. Another place that this project was deigned to grow is with data types, which as I discussed above are easy to add. I can see a future need for dynamic computation for things like checksums and variable-length arrays. The project in its current form is not an appropriate tool for dynamic computation. However, using the ideas and principles from this project one could make a more dynamic packet generator.

Appendixes

Appendix A: Project Requirements

MDL can model packets that will be sent to sensors, actuators, and controllers.

The MDL describes aspects of the module including a human-readable name, a unique ID number, a comment on what it is and a list of methods.

Each procedure in the MDL can include a human readable name, a unique (for that module), a comment on exactly how the procedure works and a list of parameters.

Each parameter for each procedure for each module has a human readable name, a comment describing what it represents and any restrictions on values (e.g. must be between 10 and 27) and how it will be encoded in the packet (its datatype).

Parameter encoding is by type and byte length.

Parameter types include integers, unsigned integers, character arrays array, Booleans, byte arrays and floating point.

New datatypes should be easy to add.

Parameters can be either constant or variable.

Constant parameters also have an associated value.

One MDL implementation will be xml.

System must support extension for many MDLs. (for example json & xml)

The system must validate the MDL.

User can specify MDL file or directory of MDL files to generate.

Using what is read in from the MDL the system generates a set of methods (in a specific programming language) for the end user to use.

The generator should be able to be extended to generate code in any language.

For each method, the generator should produce a creation method that takes in any variable values for the packets and crafts a packet.

For each method, the generator should produce a type-checking method that reads in a data packet and returns true if the data packet could be for that method.

For each variable parameter in each method, the generator should produce a parsing method that read in a data packet and returns the value of the parameter in the packet.

One implementation of the generator code should produce Java.

Appendix B : Table Describing the Final MDL

Element Type	Attribute Name	Data type	Default Value	Meaning
Module	Name	String	Module	The name of the module. Must be Unique.
	Description	String	null	Describes the module. Used for comment generation.
Method	Name	String	Method	Name of the method. Must be unique for the method.
	Description	String	Null	Describes the method. Used for comment generation.
Parameter	Name	String	Required	Must be unique for method
	Length	Positive Integer	1	The byte length of the parameter.
	Type	Enumerated Type	UInt	One of: UInt, int,chars, bytes
	Description	String	Null	Describes the parameter. Used for comment generation. Good for discussing valid values
	Value	Anything	Null	If the attribute is not mentioned, then the parameter is a variable. The value of this attribute should correspond to the parameter's type.

Appendix C : XML Schema Document

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="module">
  <xs:complexType>
    <xs:sequence>
      <xs:attribute name="name" type="xs:string" default="Module"/>
      <xs:attribute name="comment" type="xs:string" default=""/>
      <xs:element name="method">
        <xs:complexType>
          <xs:sequence>
            <xs:attribute name="name" type="xs:string" default="Method"/>
            <xs:attribute name="comment" type="xs:string" default=""/>
            <xs:element name="method">
              <xs:complexType>
                <xs:sequence>
                  <xs:attribute name="name" type="xs:string" use="required"/>
                  <xs:attribute name="length" type="xs:integer" default="1"/>
                  <xs:attribute name="comment" type="xs:string" default=""/>
                  <xs:attribute name="type" type="xs:string" default="uint"/>
                  <xs:attribute name="value" type="xs:string"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Appendix D: A Sample MDL

```
<module name="Servo" comment="YellowJacket Servo uses WASP protocol">
  <method name="putServo" comment="moves the servo to a certain position">
    <param name="versionNum" value="2" type="uint" len="1"/>
    <param name="MAC" type="bytes" len="6"/>
    <param name="methNum" value="32" type="uint" len="1"/>
    <param name="respFlag" value="0" type="uint" len="1" comment="Value is 0 if sending."/>
    <param name="dataLen" value="6" type="uint" len="1"/>
    <param name="methName" value="srvo" type="chars" len="4"/>
    <param name="servoNum" type="uint" len="1"/>
    <param name="pos" type="uint" len="1" comment="Servo position. A value between 1 and 255"/>
    <param name="checksum" value="50" type="uint" len="1"/>
  </method>
</module>
```

