The Design and Implementation of Looping Constructs for Robots

Project Team:

Rachel Hahn rhahn@wpi.edu
Ian Johnson icjohnson@wpi.edu
Benjamin Mattiuzzi bmmattiuzzi@wpi.edu

gr-roboticlang@wpi.edu

Project Advisor:
Professor Michael A. Gennert

Abstract

Our goal is to make the job of programming WPI's Atlas robot simpler for both the experienced programmer and the new programmer. After meeting with the stakeholders of the Atlas robot, we identified a few different options in order to achieve this goal. We used system engineering tools to decide that creating a looping function was our best option. We then used user stories, use cases, and test cases to help flush out what our function does and how it operates. We finally implemented it and tested it with a sample of the Atlas code to ensure we achieved our initial goal.

Acknowledgements

We would like to thank our project advisor who helped guide us through our project:

Professor Michael A. Gennert	Project Advisor and Project Sponsor

We would also like to thank the following people who met with us in order to help us understand the Atlas code:

Chris Atkeson	Professor at Carnegie Mellon University
Benzun Pious Wisely Babu	Ph.D. student in Robotics Engineering at WPI
Vinayak Jagtap	Ph.D. student in Robotics Engineering at WPI

Executive Summary

We started off by gathering information about the history of WPI's Atlas robot and the difficulties that users have when writing code for it. We interviewed a number of people who had written code for Atlas to find out where they believed the code base's weaknesses lay. We heard a lot of different, sometimes contradictory, ideas about the code base, but the one thing that was universal was that the code was very hard to understand. It would take teams weeks or a few months to fully understand the code well enough to start to make changes and add functionality. With this in mind, we decided to create a library that could lay on top of the Atlas code and abstract some of the less intelligible code into a more readable format, focusing on one of the most prominent design patterns in robotics, loops.

Often, the high-level loop used in robotic programming will have a lot of complex logic embedded in. For humans, it is natural to think that if we fail to pick up a hammer, we cannot then use that hammer to pound in a nail, but that thought process often proves very hard to implement. With this in mind, we created a suite of user stories and use cases to define the scope of our project, and then a set of test cases to measure our achievement. Our goal is to have a single looping function that would logically interpret a set of nested functions and loop through them, jumping back or forward as necessary.

We decided to start from the most complex form and then simplify, so first we created a version of the function that had all of the functionality we desired, but it was extremely verbose and put a lot of the onus on the programmer. We then worked to iteratively improve the function, giving it the ability to fill in fields that were left blank, and giving the user different options of how to input data. When attempting to minimize the number of lines of code needed, we ended up hitting some roadblocks in the nature of C++, causing our final product to not be as simplified as we desired, while still being leagues simpler than writing the code without using our function.

We then ran some timing tests on our code to make sure that our added functionality did not slow down code execution. We found that in cases of simple calculation, our code ran around 8 milliseconds slower than simply calling functions, and we concluded that this is negligible, as a robot will often take more time than that to execute its movements.

Authorship

Rachel took a lead role in writing the Abstract, Methodology chapter, the Design and Intent section in the Design and Examples chapter, and the User Guide. She also took part in editing all parts of the paper after they were written. Rachel took lead in ensuring all interviews and meetings were well documented while helping with the code.

Ian took a lead role in writing the Introduction chapter, Executive Summary, the Implementation and Testing sections of the Analysis and Testing Chapter, and the User Guide. He also took part in editing all parts of the paper after they were written. Ian took the role as the main programmer while helping with interviewing.

Benjamin took a lead role in writing the Background chapter, the Interview Feedback section in the Design and Examples chapter, the Deployment section in the analysis and testing chapter, and the Conclusions chapter. He also took part in editing all parts of the paper after they were written. Benjamin was the main point of contact for all interviewees while helping with deployment of the code.

Table of Contents

Table of Acronyms

CDR	Critical Design Review
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency
DSL	Domain Specific Language
GPL	General Purpose Language
GPU	Graphics Processing Unit
IDEA	Intelligent Distributed Execution Architecture
NASA	National Aeronautics and Space Administration
PDR	Preliminary Design Review
ROS	Robotic Operating System
RPL	Reactive Plan Language
TDL	Task Description Language
WARNER	WPI's Atlas Robot for Nonconventional Emergency Response
WPI	Worcester Polytechnic Institute

1.0 Introduction

Currently it is the standard to use low level languages like C and C++ to interact with robotic systems. Low level languages give programmers fine grain control of low level systems such as memory management and direct access to hardware devices. Worcester Polytechnic Institute's (WPI) Atlas robot is currently programmed using C, however, there are some issues.

Programming in low level languages can obscure the meaning of code and introduce difficult to find bugs in software. In addition, many programming languages are a compromise between ideals. A language that might be perfect for one use of the Atlas robot might be useless for another. Many people have attempted to create robotic programming languages in the past. However, because of the extreme variety in the design and purpose of robots, there is no consensus on what a robotic programming language should do; some requirements want code that is error-free, some want code that is adjustable on the fly, some want code that self-corrects. A programming language's requirements will have to be determined by the stakeholders of the robot.

The goal of our project is to design and implement a programming language for the Atlas robot. The language will have the intuitive structure and approachability of a high level programming language without abstracting all of the lower level functionalities that a low level language affords the programmer. A programmer using the final system should be able to:

- Develop systems that are robust and simple to debug;
- Develop systems that are easily maintainable and extensible;
- Interact directly with the hardware in the system in an straightforward and non-error prone fashion; and
- Understand the syntax easily to ensure it does not prove to be a major barrier to entry.

