RUNTIME VERIFICATION OF STATE DIAGRAM FOR ROBOTICS

By

Taylor Harvin

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of Master of Science

in

Computer Science

Middle Tennessee State University

December 2017

ACKNOWLEDGEMENTS

I would like to thank my thesis committee: Dr. Dong, Dr. Li, and Dr. Barbosa for agreeing to be my advisors and assisting me throughout this process. Dr. Dong in particular with his expertise on the field of runtime verification was specifically helpful to me during development and debugging this framework. His advice and help truly made this project possible. He made my transition from the working world back to school much smoother by asking me to participate in this project, and I am truly grateful. In addition, I would like to thank Dr. Li for being a wonderful teacher in robotics. Her classes throughout my Bachelors and Masters with robotics specifically helped with constructing a proper testing environment with the LEGO robotics; with her high school summer robotics camp, she ultimately convinced me to choose Computer Science. Furthermore, Dr. Barbosa has also been a tremendous help in this whole process. His game AI class helped me think outside the box with some of the challenges that I faced with the robotic systems in this project, and he was very helpful during some rough times in the overall integration of the framework into the robotic system.

To all of my computer science processors at MTSU: these past years throughout my undergraduate and graduate time have been challenging and fun, and I will truly miss learning from top quality best professors. All of you made my dreams possible.

This work was partially supported by AFRL under FA8750-15-2-0106. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

ABSTRACT

It is critical to develop a trustworthy system for cyber physic systems (CPS), such as unmanned aerial vehicle and robotic systems. However, it is challenging to develop trustworthy systems due to complicated system behavior and unknown or even hostile external environments that are in general unstable. It becomes even worse because of the integration of error detection and handling code in the system to react to unknown events or exceptions. To facilitate the development of trustworthy systems in CPS, we proposed a framework that allows developers to monitor system behavior at runtime easily. The framework is built around runtime verification tools and could detect any deviation from system behavior that is specified in state diagrams. One benefit of our framework is that it separates the monitoring code from system code that achieves the required functionalities. This creates a cleaner and modular system. A case study of a Lego EV3 robot is conducted to evaluate our framework.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

AOP	Aspect Oriented Programming
CPS	Cyber Physical System
LTL	Linear Temporal Logic
MOP	Monitor Oriented Programing
RV	Run-Time Verification

CHAPTER I

INTRODUCTION

A **cyber-physical system** (**CPS**) is a mechanism that is controlled or monitored by computer-based algorithms, tightly integrated with the Internet and its users. [1] Cyber physical systems are becoming more and more popular in our daily life, from autonomous car on highway, robotics on manufactures and warehouses, to unmanned aerial vehicle (UAV) in battle fields to perform different predefined services. Due to the increasing prevalence of CPS, our safety and security depends on their trustworthy operation. Any uncaught hardware failure and software malfunction may result in the loss of life or huge financial loss, especially in critical areas such as military operations, medical equipments, or power plants. This poses a challenging issue to create a trustworthy systems for CPS.

Traditionally, formal methods have placed an important role to ensure trustworthiness with high confidence in the past years. Model checking [2], theorem proving [3], static analysis [4] are the typical formal approaches to ensuring the trusted and high assurance software intensive and resilient systems. However, these approaches all have drawbacks. Model checking suffers the state space explosion problem, especially in the presence of concurrency and unbounded types. Although many heuristic methods have been developed, such as abstraction and partial-order reduction, scalability is still an issue when model checking large scale systems. Theorem proving based on deductive reasoning often requires human interaction and has limited applicability due to undecidability of many underlying theories. Yet another technique, with a limited set of generic errors, static analysis may output false positives sometimes. On the other hand, testing can function in a systematic manner while offering great benefits in the identification of programming logic errors with incompleteness.

In contrast, runtime verification [5] as a light weighted formal approach to detect errors at runtime provides a complementary, dynamic and scalable validation technique for large software intensive systems. Traditionally, error detection and handling is mixed with normal code, which tends to lead to sloppier spaghetti code although it is a common and valid approach. This reduces the confidence on software and therefore decrease trustworthiness of the system. Aspect-oriented programming (AOP) [6] was proposed to increase modularity by allowing the separation of cross-cutting concerns, which is done by weaving additional behavior to existing code in the form of advice without the need to modify the code itself. Runtime verification, on the other hand, uses the information extracted from a running system to detect and possibly react to observed behaviors satisfying or violating certain properties. Aspect-oriented programming is a natural choice for runtime verification to weave code to an existing system so that required information can be extracted during execution from a system without modifying it.

In this thesis, a framework is developed to improve trustworthiness of robotic systems using aspect-oriented programming and runtime verification by validating its behavior defined in state diagrams. Traditionally, robotic systems require an abnormally large amount of extra code to detect and possibly handle misbehavior in a unstable or even hostile environment, which ultimately introduces an extra layer of difficulty to the systems in both development and future modification. Our framework avoids this issue by separating property verification and error handling from the primary system code and therefore results in a more modular and organized set of code. During development, source code is divided into two categories representing the primary system code and the RV code, which is weaved into primary system code through aspect-oriented programming. This allows developers to focus on the actual tasks of the robot rather than worry about directly including code to prepare for every possible situation. With a fully functional system, developers can then easily shift focus back to runtime verification with little to no modification of the primary source code. If both components are correct, the system will perform its tasks and attempt to detect and handle failures.

Our framework can reduce the work to extract state related information from a running system for property verification by providing an abstraction to state related information. State representations and transitions are typically more complicated than a simple method call, so this abstraction is needed to ease RV and property development. Once this abstraction is achieved, the logic is almost directly if not directly matched to the state diagram of the system when writing property formula, therefore improving the readability of the RV portion.

Overall, the primary goal of the research is centered around improving trustworthiness of a system with better organization through modularization of behavior monitoring and handling. The headaches presented by the unique challenges during robotics development should be reduced once overcoming the slight learning curve of the concept and tools in RV development. Ensuring reliability of systems is a crucial factor in many cases, and helping developers do so with this approach will hopefully prove useful for many applications.

The rest of the thesis is organized as the following: Chapter 2 introduces required background knowledge and tools such as linear temporal logic, AspectJ, JavaMop, and LeJOS. Chapter 3 explains our framework and case study in detail. Chapter 4 discusses our experiment result of our case study. Chapter 5 gives the summary of our work.

CHAPTER II

BACKGROUND

In this chapter, we will review required preliminary knowledge for the thesis. It includes linear temporal logic (LTL), AspectJ [7], JavaMop [8], and leJOS EV3 [9]. Linear temporal logic is a modal temporal logic with modalities referring to time. In our work, properties of system behaviors to be monitored at runtime are specified as LTL formula. AspectJ is an aspect-oriented programming extension created for the Java programming language. It allows programmers to weave cross-cutting concerns into Java program without modifying it. JavaMop is a runtime verification tool developed to monitor system behaviors during its execution. The LeJOS is a tiny Java virtual machine providing support to develop and execute Java program in LEGO EV3 robotics. The rest of the chapter will cover these topics in detail.

Section 2.1: Propositional Logic

Logic is a key concept in many fields, and computer science is centered around its usage. Considering that programs are essentially just a listing of actions to be processed by a machine, including at least basic logic is critical to ensuring correct actions are performed, and propositional logic helps fulfill this need.

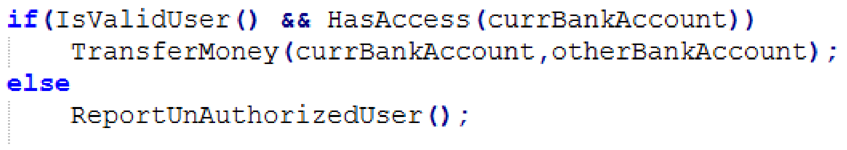
Propositional logic gives users the means to evaluate the truthfulness of a compound statement. It provides basic operators as shown in **Table 2.1**. The AND, OR, NOT, IMPLIES, and EQUIVALENCE operators allow users to build and evaluate statements in a simple yet powerful way. Furthermore, **Table 2.1** also shows the result of every possible combination of operand value combinations and their evaluations for each basic usage of the propositional operators. For example, if Op1 (one atomic statement) is true and the second atomic statement Op2 is false, the compound statement of Op1 or Op2 is true because it follows the or rule of at least one operand is true.

In computer science, propositional logic is crucial to developing software. Statements help control the overall flow of a program; from if/else statements to while and for loops, propositional logic is the central focus of these constructs in programs. If a developer only wants a loop to execute ten times, then a structure such as a counter variable along with a conditional statement such as count < 10 is needed and a way to increment the counter (from 0 to 10) count++ (where ++ says to increment the count by 1 at every loop).

Table 2.1. Truth table for basic propositional logic.

Op1	Op2	Not Op1	Op1 And Op2	Op1 Or Op2	Op1 Implies Op2	Op1 Equivalent Op2
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

Controlling the flow of actions in a program in this way is critical for various other reasons to. For example, consider the if/else statement in **Figure 2.1**. If this represented a part of an online banking program that allowed users to access their bank accounts and transfer money to other accounts, it is fairly obvious why it is important to have at least basic logic in programs. Looking at the if statement portion, two separate methods are the deciding factors as to whether or not the user can transfer money from the one bank account to another. Connecting the two functions IsValidUser() and HasAccess(…) is an **AND** operator to provide some basic security to the TransferMoney(…) method. As demonstrated, if both methods have to be true in order for the requested transfer to be processed, otherwise the request will be flagged as an unauthorized action. Without this basic logic, the money transfer would be open to anyone.



```
if(IsValidUser() && HasAccess(currBankAccount))
    TransferMoney(currBankAccount,otherBankAccount);
else
    ReportUnAuthorizedUser();
```

Figure 2.1. Basic logic in program for a bank account example.

This example is not particularly complex, but it does show the basic idea of the usage of logic in computer science. Multiple other propositional logic operators exist as shown in Table 2.1, and they can be combined to make complex conditions for programming constructs. Table 2.1 simply shows the basic usage of all of the propositional logic operators, along with all possible combinations of the operator values and the statement results. Looking at the example from **Figure 2.1** and **Table 2.1** you can see all of the combinations of true (T) and false (F) and which combination results in a true statement meaning the money transfer is authorized. Again, these operators can be combined with each other to form more complex statements, and Op1 and Op2 from **Table 2.1** also represents statements in themselves. [10]

Section 2.2: Linear Temporal Logic

Logic is clearly a crucial tool for computer science, but basic propositional logic alone lacks key qualities necessary for representing properties across multiple states. In general, propositional logic is centered around determining if a statement is true or false based on one set state. This is where linear temporal logic takes over. Linear temporal logic extends propositional logic to include time and state sequence based logic to describe state properties over an infinite sequence of states or a finite set of states with the final one assumed as repeating infinitely.

Looking back at **Figure 2.1**, the simple if/else statement will technically cover the logic needed for basic security, but it does not account for system failures or malicious attacks on the system. For example, how does the system recognize that the user always has access to the bank account? How does the system know when a session expired? Linear temporal logic helps with these time and sequence based logic issues, that basic propositional logic alone cannot easily handle.

With runtime verification, it is important to track the flow of the state changes as the system runs through its procedure. This trace of states is represented by a sequence of states over an infinite duration. Realistically, this means that over time, a group of state points will represent the overall flow of the system, and it is important in the context of runtime verification to ensure that this flow of state transitions over time matches the expected flow given a fully functional system. Linear temporal logic extends propositional logic to handle such issues. As the name implies, it is logic associated with a linear time scale. It introduces the concept of future and past aspects and operators to handle time based logic. The primary past operators include ‘*previously*’, ‘*since*’, and ‘*eventually in the past*’, and the base future operators includes ‘*eventually*’, ‘*next*’, ‘*until*’, and ‘*always*’. **Table 2.2** gives a general description of each operator. Note any reference to ‘p’ and ‘q’ each represent a generic property of a given system. [11,12,13,14,15]

Table 2.2. Linear temporal logic operators (p and q as the operands).

[] p	p remains true at every time point in the future
<> p	p is true eventually in the future
o p	p is true in the next time point
p U q	p remains true until q becomes true
<*> p	p was true at some point in the past
(*) p	p was true in the previous time point
p S q	q was true in some past point, and from that point, p has remained true

Section 2.2.1 Always Operator

First, the unary operator **always**, represented by the LTL formula ‘[] p’, ensures a certain property remains true for the entire sequence of states. For example, given a system that monitors for blockages in a water pipe required a property such as: [] (water_remains_flowing). Upon system initialization, the assumption is that water will continue to flow through the pipe forever in every state. If, however, the system at any point in time after sensing a water flow stops reading a flow, then a violation of the logic occurs and hence the statement is false as shown in **Figure 2.2**. As shown in this example, the '[] p' can refer to a subset of time rather than the entire state trace, but it requires a combination of other logic operators. [11, 12, 13, 14, 15]

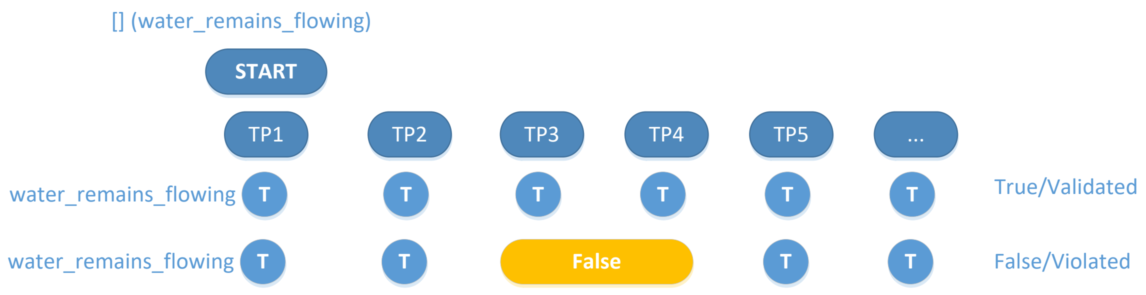


Figure 2.2. Water-flow example: state sequences that validate/violate [] p.

Section 2.2.2: Eventually Operator

Next, the LTL formula '<> p' referred to as **eventually** formula *p* is true. Usage of this formula means that at some point in the future in the state trace, a given property will eventually become true. For example, the water flow example is modified to: <> *water_remains_flowing*. The statement remains true if the system reads a water flow at some point after initialization and before termination of the system as shown in **Figure 2.3**. If the system reads the water flow at any given time point after the initiation, the statement is true even if the system detects a blockage sometime before the termination. [11, 12, 13, 14, 15]

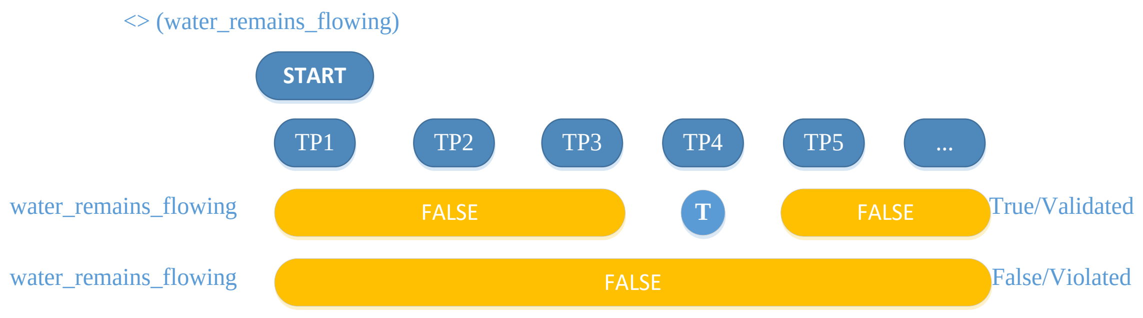


Figure 2.3. Water-flow example: state sequences that validate/violate <> p.

Section 2.2.3: Next Operator

Another major part of LTL is the next formula represented by the 'X p'. This formula will only be true if in the next state, the property is satisfied. Using the water flow example again, *X (water_remains_flowing)* requires that the following state should be that the water_remains_flowing is satisfied as shown in **Figure 2.4**. [11,12,13,14,15]

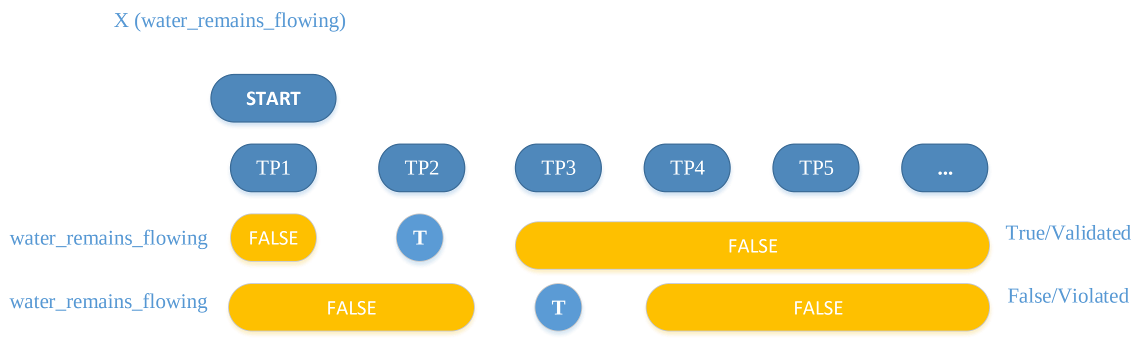


Figure 2.4. Water-flow example: state sequences that validate/violate Xp

Section 2.2.4: Until Operator

Last, the *until* LTL formula, represented by 'p U q' provides one of the more useful features of LTL. The *until* formula holds if formula *p* is true in each of the following states until a state that makes formula *q* true. Looking back at the water flow example, a good modification would be: *water_remains_flowing U valve_locked*. This extension to the water flow example shows that the *until* operator can bend the base logic to some extent to fit a more specific need as shown in **Figure 2.5**. If any of the three previous LTL formula examples were the only ones used, the system would lack the direct ability to handle a blockage versus a valve shutoff. Utilizing the *until* operator in this scenario gives a direct and simple approach to handling the logic. [11, 12, 13, 14, 15]

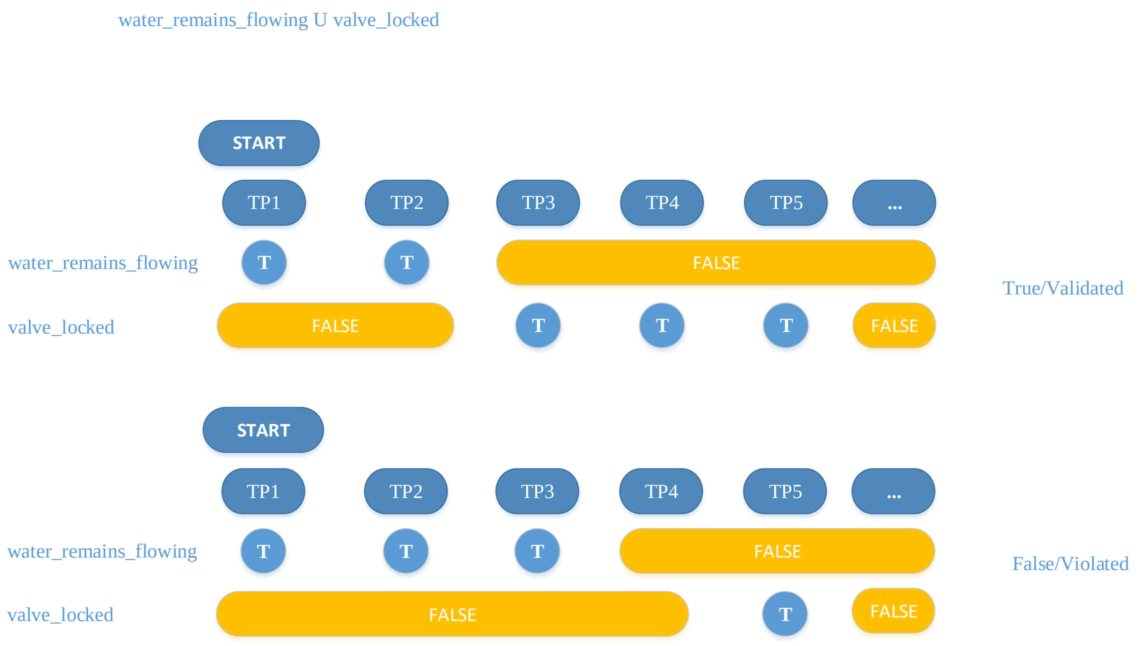


Figure 2.5. Until LTL formula example using p U q.

Section 2.2.5: Conclusion and Example

Given the main LTL operators, more complex formulas can be developed to cover a given property of a system. Looking back at the water flow example, imagine an extra property: water remains flowing in the pipe until a blockage occurs or the valve is locked and a system alarm is immediately after the water stops flowing to notify workers. Given the language of this property, it should be clear that a combination of the LTL properties is needed to cover this case. **Figure 2.6** shows the LTL formula to cover this.

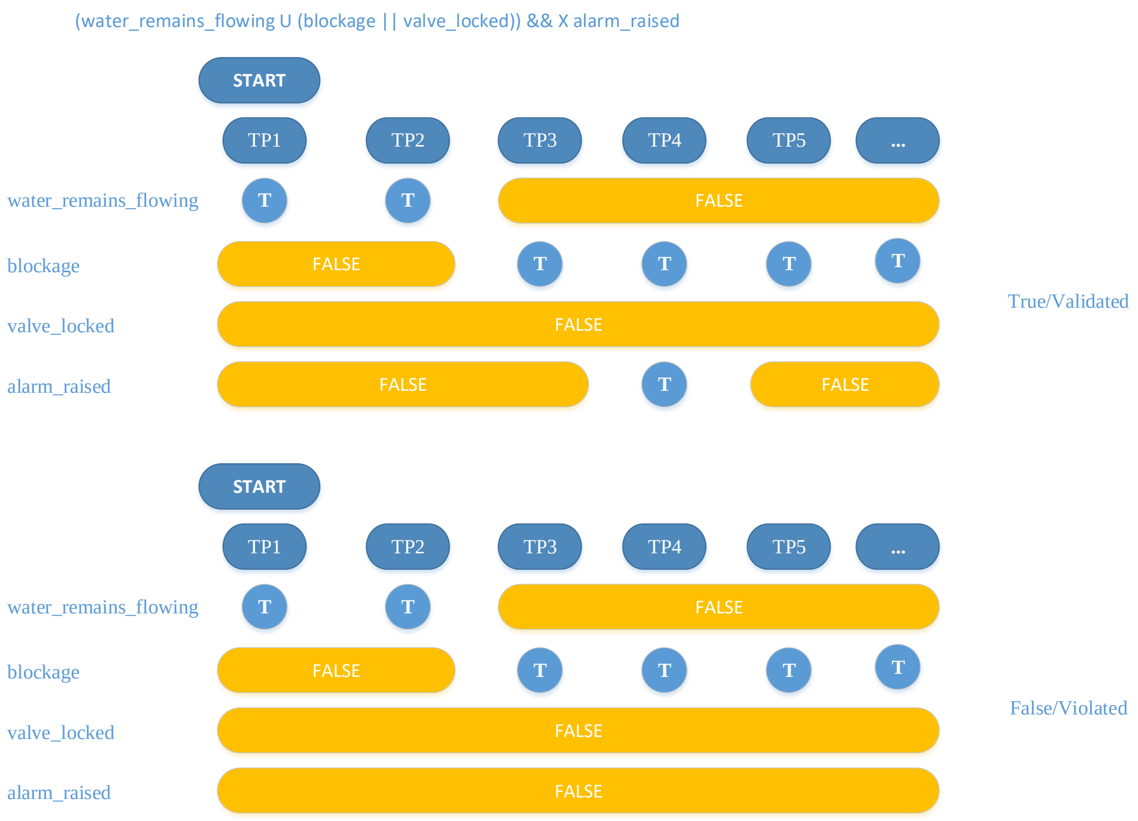


Figure 2.6: Full LTL formula for larger water flow example.

