A Flight Software Framework and State Machine DSL for Advanced Collegiate Rockets

by

Stefan deBruyn

THESIS

In Partial Fulfillment of the Requirements for

Graduation with Honors for the Degree of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2022

I wouldn't trade it for the world. In a way, I had the world.

Keith Todd

Acknowledgments

I'd like to thank my research advisor James Bornholt, whose guidance was indispensable to this project. I'd also like to thank Sarah Masimore, Sahil Ashar, and Butch Bartels, who were my introduction to flight software. Lastly I'd like to thank all members of the Texas Rocket Engineering Lab past and present for the wild ride that has been the Halcyon project.

A Flight Software Framework and State Machine DSL for Advanced Collegiate Rockets

Stefan deBruyn
The University of Texas at Austin, 2022

Supervisor: James Bornholt, PhD

Rockets are complex machines that require experts from many domains to design, build, and fly. Developing the embedded flight software that controls a rocket entails challenges like reacting to frequent vehicle design changes and implementing control software for non-software domains like propulsion and fluid systems. This thesis presents a framework to streamline flight software development for advanced collegiate rocketry projects through highly configurable APIs. The framework features a state machine API programmable with a domain-specific language (DSL) that allows a non-software domain expert to implement their own vehicle control logic. We evaluate the state machine DSL in user studies of students from various engineering backgrounds, all of whom were able to quickly learn the language. As an experiment, the DSL was integrated into the flight software of a research rocket and successfully guided the rocket through a hardware-in-the-loop flight simulation.

Table of Contents

Acknowledgments **iii**

Abstract **iv**

List of Figures **vii**

Chapter 1. Introduction **1**

1.1 Background 1
1.2 Problem Statement 3
1.3 Thesis Statement 3
1.4 Thesis Structure 3

Chapter 2. Design **4**

2.1 Design Goals 4
2.2 Example Framework Application 6
2.3 Framework Architecture 8
2.3.1 Core Library 9
2.3.2 Platform Abstraction Layer 9
2.3.3 Platform Support Layer 10
2.3.4 Config Library 10
2.4 State Vector 11
2.5 Task and Executor 12
2.6 State Machine 13
2.6.1 State Machine Logic Flow 13
2.6.2 State Machine DSL 14
2.6.2.1 State Vector Section 14
2.6.2.2 Local Variables Section 15
2.6.2.3 State Sections and Language Syntax 16

2.6.2.4 Type System 17
2.6.2.5 Built-In Functions 18
2.6.3 Use Cases 19
2.6.4 State Machine Testing 19
2.7 Analog and Digital I/O Tasks 21

Chapter 3. Implementation **23**
3.1 State Vector 23
3.2 Task and Executor 24
3.3 State Machine 25
3.4 Config Library 26
3.4.1 State Vector Compiler 28
3.4.2 State Machine Compiler 28
3.4.3 State Script Compiler 29
3.4.4 State Machine Runtime Reconfiguration 29
3.5 Framework CLI Tool 31

Chapter 4. Evaluation **33**
4.1 State Machine DSL User Studies 33
4.2 State Machine DSL Hardware-in-the-Loop Test 34
4.3 Framework Design Goals Revisited 36

Chapter 5. Future Work **37**
5.1 Improved State Machine DSL Toolchain 37
5.2 Network I/O and Entry Point Config Files 37
5.3 State Machine DLL for LabVIEW 38

Chapter 6. Conclusion **39**

References **40**

List of Figures

2.1 Example Framework flight software application 7
2.2 Framework layers 8
2.3 Framework dependency graph 9
2.4 Example state vector config file 12
2.5 Example state machine DSL state vector section 15
2.6 Example state machine DSL local section 15
2.7 Example state machine DSL state section 17
2.8 Example state script 20
2.9 Example I/O task configs 22

3.1 State machine step logic flow 26
3.2 Config file compilation process 27
3.3 Example CLI state machine compiler output 31
3.4 Example CLI state script output 32

4.1 HIL test flight profile with state transitions 35

Chapter 1

Introduction

This thesis presents a C++ framework (the "Framework") and state machine DSL that attempt to solve challenges of flight software development for advanced collegiate rockets. This chapter introduces these challenges and describes the structure of the rest of the thesis.