We will achieve this goal by conducting in-depth interviews with the stakeholders of the Atlas robot, develop a prototype of the language, gather feedback, and implement a flushed-out version of the language that should run on both the simulation of the Atlas robot as well as the robot itself. These objectives will allow us to better see what part of the robot should be improved upon as well as allow the programmers to better interact with it.

2.0 Background

This section describes the information needed to understand the goals and objectives of our project. We will discuss an introduction to general purpose and domain specific languages, the key differences between them, and define what our platform is for this project.

2.1 General Purpose Languages

In the early days of computing, programs were written in low level languages that were uncompiled and ran directly on bare hardware, such as assembly or machine code. The syntax of these low level languages was difficult to read, difficult to write, and error prone. As computing technology evolved, so did the languages we used to program them. In the late 1950's, higher level languages, such as Lisp, Cobol, and Fortran, were developed to allow programmers to more easily and effectively write code while also making them easier to learn. C was released in 1972 followed by its successor C++, which was released in 1983. These two languages were particularly popular because they both allowed low level functions, for example direct memory management, while still being easy to read and write. C and C++ had the added benefit of being less error prone and easier to debug than lower level languages. Later languages, such as Python, Perl, C#, and Java, all draw certain design choices from C and C++, such as their object oriented paradigms, and even share certain syntax elements. All of these languages are examples of general purpose languages (GPLs). They all contain a wide variety of tools, structures, and libraries for building programs to perform many tasks. Small code snippets are shown below to highlight syntactic differences between C, Java, and Lisp when printing the message "`Hello, World!`" to the console.

```
C : cout << "Hello, World!\n";
Java : System.out.println("Hello, World!\n");
Lisp : (print "Hello, World!\n")
```

In the above example, Lisp uses open and closed parentheses to enclose its statements whereas C and Java statements do not. In Lisp, printing is possible with the simple keyword '`print`' while in Java, it is necessary to specify where the "`println`" keyword comes from, specifically the package "`System.out.`" In C, the printing is performed by directly putting the output text onto the terminal line with the built in "`cout`" reference and the "`<<`" symbol. In both Java and C, statements must be concluded with a semicolon whereas doing this in Lisp would cause a syntax error preventing the code from functioning. Each of these three code snippets perform the same task, so semantically they are nearly identical, but it is easy to see their syntax varies significantly.

There are many programming languages used in academia and industry today, and each one has strengths and weaknesses that make it more or less appropriate for certain tasks. The lower level languages, for example assembly and machine code, allow the programmer to write code that is run directly on the hardware. These types of languages are commonly used when size and efficiency are a top priority since programs written in them are small and fast to run. Slightly higher level languages, for example C and C++, are commonly referred to as system programming languages since they allow the programmer to interact directly with the hardware in a computer with higher level functions like loops and conditionals. Languages similar to these are commonly used to program operating systems, low level computer drivers, and libraries for higher level languages. High level languages, for example Java and C#, start to add features that aid in building larger suites of software. These languages may include object oriented structures such as classes, methods, and interfaces to help organize code, but they typically do not include the direct hardware manipulation that lower level languages have. The highest level languages, otherwise known as scripting languages, such as Perl and Python, are used for everything from embedded projects running on Raspberry Pi to small personal servers and even large pieces of corporate software. These languages are unfit to write low level code that would be necessary to create an operating system, but they are well suited for creating web applications and smart home or internet of things devices. Every programming language has strengths and weaknesses and it is important to consider these when choosing a language for a project.

2.2 Domain Specific Languages

The powerful feature of a GPL is that it is general enough to encode many programs. One can create a website, a piece of software, a database, and an operating system using nothing but C. These tasks require very different parts of the language, and in each case there are likely to be large swaths of unused libraries and functionalities of the language that still have to included in the final product. This is where designing a domain specific language (DSL) can be a reasonable option. A DSL is a small language, either designed on top of an existing one or built from the ground up, that is used for a specific task. Syntactically, a DSL can take almost any form, since it is up to the designers to create a language that best fits the needs of the project. It is not possible to express as many different programs using a DSL, but the programs you can express are likely to be more efficient and smaller overall.

There are many examples of DSLs. The popular children's toy LEGO Mindstorm is a good example. LEGO Mindstorm is a simple, block based DSL that allows children and young adults to program a robot by dragging and dropping color coded blocks into a work environment to build up programs. The blocks each have a specific function, such as operating a motor, checking the value on a sensor, or looping a command or sequence of commands a certain number of times or until a sensor reads a specific value.

