High Assurance Controller of Self-balancing Robot II

Product Design Specification

Capstone Group 10

Forsman, Kulakevich, Mejia-Rodriguez, Wang

Electrical and Computer Engineering
Portland State University
2020-01-21

Revision History

Revision	Date	Author(s)	Description
0	01/20/2020	AF,IMR,AK,YW	Initial Release

Contents

1 Introduction

The purpose of this document is to explain the requirements that Galois has given us in our project. In addition, we will communicate the background and research associated with several of the requirements. In specific, we will provide information on the controller, firmware, verification, and robot design.

2 Objective

Tools and practices for designing and verifying high assurance software systems are not fully standardized. In order to explore the processes involved in this type of development, a test case using a control system for a two-wheeled inverted pendulum (TWIP) will be explored and documented.

3 Overview & Guidelines

The requirements have been broken into things that we must complete, should complete, and "may" complete. It should be noted that these requirements were set by Galois and may seem to be less abstract than normal.

MUST

- Extend the existing code for the inverted pendulum robot to include a Rust control library.
- Modify Lustre/Kind 2 Rust code generator to generate embedded-friendly Rust, which can be directly imported in your library.
- Develop and identify a model of a nominal self-balancing robot in Octave or MATLAB.
- Isolate the Software Under Test as a severable block to analyze only the linear/non-linear controller for specific properties.
- Develop and identify a dynamics model of the robot in Octave or MATLAB.
- Develop and verify a controller with a wider range of stable input conditions and compare its performance with the PID controller through both simulation and in the real system.

- Build another robot with the upgraded parts provided by Galois.

SHOULD

- Develop and verify a non-linear controller and compare its performance with the PID controller through both simulation and in the real system.

MAY

- Attempt to leverage Rust Quickcheck to verify the properties developed in Kind 2 in the source files after the control design is implemented.
- Develop a suite of verified nonlinear controllers and compare their performance with linear variants.
- Translate Rust code into a LEAN theorem definitions for verification.

4 Architecture Design

4.1 TWIP

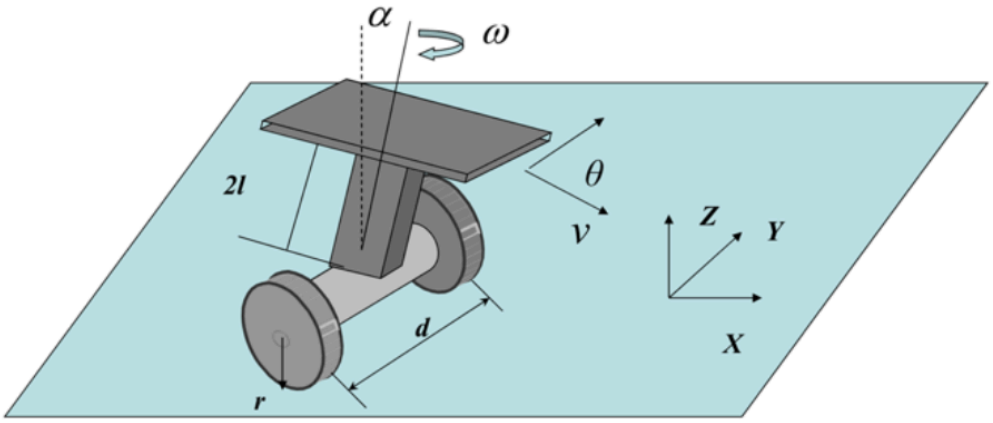

Figure 1: Simplified image of a TWIP[**?**]

The model used for the design will be dependent upon the method of control that we wish to implement. For a linear control method the model will be centered about the equilibrium point corresponding to the two-wheeled inverted pendulum (TWIP) being upright. This yields the following state space representation [**?**] :

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ t_1 & 0 & 0 & 0 & 0 \\ t_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ t_3 & 0 \\ t_4 & 0 \\ 0 & 0 \\ 0 & t_5 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$t_1 = \frac{(m + 2m_w + M)mlg}{(m + 2m_w + M)I + (m + M + 2m_w)l^2}$$

$$t_2 = -\frac{m^2 g l^2}{(M + 2m_w)I + (M + 2m_w)ml^2}$$

$$t_3 = -\frac{ml}{(M + 2m_w + m)I + (M + 2m_w)ml^2}$$

$$t_4 = \frac{1}{r}\frac{I + ml^2}{(M + 2m_w + m)I + (M + 2m_w)ml^2}$$

$$t_5 = \frac{2d}{r(2I_w + I_p)}$$

$$u_1 = \tau_1 + \tau_2$$

$$u_2 = \tau_1 - \tau_2$$

Figure 2: Linearized state space representation of TWIP Dynamics [**?**].