1.1 Background

Rockets are highly multidisciplinary engineering projects. Systems on a rocket may include propulsion, propellant feed, guidance, navigation, and control (GNC), power, radio telemetry, flight termination, payload deployment, and recovery. An onboard flight computer performs coordinated control of these systems through sensors and effectors, which the flight computer communicates with over some kind of data bus. The flight software runs an observe-orient-decide-act (OODA) loop that processes sensor readings into effector commands. This loop typically runs from a time before the rocket launches (e.g., while being fueled on the launchpad) through the end of the mission.

Flight software is a unique component of a rocket because it controls almost every other vehicle system. Control logic commonly features distinct states that vary with inputs like time and sensor readings, making finite-state machines an

effective model for many flight software control tasks. The design of a particular state machine is informed by expert knowledge of the system under control like its sensors, effectors, dynamics, and tolerances. This need for expertise in multiple domains makes development of flight software challenging, especially for amateur rocketry projects such as those found at the collegiate level.

Recent years have seen notable advancements in the state of collegiate rocketry. In 2019, a team from the University of Southern California became the first students in history to launch a rocket past the Kármán line, the internationally-recognized edge of space at 100 km (62 mi) above sea level [1]. The year prior, the Base 11 Space Challenge—a competition awarding $1 million to the first university team to launch a liquid-propellant rocket to space—spawned advanced rocketry projects at universities across the United States and Canada [2]. One such university is the University of Texas at Austin, home to the Texas Rocket Engineering Lab.

The Texas Rocket Engineering Lab (TREL) is a student-run research lab that trains aerospace professionals through applied liquid rocket engineering. As of 2022, TREL is working on the "Halcyon" rocket, one of the most advanced rockets ever built by students. Halcyon uses a number of technologies that make it unique among collegiate rockets, including a distributed real-time Linux flight computer that controls all vehicle systems. The work presented in this thesis is separate from Halcyon, though Halcyon was a backdrop for developing many of the ideas herein.

The increasing complexity of collegiate rockets is driven by pursuit of higher altitudes and industry-standard technologies like liquid propulsion. This complexity propagates into the flight software and integration process that marries the soft-

ware to the rest of the rocket. The work presented in this thesis is a step towards streamlining flight software development for advanced collegiate rockets, especially "spaceshots" like TREL Halcyon with 100+ km apogees and a wide range of computer-controlled systems.

1.2 Problem Statement

Rockets have complex computer-controlled systems. These systems fall mostly in non-software domains, especially propulsion and fluid systems, which are usually the largest systems on a rocket by sensor and effector count. Rockets are also highly-optimized systems and often change constantly in the development phase. This has two implications for the flight software that controls a rocket: much of the control logic requires expertise in non-software domains, and the software will change constantly in reaction to design changes during the rocket's development.

1.3 Thesis Statement

A domain-specific language for state machines makes flight software easier to develop for advanced collegiate rockets.

1.4 Thesis Structure

The thesis proceeds with Chapter 2, which covers the design of the Framework and state machine DSL. Chapter 3 explores their implementation. Chapter 4 describes how the Framework and state machine DSL were evaluated as solutions to the stated problem. Chapter 5 discusses opportunities for future work, and Chapter 6 concludes the thesis with a summary of results.

Chapter 2

Design

This chapter covers the design of the Framework and state machine DSL. Design goals are discussed first, followed by an example Framework application that sets up deeper discussion of the main Framework APIs.

2.1 Design Goals

The design of the Framework was driven by several design goals. These goals reflect the thesis problem statement as well as other characteristics desirable in a software framework like the one presented. The design goals are as follows:

1. Provide portable interfaces for common flight software needs
2. Allow developers to iterate the flight software quickly
3. Optimize for safety by minimizing failure modes and surfacing all errors
4. Enable non-software domain experts to program their own control logic

Design goal 1 addresses a desire for code reuse across different flight software projects. Flight software for different rockets often share features like task scheduling and data sharing between tasks. Packaging these common features into a set of portable interfaces allows faster bringup of new flight software and increased mission assurance due to reuse of heritage flight software.

Design goal 2 acknowledges that developing flight software for a new rocket

often requires hitting a (metaphorical) moving target. A rocket usually undergoes many design changes between its initial whiteboard concept and maiden flight as system requirements are revised, design flaws are discovered, etc. Many design changes will directly impact the flight software, which calls for flight software that can be rapidly reconfigured as the design of a rocket matures.