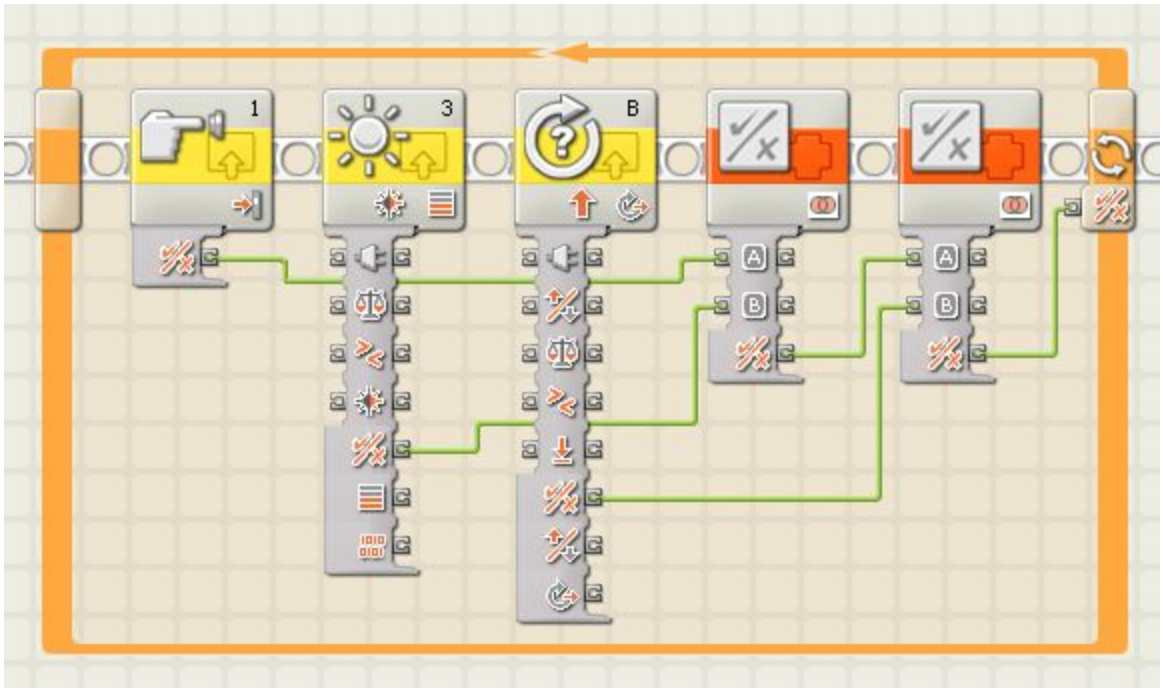

Fig 1. a simple example of the graphic, block based DSL used to program LEGO Mindstorm Robots.

This kind of graphic DSL is easy to learn, but highly restrictive in its uses. Graphic DSLs tend to be very small languages, offering only basic looping and conditional commands, and often only allowing functional programming. These features are sometimes advantageous, but sometimes can prevent a language from reaching widespread deployment. Another language that is compatible with the LEGO Mindstorm product line is Lejos, a Java derivative that is specifically designed to run on the LEGO Mindstorm systems. Lejos has more features than the default Mindstorm language. It includes most of the Java.lang, Java.util, and Java.io packages and supports error handling, many data types, and even synchronization. What makes this derivative of the Java language a DSL, is that it can only be deployed on the Mindstorm System.

Another interesting user of DSLs is the National Aeronautics and Space Administration (NASA), which uses DSLs for many different projects. Some notable examples include Intelligent Distributed Execution Architecture (IDEA), reactive plan language (RPL), and task description language (TDL). IDEA is a model based planning and execution system that was developed to help eradicate small bugs in systems at the planning, execution and diagnostic layers caused by semantic differences in the model (Verma et al., 2005, p. 3). RPL is a fairly fully featured language with a Lisp like structure that adds certain "policies" that are held true during the extent of the execution of a program (Verma et. al 6). TDL is an extension of C++ that adds support and commands for high level control of robots. It is translated with the Java compiler into pure C++ and then compiled into an executable using any C or C++ compiler (Verma et al., 2005, p. 7). TDL focuses on "high level control constructs like task

decomposition, task synchronization and coordination, execution handling and execution monitoring, and distributed synchronization of multiple agents" (Verma et al., 2005, p. 7) to create a powerful and robust language for the specific purpose of programming and controlling robots.

2.3 Comparison of GPLs and DSLs

Given the breadth of DSLs and GPLs in existence, it is important to look at some of the key differences to understand the advantages and drawbacks to each option. A DSL tends to be smaller and more efficient since many extraneous features are removed in the design process. GPLs tend to be more widely used, and remain relevant for longer in the general computing community. Some GPLs like Fortran and Cobol have been in use since the late 1950's and are still in industry today, whereas no DSL has ever reached that level of widespread deployment. A major drawback to using a DSL is that it must be designed and implemented before work on the final product can begin. Creating a DSL takes careful domain analysis and planning, implementation, and testing. These tasks can create a huge overhead for a project before the work on the actual product can begin. On the other hand though, since a DSL has to be created specifically for a project, the designers can choose to include certain critical features directly into the language. The ability to include functionality directly into a language reduces extra code, especially repetitive boilerplate type code, so that the final project can be more efficient and smaller overall.

2.4 Our Platform

Atlas is a line of humanoid robots developed by Boston Dynamics. WPI was loaned an Atlas, named WARNER (WPI's Atlas Robot for Nonconventional Emergency Response), to allow students and faculty to compete in the Defense Advanced Research Projects Agency (DARPA) Robotics Challenge (Prato, 2016). At WPI, WARNER is primarily programmed using C and C++. Atlas robots are a distributed systems of five main computers, four of which are programmable by the faculty and staff, one of which remains a 'black box,' programmed by Boston Dynamics when they released the robot to WPI. Much of the work on Atlas begins in simulators, where developers are able to test changes to the programming in an environment where the robot itself cannot get hurt. Once the simulations are tested and correct, the program can be loaded onto Atlas and tested and further refined. Developing for the Atlas robot can be very difficult, as any distributed system presents challenges to programming, and it can take a new programmer days or weeks to learn how to create even a simple program to perform a task like raising the robots arm. Using C to program Atlas presents advantages and disadvantages. Since it is a lower level language than Python or Java, it allows the programmer to directly

access hardware on the robot but it also makes communicating between the different systems on the robot more difficult.

3.0 Methodology

This section describes the steps that we took to achieve our goal. We used systems engineering concepts such as a preliminary design review, throughout the process of our project to ensure we were well prepared for implementation.

3.1 Objectives

The goal of our project is to create a programing language for the humanoid robot series, Atlas, that will work on top of the existing C/C++ code. To achieve this goal, we had set the following objectives:

1. Identify stakeholders' needs through interviews;
2. Design a function through user stories, use cases, and test cases;
3. Implement and test our language.