Ultimately, linear temporal logic is an ideal concept for ensuring a system functions properly, and it is a major contributing factor to tools such as JavaMOP. Linear temporal logic alone, however, does not provide the full solution to solving runtime verification problems. Other constructs are needed in collaboration with LTL to help ensure a reliable system. [11,12,13,14,15]

Section 2.3 Aspect Oriented Programming

Object oriented programming solved many organizational issues associated with procedural programming. Related methods and variables/properties are grouped together into a set object through encapsulation; along with concepts such as polymorphism and inheritance, code is structured in a logical and organized way. Object oriented programing alone, however, does not solve all organization issues, especially crosscutting concerns.

Some methods are designed to be used across multiple layers and sections of an application; when this usage spans across multiple sections of an application it ultimately affects those sections in one way or another. This is the core concept of cross cutting concerns. Any changes to such a method may clearly introduce many problems into a system, and centralizing it when possible would help reduce this problem. Imagine a system with 100 separate types of classes, and each class contained a variable called ID that must remain a specific value. If we want the system to log an error and attempt to reset it to the appropriate value every time the variable ID is changed during any point in execution. Since the variable ID may be modified in lots of different locations in different classes, implementing such requirement will need changes in multiple locations of multiple classes and causes redundant and spaghetti code. Aspect oriented programing was proposed to answer this challenging issue of crosscutting concerns. Rather than writing 100 separate logging and recovery methods, AOP allows programmers to implement the crosscutting concern in one location only, which makes development and maintenance much easier.

The basic concept in AOP is join points. A join point is a candidate point in the program execution of the application where a crosscutting concern can be plugged in. This point could be variables being modified, methods being called, and methods being executed. AOP takes full advantage of these join points. The basic idea is to perform some form of action before or after a given join point in the execution. At first glance, this seems like something doable in a standard OOD structure with a simple method call, but in situations where crosscutting concerns are an issue, messy code is inevitable without AOP. AOP allows for modularization of this base method by linking a pointcut in an aspect with a set of join points in the execution of this program. Only one method would have to be written, and it would be out of site from the primary code which separates secondary actions from primary ones. [16, 17]

Section 2.3.1 AspectJ Background

In order to include AOP concepts in Java programs, an appendage to Java called AspectJ was developed to provide developers all of the necessary AOP structures. AspectJ includes all the key pieces required for AOP such as aspects, pointcuts, and advice along with the allowance of standard Java code. AspectJ is simply Java with a couple of added syntax structures. AspectJ specific code goes into separate aj files and are later weaved into the primary system code during the final compile. AspectJ code can also be included in pre-compiled jar files. [16, 17]

Section 2.3.2 Join Points and Pointcuts in AspectJ

Various types of join points in a program's execution exist, and in AOP, the goal is to provide additional and modularized actions before and/or after multiple join points when needed. This is particularly useful during cross-cutting concerns, and in order to link these new actions, pointcuts in AspectJ provide a direct linking to one or more join points in the system where the AspectJ code will be weaved. Pointcuts essentially tell the AspectJ compiler the methods or variables we are interested in, and various options are available to restrict pointcuts to more specific points in execution of the system. For example, it might be desirable to have a pointcut for a method with the same signature across all classes except one. Using basic logic and AspectJ syntax, this can easily be achieved.

Various join points can be defined in AspectJ, but for our purposes the primary ones used in our work are: *call*, *set*, *cflowbelow*, and *execution*. In addition, AspectJ has operators like *within*, *args*, *this*, and *target* to impose restrictions on join points. *Call* and *execution* are the options that tells which method call/execution based join point you want to capture with the given signature, and *set* does the same except rather than pointing to a method, it looks at when a specific variable changes. (**See Figure 2.7**) If any method or variable linked by these options are called or changed, new code from AspectJ is also executed depending on other options included.

```
pointcut reqCredPC(Login loginObj) : call(public void Login.requestCred());
pointcut userNamePC(Login loginObj) : set(private String Login.userName);
```

Figure 2.7. Example call and set pointcuts from **Figure 2.5**.

Next, *cflowbelow* allows for a capture of the control flow of the join point set of a given pointcut, and *within* captures all join points in a given section of the main code. These options can be combined with various other options and logic to restrict the overall join point capture. For example, if a given method is called in two other methods but only one of those are needed for capturing, the within option will allow you to specify which section is preferred for the focus. As far as the AspectJ portion is concerned, one of these join points would be ignored even though the call signature matches. This structure clearly gives developers more power of the overall control flow of the join point captures.

The last main option set in an AspectJ pointcut are the *args*, *this,* and *target* options. The *args* option helps expose a reference to the given arguments from a method call/execution, and allows for concepts such pre-processing of a set of arguments. The operators *this* and *target,* on the other hand, aims to expose either the currently or target executing object. This helps give access to a full object during the advice execution. Given this object, advice execution could perform roughly any action that an instance of the object could accomplish. All public methods and properties are exposed for usage in this case. [16,17]

Section 2.3.3 Advice with AspectJ

Advice in AspectJ houses the direct action taken after a certain pointcut is triggered. Further ordering of the execution of the advice is set such as *after*, *before*, or *around* meaning that the advice will run after, before, or around the join points linked in the pointcut. (**Figure 2.8**) The *around* keyword represents both after and before, but this option is never used in my research. If the pointcut gives the advice access to things such as method arguments or a target object, then the advice may prove more useful depending on the application. Whether simple system logs, pre-processing, or some advanced method structures are included in the advice, it is still essentially just another function. [16,17]

```
before(Login loginObj): openDBConnPC (loginObj){
        log("Begin openDBConn");
}

after(Login userNamePC): reqCredPC (loginObj){
        log("userName set");
}
```

Figure 2.8. Example before and after advice.

Section 2.3.4 Aspect and AspectJ

Last, the aspect is the container that houses all of the AspectJ related code. Just like with Java classes, the aspect key word is used to show the type of structure. Aspects roughly follow the same general Java syntax in classes, however, rather than just bundling related methods and properties, the aspect groups pointcuts and their associated advice as seen in **Figure 2.9.** Unlike standard classes, however, aspects act similar to singletons. Aspects are not directly instantiated as individual objects, and they are not directly callable as a standard object. Methods and variables are declarable and usable within these aspects, but interaction between the main system classes and the aspect is structured somewhat as an observer. The main system performs its process without any direct knowledge of the aspect, while the aspect essentially listens for points of interest in execution and intervenes when needed by running separate advice.

Section 2.3.5 AspectJ/Java Example

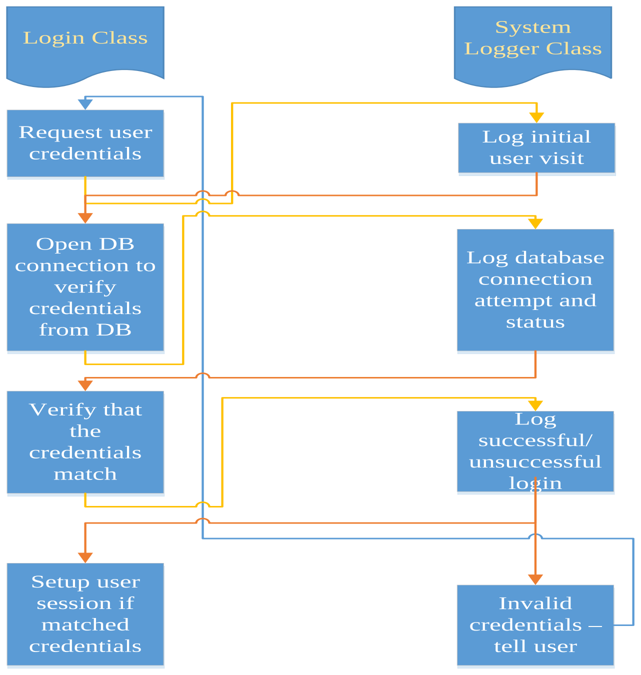

Login Class
System Logger Class
Request user credentials
Log initial user visit
Open DB connection to verify credentials from DB
Log database connection attempt and status
Verify that the credentials match
Log successful/ unsuccessful login
Setup user session if matched credentials
Invalid credentials – tell user

Figure 2.9. Crosscutting example – basic logging for a login system.

Through general object-oriented programming, the example from **Figure 2.9** would simply have an object for the login system and one for the system logger, and the login object would simply call logging methods from the system logger object as shown in **Figure 2.11**. Considering this example is small, the issues associated with this structure may not be as apparent as it should, so imagine if hundreds more additional object types were introduced into the system which all used the same system logger class. The primary objective of each object structure would be cluttered with excessive calls of the logger classes which in many ways diminish the quality of encapsulation and abstraction intended with object oriented programming. This is where aspect oriented programming comes in. Aspect oriented programming aims to separate out those pieces to alleviate the issues caused by cross-cutting concerns. Looking back at **Figure 2.9**, the login object would not directly call the methods from the system logger object in an aspect oriented structure. In fact, the system logger object would technically not exist as an object. It would exist as a separate entity in an aspect. To provide the linking between the two, join points from the login class need to be included in pointcuts within the aspect along with associated logging based advice. Clearly, this form of modularization allows this single aspect to intertwine itself with other classes without direct integration into the separate classes. With all of the system logging concerns separated from the other class, it is easier to maintain a fairly clean implementation of the main objective in the login class along with any other classes introduced in the future.

```
public class SystemLogger{
	public SystemLogger(){
		// ...
	}
	public void log(String msg){
		// Log message in file
	}
}
```

Figure 2.10. SystemLogger class for AspectJ/JavaMOP.

The original structure from **Figure 2.9** is still maintained but in a post-compile rather than cluttering source files. Various tools exist to accomplish this structure, but our primary choice due to its Java base and the easy integration with JavaMOP is AspectJ.

In our case, AspectJ provides a fairly straightforward approach to separating error checking and handling from the base system source files pre-compile. AspectJ has three base structures: pointcuts, advice, and an aspect. A full example of an aspect is given in **Figure 2.12** and is associated with the Login class presented from **Figure 2.9**. Initially, however, it is better to break it down into its individual pieces. [16,17]

```
public class Login{
        private String username;
        private String password;
        private SystemLogger sysLog;

        public Login(){
                sysLog = new SystemLogger();
                // Other constructor stuff...
        }
        public void requestCred(){
                // Enter method log
                sysLog.log("Begin requestCred");
                // Some other work...
                // End method log
                sysLog.log("End requestCred");
        }
        public void openDBConn(String dbName, String pass){
                //Enter method log
                sysLog.log("Begin openDBConn");
                // Some other work...
                // Enter method log
                sysLog.log("End openDBConn");
        }
        public boolean verifyCred(){
                // Enter method log
                sysLog.log("Begin verifyCred");
                //some other work...
                // End method log
                sysLog.log("End verifyCred");
        }
        public void setupSession(){
                // Enter method log
                sysLog.log("Begin setupSession");
                // Some other work...
                // End method log
                sysLog.log("End setupSession");
        }
}
```

Figure 2.11. Login class example.

```
public aspect LogAsp{
	private void log(String msg){
		//Log message in file...
	}

	// Trigger each time after the userName variable in the Login class is
	// changed
	pointcut userNamePC(Login loginObj) :
		set(private String Login.userName) && target(loginObj);
	after(Login loginObj): userNamePC (loginObj){
		log("userName set -- after");
	}

	// Trigger each time after the userName variable in the Login class is
	// changed
	pointcut passWordPC(Login loginObj) :
		set(private String Login.passWord)&& target(loginObj);
	after(Login loginObj): passWordPC (loginObj){
		log("passWord set -- after");
	}

	// Trigger each time (before and after) the requestCred method in the
	// login class is called
	pointcut reqCredPC(Login loginObj) :
		call(public void Login.requestCred())&& target(loginObj);
	before(Login loginObj): reqCredPC (loginObj){
		log("Begin requestCred -- before");
	}
	after(Login loginObj): reqCredPC (loginObj){
		log("End requestCred -- before");
	}

	// Trigger each time (before and after) the openDBConn method in the
	// login class is called
	pointcut openDBConnPC(Login loginObj,String dbArg,String passArg) :
		call(public void Login.openDBConn(String, String))
		&& target(loginObj) && args(dbArg, passArg);
	before(Login loginObj,String dbArg,String passArg):
		openDBConnPC (loginObj,dbArg,passArg){
		log("Begin openDBConn -- before");
	}
	after(Login loginObj,String dbArg,String passArg):
		openDBConnPC (loginObj,dbArg,passArg){
		log("End openDBConn -- after");
	}
```

Figure 2.12. Login Aspect for **Figure 2.11**.

Section 2.4 JavaMOP

JavaMOP is a monitor oriented programming tool centered around detecting and handling events and logic violations. JavaMOP is a Java based tool that ultimately compiles to AspectJ and is intended to simplify various runtime verification needs. It borrows the concepts from AspectJ too in its structure, but rather than pointcuts, advice, and aspects, JavaMOP uses a similar structure with a different naming. It includes monitors, events, logic statements, violation handlers, and various other options, and it remains in a modularized structure independent of the main system code pre-compile.

Like an aspect, a monitor is the main container for all event handlers needed for the specific runtime verification application property. It is also essentially treated as a singleton. As shown in **Figure 2.13**, no key words directly exist to signal that the structure is a monitor. Simply providing the name of the monitor along with roughly the standard Java method syntax, is enough to tell the JavaMOP compiler that it is a monitor; a key word is not used since it is directly implied being in a "*.mop" file too. A monitor also represents the grouping of events for a section of the state checking. Multiple monitors are needed to cover the entire state diagram structure.

Furthermore, typically the largest portion of the monitor are the events. Events are ultimately the same as the pointcuts and advice used in AspectJ, but they are a compact combination of the two for easier and cleaner development. The "event" keyword is used to signal a new event to be defined. Included with this is the name of the event along with before or after as in an advice and the options for the pointcut of the event as shown in **Figure 2.12**. Like the pointcut/advice structure, if the event is triggered based on the given options, the advice of the event is executed. Events in monitors also go a step further than in AspectJ. When events are triggered, the monitor internally tracks this event sequence which allows for further logic application such as linear temporal logic.

JavaMOP monitors allow for various forms of logic handling of events, but in this research, we are specifically focused on the usage of linear temporal logic within JavaMOP. As shown in **Figure 2.13**, defining an LTL definition for the monitor is straightforward. Simply using the key word “ltl:” followed by the LTL statement is all that is needed to apply LTL logic to the monitor. The LTL statement used is centered around the triggered events in the monitor; and the event names are used throughout the LTL statement to assist with RV purposes. Given that the result of an LTL statement boils down to a simple true or false, JavaMOP also provides a method to handle any false returns from the LTL. The key word used to handle this is “@violation”. When a violation is triggered, the insinuation is that the event sequence does not match the expected sequence, and error handling should take over at this point to ensure the correct performance of the system.

```
import java.io.*;

import java.util.*;


LogMonitor(Login loginObj){

        Login loginObj; // Reference to Login object

        private log(String msg){

                // Log message in file...

        }


        event usernameEvent after(Login loginObj): set(private String Login.username){

                log("username set");

        }

        event passwordEvent after(Login loginObj): set(private String Login.password){

                log("password set");

        }

        event reqCredEvent before (Login loginObj):call(public void Login.requestCred()){

                log("Begin requestCred");

        }

        event openDBConnEvent before(Login loginObj) : call(public void
Login.openDBConn(String, String))

        && args(dbArg, passArg){

                log("Begin openDBConn");

        }

        event verifyCredEvent_true after(Login loginObj) returning (boolean res):

                call(public boolean Login.verifyCred()) && condition(res){

                        log("After verifyCred -- True");

        }

        event verifyCredEvent_false after(Login loginObj) returning (boolean res):

                call(public boolean Login.verifyCred()) && condition(!res){
                    log("After verifyCred -- False");
        }
        event setupSessionEvent before(Login loginObj): call(public void Login.setupSession()){
                log("Begin setupSession");
        }
        ltl: setupSessionEvent =>(*)verifyCredEvent_true
        @violation{
                // Perform recovery or just log the issue
                __RESET;
        }
}
```

Figure 2.13. JavaMOP example – roughly represents the AspectJ example

In some cases, it is appropriate to simply report the error triggered in logging, but it is always best at this point to provide a method to do everything possible to ensure the system remains in a stable state. A decent comparison would be to the try-catch structure implemented in various languages. If a problem occurs, try to fix it if possible, and utilize the knowledge of the error reported to handle the specific problem. [18,19,20]

Section 2.5: LeJOS – Lego EV3

In order to test the runtime verification structure, it is important to have an actual system to apply the RV centered code to. In this case, we used a soccer playing LEGO EV3 robot with a Java based API called LeJOS. LeJOS is an open-source API that provides a way to control and use the robot's sensors and motors. Unfortunately, it is only in beta version 0.9.1, but it works well enough for our testing purposes with RV. The LEGO EV3 robots have many sensor and motor options, but in our case we focus on three separate sensors and three motors. The sensors: inferred, sonar, and compass all have their own classes as shown in **Table 2.3**, and the motors: two large motors and a small arm motor have their own associated classes. [21]

Table 2.3: LeJOS sensor and motor Java classes.

Sensors/Motors	**Class**
Sonar Sensor	EV3UltrasonicSensor
IR Sensor	HiTechnicIRSeekerV2
Compass Sensor	HiTechnicCompass
Arm Motor	UnregulatedMotor
Movement Motors	RegulatedMotor

Sensor initialization in LeJOS has a basic setup. Each sensor is connected to a port on the EV3 brick as shown in Figure 16, and each sensor has an associated port number used by the sensor constructors. Four sensor ports labeled SensorPort.S1 to SensorPort.S4 help the sensor classes link to the sensor hardware for value pinging. Once the sensor is linked to the proper class object, the sensor object will provide a SampleProvider object based on the requested sensor mode provided by the sensor classes. Some sensors have multiple modes; for example, the inferred sensor has both an unmodulated and modulate IR mode, and the HiTechnicIRSeekerV2 will provide two separate SampleProvider objects. Given the SampleProviders, calling the fetchSample(float[] sample, int offset) method of the SampleProvider will populate an array of sample reads from the sensors. With this data along with a few other separate properties, the robot is able to determine its current state.

Next, LeJOS links motors to ports similar to how sensors are handled in the initialization phase. Motors, however, have their own four ports listed as MotorPort.A to MotorPort.D . The classes used for the two types of motors are listed in **Table 2.3**. After initialization, the newest version of LeJOS attempts to match the class structure to the hardware structure as close as it can with a chassis structure as shown in Figure 16. Most of these supporting objects are purely used during the initialization phase. With this chassis structure, a MovePilot is created, which is ultimately another supporting object to the main Navigator. The MovePilot helps the Navigator interact with the motors and objects initialized in order to traverse waypoint and grid motion. Any usage of the motors after initialization is, for the most part, only used through the Navigator which abstracts a large amount of detail to ensure ease of use through waypoint tactics.

The Navigator is one of the most important objects for our robot. Internally, the Navigator tracks the current position of the robot based on a Cartesian coordinate grid; this abstracts out the tedious mathematics required to track the robot’s position. Unfortunately, as of LeJOS version 0.9.1, the CompassPilot has been deprecated for no clear reason, and it is unclear if it will be added back in some way in future updates. Luckily, the MovePilot is a good enough backup; it looks purely at the tachometer readings from the large motors attached to the main wheels of the robot. The tachometer simply provides the MovePilot with revolutions or the wheels per minute to track general movement for the Navigator. Utilizing this information, the Navigator is capable of tracking the overall position returned by its PoseProvider object. [9, 21]

While LeJOS is a powerful API, it does have its issues due to its beta state. Other than the deprecated CompassPilot, the intended usage structure of the API is not as well documented as it would be in a non-open source API. For example, it is possible to over-ping a sensor without any warning; rather than provide an internal delay, the API assumes that the user will provide a delay between pings to avoid a read of invalid values. Furthermore, the Navigator does not follow the 0 to 360 degree turn values with the Navigator as it does with a raw compass reading; it goes on a 0 to 180 and -180 to 0 degree turn reading. Also, the inferred sensor API only reads in 30 degree increments which can result in inaccurate readings depending on the location of the IR soccer ball. These problems in combination with common surrounding interference and a lack of detailed API documentation adds to the challenge of setting up this testing environment, but these problems are expected in an open-source beta API. Thanks to trial and error, a decent portion of these issues are non-existent, but it turns out that some of these naturally occurring issues makes some of the runtime verification testing easier since the goal of this research is to attempt recovery during error.

Given that LeJOS is just a Java based API, it is no different than writing any other Java program. It is more backend oriented rather than frontend and graphics, but no true language extensions such as what is included in AspectJ and JavaMOP are made for LeJOS. Overall, LeJOS is simply a set of Java classes that help developers interact with the robot.

CHAPTER III

FRAMEWORK TO MONITOR BEHAVIOR OF ROBOTICS AT RUNTIME

Section 3.1: Architecture of the Framework

As a cyber-physical system, robotic systems are becoming more and more popular, and used in variant occasions such as warehouses, manufactures, and military battle fields. However, robotics systems are also fragile systems that inevitably experience some form of failure during runtime due to hardware failures and hostile environments. These failures range from an unexpected change in environment to a complete system crash. Manufacturing companies, the military, and various other fields rely on continuously functioning systems; even one error could lead to loss of profit in a worst-case scenario loss of life. Regardless of how minor or major the problem may be, it is important to detect and recover from these failures at an early stage so that it can continue performing predefined missions. However, it is challenging to develop a "perfect" system that we can have high confidence in it, especially due to the mixing of error detection and handling with normal system code and crosscutting concerns.

In this thesis, we proposed an approach to increase our confidence in the robotic system by monitoring robotics behavior when it is working on its mission. The monitor is separated from functionality code of the robotic system, therefore improves the modularization of the system. In addition, the monitor allows robotics to detect behaviors that deviates from missions defined in a state diagram at an early stage so that actions can be taken to continue its predefined mission.

The overall architecture of our approach is shown in **Figure 3.1**. Robotic systems are implemented in LeJOS to achieve its mission. A state diagram is developed to define the behavior of a robotics by specifying its states and events, actions, or conditions that trigger the change of states. From this state diagram, system properties can be defined as LTL formula to describe allowed behavior. Any violation of these properties indicates an error or a deviation from predefined behavior. For example, properties can specify the allowed sequences of states when it performs its mission. It can also specify the set of events that lead to a specific state. All these properties are specified based on the state diagram only. It is not related with the implementation of the robotic system.