Appendix E: Code Generated From a Sample MDL

```
package modules;

/**
 *YellowJacket Servo uses WASP protocol
 */
public class Servo{
    /**
     * this could be useful when working in the next layer in the packet
     * @return the module's name
     */
    public String getModuleName(){
        return "Servo";
    }

    /*
     * The Follow Set of Constants are for the putServo method.
     */
    //Represents reach component of the packet
    private static byte[][] putServo_components = {
        {2, }, //versionNum:
        new byte[6], //MAC:
        {32, }, //methNum:
        {0, }, //respFlag: Value is 0 if sending.
        {6, }, //dataLen:
        {'s', 'r', 'v', 'o', }, //methName:
        new byte[1], //servoNum:
        new byte[1], //pos: Servo position. A value between 1 and 255
        {50, }, //checkSum:
    };
    //indicates which of the values are constant
    //and which ones will be determined at run time
    private static boolean[] putServo_const = {
        true, false, true, true, true, true, false, false, true,
    };
    //the total length of the packet
    //note that this could be calculated at run time
    private static int putServo_len = 17;

    /**
     *creates a packet
     *moves the servo to a certain position
     *@param MAC
     *@param servoNum
     *@param pos Servo position. A value between 1 and 255
     *@return array of bytes representing the data packet
     */
    public static byte[] putServo(byte[] MAC,int servoNum,int pos) throws
    Exception{
        int i_MAC = 1;
        putServo_components[i_MAC]= MAC;
        int i_servoNum = 6;
        putServo_components[i_servoNum][0] = (byte) servoNum;
        int i_pos = 7;
```

```

        putServo_components[i_pos][0] = (byte) pos;
        byte[] result = Utils.merge(putServo_components, putServo_len);
        return result;
    }

    /**
     * If all the constant values for a putServo match then, its this type
of packet
     * If all the constant values for a putServo match then, its this type
of packet
     * @param packet
     * @return true if all constant values match
     */
    public static boolean isputServoPacket(byte[] packet){
        //If these are not the same length, then they are not the same
packet.
        boolean matches = (packet.length ==putServo_len);
        int bytesSoFar = 0;
        for(int component = 0; component < putServo_components.length &&
matches; component++){
            for(int bytesIn = 0; bytesIn < putServo_components[component]
[bytesIn]; bytesIn++){
                if(putServo_const[component]){ //If it is a constant
value...
                    //make sure that it matches.
                    matches =
packet[bytesSoFar]==putServo_components[component][bytesIn];
                }
                bytesSoFar++;
            }
        }
        return matches;
    }
}
/**
 *gets the value of putServo
 *@param dataPacket : the packet to parse
 *should be formatted like a putServopacket
 *@Exception if it itnot the proper type of packet
 *@return putServo MAC
 */
public static byte[] getputServo_MAC(byte[] datapacket)throws Exception{
    if(!Servo.isputServoPacket(datapacket)){ throw new Exception();}
    int depthIntoPacket = 1;
    byte[] component = new byte[6];
    for(int i=0;i < 6; i++){
        component[i] = datapacket[i+ depthIntoPacket];
    }
    return component;
}
}
/**
 *gets the value of putServo
 *@param dataPacket : the packet to parse
 *should be formatted like a putServopacket
 *@Exception if it itnot the proper type of packet
 *@return putServo servoNum
 */

```

```

    public static int getputServo_servoNum(byte[] datapacket)throws
Exception{
    if(!Servo.isputServoPacket(datapacket)){ throw new Exception();}
    int depthIntoPacket = 14;
    byte[] component = new byte[1];
    for(int i=0;i < 1; i++){
        component[i] = datapacket[i+ depthIntoPacket];
    }
    int value =0;
    for(int i = 0; i < component.length; i++){
        int dig= component[i];
        value = (int) (value + dig*(Math.pow(2, i)));
    }
    return value;
}
/**
 *gets the value of putServo
 *@param dataPacket : the packet to parse
 *should be formatted like a putServopacket
 *@Exception if it itnot the proper type of packet
 *@return putServo:Servo position. A value between 1 and 255
 */
public static int getputServo_pos(byte[] datapacket)throws Exception{
    if(!Servo.isputServoPacket(datapacket)){ throw new Exception();}
    int depthIntoPacket = 15;
    byte[] component = new byte[1];
    for(int i=0;i < 1; i++){
        component[i] = datapacket[i+ depthIntoPacket];
    }
    int value =0;
    for(int i = 0; i < component.length; i++){
        int dig= component[i];
        value = (int) (value + dig*(Math.pow(2, i)));
    }
    return value;
}
}
}