3.2 Interviews

To help us better understand the scope of our project, we conducted three interviews with graduate students, as well as the professors who have previously worked with the Atlas robot. These interviews helped us understand the nature of programming WPI's Atlas robot, while helping us determine features that should be implemented in the language the we created. After completing these interviews, we refined our project goal and used the information we gathered to help us identify the stakeholder needs and requirements.

3.3 Preliminary Design Review

After establishing the background of knowledge relevant to our project, we created a preliminary design review (PDR) to help us flush out our deliverables; see Appendix A for the full review. A PDR is an assessment that helped us narrow down from multiple design concepts to one that we developed further. The assessments looked at which concepts met the maximum amount of requirements without going over on cost and time. We used this to help measure how realistic our ideas were.

3.4 Develop User Stories

Once we chose the design that we wanted to focus on, we created user stories based upon the needs and requirements that we previously established (see Appendix B). A user story is a short statement that describes what a user wants while using the system. We used these

statements as a base to help understand what actions a user might take on our function. This helped us ensure that we are creating a function based on user requirements without going too much into detail.

3.5 Develop Use Cases

Following the user stories that we developed, we created use cases which helped us flush out how a user would interact with our function (see Appendix B). A use case is a set of information that helps to define what should happen between an actor (or user) and a system when achieving a goal. Each use case contains the primary actor (those who will interact with our language), the arguments for the action, the triggers that start the action, the basic flow of the action, and the end condition that tells the actor when the action is finished. These use cases helped us understand how our system should operate while taking the user's actions into consideration.

3.6 Develop Test Cases

After we created a comprehensive set of use cases, we developed a set of test cases which allowed us to measure if our goals are being met (see Appendix B). A test case is a set of information that helps us to understand if the requirements are being met as well as ensuring that a system is running correctly. Our test cases included a summary of an action, the procedure of what the user did, the data that the user gave the function, and the expected result of that action being run. These test cases were be run as we were developing the language and helped us avoid writing useless code. Our test cases covered all the use cases we had previously defined. We also created regression tests that checked if we removed any of the pre-existing functionality of the language.

3.7 Development and Testing of the Language

Our next step was to fully develop the language. We completed development of our function by using the language C++ and by using Git to help us collaborate. As we developed, we were continuously testing to ensure we were achieving our initial goals. We also added new tests and revised existing tests depending on the issues and new concepts that we discovered throughout the process. To help a user understand how to use our function, we created a supplemental user guide that describes how to implement and use our function with their code base. Once we created a working version of the function, we attempted to deployed it to the simulation of the Atlas robot.

4.0 Design and Examples

In this section we discuss how we used the feedback from our interviews to help design our function. We also explain the flow of using our function through the flushed out user stories, use cases, and test cases.

4.1 Interview Feedback

Over the course of our project we were able to interview many experts in the robotics field and individuals who worked specifically on the ATLAS robot. They had some interesting ideas and opinions. One common feeling among many of the people we interviewed is that they believed that, in the future, C and C++ will be replaced as the primary robot language by higher level more abstract languages. These higher level languages may have different paradigms than C or C++ entirely, with different systems for very careful and specific error handling and safety features to prevent unwanted outcomes such as falling over or breaking things. Some of the individuals we interviewed were wary of domain specific languages (DSLs), something we were considering as a good solution at the beginning of the project. While DSLs have their advantages, they rarely reach widespread deployment because they are too specific.

Another idea that many of our stakeholders shared was that our initial design was too ambiguous and grandiose in scale. Many of our stakeholders believed that the safety and error handling in the program is a key feature that should be focused upon. One interesting idea that was shared with us is that robots need to get bored, meaning that if a robot keeps performing an action to a failure state, it should give up and try new things. This is putting a very human trait into the robots, if you fail a bunch of times then maybe it is necessary to try performing the action a different way or just not performing the action at all. Another similar view expressed by some of our stakeholders was that human error is the largest issue facing robotics and the best way to prevent robots from failing is to make it harder for humans to tell the robot to do something that will result in failures. The idea of better error handling and safety mechanisms to prevent the robot from doing things it knows will end in it crashing, while difficult to implement, will be the best way to prevent accidents.

Other stakeholders had more hands on suggestions to things that needed to be worked on in the ATLAS robot. The ATLAS robot is a distributed system that relies on multiple discrete systems working together to function. Not all of the hardware on the ATLAS robot is accessible by the programmers, some are unmodifiable black boxes programmed by Boston Dynamics. Others cannot be programmed directly, any update in that part of the system must be transported to it over the built in communication protocols between the discrete systems. This proposes a challenge in and of itself, while the programmer is not directly responsible for communication between each discrete system on the ATLAS bot they still need to understand how it works.

Furthermore, there is a whole suite of tools to work with the ATLAS robot, including command line tools to communicate with the robot, tools to load programs to the robot, simulators to test programs, and many more. All these things together create a system that can be difficult to work with even with experience and can be almost unapproachable to a beginner. One stakeholder expressed the need for an easier way to interface with the ATLAS robot, or if that is not possible, at least good resources for beginners to more easily learn how to interface with the robot and build programs. While this is not specifically what we are choosing to focus on in our project, this could be an interesting jumping off point for future work.

4.2 Design and Intent

When programming a robot such as WARNER, the programmer must be able to run an action multiple times to ensure success. For example, an action could be "pick up a hammer," which is made of a few sub-actions: open hand, extend arm, move arm above hammer, close hand on hammer, and move arm up. Since these are sub-actions, they need to be run sequentially, but if sub-action move arm over hammer fails, the robot should not start the entire action over. The looping function we designed allows a programmer to run an action, or a list of sub-actions, a specified amount of times or until success. Our function is designed to allow the programmer the opportunity to design how they want the sub-actions to run by using breakpoints that are placed between sets of sub-actions.