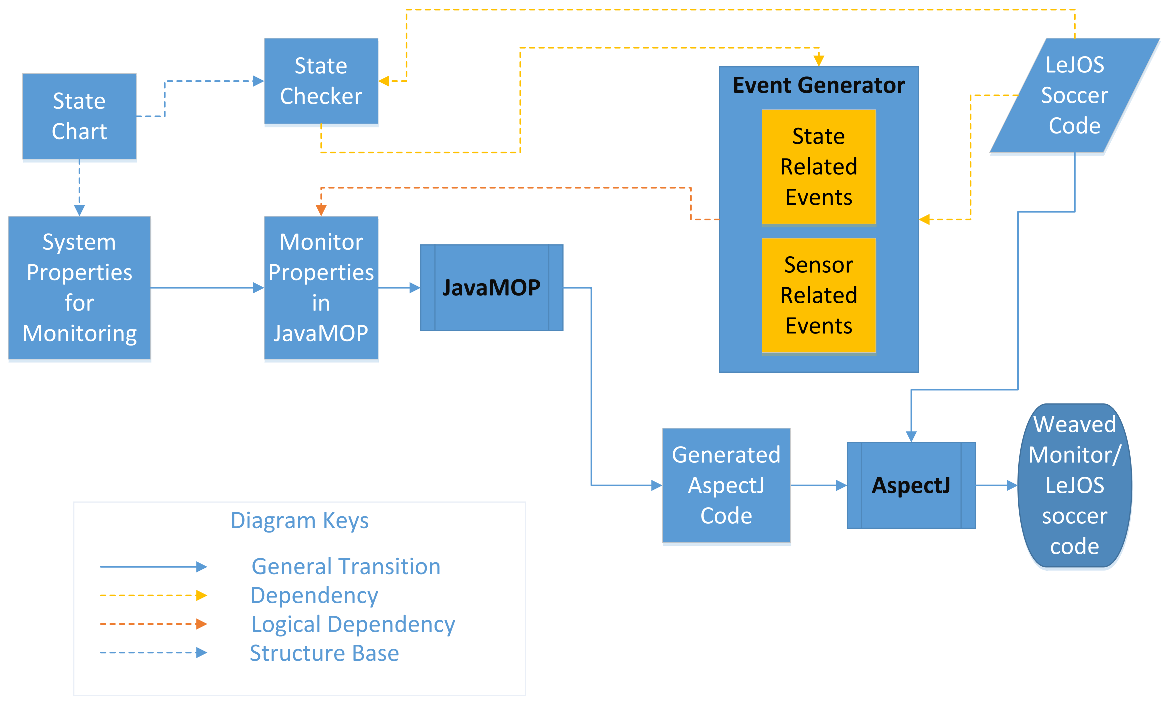


Figure 3.1. Architecture of our framework

The core components of our approach in **Figure 3.1** are: state checker, event generator, and monitor properties in JavaMop. The state checker component is used to decide the current state of the robotics at runtime based on the state diagram and the implementation of the robotic system. The event generator component is designed to generate events that are used to define properties in JavaMop. Properties defined in JavaMop is built on events, and each event is defined as a pointcut in AspectJ to represent an action or an occurrence recognized by AspectJ. Such events should not be generated by robotic system since we want to separate functionality code from monitoring code, i.e. non-functionality code. Therefore, event generator component is essential in our framework. It extracts events from robotic system through monitoring code that is weaved into functionality using AspectJ.

Overall, both state checker and event generator components are integrated into the runtime verification system as a layer between the robot's source code, and the main JavaMOP event and LTL code. These components follow the state diagram structure to ultimately generate individual events. These events are used to develop LTL to help monitor the system behavior in the separate JavaMOP code. Both components and JavaMOP code are ultimately weaved together into robotic systems through the AspectJ compiler leaving one simple jar file ready for execution on the robotic system.

Our approach organizes the system code into a cleaner and more modularized structure. Error detection and handling code is separated from the code of base system functionality. Provided that a developer follows this framework, it should be clear which parts of the source code is part of runtime verification and which parts are primary system functions.

To illustrate different components in our framework, we introduce a simple case study of a soccer scenario. In the soccer field, there is only one LEGO soccer robot, one ball, and one gate. The soccer robot will first search the ball. After finding the ball, the soccer robot will dribble the ball towards the gate, and shoot the ball when it is close enough to the gate.

The rest of the chapter is organized as follows: Section 3.2 discusses the state diagram and implementation of the soccer robot. Section 3.3 presents the state checker component to facilitate event generation. Section 3.4 discusses the event generator that extracts events needed for monitor from the soccer robot. Section 3.5 shows how to specify behavior properties and construct monitors from it. Section 3.6 gives instruction to weave different components into one executable file.

Section 3.2: Soccer Robot

An individual soccer robot is developed to illustrate the framework and demonstrates the benefits. The mission of the soccer robot is to find the ball, dribble the ball to the gate and shoot when it is close enough. The Lego EV3 robots in combination with the Java based LeJOS API provides the perfect system set to demonstrate this approach. Considering the beta state of LeJOS and the sensitive nature of sensors included in the Lego EV3 kit, errors happens constantly, and they give an ample amount of testing possibilities for this innovative approach of runtime verification of state diagrams in robotics.

Section 3.2.1: State Diagram of Soccer Robot

The soccer robot's task is to find the ball and attempt a single goal shot. As seen in **Figure 3.2**, eight possible states and twelve possible state transitions exist in the state diagram. The initial state is *Init Motors/Sensors* and the final state is *Game Over*. The robot must follow this diagram strictly. Any improper jump from one state to another state that are not represented in the diagram should be flagged as an error and should be handled immediately.

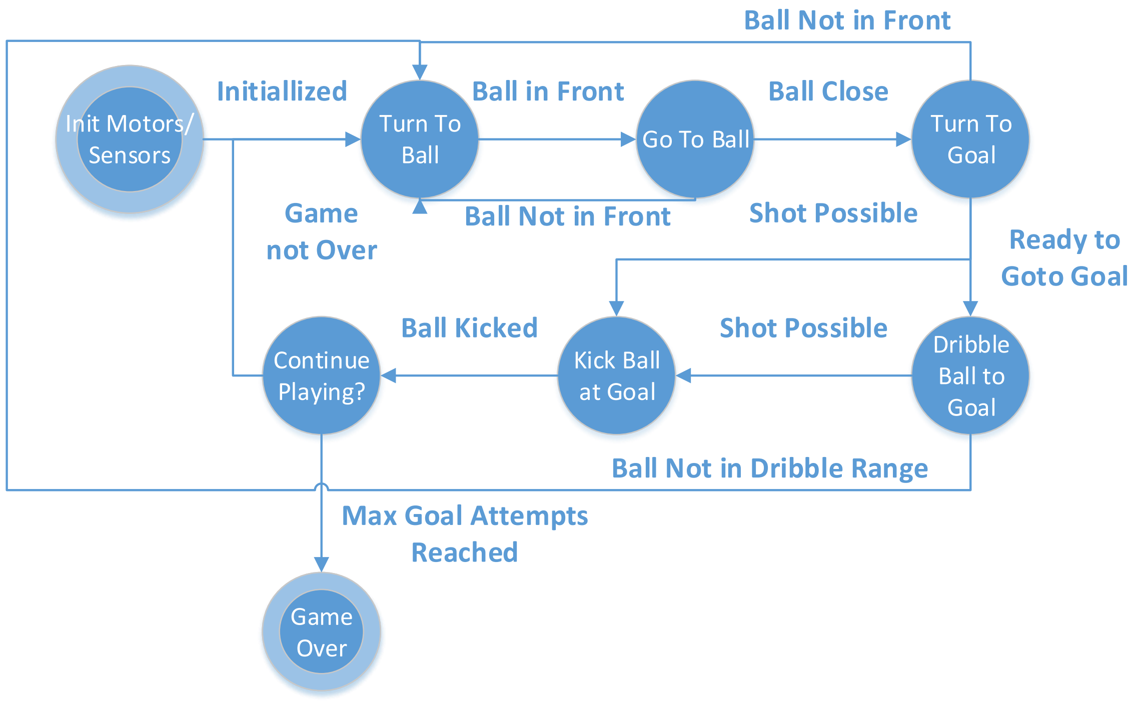


Figure 3.2. State diagram of the LEGO soccer robot

Initially, the robot and all its links to sensors and motors are setup and upon success starts the soccer playing process. From there, the robot must look for the ball using a combination of its I.R. and sonar sensors to detect the location of the I.R. emitting ball. When the ball is found, the robot should move forward to the ball until it is within the robot's arms or the ball is no longer in front. If the robot has the ball, it should find and go to the goal while dribbling the ball, and if at any point the robot no longer has the ball, it must re-search for it. When the robot is within range of the goal, it must kick the ball to the goal and then go back to looking for the ball if the current test allows for multiple attempts. This entire process is viewable in **Figure 3.3 to Figure 3.8.**

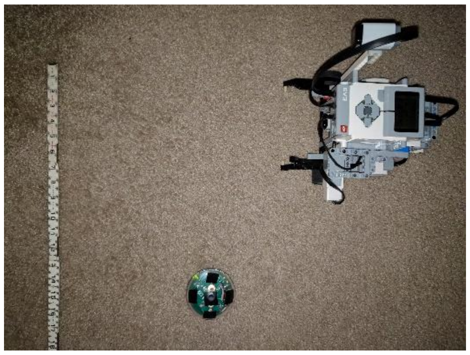


Figure 3.3. Init Motors/Sensors and Turn to Ball states

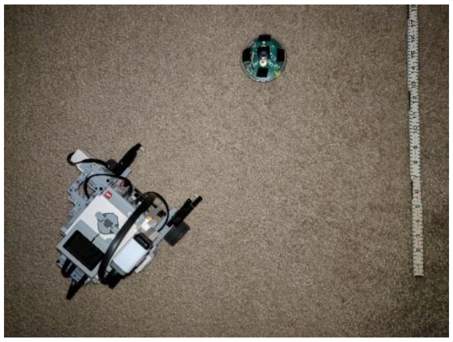


Figure 3.4. Turn to Ball

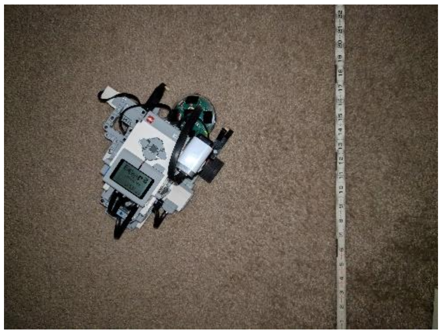


Figure 3.5. Go to Ball state

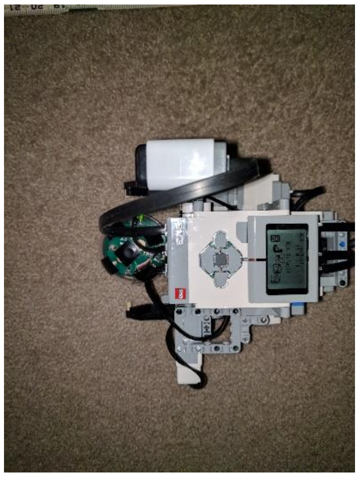


Figure 3.6. Turn to Goal state.

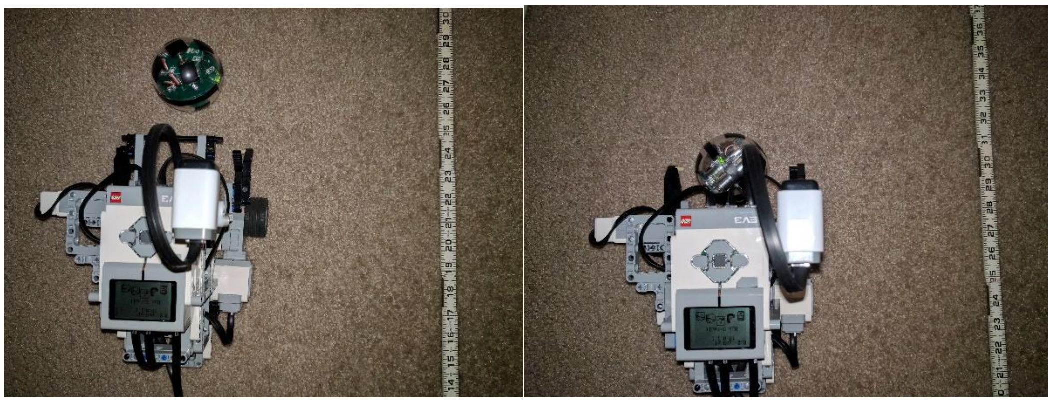


Figure 3.7. Dribble Ball to Goal

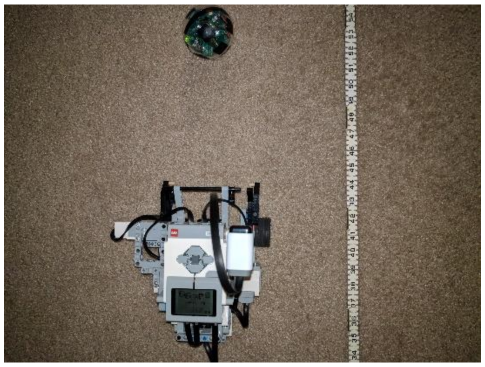


Figure 3.8. Kick Ball at Goal

Section 3.2.2: Implementation of the Soccer Robot

As with all robotic systems, a combination of sensors, motors, and software make up the entire soccer playing robot. The sensors and other components are standard Lego and Hitechnic brand hardware, and the software used for development is a third-party Java API called LeJOS. As shown in a frontal view of this robotic system in **Figure 3.9**, this robot can detect an I.R. ball along with loosely grabbing it within the side arms.

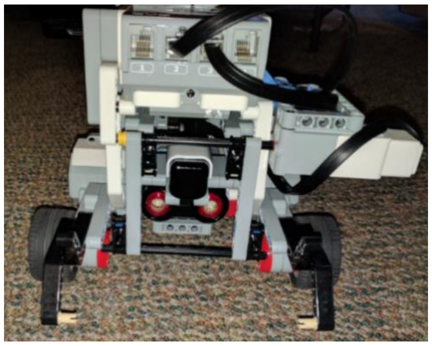


Figure 3.9. EV3 soccer robot.

Various sensors are available with the Lego EV3 robot bundles, but only a few are necessary for this soccer robot. As shown in **Figure 3.10**, these sensors include I.R.

(both modulated and unmodulated), sonar, compass, two large tachometer-equipped motors, and one small arm motor. These sensors and motors work together to ultimately allow the system to play soccer. Together, they form a complex system, but understanding the details of the individual pieces is important.

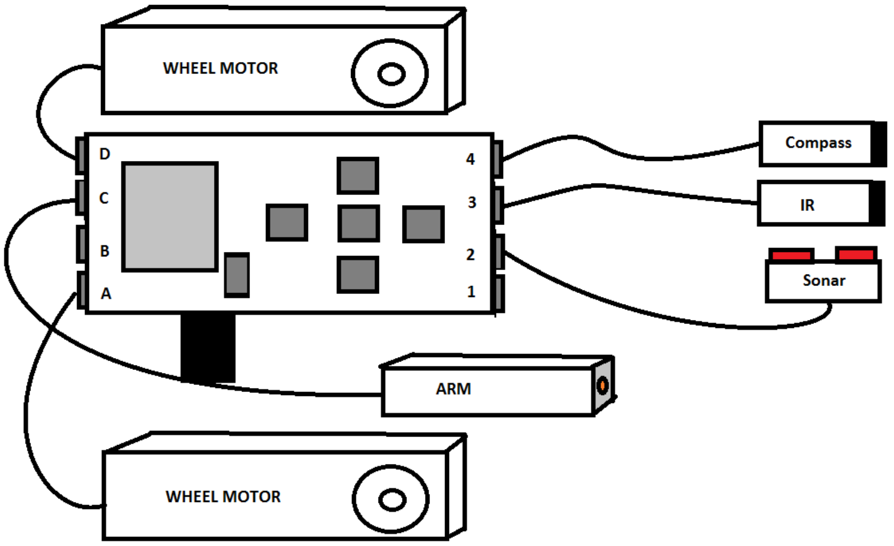


Figure 3.10. Sensor/motor placement and structure of the general soccer robot.

First, basic 360-degree motion is made possible with the two main motors up front along with a pivot point marble in the back. Speed, acceleration, and direction are all directly manageable on these motors through the Navigator class; additionally, these motors partially track the system's position through internal tachometers. The smaller motor contains comparable properties but lacks the same power level. Even with lower strength, the smaller motor in combination with the momentum of the whole robot is more than enough to kick the ball, so it is sufficient for our purposes as an arm.

Next, the inferred sensor is by far the most important part of the robot for this testing environment. The I.R. soccer ball emits an I.R. signal in all directions, and with the I.R. sensor the robot can detect a rough angle of the location of the ball (**Figure 3.12**) in relation to the sensor as shown in **Figure 3.11**. If the ball is not within the 270 degree I.R. span, the robot simply turns until a signal is found. This sensor also has two modes called modulated and unmodulated I.R. Modulated I.R. helps reduce interference such as florescent lighting, however, it sometimes has trouble finding the ball from a longer distance. Unmodulated I.R. on the other hand drastically increases the range of the sensor, yet it decreases accuracy of the angle. When combined, these two I.R. readings make up for each other's weaknesses and provides a relatively ideal I.R. reading in many situations and states.

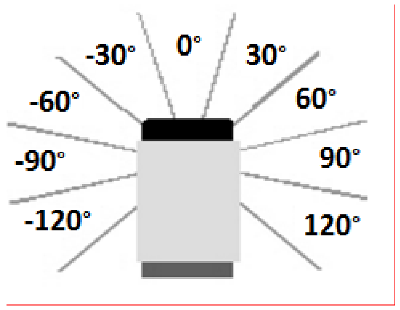


Figure 3.11. IR Sensor angle detection (Borrowed from Hitechnic and modified).

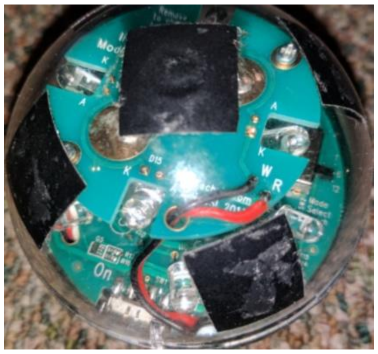


Figure 3.12. IR emitting soccer ball.

In addition to I.R., this robot utilizes a sonar sensor to track the ball location. When the ball is very close to the robot, the I.R. sensor typically fails, and the robot must rely on the sonar to indicate if it has the ball. Sonar is also able to detect other objects such as walls in order to avoid them. In combination with the I.R. sensor, the robot can improve ball location detection accuracy by simply combining the values from all sensors into basic logic.

With the combination of these sensors and motors, this robot has the ability to perform all soccer related tasks required by the state diagram. Hardware alone, however, is not all that is needed. A key component of a robotic system is the software component, and the LeJOS API makes it easier to develop software to control soccer robot's behavior. With the help of the LeJOS API, developing software that interacts with the Lego EV3 hardware is relatively straightforward. In the soccer robot, five classes were developed to dictate the actions of the soccer robot: SoccerGlobals, MotionControl, SensorControl, PlayerInit, and Kicker as shown in **Figure 3.13**. In this case, only one instance of each object exists per robot, and each has their own purpose.

First, SoccerGlobals is not exactly a standard object, but it provides a direct way to share constant properties across all the other classes such as the goal location. Only constant variable properties are contained within this class and nothing more. Next, the two core classes of the robot are MotionControl and SensorControl. MotionControl provides an abstracted wrapper around the navigation tools of the LeJOS API. Direct soccer robot related methods help hide the underlying calls to lower level motor movements, which provides a clearer representation of the actions of this specific robot. The class SensorControl on the other hand, only focuses on sensor related tasks such as reading the location of the ball. The SensorControl class is ultimately a higher-level view of sensor readings and usage.

The Kicker class integrates both SensorControl and MotionControl classes to ultimately play the soccer game from a kicker's perspective. It utilizes the higher-level methods provided within both classes to interact with underlying hardware, sensors, and drivers. All specific properties to the given robot structure are setup here too. Furthermore, the PlayerInit class simply initializes the Kicker class or the Goalie class depending on the type of robot selected by the user.

With these classes, the robot can play a simple game of soccer with modularized code and abstracted views of hardware control. This setup also makes it easier and cleaner to develop runtime verification structures. The abstract view allows for cleaner mapping to pointcuts and events and ultimately eases development as modularization was designed to do.

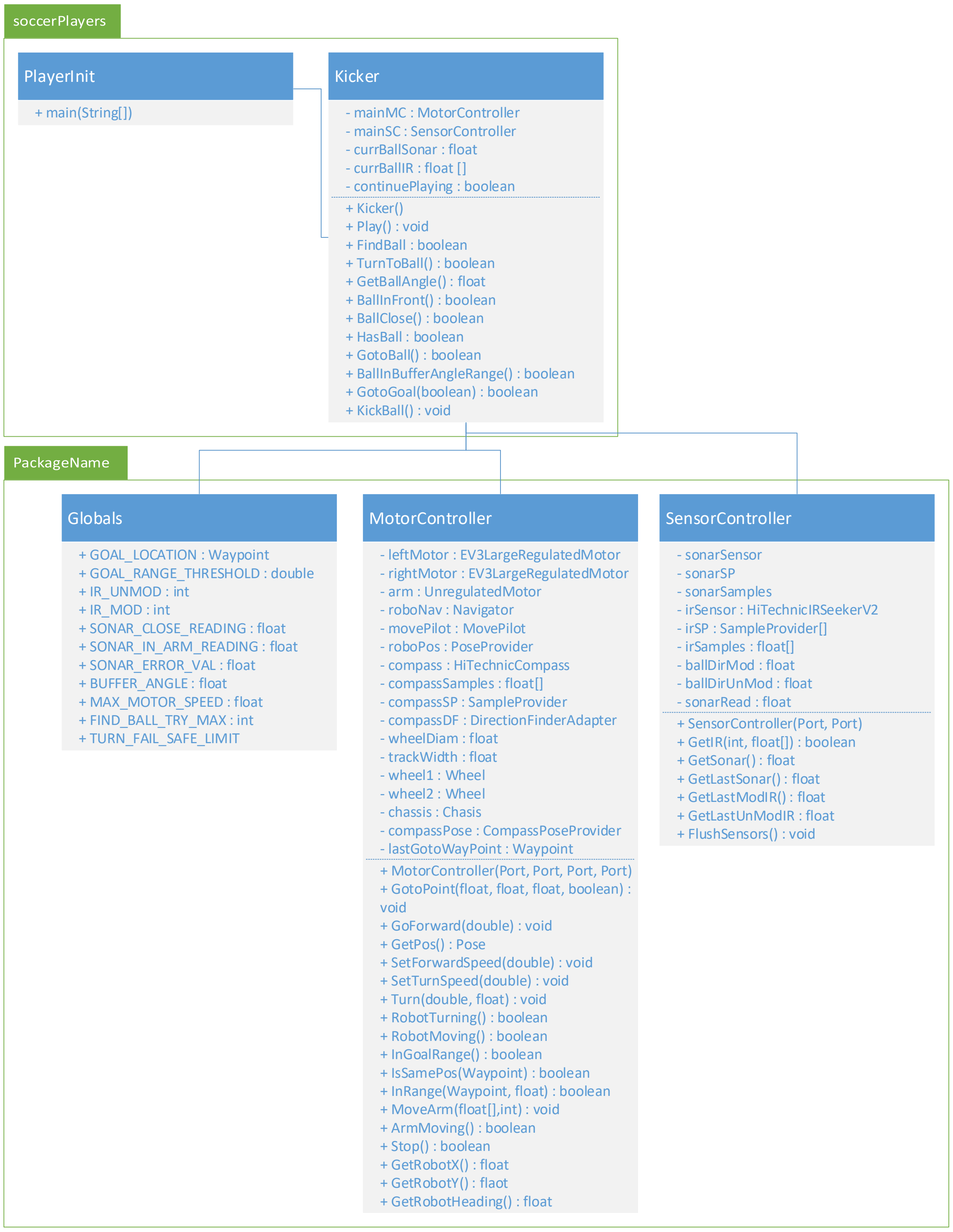

soccerPlayers
PlayerInit
+ main(String[])
Kicker
- mainMC : MotorController
- mainSC : SensorController
- currBallSonar : float
- currBallIR : float []
- continuePlaying : boolean
+ Kicker()
+ Play() : void
+ FindBall : boolean
+ TurnToBall() : boolean
+ GetBallAngle() : float
+ BallInFront() : boolean
+ BallClose() : boolean
+ HasBall : boolean
+ GotoBall() : boolean
+ BallInBufferAngleRange() : boolean
+ GotoGoal(boolean) : boolean
+ KickBall() : void
PackageName
Globals
+ GOAL_LOCATION : Waypoint
+ GOAL_RANGE_THRESHOLD : double
+ IR_UNMOD : int
+ IR_MOD : int
+ SONAR_CLOSE_READING : float
+ SONAR_IN_ARM_READING : float
+ SONAR_ERROR_VAL : float
+ BUFFER_ANGLE : float
+ MAX_MOTOR_SPEED : float
+ FIND_BALL_TRY_MAX : int
+ TURN_FAIL_SAFE_LIMIT
MotorController
- leftMotor : EV3LargeRegulatedMotor
- rightMotor : EV3LargeRegulatedMotor
- arm : UnregulatedMotor
- roboNav : Navigator
- movePilot : MovePilot
- roboPos : PoseProvider
- compass : HiTechnicCompass
- compassSamples : float[]
- compassSP : SampleProvider
- compassDF : DirectionFinderAdapter
- wheelDiam : float
- trackWidth : float
- wheel1 : Wheel
- wheel2 : Wheel
- chassis : Chasis
- compassPose : CompassPoseProvider
- lastGotoWayPoint : Waypoint
+ MotorController(Port, Port, Port, Port)
+ GotoPoint(float, float, float, boolean) : void
+ GoForward(double) : void
+ GetPos() : Pose
+ SetForwardSpeed(double) : void
+ SetTurnSpeed(double) : void
+ Turn(double, float) : void
+ RobotTurning() : boolean
+ RobotMoving() : boolean
+ InGoalRange() : boolean
+ IsSamePos(Waypoint) : boolean
+ InRange(Waypoint, float) : boolean
+ MoveArm(float[],int) : void
+ ArmMoving() : boolean
+ Stop() : boolean
+ GetRobotX() : float
+ GetRobotY() : flaot
+ GetRobotHeading() : float
SensorController
- sonarSensor
- sonarSP
- sonarSamples
- irSensor : HiTechnicIRSeekerV2
- irSP : SampleProvider[]
- irSamples : float[]
- ballDirMod : float
- ballDirUnMod : float
- sonarRead : float
+ SensorController(Port, Port)
+ GetIR(int, float[]) : boolean
+ GetSonar() : float
+ GetLastSonar() : float
+ GetLastModIR() : float
+ GetLastUnModIR : float
+ FlushSensors() : void

Figure 3.13. Class diagram of soccer playing robot.