Achieving a global model of the systems dynamics will be achieved as we make further progress in our project.

4.2 TWIP Controller

As it stands, the dynamics of the two-wheeled inverted pendulum (TWIP) are represented by a linearized model. A linearized model can be achieved by using a Taylor series expansion around an equilibrium point [**?**]. Linearizing the system allows a designer to use a vast collection of well defined control techniques. However, when the system varies significantly from the equilibrium points, the controller is no longer effective. The need for a controller that can effectively stabilize the nonlinear dynamics of the system is evident. Prominent methods used to control nonlinear systems will be discussed below:

4.2.1 Feedback linearization[?]

This technique is essentially a way to cancel out the nonlinear dynamics of the system. Rather than doing so by approximation, this is done through a state transformation of the system. Once the feedback linearization is achieved all of the methods of linear controls can be used. This sounds advantageous, however it requires an exact knowledge of the systems dynamics. Before implementing this method we would have to confirm that the dynamics model of the TWIP is accurate and precise. The basic block diagram for the feedback linearization loop is shown below:

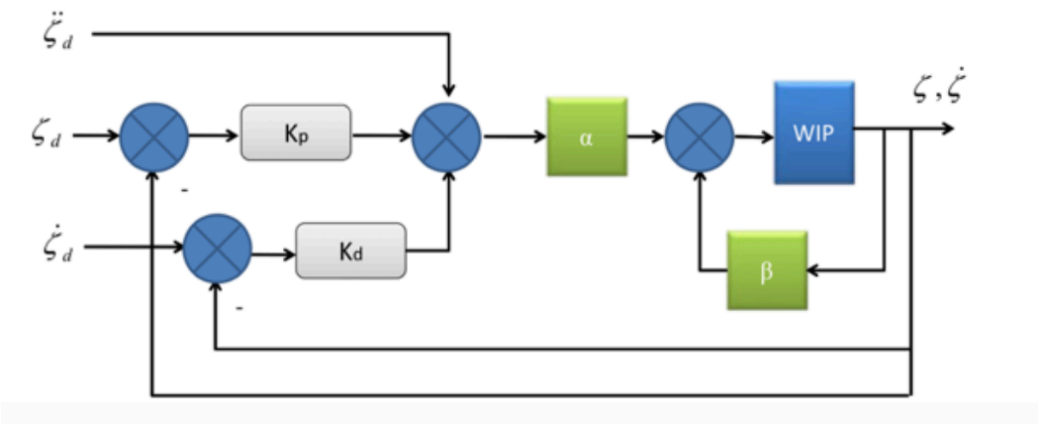

Figure 3: Feedback linearized WIP with a Proportional plus derivative controller [?].

4.2.2 Sliding mode control [?]

If an exact model of the dynamics associated with the TWIP cannot be obtained another suitable method of control must be implemented. One method is referred to as sliding mode control. This is a method where the control law is changed during process depending on the state of the system. The advantage of using this control method includes increased robustness (insensitivity to modeling errors) and operating range.

4.2.3 Model Predictive Control [?]

Model predictive control is an online optimization method that allows the control signal to be continuously optimized based off of an established model. Implementing a nonlinear model may not be feasible if the hardware of the system does not permit fast enough computations. However, because of the continuous optimization it is possible to use a linearized model and achieve better specifications than an offline controller. In addition, constraints can be imposed on the control signal to keep the physical limitations of the actuators in consideration. This method relies on a good model of the system. A good model of the system can be achieved using machine learning techniques.