Design goal 3 recognizes that flying rockets is a tricky business, where the consequences for failure are loss of expensive hardware in the best case and loss of life in the worst case. Flight software should be error-proof where possible and fault-tolerant where an error may occur. Software that surfaces all errors is easier to debug and can more effectively identify and handle faults at runtime.

Design goal 4 reflects that rocket engineering is multidisciplinary, and this makes integration challenging. For example: rocket propulsion is outside the purview of a software engineer, and software development is outside the purview of a propulsion engineer, so the two engineers must somehow cooperate to implement the software that controls a rocket engine. The propulsion engineer could simply provide pseudocode of the engine control logic that the software engineer translates into source code, but this would be slow and error-prone. A better solution would allow the propulsion engineer (or other domain expert) to express the control logic for their system in an intermediate form that requires no programming expertise and can be automatically translated to flight software.

2.2 Example Framework Application

Before discussing the Framework design in detail, we introduce the main concepts through a simple flight software application that could be built on the Framework. The application controls a rocket with a navigation system, engine, and recovery parachute. The application is decomposed into five tasks that execute concurrently:

1. Sensor task – reads sensor data into application memory
2. Navigation task – estimates the rocket's altitude based on sensor data
3. Engine task – fires the engine to begin the rocket's ascent
4. Parachute task – deploys a parachute to slow the rocket's descent
5. Mission task – directs the flight via the above tasks

The sensor, engine, and parachute tasks perform I/O with external sensors and effectors. The navigation and mission tasks do not directly interface with external devices and only deal with data in memory. All data used by tasks is stored in a shared memory region called the state vector. The state vector contains four variables:

1. Current sensor reading by the sensor task
2. Current altitude estimated by the navigation task
3. Engine task state – on or off (initially on)
4. Parachute task state – deployed or undeployed (initially undeployed)

The mission task directs the flight by changing the engine and parachute task states based on the current altitude. The mission task is a state machine with three states: powered flight, coast, and recovery. Powered flight transitions to coast when

altitude exceeds some threshold, and this transition turns off the engine. Coast transitions to recovery when altitude falls below some threshold, and this transition deploys the parachute. This state machine represents the rocket's "mission profile"—the high-level mission plan that sets the trajectory, in-flight event schedule, etc.

This example application, illustrated in Figure 2.1, demonstrates the main Framework concepts: tasks, a global state vector through which tasks communicate, and state machines that implement certain tasks. State machines may be effective at implementing some tasks and not others. Task implementations are decoupled from one another by the state vector, which serves as the application software bus. A special "mission" task controls the whole rocket from a high level by directing other tasks via the state vector. This application architecture is based on a previous work that successfully applied the same architecture on the TREL Halcyon rocket [3].