Section 3.3: State Checker Component

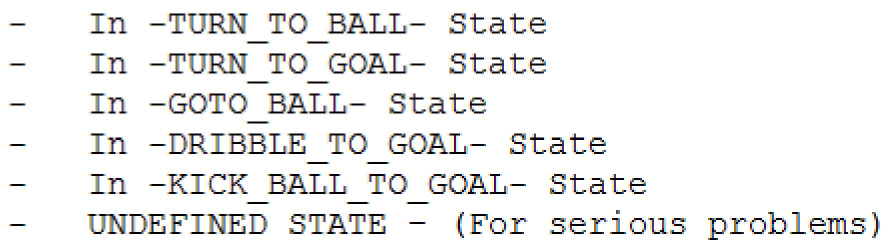
The purpose of the state checker component is to decide the current state of the soccer robot based on the values reading from various sensors and status of the motor. The implementation of the soccer robot doesn't track its state change since it is irrelevant to its mission of playing soccer. Therefore, being able to check the current state of the system is crucial to monitor its behavior based on the state diagram. The state checker component provides key helper functions to achieve this.

In the state checker component, two enum classes as shown in **Figure 3.14** are defined: State and ChangeEvent. The State enum class explicitly specifies all possible states allowed in the soccer robot; and the ChangeEvent enum class defines flags for sensors. Each flag indicates the current value from the corresponding sensor is different from previous reading. The ChangeEvent class could be used to help directly notify the state checker what has changed most recently. At the moment it is not fully utilized in this implementation, but some cases do require it.

```
package stateTools;

// State status options
public enum State {
      INIT, TURN_TO_BALL, GOTO_BALL, TURN_TO_GOAL, DRIBBLE_TO_GOAL,
      KICK_BALL_TO_GOAL, GAME_OVER
}

package stateTools;

public enum ChangeEvent {
      NONE, IR_MOD, IR_UNMOD,SONAR
}
```

Figure 3.14. State and ChangeEvent enum classes

The most important portion of the state checker component is the StateCheck class in StateCheck.java. This class provides a way to check the current state of the soccer robot based on the state diagram from **Figure 3.2**, and **Figure 3.15** gives a full list of states of the soccer robot that we want to monitor.



```
-    In -TURN_TO_BALL- State
-    In -TURN_TO_GOAL- State
-    In -GOTO_BALL- State
-    In -DRIBBLE_TO_GOAL- State
-    In -KICK_BALL_TO_GOAL- State
-    UNDEFINED STATE - (For serious problems)
```

Figure 3.15. Possible robot states from StateCheck.

In this class, only two methods are intended for use directly outside of the StateCheck class: GetState and PrintState. As the name implies, GetState() methods returns the current state of the soccer robot, and PrintState() method print out the current state. These two methods together facilitate the mapping in the event generator component from the robot's partial state into a full state associated with the state diagram. It is also helpful to log state information for viewing during runtime.

As shown in **Figure 3.16**, the GetState method takes in a ChangeEvent and Kicker object, and returns a State object. The Kicker object is the primary object controlling the physical actions of the soccer robot, and a reference to it is passed to this method to expose the necessary methods to gather current sensor data. This sensor data is collectively analyzed to determine the current state of the robot, and the resulting analysis generates a state to return back to the event generator for further processing.

There are six main properties that this method aims to gather to determine the state of the robot.

```
public static State GetState(ChangeEvent bifTriggered, Kicker
currMK){

            sonarRead = currMK.getSensorControl().getLastSonar();
            if(bifTriggered != null)
                  ballInFront = true;
            else
                  ballInFront = false;

            // NOTE: this assumes that sonar is up to date
            ballClose = BallClose(sonarRead);
            ballKickable = BallKickable(sonarRead);


            inGoalRange = currMK.getMotionControl().inGoalRange();
            robotMoving = currMK.getMotionControl().robotMoving();
            robotTurning = currMK.getMotionControl().robotTurning();


            boolean inTurnToBallState = TurnToBallState(currMK);
            boolean inGotoBallState = GotoBallState(currMK);
            boolean inTurnToGoalState = TurnToGoalState(currMK);
            boolean inDribbleBallState = DribbleBallState(currMK);
            boolean inKickBallAtGoal = KickBallAtGoal(currMK);

            if(inTurnToBallState)
                  return State.TURN_TO_BALL;
            else if(inGotoBallState)
                  return State.GOTO_BALL;
            else if(inTurnToGoalState)
                  return State.TURN_TO_GOAL;
            else if(inDribbleBallState)
                  return State.DRIBBLE_TO_GOAL;
            else if(inKickBallAtGoal)
                  return State.KICK_BALL_TO_GOAL;
            else
                  return State.INIT;

      }
```

Figure 3.16. GetState method

The first, and most important boolean flag from the StateCheck class in the GetState method from **Figure 3.16** is the ballInFront. This is required knowledge across most of the states, and it is crucial that this is accurate. A read of zero degrees on either of the IR pings or a “close” reading from the sonar will set this property as true. Furthermore, these sensors are at a high risk of receiving invalid values due to oversampling in a short period of time, which is what this overall structure aims to prevent. The solution to this issue results in technically old values obtained through GetState. Ideally, the sensors could be re-pinged at any given time, but in this case the most recent pings are used, and the event generator helps ensure that the last pings were performed as close as possible to the call to GetState. If everything runs smoothly, the time difference is mostly negligible considering the realistic speed of the ball and robot.

The remaining boolean flags are ballClose, ballKickable, inGoalRange, robotMoving and robotTurning. These flags are used to indicate the soccer robot’s status as well as the status of the external environment. All these flags should be update through the parameter Kicker class. As the name implies, ballClose indicates if ball is close enough to the soccer robot. The flag ballKickable indicates if the soccer robot can kick the ball. Both ballClose and ballKickable require a ballInFront status along with a specified closeness distance based on the sonar. The flag inGoalRange indicates if the soccer robot is within a certain distance from the goal, which then triggers the kick to the goal. The flag robotMoving and robotTurning indicate if the soccer robot is in the process of moving or making turns, respectively.

Lastly, the other methods in StateCheck directly provide a flag for which state the robot is in and all are checked until the correct state is determined. These include: TurnToBallState(…), GotoBallState(…), TurnToGoalState(…), DribbleBallState(…), KickBallAtGoal(…) which are directly called for the current state check. These functions are at the core of the event generator. For example, as shown in **Figure 3.17**, the TurnToBallState function looks at properties associated with being in a turn-to-ball state. In this state, the ball should not be in front, close, or kickable. This is simply checking the current state of the robot to report back to the overall requestor which in most cases is the RobotStateMachine aspect. The nested if statement as shown in **Figure 3.18** from the GetState function reports the actual state of the robot, and the significance of the ordering is not necessarily important in many cases. However, this ordering gains importance when handling unusual behavior from the robot. Since each test for a state is performed sequentially, a very small window of time exists for the robot to shift its state during the check, meaning the appearance of being in multiple states at once is possible. Setting the precedence for earlier states in the state diagram during testing appeared to significantly improve reliability of state reporting; this makes sense partly because the probability of the robot being in a given state in our case decreases as it continues to the end of the flow of the state chart. The return of State.INIT is the only exception to this since it should occur only once at most, so it effectively functions as an error flag.

```
public static boolean TurnToBallState(Kicker currMK){
        if(!ballInFront && !ballClose && !ballKickable)
                return true;
        else
                return false;
}
```

Figure 3.17. TurnToBallState method in StateCheck

```
if(inTurnToBallState)
        return State.TURN_TO_BALL;
else if(inGotoBallState)
        return State.GOTO_BALL;
else if(inTurnToGoalState)
        return State.TURN_TO_GOAL;
else if(inDribbleBallState)
        return State.DRIBBLE_TO_GOAL;
else if(inKickBallAtGoal)
        return State.KICK_BALL_TO_GOAL;
else
        return State.INIT;
```

Figure 3.18. Nested-if in StateCheck.GetState(…)

These specific state check methods are linked as events in the JavaMOP portion of the project as events, but the calls within StateCheck are ignored since that would generate too many useless events. The event generator helps reduce these state events to the bare minimum needed to accurately represent the robot's state at any given time point.

<u>Section 3.4: Event Generator</u>

The state checker component helps extract the current state of the soccer robot, but this class is just a helper to the event generator component. The event generator component is an AspectJ aspect that determines when a state event or sensor event should occur and therefore pass these events to the property monitor, which is implemented in JavaMOP and separated from the implementation of the soccer robot.

Section 3.4.1 RobotStateMachine Aspect Overview

To help dictate when certain events are triggered, the RoboStateMachine aspect was developed to act as a middle-man between the robot and the JavaMOP proporty monitor. Robotic sensors tend to require constant value checking for accurate up-to-date information for the robot, and they are generally not represented in binary form, meaning they are not directly represented as events. Therefore, only including a direct pointcut to a sensor value change to represent a sensor event is clearly the wrong approach due to serious performance issues along with potentially unpredictable behavior. Instead, when sensor value pointcuts are triggered, this aspect sets internal flags for sub-state tracking and proceeds with a full state event generation if all appropriate partial-state values have been obtained from the sensors. If a state check is performed, the resulting state will be generated and captured by one of more of the JavaMOP property monitors.

Section 3.4.2: RobotStateMachine Pointcuts

RobotStateMachine aspect defineds several pointcuts as shown in **Figure 3.19**. The pointcut *BallInFrontPC* specifies join points as point whenever the method

Kicker::BallInFront() is invoked. Similarly, the pointcut *ballClosePC* specifies join points as point whenever the method Kicker::ballClose() is invoked. The other pointcuts *irModChange_BIF*, *irUnModChange_BIF*, *sonarCharng_BIF* and *sonarCharng_BIF* are built around changes of values read from three different sensors within method Kicker::BallInFront() or Kicker::ballClose9). These three sensors are modulated IR, unmodulated IR, and sonar sensors. With these values, along with a couple of other properties, the StateCheck object can determine the current state of the robot. This aspect also defines other types of pointcuts and events such as entrance/exit of a state in some situations.

Although there are multiple pointcuts defined in the aspect, we should pay more attention to: irModChange_BIF, irUnModChange_BIF, and sonarCharng_BIF since they represent the change of the core values of the states and will ultimately trigger a state check. The others are simply there for performance reasons with AspectJ with *cflowbelow* calls, testing, or forcing violations for JavaMOP.

```
pointcut BallInFrontPC(Kicker MK) : call(public boolean Kicker.BallInFront()) && target(MK);

pointcut ballClosePC(Kicker MK) : call(public boolean Kicker.BallClose()) && target(MK);

pointcut irModChange_BIF(Kicker MK) : cflowbelow(BallInFrontPC(MK)) && set(float
SensorController.ballDirMod)&& within(SensorController);

pointcut irUnModChange_BIF(Kicker MK) : cflowbelow(BallInFrontPC(MK)) && set(float
SensorController.ballDirUnMod)&& within(SensorController);

pointcut sonarChange_BIF(Kicker MK) : cflowbelow(BallInFrontPC(MK)) && set(float
SensorController.sonarRead)&& within(SensorController);

pointcut sonarChange_BC(Kicker MK) : cflowbelow(ballClosePC(MK)) && set(float
SensorController.sonarRead)&& within(SensorController);
```

Figure 3.19. Pointcuts Defined in RobotStateMachine aspect

Section 3.4.3: RobotStateMachine Advice and Supporting Methods

Advices defined in RobotStateMachine for pointcuts *irModChange_BIF*, *irUnModChange_BIF*, *sonarCharng_BIF* and *sonarCharng_BIF* are similar. Therefore, we only discuss the advice for pointcut *irModChange_BIF* as shown in **Figure 3.21**. IR and sonar sensors are checked on a regular basis within the LeJOS code, and give all necessary values to allow a valid state check. The *irModChange_BIF* advice is executed every time a value is gathered from the IR sensor. The purpose of the advice is to generate events that represent entering or exiting a state. To do so, we introduce two variables. One as a flag to indicate if the appropriate condition such as if a ball is in front or if a ball is too close is true or false based on the new reading from the sensor. Appropriate flags for this will help determine the correct time to signal a state event. And the second introduced variable is to track the sensor that triggers the checking of state.

After set the values for both introduced variables, the advice checks if the robot is ready to check its state, which is represented by the return value of the helper method readyForStateCheck() defined in the aspect. If yes, it will gather the current state of the robot represented by the StateCheck object, and passes as arguments to another helper method generateStateEvent(…), which, as shown in **Figure 3.24**, is defined in the aspect to call corresponding *Gen_* methods in StateCheck class. After that, it resets all variables and flags. If the robot is not ready to check its state, nothing happens.

As shown in next section, every time a *Gen_* method in StateCheck class is invoked, a state event that represents the state change will be generated in our behavior monitor. These events form an event trace in the monitor and allows a relatively straightforward property check of any point in the state diagram. With this strictly controlled structure combination of RoboStateMachine and StateCheck, it is far less chaotic and are more accurate to the realistic state of the robot. In some cases, it is desirable to trigger other events to make up for a lack of calls from the soccer code. This structure easily allows for that and only requires a modification of the aspect code rather than the soccer. Furthermore, the overall flow of this process is partly controlled in the advice sections associated with the previously mentioned pointcuts as shown in **Figure 3.20.** The state check is triggered only when the core advice sections have been executed, and therefore set the appropriate flags.

Abstracting out the state generation parts from the soccer code and monitor code allows for a clean modularization approach. The base idea is that the soccer code is strictly setup to play soccer with little care about error checking and correcting within it.

The JavaMOP parts are centered around ensuring that behavior of the robot correctly match the given state diagram. It does not have a direct concern of the exact sub-states of the robot. It just retrieves an abstracted view of the robot. It does, however, require a little more handling detail during violation handling, but it is negligible compared to the complexities that would be required for event generation if only JavaMOP was used. RoboStateMachine and StateCheck pre-process raw data into a relatively easy to follow event listing which is captured in JavaMOP. This design provides a powerful runtime verification method with a small footprint on performance.

```
// IR -- Mod advice, handle change in IR MOD value (after new ping)
        after(Kicker MK, float newIrMod) :irModChange_BIF(MK) && args(newIrMod){
                irModSet = true;
                if(newIrMod == 0){
                        bifFlag = true;
                        bifTrigger = ChangeEvent.IR_MOD;
                }
                if(readyForStateCheck(MK)){
                        State currState = StateCheck.GetState(bifTrigger, MK);
                        MK.generateStateEvent(currState,MK);
                        resetStatePreCheck();
                }
        }

// IR -- Un-Mod advice, handle change in IR UN-MOD value (after new ping)
after(Kicker MK, float newIrUnMod):irUnModChange_BIF(MK) && args(newIrUnMod){
        irUnModSet = true;
        ***ADVICE SECTION – SAME AS irUnModChange_BIF***
}
// Sonar -- Sonar advice, handle change in Sonar value (after new ping)
after(Kicker MK, float newSonar):sonarChange_BIF(MK) && args(newSonar){
        sonarSet = true;
        ***ADVICE SECTION – SAME AS irUnModChange_BIF***
}
```

Figure 3.20. RobotStateMachine advice

```
// Generate the current state on-demand here rather than directly from state check method
public void Kicker.generateStateEvent(State currState, Kicker currMK){
        if(currState != lastState){
                switch(currState){
                        case TURN_TO_BALL:
                                StateCheck.Gen_TurnToBallState(currMK);
                                lastState = currState;
                                break;
                        case GOTO_BALL:
                                StateCheck.Gen_GotoBallState(currMK);
                                lastState = currState;
                                break;
                        case TURN_TO_GOAL:
                                StateCheck.Gen_TurnToGoalState(currMK);
                                lastState = currState;
                                break;
                        case DRIBBLE_TO_GOAL:
                                StateCheck.Gen_DribbleBallState(currMK);
                                lastState = currState;
                                break;
                        case KICK_BALL_TO_GOAL:
                                StateCheck.Gen_KickBallAtGoal(currMK);
                                lastState = currState;
                                break;
                        default:
                                break;
                }
        }
}
```

Figure 3.21. RobotStateMachine supporting methods

Section 3.5 Monitoring System Behavior

The ultimate goal of this research is to monitor system behavior defined in state diagrams during its execution. System behavior is specified as a set of LTL formula derived from state diagrams. Each LTL formula indicates an important pattern of behavior that the robot must follow. The StateCheck class and RobotStateMachine aspect discussed in previous sections provide necessary scaffold for our monitor by facilitating the generation of state events whenever the robot enters a new state.

The monitor for each LTL formula is specified and generated by JavaMOP and weaved into robot code using AspectJ. To specify monitor in JavaMOP, we need to specify the LTL formula based on events and definition of these events as appropriate pointcuts. In general, there are two types of events we may need: events that represent entering a new state and events that represent actions of transitions in state diagrams. The former events can be defined as pointcuts in corresponding *Gen_* method defined in StateCheck class. For example, as shown in **Figure 3.22**, *turn_to_ball_state_true* event occurs if the method *Gen_TurnToBallState()* is called. Occurrence of event *turn_to_ball_state_true* implies that the robot now enters into state turn_to_ball_state. This is possible because of the RobotStateMachine aspect, which calls Gen_TurnToBallState() whenever it detects that the robot enters into TurnToBallState state. This is independent of how soccer robot is implemented.

```
event turn_to_ball_state after(Kicker MK) returning(boolean res) :
        call(public boolean StateCheck.Gen_TurnToBallState(Kicker))
        && args(MK)
{  //advice
}
```

Figure 3.22 Definition of event turn_to_ball_state_true

The latter events that represent actions of transitions in state diagrams can be defined as pointcuts of corresponding join points based on the nature of the action. For example, as shown in **Figure 3.23**, we can have two different events defined for the call of method Kicker.KickBall(): one represents before the action kick ball and one represents after the action kick ball. Such event definitions requires the knowledge of robot code.

```
event kick_ball_before before(Kicker MK):
        call(public void Kicker.KickBall()) && target(MK)
{  //advice
}


event kick_ball_after after(Kicker MK):
        call(public void Kicker.KickBall()) && target(MK)
{  //advice
}
```

Figure 3.23: Definition of events KickBall

In our experiment, 7 separate properties were developed for the soccer robot directly utilizing the event generator and tested (See Chapter 5 for results). These properties are: *the initialization property, ball-in-front until after-kick, ball close at kick, IR always reads a value, go to goal until in range, and eventually kicks ball*. Some of these properties are challenging if not impossible to test under normal circumstances, so some code modifications were made to force a property violation to see how this system would handle such a situation. We will discuss these properties one by one in the rest of this section.

Section 3.5.1: Initialization Property

For the soccer robot to perform any kind of task, it is critical to ensure that all motors and sensors have been initialized. The initialization property, as shown in **Figure 3.24** is design for this purpose. In this monitor, we defined several events: *ready_true*, *ready_false*, and *play_before*. The events *ready_true* and *ready_false* represents the call to method Kicker.Ready(), which checks if motor and sensors are initialized successfully. The only difference among these two events lies in the boolean value returned from the method. The play_before event represents the time before method Kicker.play() is invoked. The LTL formula to be monitored is: *[](play_before* -> <*> *ready_true),* which implies that it is always true that when *play_before* occurs, event *ready_true* must occur sometime in the past. This formula guarantees that the robot is ready to leave the first state in the state diagram only after the motor and sensors are initialized successfully, and a violation will trigger a separate set of code in JavaMOP as shown in the violation section. Rather than attempting to proceed through the other states and failing later, the property monitor will attempt to fix the problem before going any further. In some cases, the LeJOS library will fail to initialize the motors and/or sensors possibly due to disconnected wiring, but for testing purposes, one of the sensor cables were pulled out temporarily. The monitor will recognize that the robot is attempting to transition to a turn-to-ball state without proper sensor/motor setups, and will attempt to recover before releasing the robot to regular operation.

```
event ready_true after(Kicker MK) returning(boolean res):
        call(public boolean Kicker.Ready()) && condition(res) && target(MK)
{  /*advice*/   }

event ready_false after(Kicker MK) returning(boolean res):
        call(public boolean Kicker.Ready()) && condition(!res) && target(MK)
{  /*advice*/   }

event play_before before(Kicker MK):
        call(public void Kicker.Play()) && target(MK)
{  /*advice*/   }

ltl: [](play_before => <*> ready_true)
@violation{
        while(!currMK.Ready()){
                System.out.println("Please plug in all cables correctly, then press any
        button.");
                Button.waitForAnyPress();
                currMK.Init();
        }
        __RESET;
}
```

Figure 3.24. Init.mop

Section 3.5.2: Ball in Front After Turn Until Kick

When the robot successfully turns to the ball, we anticipate the ball remaining directly in front of the robot under normal circumstances, but various situations may cause a violation of this property. The ball's battery could die, leaving the robot somewhat blind or some other force could move the ball.