Looking back at our hammer example, we can look at the user story for it: I want action *pick_up_hammer* with sub-actions (1) *open_hand*, (2) *extend_arm*, (3) *move_arm_above_hammer*, (4) *close_hand_on_hammer*, and (5) *move_arm_up* to be run with the grouping:

```
{1, {2, 3, 4}, 5}.
```

Actions 2, 3, and 4 are grouped because they depend on each other; if *close_hand_on_hammer* fails, it may be because *extend_arm* did not fully complete, or because *move_arm_above_hammer* did not end up in the correct location. This allows the function to jump back to sub-action 2 if sub-action 3 or 4 fails, instead of jumping all the way back to the beginning.

Next, we will look at the use case for our hammer example:

Actor	The programmer
Arguments	The sub-actions need to be run with the grouping `1, {2, 3, 4}, 5`
Triggers	A success on a sub-action
Basic Flow	If a sub-action succeeds, move onto the next sub-action. If sub-action

	fails, jump back to the last breakpoint.
End Condition	Action *pick_up_hammer* is completed successfully

This use case shows a bit more information about how the user will interact with our function. The programmer needs to specify the groupings of the sub-actions in order for the function to run properly. It then shows how the function will operate on a high level given the user input. It also shows that the function will run indefinitely until the full action is completed; this is the default option of function but the user does have the option to specify how many times they would want the sub-actions to run before termination of the action.

Lastly, we will look at a test case of the hammer example to show how we tested our function to ensure it was working properly.

Summary	I want action *pick_up_hammer* with sub-actions (1) *open_hand*, (2) *extend_arm*, (3) *move_arm_above_hammer*, (4) *close_hand_on_hammer*, and (5) *move_arm_up* to be run with the grouping 1, {2, 3, 4}, 5
Test Procedure	Call looping function with a list of sub-actions with the specified grouping
Test Data	sub-actions: (1) *open_hand*, (2) *extend_arm*, (3) *move_arm_above_hammer*, (4) *close_hand_on_hammer*, (5) *move_arm_up* Grouping: 1, {2, 3, 4}, 5
Expected Result	1. Each sub-action will be attempted and if success, it will move onto the next sub-action, or complete the action and exit the function 2. sub-action 1 will be attempted and if failure, it will jump to sub-action 1 and repeat 3. sub-action 2, 3, and 4 will be attempted and if failure, it will jump to sub-action 2 and repeat 4. sub-action 5 will be attempted and if failure, it will jump to sub-action 5 and repeat

Test cases are used to show even more information about how exactly our function will operate under certain conditions. More specifically, it shows how the function uses the specified groupings to repeat sub-actions upon failure. The function uses breakpoints in order to set where a sub-action should jump back to upon failure. For example, consider the grouping `1, {2, 3, 4}, 5` to action be a set of 3 groups: `{1}, {2, 3, 4}, {5}`. Our function sets

breakpoints (b) to be at the beginning of each group: `{b1}, {b2, 3, 4}, {b5}`; if any sub-action in a group fails, the function will jump to the first sub-action in that same grouping.

5.0 Analysis and Testing

In this section, we discuss how a programmer would use our function, how we tested our function to ensure it ran as we intended, as well as how we used our function to help simplify the Atlas code.

5.1 Implementation

Our function operates by running through a list of `ActionObject`, which contain a pointer to a function that executes an action, as well as some metadata about the action such as how far to jump back on failure, etc. The core code only takes in three arguments: a `std::list` of `ActionObject` to be run, a `std::list` of indexes representing breakpoints, and an enum that describes how to loop through these objects in different ways. All other data is encapsulated within `ActionObject`, which contains a pointer to a function to be run, a minimum number of times the function should pass to be considered a success, and a list of pairs of ints, with each pair representing what the looping function should do on a given number of failures. For example, the list

```
{{1, 3}, {2, 4}, {2, 5}, {3, 10}}
```

indicates that on the first failure, the program should jump back 3 steps. If the function later fails 2 times, it should jump back four steps, then if it later fails an additional 2 times, it should jump back 5 steps, etc.

In order to make the function more readable and easier to use, we added a number of default options that drastically reduce the burden on the programmer. First, we assumed that every action wants to jump back to the nearest breakpoint. This allows the programmer to eschew manually creating breakpoint indexes, and instead structure the input in such a way that the breakpoints can be logically inferred. For example the in the series:

```
{{A, B}, {C}, {D, {E, F} G}
```

The default will put breakpoints at A, C, D, and E. We did this by automatically putting breakpoints at the beginning of each grouping, then setting the jumpback distance to be arbitrarily large.