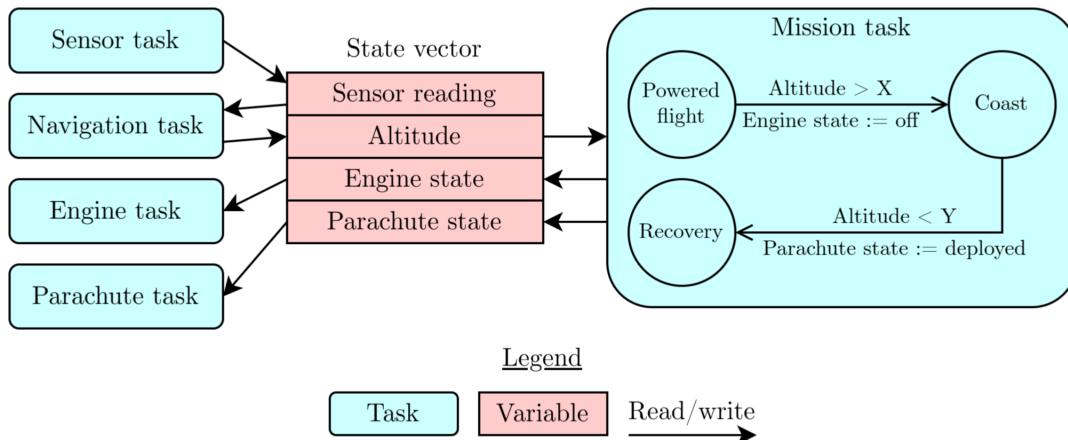

Figure 2.1: Example Framework flight software application

2.3 Framework Architecture

The Framework is written in C++11 and has three layers: the Core Library, the Platform Abstraction Layer (PAL), and the Platform Support Layer (PSL). A Framework user's flight software application rests on the Core Library and possibly PAL. The PAL acts as an abstraction layer between the Core Library and operating system or machine that the Framework is deployed to. The PSL implements abstract interfaces in the PAL for the target platform. Below this are platform-specific APIs like system headers used to implement the PSL. Figure 2.2 illustrates the Framework layers.

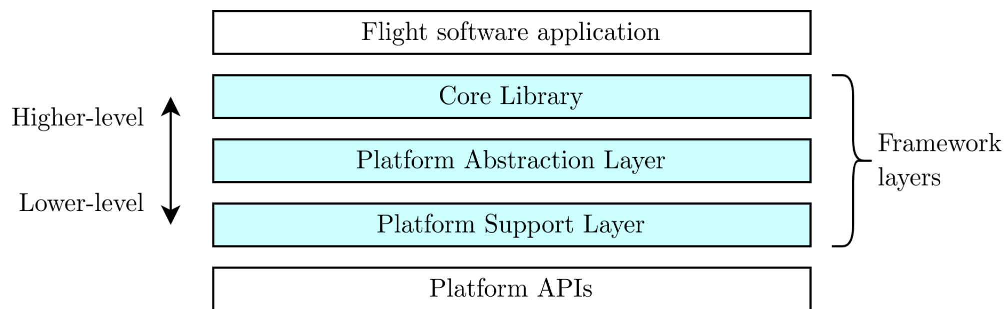

Figure 2.2: Framework layers

Orthogonal to this stack is the Config Library, a utility library for creating Core Library objects from config files. The Config Library depends on the Core Library but not vice versa, so linking the Config Library to a flight software application is optional. The Config Library also depends on the C++ Standard Library. Figure 2.3 illustrates the Framework dependency graph.

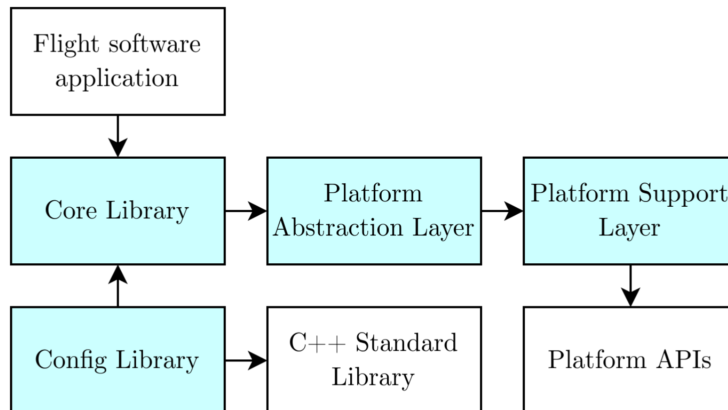

Figure 2.3: Framework dependency graph

2.3.1 Core Library

The Core Library provides general-purpose interfaces for common flight software needs:

- `Task` – abstract interface for a unit of application business logic
- `Executor` – abstract interface for a `Task` scheduling algorithm
- `StateVector` – memory region used to share data between `Task`s
- `StateMachine` – deterministic finite-state machine with actions and transitions based on `StateVector` data

The Core Library depends only on the PAL and does not allocate memory, which makes it highly portable.

2.3.2 Platform Abstraction Layer

The PAL abstracts the platform on which the flight software application runs. The PAL provides the following interfaces:

- `Clock` – system clock interface for timekeeping
- `Console` – console interface for printing to standard output
- `Thread` – thread interface for creating and joining threads
- `Lock` – lock interface for implementing mutual exclusion
- `Socket` – socket interface for network communications
- `AnalogIO` – interface for accessing analog I/O pin hardware
- `DigitalIO` – interface for accessing digital I/O pin hardware

The PAL contains little executable code and mostly function prototypes. These prototypes are implemented by a PSL for the platform targeted by a flight software application.