To monitor this property, four separate events were utilized as shown in **Figure 3.25**, and the LTL to be checked adds a layer of complexity yet still retains a relatively simplistic structure thanks to the event generator: *[](turn_to_ball_state_true => o(ballinfront_true U kick_ball_after))*. The event *turn_to_ball_state_true* indicates that the soccer robot is in *turn_to_ball* state, while the events *ballinfront_true* and *ballinfront_false* indicates if ball is in front of the robot or not. The event *kick_ball_after* represents the action of kicking ball. If the property is violated, which typically means ball is removed from its previous detected position either by opponents or teammates, the soccer robot starts searching for the ball. The advantages of the event generator should be especially clear with this property. Once the definition of events are provided by associating them to join points of method calls and value changes of variables deciding states, the LTL logic is drastically simplified. It ultimately utilizes abstraction to make error checking and correction setup simpler.

```
BallInFrontAfterTurnUntilKick(Kicker MK) {
        Kicker currMK = null; // Allows for usage of MK in the ltl violation
        event turn_to_ball_state_true after(Kicker MK) returning(boolean res) :
                        call(public boolean StateCheck.Gen_TurnToBallState(Kicker)) && condition(res)
                        && args(MK){/* Advice */}
        event ballinfront_true after(Kicker MK) returning(boolean res):
                call(public boolean Kicker.BallInFront()) && condition(res) && target(MK){/* Advice */}

        event ballinfront_false after(Kicker MK) returning(boolean res):
                call(public boolean Kicker.BallInFront()) && condition(!res) && target(MK){ /* Advice */}
        event kick_ball_after after(Kicker MK):
                call(public void Kicker.KickBall()) && target(MK){ /* Advice */}

        ltl: [](turn_to_ball_state_true => o(ballinfront_true U kick_ball_after))
        @violation {
                while(!currMK.FindBall());
                __RESET;
        }
}
```

Figure 3.25. BallInFrontAfterTurnUntilKick.mop

Section 3.5.3: Ball Close at Kick

When the robot is preparing to kick the ball, it must have the ball, so ensuring this is clearly important. As shown in **Figure 3.26**, if the LTL: [](kick_ball_before => (*) kick_ball_state_true) is violated, the monitor attempts to find the ball before the kick is actually performed, and then it has to re-position itself for a proper shot. In general, this property monitor maintains the same general approach as the others. If a violation occurs at its given point in the state diagram, it back tracks to an earlier point in the state diagram. Even during a violation, the robot should generally remain within the rules of the state diagram too. (See **Figure 3.26**)

```
BallCloseAtKick(Kicker MK) {
        Kicker currMK = null;
        event kick_ball_state_true after(Kicker MK) returning(boolean res) :
                        call(public boolean StateCheck.Gen_KickBallAtGoal(Kicker)) && condition(res)
                        && args(MK){ /* Advice */}

        event kick_ball_before before(Kicker MK):
                call(public void Kicker.KickBall()) && target(MK){ /* Advice */}

        ltl: kick_ball_before => (*) kick_ball_state_true
        @violation{
                while(!currMK.FindBall()){}
                currMK.GotoGoal(true);
                __RESET;
        }
}
```

Figure 3.26. BallCloseAtKick.mop

Section 3.5.4: IR Always Reads a Value

After all the sensors are properly initialized, the robot should always have some valid value. It must be from UnMod and/or Mod IR; and this is the case because the ball should always be emitting a signal, and the sensor should always be able to pick it up except in the dead-zone directly behind it. If the ball is directly behind it, the violation will flush the sensors and then force the robot to turn a little to try to pick up the signal of the IR ball again. If the ball is turned off or the cable for IR is unplugged it will try to recover until it gets a signal. The LTL for this property is simple: *[](ir_read_true)*. (See **Figure 3.27**) Once the sensors are initialized, it is expected that an IR signal is read. If not, one of the previously mentioned errors probably occurred. If it fails, the violation code continuously flushes the IR sensor and retries a pull of the values.

```
AlwaysIR(Kicker MK) {
        SensorController currSC = null; // Allows for usage of SC in the ltl violation
        Kicker currMK = null;
        event ir_read_true after(SensorController SC)    returning(boolean res):
                call(public boolean SensorController.GetIR(int,float[])) && condition(res) && target(SC){
                        currSC = SC;
                }

        event ir_read_false after(SensorController SC) returning(boolean res):
                call(public boolean SensorController.GetIR(int,float[])) && condition(!res) &&
                target(SC){
                        currSC = SC;
                }
        ltl: [] ir_read_true
        @violation {
                while(currSC.GetIR(Globals.IR_MOD,tmpIR)){
                        currSC.FlushSensors();
                        System.out.println("Flush IR Loop");
                }
                __RESET;
        }
}
```

Figure 3.27. AlwaysIR.mop

Section 3.5.5: Goes to Goal Until in Range of Goal

For additional complexity, rather than going to a specific point every time and shooting to the goal, the robot heads in the direction of the given goal point, and kicks the ball once it is within a scorable radius from the goal. Once the robot starts turning to the goal with the ball, it is expected that the robot will then dribble the ball to the goal until at least being within range of it. The robot should never lose the ball in this process, and would therefore raise a violation if such an event occurred. The LTL for this property as shown in **Figure 3.28** is *[](turn_to_goal_state_true => o(dribble_ball_state_true U goto_goal_true)).* Note that a successful goto_goal_true event mainly means that the robot is in range of the goal. The property maybe violated if the ball is taken away during the dribble to goal phase, and the robot goes into violation handling mode.

```
GoToGoalUntilInRange(Kicker MK) {

        Kicker currMK = null;

        event dribble_ball_state_true after(Kicker MK) returning(boolean res):

                        call(public boolean StateCheck.Gen_DribbleBallState(Kicker)) && condition(res)
                        && args(MK){ currMK = MK;}


        event turn_to_goal_state_true after(Kicker MK) returning(boolean res):

                        call(public boolean StateCheck.Gen_TurnToGoalState(Kicker)) && condition(res)
                        && args(MK){currMK = MK;}

        event goto_goal_true after(Kicker MK) returning(boolean res):

                call(public boolean Kicker.GotoGoal(boolean)) && condition(res) &&
                target(MK){/*Advice*/}

        event goto_goal_false after(Kicker MK) returning(boolean res):

                call(public boolean Kicker.GotoGoal(boolean)) && condition(!res) &&
                target(MK){/*Advice*/}

        event kick_ball_state_true after(Kicker MK) returning(boolean res) :

                call(public boolean StateCheck.Gen_KickBallAtGoal(Kicker)) && condition(res) &&
                args(MK){/*Advice*/}

        ltl: [](turn_to_goal_state_true => o(dribble_ball_state_true U goto_goal_true))

        @violation{

                currMK.KickBall();

                __RESET;

        }

}
```

Figure 3.28. GoToGoalUntilInRange.mop

Section 3.5.6: Kicks to Goal Eventually

Finally, the ultimate task of the robot is to kick the ball to the goal, so some time from the beginning of the execution of the code to the ending, the robot must kick the ball to the goal. The LTL for this given property is *(game_over => <*> kick_ball_state_true) and (game_over => <*>kick_ball)*. Once a game over occurs, sometime in the past the robot has to have been in a kick-ball state and have completed a kick successfully. A violation is challenging to force without any code manipulation, so to force it, the kick method was simply taken out. If the violation occurs, the robot will essentially determine what is left to do and attempt to finish the overall process. In most testing situations, all the robot should have to do is to kick the ball unless the ball is completely moved. The full code can be viewed in **Figure 3.29.**

```
EventuallyKicksToGoal(Kicker MK) {
        Kicker currMK = null;
        event kick_ball after(Kicker MK):
                call(public void Kicker.KickBall()) && target(MK){currMK = MK;}
        event game_over before(Kicker MK):
                call(public void Kicker.GameOver()) && target(MK){currMK = MK;}
        event kick_ball_state_true after(Kicker MK) returning(boolean res):
                call(public boolean StateCheck.KickBallAtGoal(Kicker)) && condition(res) && args(MK){
                        currMK = MK;
                                Logger.log(LogFile.EVENTUALLY_KICKS,"kick_ball_state_true");
                        }
        ltl: (game_over => <*> kick_ball_state_true) and (game_over => <*>kick_ball)
        @violation{
                currMK.KickBall();
                currMK.SetBallKickedAtGoal(currMK.BallInFront());
                __RESET;
        }
}
```

Figure 3.29. EventuallyKicksToGoal.mop

Section 3.6: Final Compiling/Weaving

Using the event generator with JavaMOP property monitoring is straightforward. When developing JavaMOP property monitoring code, all a developer must know at this point is the signature of the state related methods (and maybe a couple of other methods), the state chart diagram, and of course LTL formula derived from the state chart diagram. This allows the developer to focus mostly on the higher-level view of how the property should be structured rather than being distracted by the lower-level implementation details.

Weaving new JavaMOP code in with the event generator is straightforward too. It simply weaves the following pieces of code into soccer robot code using AspectJ compiler *ajc*: Java and AspectJ code of our framework (StateCechk.java, RobotStateMachine.aj and other support files) and monitor code generated from LTL formula using JavaMOP . As a result, a single runnable jar file will be generated. An example of generating executable to monitor LTL forumla specified in AlwaysIRMonitorAspect.aj is shown in **Figure 3.30**. The command-line options to the *ajc* weaver are provided to specify the runtime libraries it needs: ev3classes.jar for LeJOS library, aspectjrt.jar for AspectJ library, rv-monitor-rt.jar for RVMonitor library needed by JavaMop and StateTools.jar for our framework. This allows us to monitor one property per time. If we want to monitor another property, just replace AlwaysIRMonitorAspect.aj with another property file. It is also possible to monitor multiple properties at the same time by providing multiple *.aj files that contain different LTL formula.

```
ajc -1.6 RoboStateMachine.aj -cp "ev3classes.jar" -inpath "./PlayerInit.jar;aspectjrt.jar;rv-
monitor-rt.jar;StateTools.jar;"  AlwaysIRMonitorAspect.aj -outjar test2.jar
```

Figure 3.30. Command to weave monitor into robot code

CHAPTER IV

RESULTS

This chapter describes the results we have on the case study by monitoring all properties discussed in **Section 3.5**. Three measurements were taken to evaluate the effectiveness of our approach: code size change with additional RV code, time cost related with runtime verification code, and finally the resulting event sequence for each given property. As shown in the rest of the chapter, consistent patterns exist for each property monitor and event generator appendage to the base soccer code.

Section 4.1: Change of Code Size

In many cases, it is crucial for systems to retain minimal code due to space restrictions, especially in cyber physic systems like robotics, and it may not even be possible to integrate this event generator and monitoring method into any given system. For this given research, the rough size of the PlayerInit.jar file (pure soccer code without RV parts), is roughly 10 KB, but on average the additional RV significantly increases the size to an average of roughly 400KB from 12KB. This can be viewed in **Figure 4.1**. As shown in this table, the size change is drastic from the original code, but it stays relatively the same across each property because the libraries associated with AspecJ and JavaMOP account for the vast majority of it. One route not tried in this research is the application of this concept on much larger programs. In this case, the soccer robot code is relatively small, and larger projects may not see this drastic of a difference depending on how much RV code is needed. For example, if the same basic sensors are used in much larger projects, the event-generator may not change in size by much, and any dependencies associated with JavaMOP and AspectJ would remain constant. Ultimately, however, most modern systems should be able to handle the size change. Even our Lego EV3 system has very limited storage and no problems occurred due to storage limits.

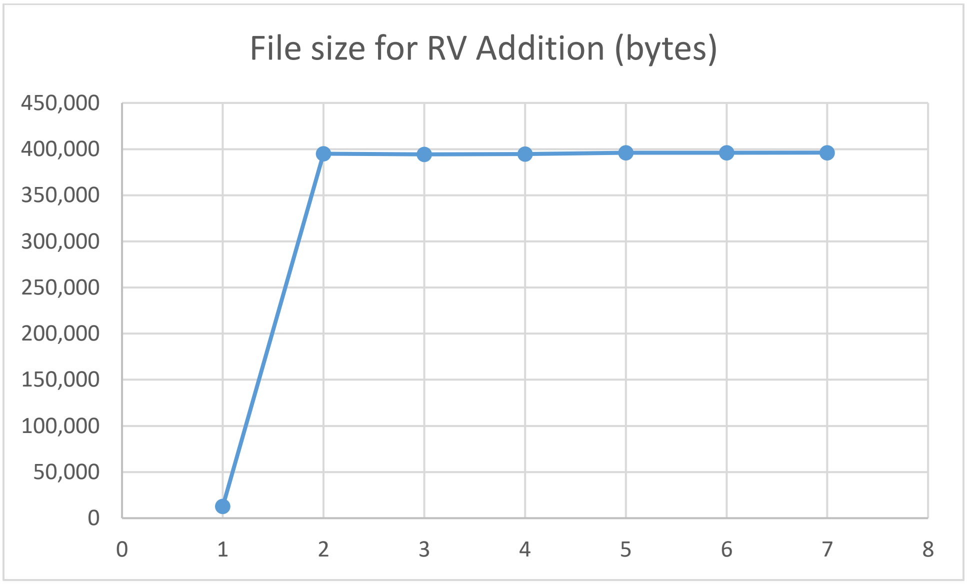


Figure 4.1. RV file size impact; the ordering is: NO-RV, *Init, AlwaysIR, BallCloseAtKick, BallInFrontAfterTurnUntilKick, EventuallyKicksToGoal, GoToGoalUntilInRange*

<u>**Section 4.2: Runtime Overhead of Monitor**</u>

Robotic systems are reactive computing systems that are required to provide real-time response to changes in the external environment. Runtime verification code introduced to monitor system behavior during its execution incurs runtime overhead. Such runtime overhead is typically caused by two actions: intercepting pointcuts in AspectJ, and detecting and handling property violation in monitor. We need to make sure the runtime overhead of our monitor doesn’t affect the robotics to complete its predefined mission. This section will analyze the runtime overhead of the monitor for each property discussed in the previous chapter.

To gather runtime overhead of runtime verification code, a time logger class was developed and used. When an event or pointcut is linked with a method or a value change, new code from the advice sections will be weaved before or after the method call or value change. To capture this overhead, the log method in the Logger class was inserted around the method calls where appropriate; this ultimately captures the overhead associated with the runtime verification parts. If a method such as Play() have drastically different timing on its own without RV, due to external factors such as missing the ball on a first go-to-ball, then the log method calls were placed to avoid this uncertainty while still capturing the RV part. Note, however, that all prints to screen were disabled for these runs. As with any system logging, printing to screen gives excessive time delays unimportant to this measurement. Based on the data we gathered, even with a high size change, the runtime overhead of the monitor added to the system is relatively low. The Lego EV3 robot contains relatively low processing power, but performance based on the time recordings wasn’t impacted too heavily on average. Given these results, developers who utilize this RV method shouldn’t see too much of an impact on processing time. From **Figures 4.2** to **4.9**, each graph represents the timing of all possible recordings to capture additional RV processing time. The first box and whisker plot for all of these figures is the original processing time of the specified section before the introduction of the RV code. For each test set (each figure), the logging statements in the base soccer code were left unchanged with the RV code weaved in. If the logging statements are placed correctly, this ultimately captures the change since the RV code is integrated between the logging statements. In addition, other than the first plot in each figure, all the plots represent when an event in one of the monitors is linked with the specified method call or value; and each box also represents multiple runs of the same code. These repeated runs were performed until the processing time appeared to have hit peak points from lowest to highest. Only a few exceptions exist, however; some additional recordings for things like ball-close and ball-in-front were included across all because pointcuts exist in the event generator.

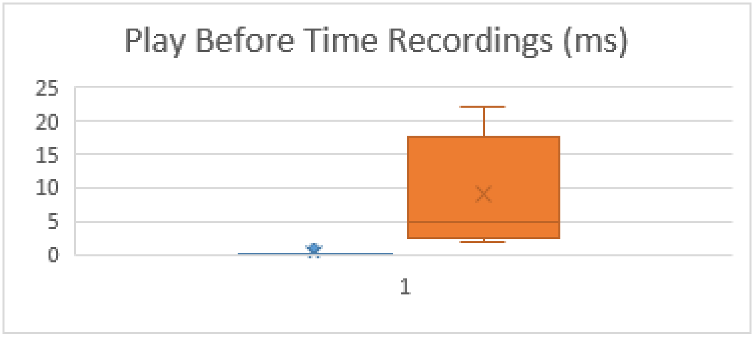


Figure 4.2. RV time impact attached before Play method (1st is without RV).

In **Figure 4.2**, only the Init monitor has an event for before the Play method, so only one additional box was included in this graph. This recording only captures the transition from a call to Play and the start of it rather than timing the entire method because it avoids random values due to unpredictable nature of the soccer sequence. The time it takes ultimately get to the goal with the ball is not fully consistent, and therefore useless data. The box shows that the timing has a tendency to be on the higher end with additional RV code, but the additional RV code clearly doesn't add too much additional processing time since it appears to peak at about 22ms with a median around 10ms.

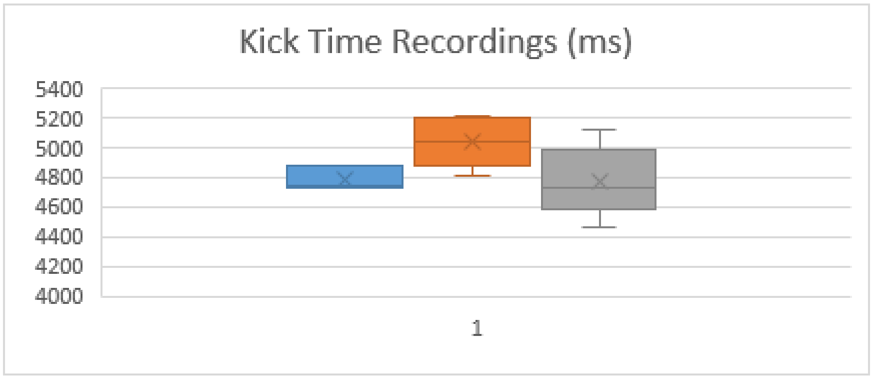


Figure 4.3. RV time impact attached to the Kick method (1st is without RV).

Next, **Figure 4.3** displays full recordings of the Kick method since it is mostly consistent as the first box plot shows, without any RV code it basically stays around 4.8 seconds. Both the *BallInFrontAfterTurnUntilKick* and *BallCloseAtKick* monitors (same order as the graph) have an associated event with the KickBall method. The central tendency for both still come close to the runs without RV code, the max going to about 5.2 seconds. As for some recordings being lower than the no-RV set is most likely due to a slight malfunction in the way the LeJOS API handles the delay or arm movement methods, which is part of the method and not the RV parts.

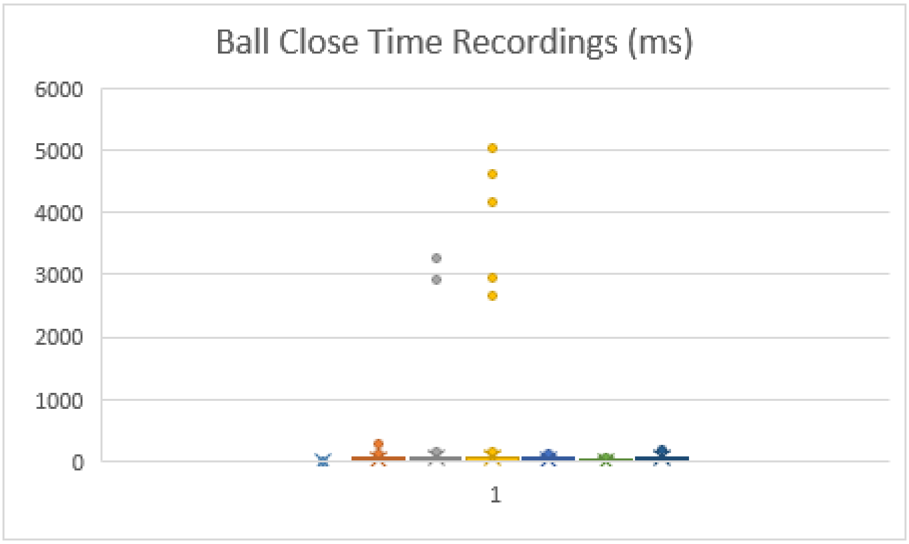


Figure 4.4. RV time impact attached to the Ball Close method (1st is without RV).

The BallClose method in **Figure 4.4** is called in various places throughout the soccer code and a pointcut exists the event generator. No JavaMOP events exist for this method in the monitors, but it is still an important recording since RV code is associated with it. The ordering for the graph is*: AlwaysIR*, *GotoGoalUntilInRange*, *BallCloseAtKick*, *EventuallyKicksToGoal*, *HasBallAfterGotoUntilKick, and Init*. All of them have a central tendency not too far off from the no-RV run with a median around 100ms to 200ms additional runtime. However, *BallCloseAtKick* and *EventuallyKicksToGoal* have clear spikes with nearly 3 to 5 additional seconds added to runtime. The addition of those two monitors most likely resulted in more usage of the sonar at faster rates, which typically hurts performance of the sonar. The LeJOS API may even have some level of internal recovery for this that may cause rare delays, but this is not explicitly stated anywhere. In general though, these are simply outliers.

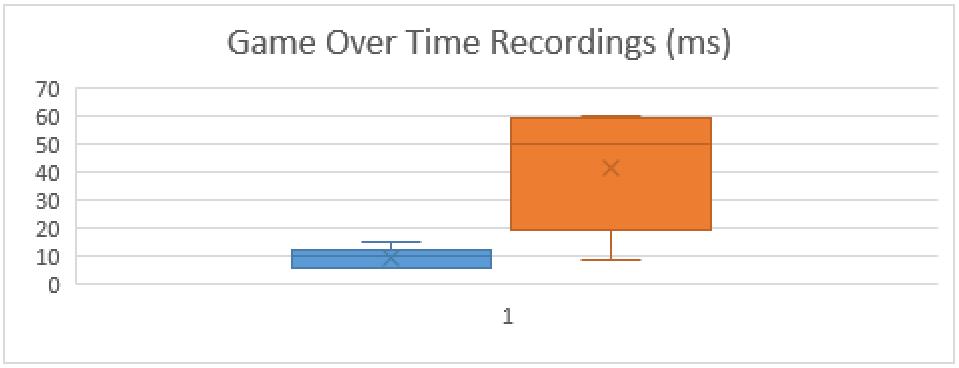


Figure 4.5. RV time impact attached to the Game-Over method (1st is without RV).

For **Figure 4.5** only the *EventuallyKicksToGoal* monitor is linked to the GameOver method. Based on the graph, the rough range of additional processing time when RV is included is around 10ms to 60ms with a central tendency hovering around 40ms total. Note that these measurements include the runtime of the GameOver method.

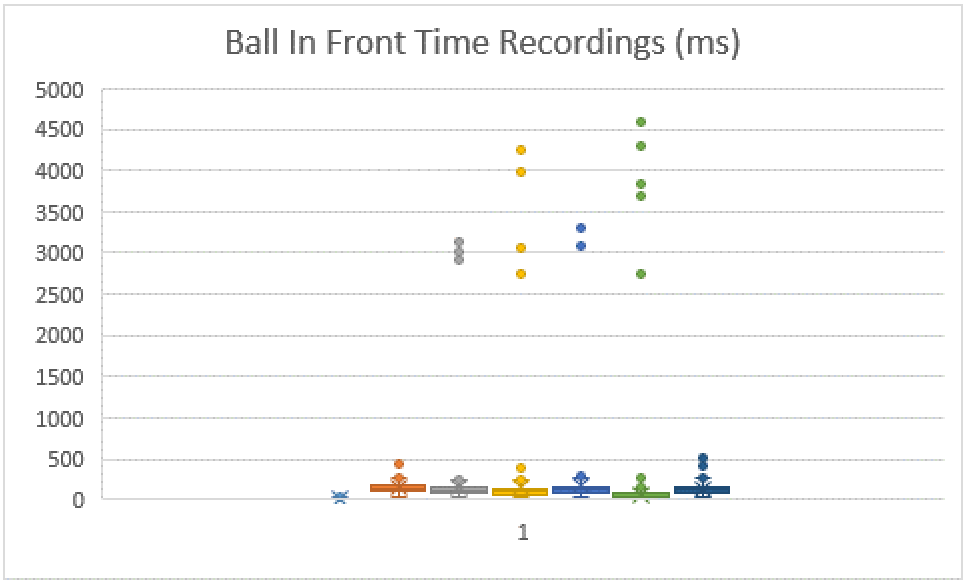


Figure 4.6. RV time impact attached to the Ball-In-Front method (1st is without RV).