Second, we added a lot of syntactic sugar to our inputs. In its most detailed, our function can be called by:

```
	//Declare the pointers to functions
	bool(*pointerToFunctionA)(); pointerToFunctionA = &FunctionA;
	bool(*pointerToFunctionB)(); pointerToFunctionB = &FunctionB;

	//Create pairs of failure-nums and jump-backs
	PairInt pair1(2, 2);
	PairInt pair2(4, 5);

	//Create the list of pairs
	std::list<PairInt> lpairs1 = { pair1, pair2 };
	std::list<PairInt> lpairs2 = {};

	//Create ActionObjects with a pointer to a function, a list of pairs, and
a minimum number of successes
	ActionObject myObjectA(pointerToFunctionA, lpairs1, 3);
	ActionObject myObjectB(pointerToFunctionB, lpairs2, 2);

	//Create the list of ActionObjects
	std::list<ActionObject> lActionObjects = { myObjectA, myObjectB };

	//Call atlas loop with style Standard, the list of action objects, and an
empty list of breakpoints
	atlasLoop(Standard, lActionObjects, std::list<int>{});
```

This method of calling is extremely verbose, but very powerful, allowing any desired modifications to be made to the default actions. By adding a few interpretation helper functions, we allowed our code to be called with this:

```
	 //Declare the pointers to functions
	bool(*pointerToFunctionA)(); pointerToFunctionA = &FunctionA;
	...

	//Create the array of objOrLists
	ool arrayOfObjOrList[] = {
		ool(pointerToFunctionA),
		ool(pointerToFunctionB, pointerToFunctionC)
		ool(pointerToFunctionD)};

	//Call looping function with size of list
	atlasLoop(arrayOfObjOrList, 3);
```

This code does the same thing as the code above, with more flexibility and power, in fewer lines. This is done by making some assumptions about the way the programmer inputs information to our function. The array that is passed into the `atlasLoop` contains at each index an `objOrList` (called an "`ool`" for brevity) that was created with either a single

function pointer, or a number of comma-seperated function pointers. Internally, when `atlasLoop` is passed an array of type `objOrList`, it runs two desugaring functions, first to "flatten" the array into a lone array of pointers to functions, and second to turn those pointers to functions into `ActionObjects`, each with appropriate jumpback values and breakpoints.

Of course, it is also possible to use a combination of the two input styles. Any values that are not manually specified, the function will attempt to fill in.

5.2 Testing

In order to test that our code was not significantly slower than a manual loop, we wrote 10 functions that each created 50,000 arrays that were 50,010 elements long, then iterated through each array to assign each element to the value of the index. We then wrote two testing functions: our baseline function would call each of the 10 functions once, and our looping function would call one large loop with the 10 functions in it once. We ran each of these testing functions 30 times and averaged the results. On average, the baseline code took 50.2495 seconds to run on a group member's computer, while the looping code took 49.9661 seconds, making our code slight *faster* than simply writing a custom loop. However, the standard deviations of this run were 0.179035 and 0.49882, respectively.This means that the slight time difference is completely insignificant. To look for a more significant difference, we changed our 10 functions to create 500 arrays that were 510 elements long, and instead ran this code 300 times to get more data points. This showed that the baseline code took 0.00541667 seconds to run while our code took 0.0134767. This is more in line with our expectations, as the ~8 milliseconds small time differential is likely due to the more complete edge cases that our looping code handles. The standard deviations of this run were 0.0087 for both cases.

5.3 Deployment

The Atlas robot is made of many interworking parts and our function runs in conjunction with the actual program that runs on the Atlas robot. The Atlas robot uses the Robotic Operating System (ROS) to operate. ROS programs are made of many nodes of different types that communicate with each other during the execution of a program. Each of these nodes can be written in C++ (among other languages) and can therefore be rewritten using our system.

Unfortunately we were unable to fully deploy our system to the Atlas robot due to some hardware constraints. In order to fully run the Atlas robot simulation, it is necessary to have a computer running Ubuntu natively (a virtual machine will not suffice) and that system needs a large amount of RAM, a fairly high end central processing unit (CPU) and, probably most importantly, a discrete graphics processing unit (GPU). In order to truly run the simulation, a discrete GPU is required for the OpenGL library, for both the rendering and the actual simulation of the distributed system on the Atlas robot.

One solution we discussed with some of our stakeholders would be to run the simulations without any of the ‘extra’ or computationally expensive systems. While this may have worked, it was brought to our attention that there are no ‘extra’ systems on the robot and that it is overall a very complex system and its difficult to turn off the unused parts. Even if we could, our hardware would still be incapable of fully running the system because we still lacked a discrete GPU.

One solution we explored briefly was to simulate a simpler system, using the Turtlebots, a system used for an undergraduate course at WPI. The system is fairly simple and we could have obtained the starter code with ease and began work transitioning parts of the existing code to use our structure. We ultimately ran out of time to perform this exploration and left it as a suggestion for future work.

6.0 Conclusions

In this section, we discuss how our project could be extended in the future.

6.1 Future Work

While we made many strides forward in creating a cleaner looping structure, there is plenty of room for future groups to improve our system. A few key assumptions we made are that functions took no parameters and returned only booleans. This is a large assumption, but it is easily remedied. C++ allows the programmer to create pointers to any function, but it is necessary to provide the type of all of the input variables and to provide the type of the return value. These are easily created, and creating a set of general use function pointers would be fairly easy. By creating a larger set of function pointers future groups could rewrite many different existing programs using our looping tools.

A second opportunity for future work would be to create a new syntax for the looping structure. Currently, it is necessary to write multiple lines to create function pointers, and then fill them with the necessary input and output types. A future group could create a set of macros using C++ or another language such as Racket or Lisp to create functions that take in the requisite information and expands the input into the proper function pointers. One issue that we faced was that C++ does not allow arbitrary strings to be executed as code, as this poses a major security hazard. Racket and Lisp have built in functions for creating macros and code expansion that could be used to perform this functionality, the difficult part would be then converting the expanded Racket or Lisp into C++. We leave this execution to future groups.

Another opportunity for future projects would be to complete extensive tests using the robot Atlas. We designed the function to run with any code base, but with Atlas specifically in mind. That being said, we were not able to run any tests on the physical robot Atlas; we only ran some tests on the simulation of Atlas. We believe that the function would be much more robust

and tailored to Atlas after it went through extensive testing with the physical robot. Additionally, in order to show the generality of our system and how any pre-existing robot can be rewritten using our structure, future groups could also perform similar testing on WPI's turtlebots. These are smaller, easier to use robots that could act as a very realistic small scale test before upgrading to the full sized Atlas robot.