4.3 Rust

Rust is an evolving language thats still in its early stages, its main aspects are performance, reliability, and productivity, with a strong emphasis on

safety. Rust was designed to compete with programming languages like C and C++, while adding features that are meant to make Rust safer than the other two languages. Ownership and Lifetimes are two of the most important features exclusive to Rust. These are used to eliminate the need for garbage collection, and reduce the risk of a data race (when two pointers access the same memory location at once), respectively. Rust has a strict compiler that attempts to enforce ownership and lifetime rules before the code can go into runtime. The goal is to make it significantly easier to catch inconspicuous errors that often plague other lower level languages before the code ever leaves the compiler.

Another benefit of Rust is that the compiler is very helpful in describing possible issues and finding alternative solutions. Additionally, the standard library provides many features that are borrowed from functional programming, some of which are shown in the figure below. Unfortunately, for our implementation we will be using Embedded Rust with our microprocessor that removes many of the features provided by the standard library:

feature	no_std	std
heap (dynamic memory)	*	✓
collections (Vec, HashMap, etc)	**	✓
stack overflow protection	✘	✓
runs init code before main	✘	✓
libstd available	✘	✓
libcore available	✓	✓
writing firmware, kernel, or bootloader code	✓	✘

Figure 4: std vs no-std feature overview [**?**].

Our required model verification tool Kind 2 has the option to compile Rust from Lustre, allowing for traceability between our model and our coded implementation. This will allow us to generate Rust code that has properties that have been verified as invariant, and test it on our project without the risk for human error between the model we have created and code we have written for the robot.

Currently Kind 2 is not configured to generate embedded-friendly code, and one of our requirements will be to make Lustre to Rust embedded-

friendly compilation an option. This will require identifying the components of the Lustre to Rust compiler that introduces code not suitable for an embedded environment, and try to find alternatives in the core library or identify necessary limits on the Lustre code that will allow for embedded code generation.

Additional stretch goals have also been defined for our Rust implementation that user other features and external libraries that are available to Rust. Cargo is a package manager for Rust that allows for easy distribution and documentation of Rust libraries, with a defined library layout. One of our stretch goals is to implement a suite of Rust libraries, and following the Cargo documentation process will make it much easier to other to implement our code suite in other projects. Other stretch goals include using Rust's Quickcheck feature to perform verification.

4.4 Verification

In order to ensure that the controller we have designed is what has been implemented into the Rust language, and that it will perform as we expect, we will need to use formal verification. We will use the method of model checking to achieve this goal. Model checking is a good candidate for this as model checking software is a mature area, and the softwares performance has been optimized.[**?**]

Model checking involves verifying properties of the model that are invariant. For our system, that would be something like for any input, our system remains stable. To do this in a systematic, traceable way, we will use the Kind 2 model checker, an SMT-based model checker. Kind 2 takes a Lustre (a programming language) file as an input and checks to see whether specified properties of the code are invariant for all inputs. If they are not invariant for all inputs, it reports the sequence(s) of inputs that result in the variance.

To get to point where we are checking that our controller is stable for all inputs, we will need to develop a model of both the robots hardware, as well as the controller, in Octave/MATLAB and then translate those models into Lustre code. Our general process will look something like this:

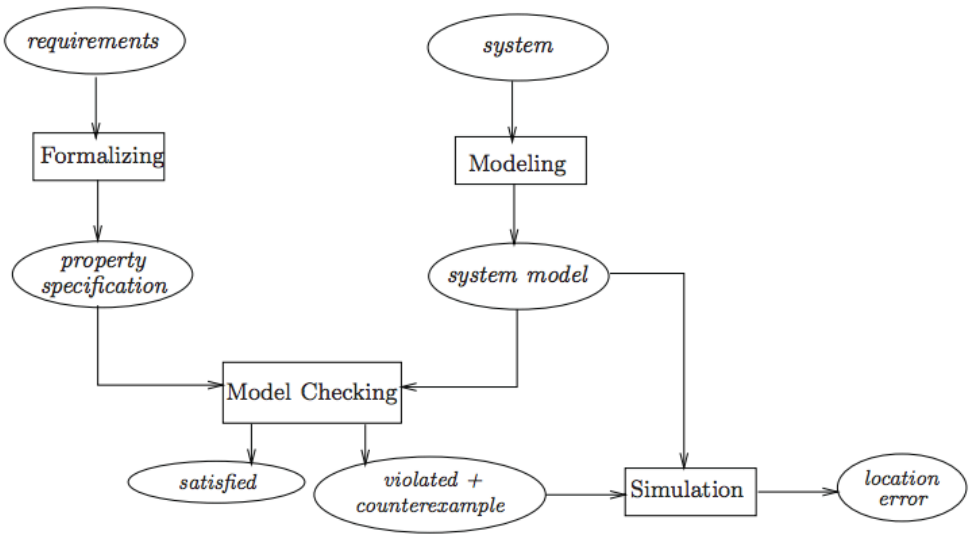

Figure 5: Model Checking Verification Process [**?**].