Based on **Figure 4.6**, the rough range of additional processing time added on from the RV code pushed the total to around 100ms more than the no-RV version. BallInFront is another method with a pointcut in the event generator, and for the most part the RV addition didn't make too bad of an impact on the runtime. The somewhat drastic outlier points from the 3rd to the 6th plots stem from the same problem in **Figure 4.5**. Over pinging the sonar (and in this case also the IR) sensors may cause some strange yet rare behavior. It is important to reiterate that the LeJOS API is a beta version, and at the moment momentum on development has slowed, so bizarre behavior such as this may be more of a LeJOS API problem more so than an RV addition problem.

Figures 4.7 to **4.9** are all linked with the value changes in sensors and monitored by the event generator. All runs of the monitors show a relatively consistent pattern for most recordings. IR MOD readings contain many outlier times, but the overall central tendency of the time is roughly the same as the version without RV code. IR UNMOD clearly has some more tendency to be affected by the additional RV code, but the impact is usually only around 25ms. Lastly, other than a few very rare outlier points in **Figure 4.9**, sonar basically not affected by the additional RV code. When an over-ping happens, however, it will obviously have a substantial hit in performance by about 3 to 5 seconds, but again this is a very rare situation based on the data. These sensor reading points represent hundreds of data points per plot, so these outlier points are clearly very rare.

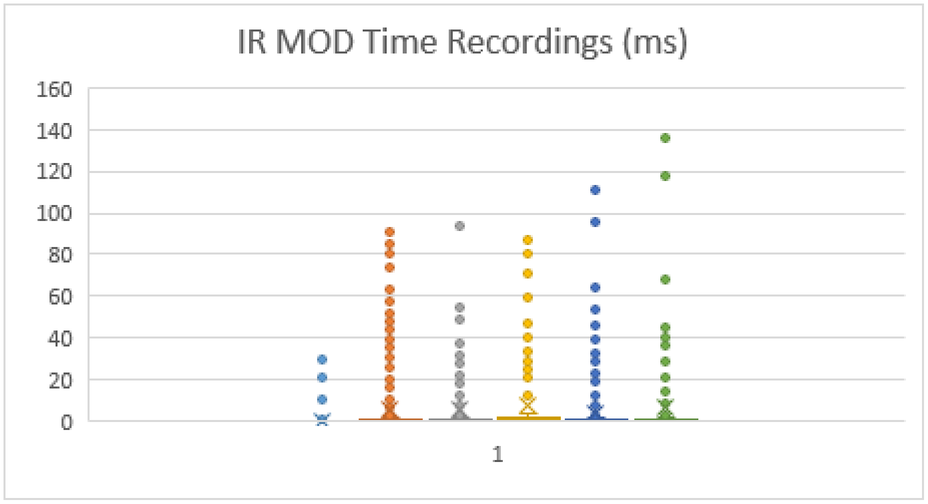


Figure 4.7. RV time impact attached to the IR Modulated method (1st is without RV).

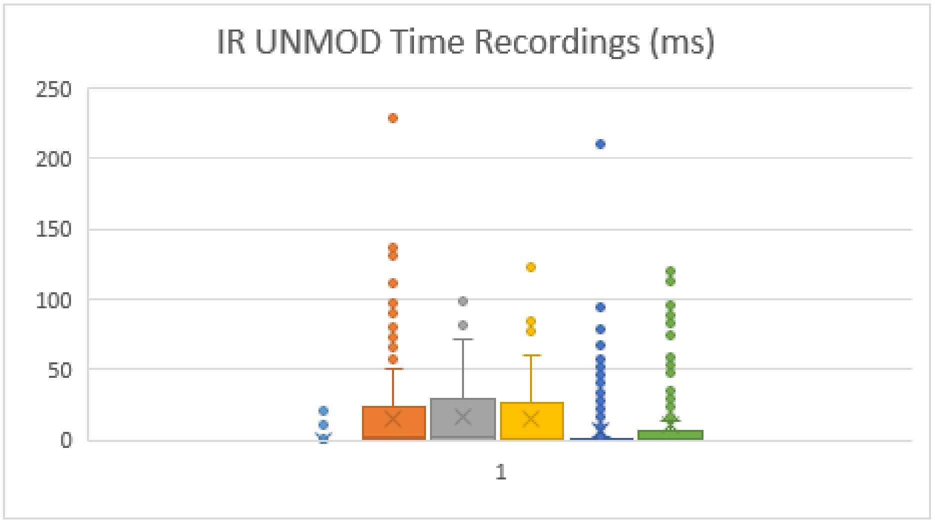


Figure 4.8. RV time impact attached to the IR Unmodulated method (1st is without RV).

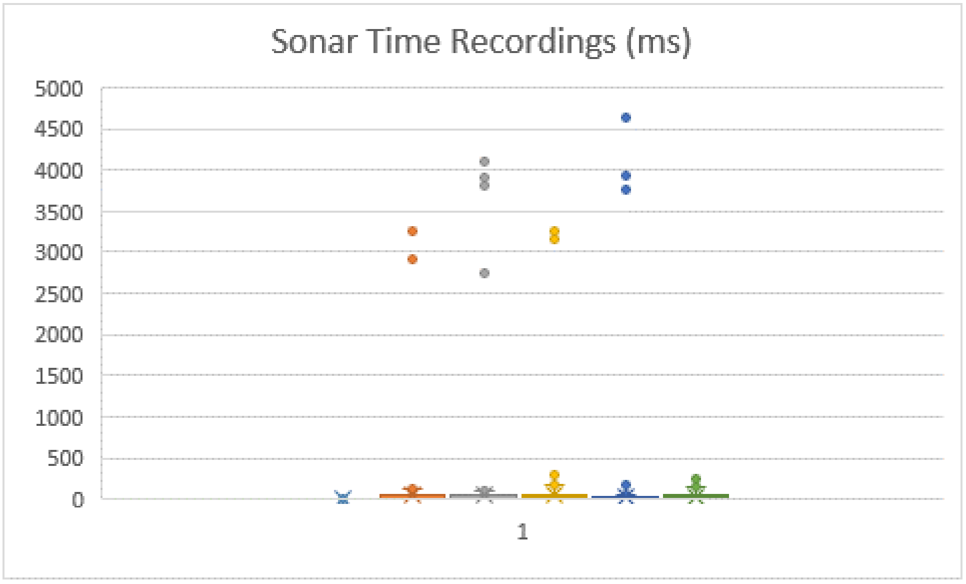


Figure 4.9. RV time impact attached to the IR Sonar method (1st is without RV).

In general, for the first set of measurements from **Figures 4.2 to 4.6** other than a few outliers, the general impact on processing time is minimal given the processing power of the system. Even the bigger looking plots such as the ones from **Figures 4.2 and 4.5** the additional processing time for the play and game over methods only around 20ms. In some applications, 20ms might be too great of impact, but in most cases, this is negligible. Central tendencies of the time all hover relatively close to the runs without any RV code as shown in **Figures 4.3, 4.4, and 4.6**. The next set of recordings from **Figures 4.7 to 4.9** directly impact sensor readings. These sensor value changes are the key to the event generator structure, and have the highest likelihood for the higher outliers. When IR or sonar is read, a state check may or may not happen, so additional RV code has nearly a non-existent impact on processing time until enough data is collected to perform a state check. A state check would ideally occur around once every three sensor readings since it will occur when these three provide valid values.

Unmodulated IR seems to have taken the biggest impact on the general processing time, which implies that state checks occur more frequently after reading in this value as shown in **Figure 4.8**. Impact on sonar in **Figure 4.9** contains some drastic outliers, but it appears that this is just an initialization impact during the first read.

Section 4.3: Property Monitor Outputs

Finally, each individual property monitor will generate its own event sequence, and that sequence is what is checked internally by the JavaMOP property monitors. In **Figures 4.10** to **Figure 4.7**, the event sequence can be seen during test runs. These event listings also include the parts where violations were attempted, and are not official events.

Without Violation	With Violation
ready_true EVENT	ready_false EVENT
Play Before EVENT	Play Before EVENT
	!!!Init LTL Violated!!!

Figure 4.10. Event sequences for the Init property monitor.

First, we will examine the event sequence for the Init property monitor. The Init property is relatively simple. It ensures that the Ready method, which ensures the preparedness of the robot's sensors and motors, returns true before the robot begins playing. In **Figure 4.10**, it simply shows just that. It tries to ensure that the robot's sensors and motors are properly setup. Note too that the points in any violation runs that say something to the extent of "!!!...Violated!!!" is when the violation occurs in the sequence and is therefore where the violation handler takes over. The LTL for this property is: *[](play_before => <*> ready_true).* In other words the ready_true event must appear sometime before the play event (linked to the Play method). To violate this for testing, part of the Init method was commented out, and therefore some of the sensors/motors were not started, which caused a violation in this sequence. It attempts to recover itself by attempting to initialize everything again.

Without Violation	**With Violation**
dribble_ball_state_true	dribble_ball_state_true
turn_to_goal_state_true	dribble_ball_state_true
dribble_ball_state_true	turn_to_goal_state_true
in_goal_range_true EVENT	dribble_ball_state_true
in_goal_range_true EVENT	kick_ball_state_true
in_goal_range_true EVENT	!!!GoToGoalUntilInRange LTL Violated!!!
in_goal_range_true EVENT	
in_goal_range_true EVENT	
in_goal_range_true EVENT	
in_goal_range_true EVENT	
in_goal_range_true EVENT	
in_goal_range_true EVENT	
in_goal_range_true EVENT	
in_goal_range_true EVENT	
in_goal_range_true EVENT	

Figure 4.11. Event sequences for the *GoToGoalUntilInRange* property monitor.

Second, as **Figure 4.11** shows we will now look at the *GoToGoalUntilInRange* property monitor. As the sequence shows, on the correct side, the robot made its way to the goal until at least the *in_goal_range* event was true. Note too that the dribble state occurred before the turn state due to the robot thinking it was in the correct goal heading until it started to move. Most of the time this is ok, but occasionally the LeJOS API navigator will get a bad read from the compass until a moment of forward motion. In the violation sequence, the robot clearly tried to kick the ball before it was in the range of the goal. Hence a lack of *in_goal_range* events.

Next, in **Figure 4.12**, we examine the *EventuallyKicksToGoal* property. The overall goal of the robot is to make a shot to the goal, so it is critical to ensure a completed kick before the end of the execution of the full sequence. In this scenario, if a *kick_ball_state_true* event occurs prior to a game over, no violation should occur. In this case, to force a violation, the KickBall method was simply never called, and the game was allowed to finish. Therefore, the RV code took over and made the kick.

Without Violation	**With Violation**
kick_ball	game_over EVENT
kick_ball_state_true	!!!EventuallyKicksToGoal LTL Violated!!!
game_over EVENT	kick_ball_state_true

Figure 4.12. Event sequences for the *EventuallyKicksToGoal* property monitor.

Another relatively simple property as shown in **Figure 4.13** is the ball be close to the robot prior to the kick. With the successful run, the ball will eventually get close to the robot and stay that way until the actual kick recorded as *kick_ball_before*. To force this violation, the call to KickBall() was made at the beginning of Play(). Therefore, the event sequence is so short; everything is handled in violation section and therefore no events were triggered beyond that.

Without Violation	**With Violation**
ballclose_false	kick_ball_before
•••	!!!BallCloseAtKick LTL Violated!!!
ballclose_false	
ballclose_true	
•••	
ballclose_true	
kick_ball_state_true	
ballclose_true	
kick_ball_before	
ballclose_false	

Figure 4.13. Event sequences for the *BallCloseAtKick* property monitor.

Additionally, **Figure 4.14** shows the event sequences for the *AlwaysIR* property, which simply ensures that the robot always reads an IR value. Turning the IR ball off is a simple way to force a violation, and it is clear in this example where this happened. Note the "…" part of the listing simply represents hundreds of the same event wrapped around it.

Without Violation	**With Violation**
IR_read_TRUE_EVENT	IR_read_TRUE_EVENT
IR_read_TRUE_EVENT	IR_read_TRUE_EVENT
IR_read_TRUE_EVENT	IR_read_TRUE_EVENT
IR_read_TRUE_EVENT	IR_read_TRUE_EVENT
•••	•••
IR_read_TRUE_EVENT	IR_read_TRUE_EVENT
IR_read_TRUE_EVENT	IR_read_TRUE_EVENT
IR_read_TRUE_EVENT	IR_read_FALSE_EVENT
	!!!IR Read Fail LTL FAIL!!!
	IR_read_TRUE_EVENT
	IR_read_FALSE_EVENT
	!!!IR Read Fail LTL FAIL!!!
	IR_read_FALSE_EVENT
	!!!IR Read Fail LTL FAIL!!!
	IR_read_FALSE_EVENT
	!!!IR Read Fail LTL FAIL!!!
	IR_read_FALSE_EVENT
	!!!IR Read Fail LTL FAIL!!!
	IR_read_FALSE_EVENT
	!!!IR Read Fail LTL FAIL!!!
	IR_read_TRUE_EVENT
	IR_read_TRUE_EVENT
	IR_read_TRUE_EVENT
	IR_read_TRUE_EVENT
	IR_read_TRUE_EVENT
	IR_read_TRUE_EVENT

Figure 4.14. Event sequences for the *AlwaysIR* property monitor.

Finally, *BallInFrontAfterTurnUntilKick* is the most complicated property for our example and the event sequences are shown in **Figure 4.15**. When the run is successful, upon a successful *turn_to_ball* state, the ball remains in front until the *kick_ball_after* event. Note that it is ok for the last *ball_in_front* event because in our case we only care if the ball is in front at least until the kick. It doesn't really matter if it is in front at the end of the process.

Ultimately, the results of this all of these tests from run time recording to event sequence recordings were successful, and show the potential of this event generator approach. Additional yet minimal delays are inevitable, but in most cases, it has very little impact on the performance of even this low-powered robotic system.

<u>**Without Violation**</u>	<u>**With Violation**</u>
turn_to_ball_state_true	turn_to_ball_state_true
ballinfront_true	ballinfront_false
ballinfront_true	!!!HasBallUntilAfterKick LTL FAIL!!!
•••	ballinfront_false
ballinfront_true	ballinfront_true
ballinfront_true	ballinfront_true
kick_ball_after	ballinfront_true
ballinfront_true	ballinfront_true
	ballinfront_true
	turn_to_ball_state_true
	ballinfront_false
	!!!HasBallUntilAfterKick LTL FAIL!!!
	ballinfront_false
	ballinfront_true
	ballinfront_true
	•••
	ballinfront_true
	ballinfront_true
	turn_to_ball_state_true
	ballinfront_false
	!!!HasBallUntilAfterKick LTL FAIL!!!
	ballinfront_true
	ballinfront_true
	•••
	ballinfront_true
	ballinfront_true
	kick_ball_after
	ballinfront_true

Figure 4.15. Event sequences for the *BallInFrontAfterTurnUntilKick* property monitor.

CHAPTER V

SUMMARY

In this research, we proposed our framework to monitor behavior of robotic systems at runtime based on its defined state chart diagrams. This facilitates the development of a trustworthy system in robotics. Our framework contains two major parts: an event generator and property monitoring structure. Our framework has its pros and cons. Up front without any previous knowledge of runtime verification, the learning curve can be a bit steep simply because of some unique syntax and logic structures presented in LTL, JavaMOP, and AspectJ. But once the general flow is understood, the process should be straightforward. In addition, separating out error detection and handling parts from the direct base code feels unnatural at first since the normal approach in software development would be to fill the base code with various calls and conditionals to ensure everything runs smoothly.

In our experiment of soccer robot, we faced challenging issues. Most issues in fact came not from those concepts/structures but from the robot hardware and the third-party beta API LeJOS. Issues include things such as: IR sensor failure from different lighting, the sonar sensor occasionally failing to detect the ball when directly in front of it, poor documentation in some key parts of LeJOS, and various other things. These problems, however, help highlight the need for the integration of this research into other systems. Failure of this robot lacks wide reaching consequences, but what if larger scale military robotics or medical robotics fail in similar but larger scale ways? A simple malfunction of a system like that could result in serious issues, and this research aims to help developers easily ensure that such events are minimized.

Even with a high increase in the jar file size in our experiment, the general performance hit is minimal. The processing power of these EV3 robots is minimal, and it is still able to handle the monitoring code. It all comes down to trade-offs for each individual situation. In systems with strictly limited memory, this approach may not be plausible, and in some cases the additional code may overload the system depending on the situation. Modern computing power, however, should easily handle this.

BIBLIOGRAPHY

[1] U. S. N. S. Foundation, "Cyber-Physical Systems (CPS)." Arlington, VA, 2008.

[2] E. M. Clarke and E. A. Emerson, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications."

[3] J. H. Gallier and J. H., Logic for computer science : foundations of automatic theorem proving. Harper & Row, 1986.

[4] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," Proceedings of the 12th conference on USENIX Security Symposium - Volume 12. USENIX Association, pp. 12–12, 2003.

[5] M. Leucker and C. Schallhart, "A brief account of runtime verification," J. Log. Algebr. Program., vol. 78, no. 5, pp. 293–303, May 2009.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," ECOOP'97 — Object-Oriented Program., pp. 220–242, 1997.

[7] S. Soares, E. Laureano, P. Borba, S. Soares, E. Laureano, and P. Borba, "Implementing distribution and persistence aspects with aspectJ," in Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '02, 2002, vol. 37, no. 11, p. 174.

[8] D. Jin, N. Meredith, C. Lee, and G. Rou, "JavaMOP: Efficient Parametric Runtime Monitoring Framework," in ICSE'12, IEEE, 2012, pp. 1427–1430.

[9] Maximum LEGO EV3: Building Robots with Java Brains. Variant Press.

[10] M. Genesereth and E. Kao, “Introduction to logic,” 2012. [Online]. Available: http://logic.stanford.edu/intrologic/secondary/notes/cover.html. [Accessed: 20-Jul-2017].

[11] A. Bauer, M. Leucker, and C. Schallhart, “Runtime Verification for LTL and TLTL,” ACM Trans. Softw. Eng. Methodol., vol. 20, no. 4, pp. 1–64, Sep. 2011.

[12] M. Hammer, A. Knapp, and S. Merz, “Truly On-The-Fly LTL Model Checking,” Nov. 2005.

[13] E. Ghassabani and M. A. Azgomi, “A New Approach to Stateless Model Checking of LTL Properties,” Mar. 2016.

[14] R. Medhat, Y. Joshi, B. Bonakdarpour, and S. Fischmeister, “Accelerated Runtime Verification of LTL Specifications with Counting Semantics,” Nov. 2014.

[15] D. Giannakopoulou and K. Havelund, “Runtime Analysis of Linear Temporal Logic Specifications,” RIACS, 2001.

[16] J. Hannemann and G. Kiczales, “Design pattern implementation in Java and aspectJ,” ACM SIGPLAN Not., vol. 37, no. 11, p. 161, 2002.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “Getting Started with ASPECTJ,” Commun. ACM, vol. 44, no. 10, pp. 59–65, Oct. 2001.

[18] F. Chen, G. Roşu, F. Chen, and G. Roşu, “Mop,” in Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA ’07, 2007, vol. 42, no. 10, p. 569.

[19] D. IEEE Computer Society., P. O. IEEE Computer Society. Technical Council on Software Engineering., C. Lee, and G. Roşu, Proceedings, 34th International Conference on Software Engineering (ICSE 2012) : June 2-9, 2012, Zurich, Switzerland. IEEE, 2012.

[20] F. Chen, M. D'Amorim, and G. Rou, "Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP," Electron. Notes Theor. Comput. Sci., vol. 144, no. 4 SPEC. ISS., pp. 3–20, 2006.

[21] L. Griffiths, A. Shaw, R. Glassey, S. Köhler, B. Bagnall, J. A. B. Moral, J. Stuber, M. P. Scholz, and J. Solorzano, "LeJOS, Java for Lego Mindstorms http://lejos.sourceforge.net/," 2008. [Online]. Available: http://www.lejos.org/ev3.php. [Accessed: 08-June-2015].