```

Appendix F: Using Generated Code to Communicate with a Module

```
//Sample code adapted from http://zerioh.tripod.com/ressources/sockets.html
import java.io.*;
import java.net.*;
import java.util.Scanner;
public class Requester{
    String host = "130.215.23.57"; //IP of module
    byte[] MAC = {0, 4,(byte)163, 0,0,3}; //mac of module
    Socket requestSocket;
    PrintStream out;
    Scanner in;
    int servoPos;
    void run(){
        try{
            //1. creating a socket to connect to the server
            requestSocket = new Socket(host, 1865); //port 1865 :)
            System.out.println("Connected");
            //2. get Input and Output streams
            out = new PrintStream(requestSocket.getOutputStream()),
true);
            in = new Scanner(System.in);

            //3: Communicating with the server
            boolean done = false; //not done yet
            while(!done){
                try{
                    servoPos = in.nextInt(); //read a servo position
                    done = (servoPos < 0); //less than zero
                    indicated we are done

                    if (!done){
                        //Call the putServo method in servo
                        out.write(Servo.putServo(MAC, 11,
servoPos));

                        //Wait for a response back
                        byte[] back = new byte[17];
                        requestSocket.getInputStream().read(back);

                        System.out.println(Utils.printArray(back));; //print it
                    }

                    }
                catch(ClassNotFoundException classNot){
                    System.err.println("data received in unknown
format");
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                    done = true;
                } catch (Exception e) {
                    e.printStackTrace();
                    done=true;
                }
            }
        }
    }
}
```

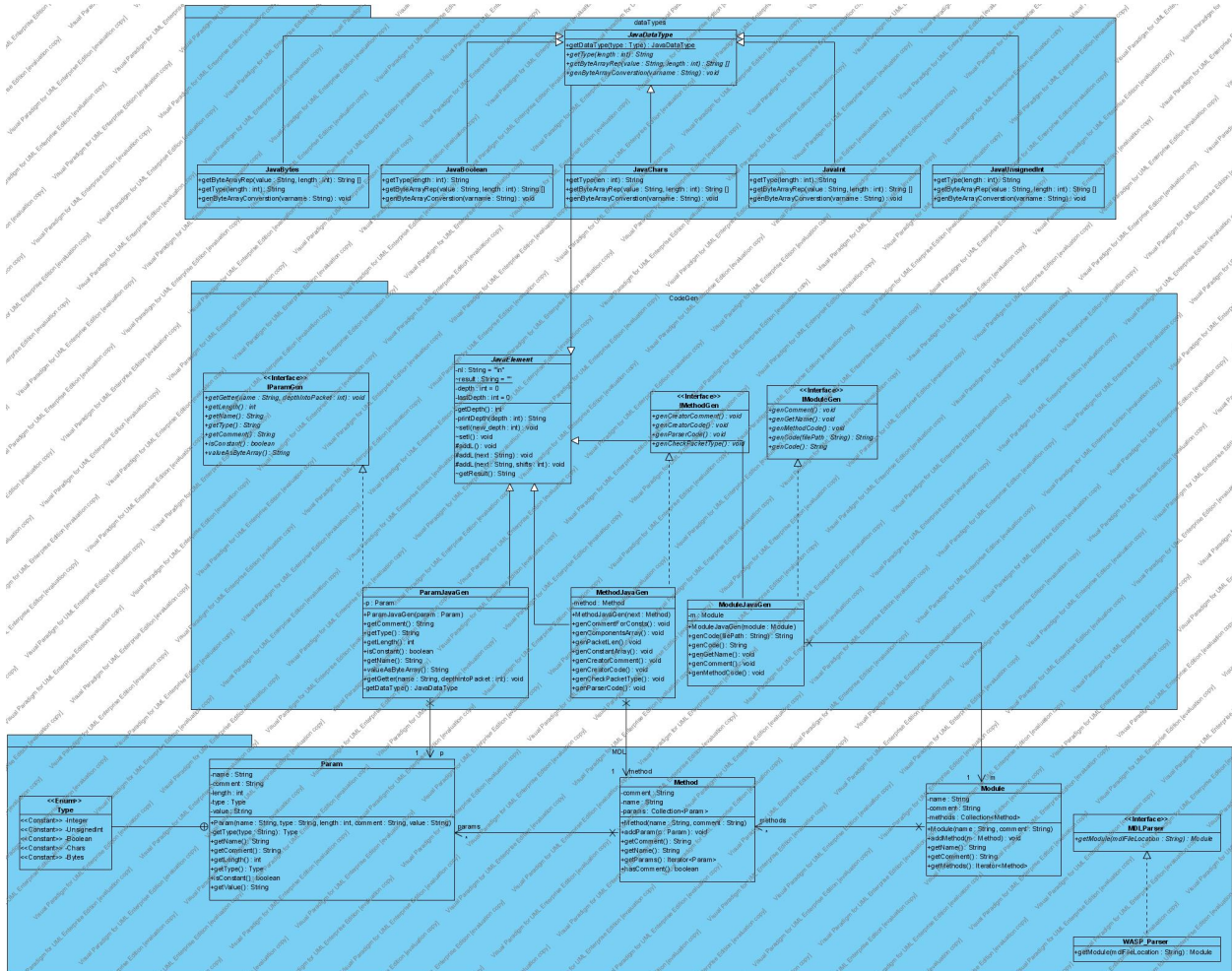
```

        }
    }
}
catch(UnknownHostException unknownHost){
    System.err.println("You are trying to connect to an unknown
host!");
}
catch(IOException ioException){
    ioException.printStackTrace();
}
finally{
    //4: Closing connection
    try{
        out.flush();
        in.close();
        out.close();
        requestSocket.close();
    }
    catch(IOException ioException){
        ioException.printStackTrace();
    }
}
}

public static void main(String args[])
{
    Requester client = new Requester();
    client.run();
}
}