6.2 Conclusion

Through the extent of the project, it became painfully clear that there is no simple or easy solution to creating more readable and extensible code. While a DSL is good solution for one system, they rarely pose a realistic solution for more general problems as we originally had hoped. Building libraries in C++ to help create more readable and easier to write code works well but can be just as extensive as the robots program itself. Striking a balance between robust and brief code is quite difficult and we have done just this. Our system works in many cases within .5% as quickly as normally structured C++ code but is easier to read and maintain.

We also considered the many groups and individuals that our project would have an impact on. In the small scale, our project will hopefully improve the lives and productivity of programmers working on Atlas, and on many other robotic platforms that this project can be deployed to. Atlas has many inherent safety protocols such as always being on a tether, having and emergency stop, and always having two people present when in use. It is important to us that our project would not impede the execution of these protocols, and hopefully it can help make these protocols safer in the future. In the longer term, we hope this project enables others to better achieve their goals and improve the world.

References

Atkeson, Chris. Personal Interview. 17 September 2018.

Babu, Benzun Pious Wisely. Personal Interview. 24 September 2018.

Jagtap, Vinayak. Personal Interview. 18 September 2018.

Prato, C. (2016, January 28). WPI Home for WARNER. Retrieved September 26, 2018, from https://www.wpi.edu/news/wpi-home-warner

V. Verma, A. Jonsson, R. Simmons, T. Estlin, R. Levinson, (2005) “Survey of Command Execution Systems for NASA Robots and , Spacecraft’, “Plan Execution: A Reality Check” Workshop at The International Conference on Automated Planning & Scheduling (ICAE’S)

Appendix A: Our Preliminary Design Review

Project Objectives

Our project goal is to create a system to simplify or improve the experience of working with WPI's Atlas Robot. This system could take many forms, such as a library that adds commonly used features to make writing programs simpler, or a synchronization or type system change to improve the workflow with programming the robot. Some of the ideas we considered are listed below. Each of these systems has its own specifications, challenges, and requirements.

- Type Systems and Type Inference
- Synchronization
- Error and Exception Handling systems
- Control Systems, such as looping and branching statements
- Safety Systems

Specifications and Challenges

Type systems are an important part of many languages. Most languages that are higher level than assembly and machine code have some sort of type systems. Some languages, like OCaml and Haskell, use their own type system, in addition to their type inference system, to express aspects of the language beyond simple things such as integers and strings. OCaml and Haskell use their type systems to express asynchronous code and error/exception handling. Currently, since WPI's Atlas robot is programmed mostly in C, this type of expression is not possible. Implementing a system like this for the robot could decrease run time errors in code.

Synchronization is an important functionality in WPI's Atlas Robot, since it has multiple discrete systems that must communicate with each other in order to make it function. If we were to implement new synchronization systems, the primary goal would be to create an easier way to program the robot, while not sacrificing the expressiveness of the language.

Error and exception handling systems in C/C++ are clunky, and can be difficult to understand. If we were to improve the error and exception handling systems, our main objectives would be to create a better way to show errors and exceptions. Some secondary objectives would be to implement 'safe states,' where, if the robot happened to experience an error or exception, it would enter a state where it prevents self harm. This prevention could be avoiding falling over or avoiding self collisions. This potential objective pairs nicely with the safety system objective, and could be implemented beside it.

Control systems, while a broad category, do impose some interesting challenges to programming WPI's Atlas robot. Looping and conditional statements are frequently used, and offer no built in functionality to care to robotic programming. New looping structures could be developed to offer the programmers a finer built in control of how to loop and what to do in the case of an error. For example, if three items, A, B, and C, need to be looped in a particular order, or with a particular grouping; a custom looping structure can make it easier for the programmer to follow these constraints. Custom loops can also natively include certain safety and error handling measures, such as performing certain actions in the event of an error to prevent harm to the robot.

Safety systems can accompany many of the previously described systems. The primary objectives in a safety system are to create protocols to prevent the robot from harming itself in any situation where something has caused an error within the primary system. A safety system can be thought of as a background system that the primary system can start in any kind of emergency or error situation. This objective would pair particularly well with the control system.

Requirements

Implementing a new **type system** would require us to change the existing type system. Once the type rules have been developed and mathematically analyzed, we will develop a type checker for our language based on our type rules. This could take many forms, one could be part of a compiler or as a system beside a semantic analyzer built on top of the existing C code. Once the type checker has been implemented, we will have to either design new code in our type system that is compatible with older C code, or we will have to retrofit all the existing C code.

Synchronization requires that our product be able to safely and repeatedly run code in an asynchronous manner. The system may be built with native C/Unix constructs such as semaphores and system/user level threads. The system will have to be compatible with the Atlas robot's current synchronization system and its communication system. Since one of the computers running on the robot is not (re)programmable, our synchronization system must be able to interface with it.

Error and exception handling systems require an extensive set of robust and descriptive exceptions. These will require careful definitions and class hierarchy setup. After the library of exceptions is created, a system similar to the Java try/catch block will be implemented in order to obtain the most out of the new exception hierarchy system; errors will follow a similar pattern. In either case, a two path execution system, whereby one path of execution corresponds to correct and error free executions, while the other path corresponds to a path that has *any* errors along its executions. This second path can then later report the error to the user.

Control systems must be easily integratable with existing code. It must also be fully featured to express things like failure specific execution loops.