APPENDICES

APPENDIX A

soccerPlayers/PlayerInit.java

```
package soccerPlayers;

import java.io.FileNotFoundException;

import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3GyroSensor;
import lejos.hardware.sensor.HiTechnicCompass;
import lejos.robotics.DirectionFinderAdapter;
import lejos.robotics.Gyroscope;
import lejos.robotics.GyroscopeAdapter;
import lejos.robotics.RegulatedMotor;
import lejos.robotics.SampleProvider;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.localization.CompassPoseProvider;
import lejos.robotics.localization.PoseProvider;
import lejos.robotics.navigation.MovePilot;
import lejos.robotics.navigation.Navigator;
import lejos.robotics.navigation.Pose;
import lejos.robotics.navigation.Waypoint;
import lejos.utility.Delay;
import lejos.utility.GyroDirectionFinder;
import soccerControllers.Globals;
import soccerControllers.MotorController;
import soccerControllers.SensorController;
import loggingTools.*;


public class Kicker {
	private MotorController mainMC = null;
	private  SensorController mainSC = null;
	private float[] currBallIR;
	private float currBallSonar;
	private boolean continuePlaying = true;
	private boolean ballKickedAtGoal = false;

	public Kicker(){
        Init();
        currBallIR = new float[2];
        boolean ready = Ready();
}

public void Init() {
                mainMC = new MotorController(MotorPort.A,MotorPort.D,
                MotorPort.C, SensorPort.S1);
                mainSC = new SensorController(SensorPort.S3,SensorPort.S2);
}

public boolean Ready() {
        return mainMC != null && mainMC.MotorsReady() && mainSC != null
        && mainSC.SensorsReady();
}

public  MotorController GetMotorController(){
        return mainMC;
}

public  SensorController GetSensorController(){
        return mainSC;
}

public void SetBallKickedAtGoal(boolean kicked) {
        ballKickedAtGoal = kicked;
}


public void GameOver() {
        System.out.println("GAME OVER");
}

public void Play(){
        mainSC.FlushSensors();
        boolean bif = false;
        boolean hasBall = false;
        boolean tmp = false;
        boolean gotoGoalStatus = false;

        while(!ballKickedAtGoal){
                if(FindBall()){
                        hasBall = this.HasBall();
                        gotoGoalStatus = GotoGoal(true);
                        if(gotoGoalStatus){
                              System.out.println("GOTO GOAL SUCCESS");
                              this.KickBall();
                              bif = this.BallInFront();
                              SetBallKickedAtGoal(bif);
                         }
                         else{
                              System.out.println("Lost BALL!");
                         }
               }
          }
          GameOver();
}



// Wonders for the ball until it has the ball in it's arms
public  boolean FindBall(){
          boolean gotoBallSuccess = false;
          int findTrys = 0;
          do{
               TurnToBall();
               findTrys++;
               if(findTrys > Globals.FIND_BALL_TRY_MAX)
                         return false;
                         gotoBallSuccess = GotoBall();
          }while(!gotoBallSuccess);

          return true;
}

public  boolean TurnToBall(){
          Pose currPos;
          float avgBallAngle = 0;
          float lastBallAngle = 0;
          float currTurnSpeed = 0;
          int newTurnCount = 0;

          boolean bif = false;
          bif = BallInFront();
          if(bif){
               return true;
          }

          do{
               currPos = mainMC.GetPos();
```

```
                avgBallAngle = GetBallAngle();
                if(avgBallAngle < 0)
                        avgBallAngle -= 10.0;
                else if(avgBallAngle > 0)
                        avgBallAngle += 10.0;

                currTurnSpeed = 25;
                mainMC.SetTurnSpeed(currTurnSpeed);
                if(lastBallAngle != avgBallAngle || newTurnCount >=
                 Globals.TURN_FAIL_SAFE_LIMIT){
                        mainMC.Turn(currTurnSpeed, avgBallAngle*-1);
                        newTurnCount = 0;
                }


                lastBallAngle = avgBallAngle;
                newTurnCount++;
                bif = BallInFront();

        }while(!bif);
        return true;
}

public  float GetBallAngle(){
        float finalAngle = (float) 0.0;
        int numAngleVals = 0;
        mainSC.GetIR(Globals.IR_UNMOD,currBallIR);
        mainSC.GetIR(Globals.IR_MOD,currBallIR);

        if(!Float.isNaN(currBallIR[Globals.IR_MOD])){
                finalAngle += currBallIR[Globals.IR_MOD];
                numAngleVals++;
        }
        if(!Float.isNaN(currBallIR[Globals.IR_UNMOD])){
                finalAngle += currBallIR[Globals.IR_UNMOD];
                numAngleVals++;
        }

        // No valid angle returned, use a default 180
        // to try to get the ball in range of IR
        if(numAngleVals == 0){
                finalAngle = 180;
                numAngleVals = 1;
        }
```

```
		return (float) (finalAngle/2.0);
	}


	public  boolean BallInFront(){
		boolean ballClose = false;
		mainSC.GetIR(Globals.IR_UNMOD,currBallIR);
		mainSC.GetIR(Globals.IR_MOD,currBallIR);
		ballClose = BallClose();
		currBallSonar = mainSC.GetSonar();

		if(currBallIR[Globals.IR_UNMOD] == 0 || currBallIR[Globals.IR_MOD]
		== 0)
			return true;
		else if(ballClose)
			return true;
		else
			return false;
	}

	public  boolean BallClose(){
		currBallSonar = mainSC.GetSonar();
		return (currBallSonar <= Globals.SONAR_CLOSE_READING &&
currBallSonar != Globals.SONAR_ERROR_VAL);
	}

	public  boolean HasBall(){
		currBallSonar = mainSC.GetSonar();
		return (currBallSonar <= Globals.SONAR_IN_ARM_READING);
	}

	public  boolean GotoBall(){
		float lastSonar = 0;
		currBallSonar = mainSC.GetSonar();
		lastSonar = currBallSonar;
		boolean hasBall = false;
		boolean bif = false;


		if(currBallSonar != Globals.SONAR_ERROR_VAL &&
		!Float.isInfinite(currBallSonar))
			mainMC.GoForward(7);
		else
			mainMC.GoForward(7);
```

```
        do{
                currBallSonar = mainSC.GetSonar();
                if(currBallSonar < lastSonar &&
                !Float.isInfinite(currBallSonar) && currBallSonar
                != Globals.SONAR_ERROR_VAL)
mainMC.SetForwardSpeed((float)(((currBallSonar/360.0)*Globals.MAX_MOTO
R_SPEED) + 1));
                lastSonar = currBallSonar;
                hasBall = HasBall();
                bif = BallInFront();
        }while(bif && !hasBall);
        hasBall = HasBall();
        return hasBall;
}

public  boolean BallInBufferAngleRange(){
        float avgBallAngle = GetBallAngle();

        return (avgBallAngle >= (-1.0*Globals.BUFFER_ANGLE) &&
avgBallAngle <= Globals.BUFFER_ANGLE);
}

public  boolean GotoGoal(boolean withBall){
        boolean ballClose = false;
        float prevHeading = -1;
        float currHeading = 9999;
        boolean bif = false;

        mainMC.SetForwardSpeed(Globals.MAX_MOTOR_SPEED/3);
        mainMC.SetTurnSpeed(Globals.MAX_MOTOR_SPEED/3);
        Waypoint goalLoc = Globals.GOAL_LOCATION;
        mainMC.GotoPoint(goalLoc.x, goalLoc.y, Float.NaN, false);

        while(!mainMC.InGoalRange()){
                ballClose = BallClose();
                if(withBall){
                        currHeading = mainMC.GetRobotHeading();
                        bif = BallInFront();
                        if(!bif){
                                return false;
                        }
                        if(ballClose){
                                //Dribble
                        }
                }
```

```
		}
		return true;
	}


	public void KickBall(){
		float[] kickBallSeq = {0,1000,1,1000};
		mainMC.GoForward(Globals.MAX_MOTOR_SPEED);
		Delay.msDelay(2000);
		mainMC.MoveArm(kickBallSeq, 100);
		mainMC.Stop();
	}


	public void DribbleBall() {
		mainMC.SetArmPower(35);
		mainMC.OpenArm(200);
		mainMC.CloseArm(200);
	}
}
```

APPENDIX B

soccerPlayers/Kicker.java

```
package soccerPlayers;

import loggingTools.LogFile;
import loggingTools.Logger;

public class PlayerInit {
	public static void main(String[] args) {
		Kicker currKicker = new Kicker();
		currKicker.Play();
	}

}
```

APPENDIX C

soccerControllers/Globals.java

```
package soccerControllers;

import lejos.robotics.navigation.Waypoint;

public class Globals {
	public final static Waypoint GOAL_LOCATION = new Waypoint(100,-100);
	public final static double GOAL_RANGE_THRESHOLD = 70;
	public final static int IR_UNMOD = 0;
	public final static int IR_MOD = 1;
	public final static float SONAR_CLOSE_READING = (float) 0.19;
	public final static float SONAR_IN_ARM_READING = (float) 0.07;
	public final static float SONAR_ERROR_VAL = 0;
	public final static float BUFFER_ANGLE = 30;
	public final static float MAX_MOTOR_SPEED = 30;
	public final static int FIND_BALL_TRY_MAX = 20;
	public final static int TURN_FAIL_SAFE_LIMIT = 15;
}
```

APPENDIX D

soccerControllers/MotorController.java

```
package soccerControllers;

import lejos.hardware.Button;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.motor.UnregulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.hardware.port.Port;
import lejos.hardware.port.TachoMotorPort;
import lejos.hardware.sensor.HiTechnicCompass;
import lejos.robotics.DirectionFinderAdapter;
import lejos.robotics.SampleProvider;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.localization.CompassPoseProvider;
import lejos.robotics.localization.PoseProvider;
import lejos.robotics.navigation.MovePilot;
import lejos.robotics.navigation.Navigator;
import lejos.robotics.navigation.Pose;
import lejos.robotics.navigation.Waypoint;
import lejos.utility.Delay;


public class MotorController {
	private EV3LargeRegulatedMotor leftMotor;		// direct left motor controller
	private EV3LargeRegulatedMotor rightMotor;	// direct right motor controller
	private UnregulatedMotor arm;				// direct arm
controller
	private Navigator roboNav;				// Primary navigator
	private MovePilot movePilot;
	private PoseProvider roboPos;



	private HiTechnicCompass compass;			// Compass for angles in
navigator
	private float[] compassSamples;				// compass sample
values
	private SampleProvider compassSP;			// compass sample provider
	private DirectionFinderAdapter compassDF;
	private float wheelDiam = (float) /*3.3*/4.746;
	private float trackWidth = (float) 6.6 /*7.6*/;


	private Wheel wheel1;
	private Wheel wheel2;
	private Chassis chassis;

	private CompassPoseProvider compassPose;
	private Waypoint lastGotoWayPoint;



	/* Constructor for the MotorController
	 * Parameters:
	 *	1. MotorPort leftMotorPort
	 *		* The port for the left robot motor (A-D)
	 *  2. MotorPort rightMotorPort
	 *		* The port for the right robot motor (A-D)
	 *  3. MotorPort arm
	 *		* The port for the robot motor arm (A-D)
	 * Result:  All motors are initialized, unless incorrect ports are given
	 */
	public MotorController(Port leftMotorPort,Port rightMotorPort, Port armPort,Port
compassPort){
		Init(leftMotorPort,rightMotorPort,armPort,compassPort);
	}

	public void Init(Port leftMotorPort,Port rightMotorPort, Port armPort,Port
compassPort) {
		leftMotor = new EV3LargeRegulatedMotor(leftMotorPort);
		rightMotor = new EV3LargeRegulatedMotor(rightMotorPort);
		arm = new UnregulatedMotor(armPort);

		compass = new HiTechnicCompass(compassPort);

		compassSamples = new float[5];
		compassSP = compass.getAngleMode();
		compassDF = new DirectionFinderAdapter(compassSP);
		wheel1 = WheeledChassis.modelWheel(leftMotor,
		wheelDiam).offset(-1*trackWidth);
				wheel2 = WheeledChassis.modelWheel(rightMotor,
		wheelDiam ).offset(trackWidth);
		chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2
		}, WheeledChassis.TYPE_DIFFERENTIAL);
		movePilot = new MovePilot(chassis);
		compassPose = new
			CompassPoseProvider(movePilot,compassDF);
		roboNav = new Navigator(movePilot);
		roboPos = roboNav.getPoseProvider();

		lastGotoWayPoint = new Waypoint(0,0,0);
	}

	public boolean MotorsReady() {
		if(leftMotor == null || rightMotor == null || arm == null)
			return false;
		return true;
	}



	public void GotoPoint(float xPos, float yPos, float heading, boolean blocking){
		boolean newPointGiven = false;
		Pose currPos = roboPos.getPose();

		roboNav.clearPath();

		if(lastGotoWayPoint.getX() != xPos || lastGotoWayPoint.getY() != yPos ||
lastGotoWayPoint.getHeading() != heading){
			newPointGiven = true;
		}
		else if(!roboNav.isMoving() && currPos.getX() != xPos &&
currPos.getY() != yPos && currPos.getHeading() != heading){
			newPointGiven = true;
		}

		if(newPointGiven){
			roboNav.clearPath();
			if(!Float.isNaN(heading)){
				roboNav.goTo(xPos, yPos, heading);
			}
			else{
				roboNav.goTo(xPos, yPos);
			}

			// Block return until the robot is at the requested spot if requested
			if(blocking){
				roboNav.waitForStop();
			}
		}

	}

	public void GoForward(double speed){
		SetForwardSpeed(speed);
		movePilot.forward();
	}

	public Pose GetPos(){
		return roboPos.getPose();
	}

	public void SetForwardSpeed(double speed){
		movePilot.setLinearSpeed(speed);
	}

	public void SetTurnSpeed(double speed){
		movePilot.setAngularSpeed(speed);
	}

	public void Turn(double speed, float angle){
		SetTurnSpeed(speed);
		movePilot.rotate(angle);
	}

	public boolean RobotTurning(){
		float leftMotorSpeed = leftMotor.getRotationSpeed();
		float rightMotorSpeed = rightMotor.getRotationSpeed();

		if(leftMotorSpeed > 0 && rightMotorSpeed < 0)
			return true;
		else if(leftMotorSpeed < 0 && rightMotorSpeed > 0)
			return true;
		else
			return false;
	}

	public boolean RobotMoving(){
		return movePilot.isMoving();
	}


	public boolean InGoalRange(){
		double distToGoal = Math.sqrt(
                                Math.pow(Globals.GOAL_LOCATION.getX() -
GetRobotX(),2) + Math.pow(Globals.GOAL_LOCATION.getY() - GetRobotY(),2)
                                );

                if(distToGoal <= Globals.GOAL_RANGE_THRESHOLD)
                        return true;

                return false;

        }

        public boolean IsSamePos(Waypoint otherPoint){
                Pose currPos = GetPos();
                return otherPoint.x == currPos.getX() && otherPoint.y == currPos.getY();
        }

        public boolean InRange(Waypoint refPoint, float thresDist){
                Pose currPos = GetPos();
                return refPoint.distance(currPos.getX(), currPos.getY()) <= thresDist;
        }

        public void MoveArm(float[] armPosSeq, int speed){
                arm.setPower(speed);
                for(int i = 0; i < armPosSeq.length;i+=2){
                        if(armPosSeq[i] == 0)
                                arm.forward();
                        else
                                arm.backward();
                        Delay.msDelay((long) armPosSeq[i+1]);
                }
        }

        public boolean ArmMoving(){
                return arm.isMoving();
        }
        public void Stop(){
                roboNav.clearPath();
                roboNav.stop();
        }




        public float GetRobotX(){
                return GetPos().getX();
```

```
	}

	public float GetRobotY(){
		return GetPos().getY();
	}

	public float GetRobotHeading(){
		return GetPos().getHeading();
	}

	public void OpenArm(int armDelay) {
		arm.backward();
		Delay.msDelay(armDelay);
	}

	public void CloseArm(int armDelay) {
		arm.forward();
		Delay.msDelay(armDelay);
	}

	public void SetArmPower(int power) {
		arm.setPower(power);
	}

}
```

APPENDIX E

soccerControllers/SensorController.java

```
package soccerControllers;

import lejos.hardware.port.Port;
import lejos.hardware.sensor.EV3UltrasonicSensor;
import lejos.hardware.sensor.HiTechnicIRSeekerV2;
import lejos.robotics.SampleProvider;
import loggingTools.LogFile;
import loggingTools.Logger;

public class SensorController {
	private EV3UltrasonicSensor sonarSensor;	// Actual Sonar sensor
	private SampleProvider sonarSP;				// Sonar Data Grabber
	private float[] sonarSamples;			// Actual sonar values

	private HiTechnicIRSeekerV2 irSensor;		// Actual IR Seeker
	private SampleProvider irSP[];				// IR Data Grabber: 0
-> unMod, 1 -> Mod
	private float[] irSamples;				// Actual IR values

	private float ballDirMod = 0;
	private float ballDirUnMod = 0;
	private float sonarRead = 0;



	public SensorController(Port irPort, Port sonarPort){
		Init(irPort, sonarPort);
	}


	public void Init(Port irPort, Port sonarPort) {
		sonarSensor = new EV3UltrasonicSensor(sonarPort);
		sonarSP = sonarSensor.getDistanceMode();
		sonarSamples = new float[5];

		irSensor = new HiTechnicIRSeekerV2(irPort);
		irSP = new SampleProvider[2];
		irSP[0] = irSensor.getUnmodulatedMode();
		irSP[1] = irSensor.getModulatedMode();
		irSamples = new float[5];
```

```
}


public boolean SensorsReady() {
	if(irSensor == null || sonarSensor == null)
		return false;
	return true;
}


public boolean GetIR(int mode,float[] irReturn){
	irSP[mode].fetchSample(irSamples, 0);
	if(mode == Globals.IR_UNMOD) {
		this.ballDirUnMod = irSamples[0];
	}
	else {
		this.ballDirMod = irSamples[0];
	}

	irReturn[mode] = irSamples[0];

	return !Float.isNaN(irReturn[mode]);
}

public float GetSonar(){
	sonarSP.fetchSample(sonarSamples, 0);
	this.sonarRead = sonarSamples[0];
	return sonarSamples[0];
}

public float GetLastSonar(){
	return this.sonarRead;
}

public float GetLastModIR(){
	return this.ballDirMod;
}

public float GetLastUnModIR(){
	return this.ballDirUnMod;
}


public void FlushSensors(){
	float[] tmp = new float[2];
```

```
			System.out.println("-----Flushing Sensors Start-----");
			if(!GetIR(Globals.IR_MOD,tmp)){
				System.out.println("IR_MOD -- NO BALL");
			}
			if(!GetIR(Globals.IR_UNMOD,tmp)){
				System.out.println("IR_UNMOD -- NO BALL");
			}
			if(GetSonar() == Globals.SONAR_ERROR_VAL){
				System.out.println("TMP Sonar Error");
			}
			System.out.println("-----Flushing Sensors End-----");
		}
	}
```

APPENDIX F

stateTools/ChangeEvent.java

```
package stateTools;

public enum ChangeEvent {
	NONE, IR_MOD, IR_UNMOD,SONAR
}
```

APPENDIX G

stateTools/State.java

```
package stateTools;

// State status options
public enum State {
	INIT, TURN_TO_BALL, GOTO_BALL, TURN_TO_GOAL,
DRIBBLE_TO_GOAL,
	KICK_BALL_TO_GOAL, GAME_OVER
}
```

APPENDIX H

stateTools/StateCheck.java

```
package stateTools;

import soccerControllers.*;
import soccerPlayers.*;


// State status options


public final class StateCheck{
	public static boolean sonarSuccess = false;
	public static boolean modIrSuccess = false;
	public static boolean unModIrSuccess = false;

	public static float irModRead = -1;
	public static float irUnModRead = -1;
	public static float sonarRead = -1;

	public static boolean inGoalRange = false;
	public static boolean ballInFront = false;
	public static boolean ballClose = false;
	public static boolean ballKickable = false;
	public static boolean robotMoving = false;
	public static boolean robotTurning = false;
	public static boolean bifStateGen = false;
	public static boolean hasBall = false;
	public static boolean armMoving = false;


	public static boolean BallInFront(float sonar, float irMod, float irUnMod){
		if(sonarRead < Globals.SONAR_CLOSE_READING){
			return true;
		}
		//************** IR Check -- Modulated*****************//
		if(!Float.isNaN((irMod))){
			if(irMod == 0){
				return true;
			}
		}
		//************** IR Check -- UnModulated***************//
		if(!Float.isNaN(irUnMod)){
			if(irUnMod == 0){
				return true;
			}
		}
		return false;
	}

	public static boolean BallClose(float sonar){
		if(sonar < Globals.SONAR_CLOSE_READING)
			return true;
		return false;
	}

	public static boolean BallKickable(float sonar){
		if(sonar < Globals.SONAR_CLOSE_READING)
			return true;
		return false;
	}

	public static boolean HasBall(float sonar){
		return (sonar <= Globals.SONAR_IN_ARM_READING);
	}

	// NOTE: Returning true => that the robot should be in this state
	public static boolean TurnToBallState(Kicker currMK){
		if(!ballInFront && !ballClose && !ballKickable)
			return true;
		else
			return false;
	}

	// Going to ball state
	public static boolean GotoBallState(Kicker currMK){
		// Check if the robot should remain in the goto ball state
		if(ballInFront && /*robotMoving &&*/ !hasBall/*&& !ballClose*/)
			return true;
		else
			return false;
	}

	// Turning to goal state
	public static boolean TurnToGoalState(Kicker currMK){
		// Ball should remain in front of the robot while turning
		if(ballInFront && /*ballClose*/ hasBall&& robotTurning)
			return true;
		else
			return false;
	}


	// !!!NOTE!!! --> NO LONGER DRIBBLE -- IT IS GOTO GOAL
	public static boolean DribbleBallState(Kicker currMK){
		// Ball should be near and in front robot while moving to goal
		if(!inGoalRange && ballInFront && ballClose /*&& armMoving*/)
			return true;
		else
			return false;
	}


	// Kick state
	public static boolean KickBallAtGoal(Kicker currMK){
		// Ball should be with the robot and in the goal range until kick
		if(inGoalRange && ballInFront && ballClose)
			return true;
		else
			return false;
	}



	//****** GEN ****************
	// NOTE: Returning true => that the robot should be in this state
	public static boolean Gen_TurnToBallState(Kicker currMK){
		return StateCheck.TurnToBallState(currMK);
	}

	// Going to ball state
	public static boolean Gen_GotoBallState(Kicker currMK){
		return StateCheck.GotoBallState(currMK);
	}

	// Turning to goal state
	public static boolean Gen_TurnToGoalState(Kicker currMK){
		return StateCheck.TurnToGoalState(currMK);
	}
	// !!!NOTE!!! --> NO LONGER DRIBBLE -- IT IS GOTO GOAL
	public static boolean Gen_DribbleBallState(Kicker currMK){
		return StateCheck.DribbleBallState(currMK);
	}



	// Kick state
	public static boolean Gen_KickBallAtGoal(Kicker currMK){
		return StateCheck.KickBallAtGoal(currMK);
	}
	//****** GEN ****************




	// Get the state of the robot
	// NOTE: This may be the incorrect way of doing this -- depending on what is
needed
	public static State GetState(ChangeEvent bifTriggered, Kicker currMK){

		//System.out.println("*PERFORMING STATE CHECK*");

		sonarRead = currMK.GetSensorController().GetLastSonar();
		// If any sensor reads as ball in front, then set the
		// appropriote flags
		if(bifTriggered != null)
			ballInFront = true;
		else
			ballInFront = false;

		// NOTE: this assumes that sonar is up to date
		ballClose = BallClose(sonarRead);
		ballKickable = BallKickable(sonarRead);
		hasBall = HasBall(sonarRead);



		inGoalRange = currMK.GetMotorController().InGoalRange();
		robotMoving = currMK.GetMotorController().RobotMoving();
		robotTurning = currMK.GetMotorController().RobotTurning();
		armMoving = currMK.GetMotorController().ArmMoving();


		boolean inTurnToBallState = TurnToBallState(currMK);
		boolean inGotoBallState = GotoBallState(currMK);
		boolean inTurnToGoalState = TurnToGoalState(currMK);
		boolean inDribbleBallState = DribbleBallState(currMK);
		boolean inKickBallAtGoal = KickBallAtGoal(currMK);

		//System.out.println("STATE INFO: "+ballInFront+" , "+robotMoving +"
, "+hasBall);


		if(inTurnToBallState)
			return State.TURN_TO_BALL;
		else if(inGotoBallState)
			return State.GOTO_BALL;
		else if(inTurnToGoalState)
			return State.TURN_TO_GOAL;
		else if(inDribbleBallState)
			return State.DRIBBLE_TO_GOAL;
		else if(inKickBallAtGoal)
			return State.KICK_BALL_TO_GOAL;
		else
			return State.INIT;

	}

	// Print the current state of the robot
	public static void PrintState(State currState){
		switch(currState){
			case TURN_TO_BALL:
				System.out.println("In -TURN_TO_BALL- State");
				break;
			case TURN_TO_GOAL:
				System.out.println("In -TURN_TO_GOAL- State");
				break;
			case GOTO_BALL:
				System.out.println("In -GOTO_BALL- State");
				break;
			case DRIBBLE_TO_GOAL:
				System.out.println("In -DRIBBLE_TO_GOAL- State");
				break;
			case KICK_BALL_TO_GOAL:
				System.out.println("In -KICK_BALL_TO_GOAL- State");
				break;
			case INIT:

				break;
			default:
				System.out.println("UNDEFINED STATE");
				break;
```

```
                }
        }
}
```