As mentioned previously, we will use the Kind 2 model checker to generate Rust code that has been verified. By leveraging the model checker to produce Rust, we will be able to provide traceable, verifiable, stable, and safe code that should satisfy the "high-assurance" portion of this project's title.

4.5 Robot

There are many types of existing self-balancing robot structures. The structure shown in the figure below consists of three layers: the upper battery layer, the middle main control layer, and the bottom motor drive layer. The battery layer is used to place the 12V battery that powers the entire system. The main control layer consists of the minimum system of the main control chip and the sensor module. The motor drive layer receives the controller signals and controls the motor. Each layer is a functional module, and the circuit boards are supported and fixed between the layers. The motor shell is fixed to the motor drive circuit board. The motor shaft is connected to two tires.

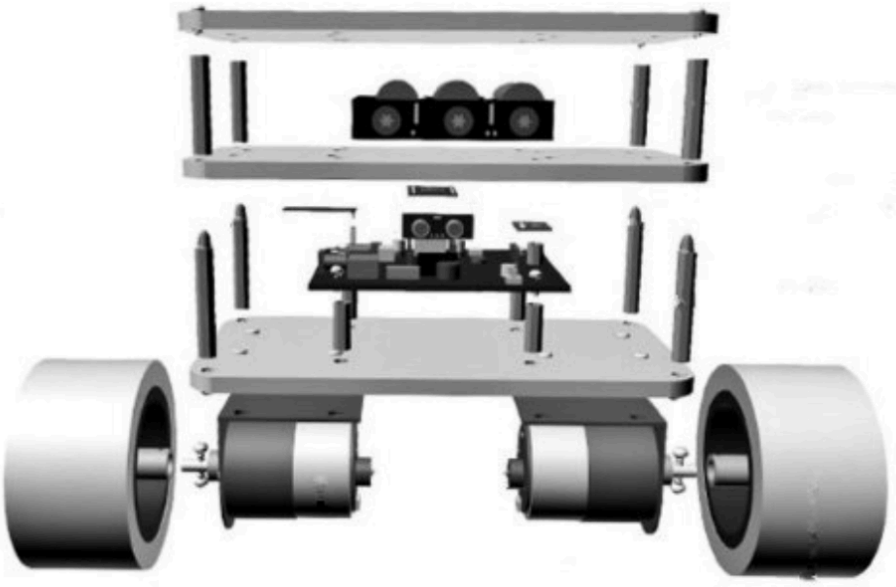

Figure 6: Mechanical Structure of TWIP Robot.

The core problem of a two-wheeled balancing robot is the problem of motion balance control. Two-wheeled robots must always maintain an upright posture. While maintaining a balanced posture, they also need to complete various tasks, such as traveling, rotating, climbing, and crossing obstacles. The completion of these tasks will inevitably result in the elevation angle Θ. Therefore, in order to balance the cart, the angle of elevation Θ must be reduced or even eliminated. In our design, we need to control Θ within a certain range. An effective method to eliminate the Θ angle is to use a feedback loop to engage the SAMD21 control motor, causing the chassis to move, ensuring that it is on the same vertical line as the upper and lower parts of the vehicle body.

Building on the last team's work, we will build a robot using parts provided by Galois. Since the last team's robot works fairly well already, we will be refining the design that they implemented, in order to provide a more stable version, hopefully making the controls and modeling portions of the project easier to achieve.

5 Approvals

The undersigned acknowledge they have reviewed the Team 10 Product Design Specification document and agree with the approach it presents. Any changes to this Requirements Definition will be coordinated with and approved by the undersigned or their designated representatives.

Signature: _____
Print: _____
Title: _____
Date: _____

Signature: _____
Print: _____
Title: _____
Date: _____

Signature: _____
Print: _____
Title: _____
Date: _____