Safety systems must be easily implementable and easy to switch into and out of. A safety system must allow programmers to create routines that prevent harm and damage, but also allow control of the system to be returned to the primary execution system once the danger has been avoided.

Appendix B: User Stories, Use Cases, and Test Cases

User Stories

1. I want action a with sub-actions a_1, a_2, ...a_N to be run X times, or until success.
2. I want action a with sub-actions a_1, a_2, ...a_N to be run X times, switching to action b upon success.
3. I want to do sub-actions a_1, a_2, ...a_N sequentially X times, skipping back the lesser of either Y step(s) or to a_Z upon failure.
4. I want to do sub-actions a_1, a_2, ...a_N sequentially X times, skipping back a variable number of steps depending on type of failure.
5. I want action a with sub-actions a_1, a_2, ...a_N, to be run X times, skipping back Y steps(s) upon failure, but never skipping past a_z. (breakpoint/savepoint)
6. I want action a to be run X times, switching to action b after Y failures, then to c after Z more failures.
7. I want action a to be run X times or until success, then I want action b to be run Y times, where Y is dependent on the number of loops of a.

Use Cases

Actor	The programmer
Preconditions	Actions a_1, a_2, ...a_N that need to be run X times each
Triggers	A success on an action
Basic Flow	If action a_n succeeds move on to action a_{n+1}
End Condition	Action a_N is completed successfully.

Actor	The programmer
Preconditions	Actions a_1, a_2, ...a_N that needs to be run X times each
Triggers	A failure on action a_n
Basic Flow	After a Failure on action a_n go back to previous action a_{n-i}
End Condition	After some number of failures on the same action, exit loop with an error.

Actor	The Programmer
Preconditions	Actions a_1, a_2, ...a_N that needs to be run X times each
Triggers	A Failure on action a_n
Basic Flow	After a failure on action a_n begin execution on action b
End Condition	Action a_N completes or action b completes.

Actor	The Programmer
Preconditions	Actions a_1, a_2, ...a_N that needs to be run X times each and a state a_m that the program will not go further back than
Triggers	A failure on one action a_n
Basic Flow	If action a_n fails, go back to action a_{n-i} unless that action is less than a_m,in which case go to a_m
End Condition	Action a_n is reached or an error is thrown

Actor	The Programmer
Preconditions	Actions a that needs to be run X times each, a separate action b, and a number of times Y to fail action a.
Triggers	Failure on action a
Basic Flow	Run action a until it fails, then repeat action a until it has failed Y times. Then after a has failed Y times run b
End Condition	Either action a or b passes fully and successfully.

Test Cases

Summary	I want action *a* with sub-actions $a_1, a_2, ...a_N$ to be run X times, or until success.
Test Procedure	Call looping function with an action and with a number
Test Data	Action: move arm Number: 5
Expected Result	1. Action will be attempted 5 times and if no success, an error message will appear and the action will be terminated. 2. Action will be attempted 5 times and if success, the arm will have moved and the function will be exited.

Summary	I want action *a* with sub-actions $a_1, a_2, ...a_N$ to be run X times, switching to action *b* upon success.
Test Procedure	Call looping function with a primary action, a number, and a secondary action
Test Data	Primary action: make a fist Number: 5 Secondary action: punch
Expected Result	1. Fist action will be attempted 5 times and if no success, an error message will appear and the action will be terminated. 2. Fist action will be attempted 5 times and if success, it will move on to the punch action and the function will be exited.

Summary	I want to do sub-actions $a_1, a_2, ...a_N$ sequentially X times, skipping back Y step(s) upon failure.
Test Procedure	Call looping function with an action composed of sub-actions and two numbers
Test Data	Action: make a fist Sub-actions: (1) close pinky finger, (2) close ring finger, (3) close middle finger, (4) close pointer finger, (5) close thumb Number X: 5 Number Y: 2

Expected Result	1. Each sub-actions will be attempted 5 times and if no success on sub-actions 3-5, jump back 2 sub-actions and try again. 2. Each sub-action will be attempted 5 times and if no success on sub-actions 1-2, jump back to sub-action 1 and try again. 3. Each sub-action will be attempted 5 times and if success on all, the robot will have a fist and the function will be exited.

Summary	I want to do sub-actions a_1, a_2, ...a_N sequentially X times, skipping back to a_Y upon failure.
Test Procedure	Call looping function with an action composed of sub-actions and two numbers
Test Data	Action: make a fist Sub-actions: (1) close pinky finger, (2) close ring finger, (3) close middle finger, (4) close pointer finger, (5) close thumb Number X: 5 Number Y: 1
Expected Result	1. Each sub-actions will be attempted 5 times and if no success, jump back to sub-action 1 and try again. 2. Each sub-action will be attempted 5 times and if success on all, the robot will have a fist and the function will be exited.

Summary	I want action *a* to be run X times, switching to action *b* after Y failures.
Test Procedure	Call looping function with two actions and two numbers
Test Data	Action a: make a fist Action b: move arm Number X: 6 Number Y: 3
Expected Result	1. Fist action will be attempted 6 times and if it failed 3 times, arm action will be run. 2. Fist action will be attempted 6 times and if success without 3 failures, the arm will have moved and the function will be exited.

Summary	I want action *a* to be run X times, switching to action *b* after Y successes.

Test Procedure	Call looping function with two actions and two numbers
Test Data	Action a: clap hands Action b: move arms up Number X: 6 Number Y: 3
Expected Result	1. Clapping action will be attempted 6 times and if it failed 4 or more times, an error message will appear and the action will be terminated. 2. Clapping action will be attempted 6 times and if success 3 times, arm action will be run.