APPENDIX I

RobotStateMachine.aj

```
/* Developer: Taylor Harvin
 * Date Last Changed: 8/14/2016
 * Purpose: Provide state properties of the robot
 *
 */

import soccerControllers.*;
import soccerPlayers.*;
import stateTools.*;
import lejos.robotics.navigation.Waypoint;
import lejos.robotics.RegulatedMotor;
import lejos.hardware.motor.UnregulatedMotor;
import lejos.robotics.navigation.Navigator;
import lejos.hardware.sensor.EV3UltrasonicSensor;
import lejos.hardware.sensor.HiTechnicCompass;
import lejos.hardware.sensor.HiTechnicIRSeekerV2;
import lejos.robotics.SampleProvider;
import lejos.utility.Delay;


aspect RoboStateMachine{
	private long lastStateCheck = System.currentTimeMillis();
	private final long PING_TIME_LIMIT = 10000;
	private boolean sonarSet = false;
	private boolean irModSet = false;
	private boolean irUnModSet = false;
	private boolean armOpen = false;
	private ChangeEvent bifTrigger = null;

	private static State lastState = State.GAME_OVER;

	//*************PC Flags************************
	private boolean bifFlag = false;
	private boolean bcFlag = false;
	private boolean bKickable = false;
	//*******************************************


	// Enough information is set to do a valid state check
	public boolean readyForStateCheck(Kicker MK){
		// All sensor values are set -- allow state check
		// Otherwise -- look at the
		if(sonarSet && irModSet && irUnModSet)
			return true;
		else if(sonarSet && MK.GetSensorController().GetLastSonar() <=
Globals.SONAR_IN_ARM_READING)
			return true;
		else if(irModSet && MK.GetSensorController().GetLastModIR() == 0)
			return true;
		else if(irUnModSet && MK.GetSensorController().GetLastUnModIR()
== 0)
			return true;
		else
			return false;
	}

	public void resetStatePreCheck(){
		sonarSet = false;
		irModSet = false;
		irUnModSet = false;

		bifFlag = false;
		bcFlag = false;
		bKickable = false;
		bifTrigger = null;
	}

	// Generate the ball in front true event on-demand rather than through Kicker
	public void Kicker.generateBallInFrontState(){
		//System.out.println("Generated BIF");
	}

	// Check the current ball in front state from the Kicker (without triggering any
events)
	public boolean Kicker.checkBallInFront(Kicker currMK){
		return currMK.BallInFront();
	}


	// Generate the ball close true event on-demand rather than through Kicker
	public void Kicker.generateBallCloseState(){
		//System.out.println("Generated Ball Close");
	}
	// Check the current ball close state from the Kicker (without triggering any
events)
	public boolean Kicker.checkBallClose(Kicker currMK){
		return currMK.BallClose();
	}
```

```
	// Generate the current state on-demand here rather than directly from state check
method
	public void Kicker.generateStateEvent(State currState, Kicker currMK){
		if(currState != lastState){
			switch(currState){
				case TURN_TO_BALL:
					//System.out.println("GEN TURN_TO_BALL");
					// NOTE: Ensure that wonder event is triggered
before BIF
					StateCheck.Gen_TurnToBallState(currMK);
					lastState = currState;
					break;
				case GOTO_BALL:
					//System.out.println("GEN GOTO_BALL");
					StateCheck.Gen_GotoBallState(currMK);
					lastState = currState;
					break;
				case TURN_TO_GOAL:
					//System.out.println("GEN TURN_TO_GOAL");
					StateCheck.Gen_TurnToGoalState(currMK);
					lastState = currState;
					break;
				case DRIBBLE_TO_GOAL:
					//System.out.println("GEN
DRIBBLE_TO_GOAL");
					StateCheck.Gen_DribbleBallState(currMK);
					lastState = currState;
					break;
				case KICK_BALL_TO_GOAL:
					//FORCE MOVE BALL
```

```
                                                  //System.out.println("!!!MOVE BALL
NOW!!!");
                                                  //Delay.msDelay(5000);
                                        // FORCE MOVE BALL

                                        //System.out.println("GEN BALL CLOSE");
                                        //currMK.BallClose();
                                        //System.out.println("GEN
KICK_BALL_TO_GOAL");
                                        StateCheck.Gen_KickBallAtGoal(currMK);
                                        lastState = currState;
                                        break;
                              default:
                                        //System.out.println(currState);
                                        //System.out.println("GEN NONE");
                                        break;
                    }

          }

     }




     //POINTCUT
SECTION***************************************************************
***********
     // General pointcut to allow for access to Kicker object for other pointcuts
(through cflowbelow)
     //pointcut PlayPC(Kicker MK) : call(public void Kicker.Play()) && target(MK);
     pointcut BallInFrontPC(Kicker MK) : call(public boolean Kicker.BallInFront())
&& target(MK);
     pointcut ballClosePC(Kicker MK) : call(public boolean Kicker.BallClose()) &&
target(MK);
     //pointcut ballKickablePC(Kicker MK) : call(public boolean
Kicker.ballKickable(boolean)) && target(MK);
     //ADVICE
SECTION***************************************************************
**************
```

```
	//BIF -- Cflow 
Section***************************************************************************
************
	// IR -- Mod advice, handle change in IR MOD value (after new ping)
	pointcut irModChange_BIF(Kicker MK) : cflowbelow(BallInFrontPC(MK)) && 
set(float SensorController.ballDirMod)&& within(SensorController);
	// IR -- Mod advice, handle change in IR MOD value (after new ping)
	after(Kicker MK, float newIrMod) :irModChange_BIF(MK) && 
args(newIrMod){
		irModSet = true;


		if(newIrMod == 0){
			bifFlag = true;
			bifTrigger = ChangeEvent.IR_MOD;
		}

		if(/*bifFlag ||*/ readyForStateCheck(MK)){
			//System.out.println("***IR MOD Changed BIF***");
			State currState = StateCheck.GetState(bifTrigger, MK);
			MK.generateStateEvent(currState,MK);
			//StateCheck.PrintState(currState);
			resetStatePreCheck();
		}
	}


	// IR -- Un-Mod advice, handle change in IR UN-MOD value (after new ping)
	pointcut irUnModChange_BIF(Kicker MK) : cflowbelow(BallInFrontPC(MK)) 
&& set(float SensorController.ballDirUnMod)&& within(SensorController);
	// IR -- Un-Mod advice, handle change in IR UN-MOD value (after new ping)
	after(Kicker MK, float newIrUnMod):irUnModChange_BIF(MK) && 
args(newIrUnMod){
		irUnModSet = true;

		if(newIrUnMod == 0){
			bifFlag = true;
			bifTrigger = ChangeEvent.IR_UNMOD;
		}

		if(/*bifFlag ||*/ readyForStateCheck(MK)){
			//System.out.println("***IR UN-MOD Changed BIF***");
			//StateCheck.GetState(ChangeEvent.IR_UNMOD, MK);
			State currState = StateCheck.GetState(bifTrigger, MK);
			MK.generateStateEvent(currState,MK);
```

```
				//StateCheck.PrintState(currState);
				resetStatePreCheck();
			}
		}

		// Sonar -- Sonar advice, handle change in Sonar value (after new ping)
		pointcut sonarChange_BIF(Kicker MK) : cflowbelow(BallInFrontPC(MK)) &&
set(float SensorController.sonarRead)&& within(SensorController);
		// Sonar -- Sonar advice, handle change in Sonar value (after new ping)
		after(Kicker MK, float newSonar):sonarChange_BIF(MK) && args(newSonar){
			sonarSet = true;

			if(newSonar <= Globals.SONAR_IN_ARM_READING){
				bifFlag = true;
				bifTrigger = ChangeEvent.SONAR;
			}

			if(bifFlag || readyForStateCheck(MK)){
				//System.out.println("***Sonar Changed BIF***");
				State currState = StateCheck.GetState(bifTrigger, MK);
				MK.generateStateEvent(currState,MK);
				//StateCheck.PrintState(currState);
				resetStatePreCheck();
			}
		}
		//**************************************************************
**********************************


		//Ball Close -- Cflow
Section**************************************************************
***********

		// Sonar -- Sonar advice, handle change in Sonar value (after new ping)
		pointcut sonarChange_BC(Kicker MK) : cflowbelow(ballClosePC(MK)) &&
set(float SensorController.sonarRead)&& within(SensorController);
		// Sonar -- Sonar advice, handle change in Sonar value (after new ping)
		after(Kicker MK, float newSonar):sonarChange_BC(MK) && args(newSonar){
			sonarSet = true;

			if(newSonar <= Globals.SONAR_IN_ARM_READING){
				bifFlag = true;
				bifTrigger = ChangeEvent.SONAR;
			}
		if(bifFlag || readyForStateCheck(MK)){
			//System.out.println("***Sonar Changed BC***");
			State currState = StateCheck.GetState(bifTrigger, MK);
			MK.generateStateEvent(currState,MK);
			//StateCheck.PrintState(currState);
			resetStatePreCheck();
		}
	}
	//*****************************************************************
***********************************************************


	// Sonar -- Sonar advice, handle change in Sonar value (after new ping)
	//pointcut sonarChange_bk(Kicker MK) : cflowbelow(ballKickablePC(MK)) &&
set(float SensorController.sonarRead)&& within(SensorController);
	// Sonar -- Sonar advice, handle change in Sonar value (after new ping)
	/*after(Kicker MK, float newSonar):sonarChange_bk(MK) && args(newSonar){
		sonarSet = true;

		if(newSonar <= Globals.SONAR_IN_ARM_READING){
			bifFlag = true;
			bifTrigger = ChangeEvent.SONAR;
		}

		if(bifFlag || readyForStateCheck(MK)){
			//System.out.println("***Sonar Changed BK***");
			State currState = StateCheck.GetState(bifTrigger, MK);
			MK.generateStateEvent(currState,MK);
			//StateCheck.PrintState(currState);
			resetStatePreCheck();
		}
	}*/
	//*****************************************************************
***********************************************************



	// Trigger needed events for after turn to ball state
	//pointcut turnto_ball_state_exit(Kicker MK) : call(public boolean
Kicker.TurnToBall()) && this(MK);
	//after(Kicker MK):turnto_ball_state_exit(MK){
		//System.out.println("TURN Event EXIT");

		//************BIF FAIL TEST*************
		/*System.out.println("MOVE BALL FOR FAIL -- NOW!");
		//Delay.msDelay(5000);
		//System.out.println("Delay Finished!");*/
		//***************************************

		// Generate ballinfront_true event if the ball is in front
		//if(MK.checkBallInFront(MK))
			//MK.generateBallInFrontState();
	//}


	// ARM*************************************
	/*pointcut openArmPC(MotorController MC) : call(public void
MotorController.openArm(int)) && target(MC);
	after(MotorController MC):openArmPC(MC){
		System.out.println("***Arm Open***");
		armOpen = true;
	}
	pointcut closeArmPC(MotorController MC) : call(public void
MotorController.closeArm(int)) && target(MC);
	after(MotorController MC):closeArmPC(MC){
		System.out.println("***Arm Close***");
		armOpen = false;
	}*/
	//*****************************************

}
```

APPENDIX J

AlwaysIR.mop

```
// From LTL 5
// 5.
//package mop;

import java.io.*;
import java.util.*;
import soccerPlayers.*;
import soccerControllers.*;
import loggingTools.*;

AlwaysIR(Kicker MK) {
        SensorController currSC = null; // Allows for usage of SC in the ltl violation
        Kicker currMK = null;

        /*event play_before before(Kicker MK):
                call(public void Kicker.Play()) && target(MK){
                        System.out.println("Before Play EVENT");

        Logger.log(LogFile.ALWAYS_IR_EVENTS,"Before_Play_EVENT");
                        currMK = MK;
        }*/


        // Handle for the turntoball event
        event ir_read_true after(SensorController SC)        returning(boolean res):
                call(public boolean SensorController.GetIR(int,float[])) && condition(res)
&& target(SC){
                        currSC = SC;
                        //System.out.println("IR read TRUE EVENT");

        //Logger.log(LogFile.ALWAYS_IR_EVENTS,"IR_read_TRUE_EVENT");
        }

        event ir_read_false after(SensorController SC) returning(boolean res):
                call(public boolean SensorController.GetIR(int,float[])) &&
condition(!res) && target(SC){
                        currSC = SC;
                        //System.out.println("IR read FALSE EVENT");

        //Logger.log(LogFile.ALWAYS_IR_EVENTS,"IR_read_FALSE_EVENT");
```

```
	}

	ltl: [] ir_read_true
	@violation {
		//Logger.log(LogFile.ALWAYS_IR_EVENTS,"!!!IR Read Fail LTL 
FAIL!!!");
		/*System.out.println("!!!IR Read Fail LTL FAIL!!!");*/
		//System.out.println("Flush IR");
		//float[] tmpIR = new float[2];
		// Try to rest IR -- Loop forever if IR is in complete fail
		//while(currSC.GetIR(Globals.IR_MOD,tmpIR)){
			//currSC.FlushSensors();
			//System.out.println("Flush IR Loop");
		//}
		// DO A DEFAULT 30 degree TURN for Find
		//__RESET;
	}
}
```

APPENDIX K

BallCloseAtKick.mop

```
//package mop;

import java.io.*;
import java.util.*;
import soccerPlayers.*;
import soccerControllers.*;
import loggingTools.*;
import stateTools.*;

BallCloseAtKick(Kicker MK) {
        Kicker currMK = null;
        event kick_ball_state_true after(Kicker MK) returning(boolean res) :
                        call(public boolean StateCheck.Gen_KickBallAtGoal(Kicker)) &&
condition(res) && args(MK){
                                //System.out.println("Kick Event TEST TRUE");

        Logger.log(LogFile.BALL_CLOSE_AT_KICK,"kick_ball_state_true");
                        }

        event kick_ball_before before(Kicker MK):
                call(public void Kicker.KickBall()) && target(MK){
                        currMK = MK;
                        //System.out.println("Ball Close TRUE EVENT");

        Logger.log(LogFile.BALL_CLOSE_AT_KICK,"kick_ball_before");
        }

        //ltl: [](kick_ball_state_true => ((<*>ballclose_true) and (!(*)ballclose_false)))
        ltl: kick_ball_before => (*) kick_ball_state_true
        @violation{
                Logger.log(LogFile.BALL_CLOSE_AT_KICK,"!!!BallCloseAtKick LTL
Violated!!!");
                //System.out.println("!!!BallCloseAtKick LTL Violated!!!");
                //System.out.println("Re-attempt Find Ball "+currMK.FindBall());
                while(!currMK.FindBall()){
                        System.out.println("Re-attempt Find Ball "+currMK.FindBall());
                }
                currMK.GotoGoal(true);
                __RESET;
        }
}
```

APPENDIX L
BallInFrontAfterTurnUntilKick.mop

```
// From LTL 2
// 2.

//package mop;

import java.io.*;
import java.util.*;
import soccerPlayers.*;
import soccerControllers.*;
import loggingTools.*;
import stateTools.*;

BallInFrontAfterTurnUntilKick(Kicker MK) {
	Kicker currMK = null; // Allows for usage of MK in the ltl violation

	// Handle for the Gotoball event
	event turn_to_ball_state_true after(Kicker MK) returning(boolean res) :
			call(public boolean StateCheck.Gen_TurnToBallState(Kicker))
&& condition(res) && args(MK){
				//System.out.println("turn_to_ball_state_true");

	Logger.log(LogFile.TURN_GTB,"turn_to_ball_state_true");
			}



	event ballinfront_true after(Kicker MK) returning(boolean res):
		call(public boolean Kicker.BallInFront()) && condition(res) &&
target(MK){
		currMK = MK;
		Logger.log(LogFile.TURN_GTB,"ballinfront_true");
	}

	event ballinfront_false after(Kicker MK) returning(boolean res):
		call(public boolean Kicker.BallInFront()) && condition(!res) &&
target(MK){
		currMK = MK;
		Logger.log(LogFile.TURN_GTB,"ballinfront_false");
	}


	event kick_ball_after after(Kicker MK):
		call(public void Kicker.KickBall()) && target(MK){
```

```
                    currMK = MK;
                    Logger.log(LogFile.TURN_GTB,"kick_ball_after");
        }

        //ltl: [](Gotoball_true => o ballinfront_true)
        ltl: [](turn_to_ball_state_true => o(ballinfront_true U kick_ball_after))
        //ltl: [](gotoball_state_true => o(Gen_DribbleBallState U kick_ball_after))
        @violation {
                __RESET;
                Logger.log(LogFile.TURN_GTB,"!!!HasBallUntilAfterKick LTL
FAIL!!!");
                //System.out.println("!!!BallInFrontBetweenTurnAndKick LTL
FAIL!!!");
                //System.out.println("Re-Attempting FindAndGrabBall");
                while(!currMK.FindBall());
                __RESET;
        }
}
```

APPENDIX M
EventuallyKicksToGoal.mop

```
//package mop;

import java.io.*;
import java.util.*;
import soccerPlayers.*;
import soccerControllers.*;
import loggingTools.*;
import stateTools.*;

EventuallyKicksToGoal(Kicker MK) {
        Kicker currMK = null;
        event kick_ball after(Kicker MK):
                        call(public void Kicker.KickBall()) && target(MK){
                                currMK = MK;
                                //System.out.println("Kick Event TRUE");
                                Logger.log(LogFile.EVENTUALLY_KICKS,"kick_ball");
                        }

        event game_over before(Kicker MK):
                call(public void Kicker.GameOver()) && target(MK){
                        currMK = MK;
                        //System.out.println("GameOver EVENT");
                        Logger.log(LogFile.EVENTUALLY_KICKS,"game_over
EVENT");
        }


        event kick_ball_state_true after(Kicker MK) returning(boolean res):
                        call(public boolean StateCheck.KickBallAtGoal(Kicker)) &&
condition(res) &&  !execution(State StateCheck.GetState(ChangeEvent, Kicker)) &&
args(MK){
                                currMK = MK;
                                //System.out.println("dribble_ball_state_true TRUE");

        Logger.log(LogFile.EVENTUALLY_KICKS,"kick_ball_state_true");
                        }


        ltl: (game_over => <*> kick_ball_state_true) and (game_over => <*>kick_ball)
        @violation{
                Logger.log(LogFile.EVENTUALLY_KICKS,"!!!EventuallyKicksToGoal
LTL Violated!!!");
                System.out.println("!!!EventuallyKicksToGoal Violated!!!");
                currrMK.KickBall();
                currMK.SetBallKickedAtGoal(currMK.BallInFront());
                //__RESET;
        }
}
```

APPENDIX N
GoToGoalUntilInRange.mop

```
//package mop;

import java.io.*;
import java.util.*;
import soccerPlayers.*;
import soccerControllers.*;
import loggingTools.*;
import stateTools.*;

GoToGoalUntilInRange(Kicker MK) {
        Kicker currMK = null;
        event dribble_ball_state_true after(Kicker MK) returning(boolean res):
                        call(public boolean StateCheck.Gen_DribbleBallState(Kicker))
&& condition(res) && args(MK){
                                currMK = MK;
                                //System.out.println("dribble_ball_state_true TRUE");

        Logger.log(LogFile.GOTO_GOAL_UNTIL_IN_RANGE,"dribble_ball_state_tru
e");
                        }

        event turn_to_goal_state_true after(Kicker MK) returning(boolean res):
                        call(public boolean StateCheck.Gen_TurnToGoalState(Kicker))
&& condition(res) && args(MK){
                                currMK = MK;
                                //System.out.println("turn_to_goal_state_true TRUE");

        Logger.log(LogFile.GOTO_GOAL_UNTIL_IN_RANGE,"turn_to_goal_state_tru
e");
                        }


        event goto_goal_true after(Kicker MK) returning(boolean res):
                call(public boolean Kicker.GotoGoal(boolean)) && condition(res) &&
target(MK){
                        //System.out.println("goto_goal_true EVENT");

        Logger.log(LogFile.GOTO_GOAL_UNTIL_IN_RANGE,"goto_goal_true
EVENT");
        }
        event goto_goal_false after(Kicker MK) returning(boolean res):
                call(public boolean Kicker.GotoGoal(boolean)) && condition(!res) &&
target(MK){
```

```
                        //System.out.println("goto_goal_false EVENT");

        Logger.log(LogFile.GOTO_GOAL_UNTIL_IN_RANGE,"goto_goal_false
EVENT");
        }

        event kick_ball_state_true after(Kicker MK) returning(boolean res) :
                        call(public boolean StateCheck.Gen_KickBallAtGoal(Kicker)) &&
condition(res) && args(MK){
                                //System.out.println("Kick Event TEST TRUE");

        Logger.log(LogFile.GOTO_GOAL_UNTIL_IN_RANGE,"kick_ball_state_true")
;
                        }


        ltl: [](turn_to_goal_state_true => o(dribble_ball_state_true U goto_goal_true))
        @violation{

        Logger.log(LogFile.GOTO_GOAL_UNTIL_IN_RANGE,"!!!GoToGoalUntilInR
ange LTL Violated!!!");
                System.out.println("!!!GoToGoalUntilInRange Violated!!!");
                currMK.KickBall();
                __RESET;
        }
}
```

APPENDIX O
Init.mop

```
//package mop;

import java.io.*;
import java.util.*;
import soccerPlayers.*;
import soccerControllers.*;
import loggingTools.*;
import stateTools.*;
import lejos.hardware.Button;

Init(Kicker MK) {
	Kicker currMK = null;

	event ready_true after(Kicker MK) returning(boolean res):
		call(public boolean Kicker.Ready()) && condition(res) && target(MK){
			currMK = MK;
			//System.out.println("ready_true EVENT");
			Logger.log(LogFile.INIT,"ready_true EVENT");
	}

	event ready_false after(Kicker MK) returning(boolean res):
		call(public boolean Kicker.Ready()) && condition(!res) && target(MK){
			currMK = MK;
			//System.out.println("ready_false EVENT");
			Logger.log(LogFile.INIT,"ready_false EVENT");
	}


	event play_before before(Kicker MK):
		call(public void Kicker.Play()) && target(MK){
			currMK = MK;
			//System.out.println("Play Before EVENT");
			Logger.log(LogFile.INIT,"Play Before EVENT");
	}




	ltl: [](play_before => <*> ready_true)
	@violation{
		Logger.log(LogFile.INIT,"!!!Init LTL Violated!!!");
		System.out.println("!!!Init Violated!!!");
```

```
                while(!currMK.Ready()){
                        System.out.println("Please plug in all cables correctly, then press
any button.");
                        Button.waitForAnyPress();
                        currMK.Init();
                }

                __RESET;
        }
}
```

APPENDIX P

loggingTools/LogFile.java -- Logger.java

```
package loggingTools;

// State status options
public enum LogFile {
	ALL_EVENTS, TURN_GTB, GTB_TURN, GTB_TTG, ALWAYS_IR_TIME,
ALWAYS_IR_EVENTS,HAS_BALL_AFTER_GOTO_BALL,BALL_CLOSE_AT_KI
CK,EVENTUALLY_KICKS,GOTO_GOAL_UNTIL_IN_RANGE,INIT,ROBO_STATE
_MACHINE
}
//-----------------------------------------------------------------------------------------------------


package loggingTools;


import java.io.*;

public class Logger{
	private static PrintWriter  outFile = null;
	private static FileOutputStream outFileStream = null;
	private static LogFile lastLogType;
	private static long startTime = 0;
	private static long endTime = 0;

	public static void setFile(LogFile fileType) throws FileNotFoundException{
		outFileStream = new FileOutputStream(new File(fileType + ".out"),true);
		//outFileStream = new FileOutputStream(new File("EventList.out"),true);
	}

	public static void log(LogFile fileType, String msg){



		try{
			setFile(fileType);
			outFile = new PrintWriter (outFileStream);
			if(lastLogType != fileType){
				lastLogType = fileType;
				outFile.append("***FROM: "+fileType+" ***\n");
			}
```

```
				outFile.append(msg+"\n");
				//outFileStream.println(msg);
			}
			catch(FileNotFoundException e){
				//System.out.println("Error");
			}
			outFile.close();
		}

		public static void log_time(LogFile fileType,String msg,boolean isStart){
			if(isStart){
				startTime = System.currentTimeMillis();
			}
			else{
				endTime = System.currentTimeMillis();
				long finalTime = (endTime - startTime);
				log(fileType,msg+" : "+finalTime + "ms");
			}
		}
	}
```