```


Appendix G: Full UML Diagram of the Generator



Appendix H: A Tutorial in using the Generator

- 1) Import the project into Eclipse. There will be three packages CodeGen, JavaCodeGen and MDL.
- 2) Create a new class with a main method that will run the generator.
- 3) Create an instance of MDLParser.
- 4) MDLParser has a method called Module getModule(String filename). Run this method giving it the file path of the MDL file. It will generate an instance of a Module.
- 5) Now create an instance of IModuleGen and pass it the module.

```
/**
 * Reads a file called Servo.mdl and
 * Generates a file called Servo.java
 * @author Galia
 */
public class MDL_to_Module_to_Code_Driver {
    public static void main(String[] args) {
        //this line is WASP specific.
        MDLParser parser= new WASP_Parser();
        //is is just the model
        Module module = parser.getModule("Servo.mdl");
        //this line is JAVA specific
        IModuleGen genny= new ModuleJavaGen(module);
        genny.genCode("Servo.java");
    }
}
```

References

- Attiya, H., & Welch, J. (2004). *Distributed computing : Fundamentals, simulations, and advanced topics* (2nd ed.). Hoboken, NJ: Wiley. Retrieved from <http://www.loc.gov/catdir/bios/wiley046/2004040728.html>; <http://www.loc.gov/catdir/description/wiley041/2004040728.html>; <http://www.loc.gov/catdir/toc/wiley041/2004040728.html>
- Barr and Paatz. (2007). *Describing anatomy of an industrial robot.*, 2009, from <http://www.manufacturingtalk.com/news/baj/baj116.html>
- Cleveland, C. J. (2001). Program generators with XML and java.
- Czarnecki, K., & Eisenecker, U. (2000). *Generative programming : Methods, tools, and applications*. Boston: Addison Wesley.
- Dissanaikie, S., Wijkman, P., & Wijkman, M. (2004). Utilizing XML-RPC or SOAP on an embedded system. Paper presented at the , 3
- Kato, H. In Mitsubishi Denki Kabushiki Kaisha (Ed.), *Industrial robot control system G05B* 1942.
- Kerievsky, J. (2005). *Refactoring to patterns*. Boston: Addison-Wesley.
- Lawrenz, W. (1997). *CAN system engineering : From theory to practical applications*. New York: Springer. Retrieved from <http://www.loc.gov/catdir/enhancements/fy0815/96039961-d.html>; <http://www.loc.gov/catdir/enhancements/fy0815/96039961-t.html>
- Perdue, D. J. (2008). *The unofficial LEGO mindstorms NXT inventor's guide*. San Francisco: No Starch Press. Retrieved from <http://www.loc.gov/catdir/enhancements/fy0913/2007033731-b.html>
- Pollice, G. *Component generation as A software reuse technique illustrated with A C scanner generator*. Unpublished University of Massachusetts Lowell,
- Sarkar, S., & Cleaveland, C. Code generation using XML based document transformation. *Published on the Server Side–Your J2EE Community*,
- Unknown. (2007). *Undergraduate program: Mission statement.*, 2009, from <http://www.wpi.edu/Academics/Majors/RBE/academics.html>
- VEX Robotics. (2008). *QUICK START GUIDE for building the protobot*. Retrieved July 25, 2009, from <http://www.vexrobotics.com/docs/Vex-Quickstart-Guide-10072008.pdf>