2.3.3 Platform Support Layer

The PSL implements the abstract interfaces of the PAL. The Framework provides PSLs for Linux and NI sbRIO.[1] A partial PSL was developed for Arduino to test deployment on a bare-metal platform. The Framework can be ported to a new platform by implementing a PSL for it.

2.3.4 Config Library

The Config Library provides configuration languages for certain Core Library objects. Manually creating these objects (which often requires defining large arrays or trees) can be tedious and error-prone, so the Config Library allows automatically creating them from config files. The Config Library can compile a config file to an

[1] sbRIO is a family of real-time embedded controllers made by National Instruments.

object in memory or C++ source code on disk, also known as autocode. Autocoding is intended for pre-compile time configuration of applications that cannot call the Config Library at runtime, e.g., due to memory constraints.

2.4 State Vector

The Core Library `StateVector` object represents a set of variables globally accessible to flight software application `Task`s. A variable in the state vector is called an "element". 11 element types are supported:

- Signed and unsigned integer types `I8`, `I16`, `I32`, `I64`, `U8`, `U16`, `U32`, and `U64`
- Single- and double-precision floating-point types `F32` and `F64`
- Boolean type `bool`

The state vector is divided into contiguous "regions" of one or more elements. Regions are a mechanism for accessing the backing memory of multiple elements at once. The state vector API allows accessing data on the scale of individual elements, regions, or the entire state vector.

`Task`s communicate by reading and writing the state vector. Ideally, the state of all `Task`s in a flight software application is stored in the state vector, so that the state vector represents a snapshot of the complete application state at one point in time. This makes large-scale state operations easier, like logging the state to a file or restoring the state following a crash.

An element declaration in a `StateVector` config file is a type followed by a name. Element declarations are grouped into one or more region sections beginning with a bracketed section name like `[foo]`. A special section `[options]` may specify

configuration options like `lock`, which makes the state vector thread-safe via a lock. Figure 2.4 shows an example state vector config file.

```
[options]
lock

[SensorsRegion]
F64 tankPressure
F64 verticalAcceleration

[EffectorsRegion]
bool valveOpen
I8   finAngle
```

Figure 2.4: Example state vector config file

2.5 Task and Executor

The Core Library `Task` and `Executor` abstractions are the main APIs used to implement the business logic of a flight software application. A `Task` wraps a short-running piece of code called at high frequency in order to do work. A single call to a task is a "step". The inputs and outputs of a task are reflected in `StateVector` elements that the task reads and writes each step. Example tasks include sensor data acquisition, effector control algorithms, network I/O, and system health monitoring.

A `Task` is in one of three modes: enabled, safed, or disabled. The mode is controlled by a `U8` state vector element. The behavior of a task when enabled or safed is implementation-defined, and a disabled task does not run. Safe mode is intended to implement fail-safes or other safing logic for dangerous tasks. A task mode is set by code external to the task and may be changed throughout runtime.

An `Executor` executes a set of `Task`s according to some scheduling algorithm.

The task set is fixed once an executor is created, but changing task modes can create an effect similar to modifying the task set, e.g., temporarily descheduling a task by disabling it. In the nominal case, an executor never stops executing tasks and holds the flight software application in an infinite loop for all of runtime.

2.6 State Machine

The Core Library `StateMachine` object is a configurable deterministic finite-state machine that interfaces with `StateVector`. The Framework user defines a state machine using a high-level DSL that the Config Library can compile to an in-memory `StateMachine` object or C++ source code. A `StateMachine` acts like a virtual machine for the user's state machine program. A `StateVector` acts like the address space that the state machine program may access, allowing it to share data with other objects like `Task`s. A `StateMachine` does not execute by itself and will usually be wrapped in a `Task` that executes it (read: a `StateMachine` is one possible tool for implementing a `Task`).

2.6.1 State Machine Logic Flow

From the Framework user's perspective, a state machine is a short-running method called in a loop to do work. Each call executes one "step" of the current state. Each state has three logic "blocks":

1. Entry block – executes on the first step of the state
2. Step block – executes every step of the state
3. Exit block – executes just prior to transitioning from the state

A block contains statements that execute sequentially. A statement may be a branching conditional, variable assignment, or state transition. A transition immediately stops execution of the current block, executes the current state's exit block, and transitions to the new state. Transitioning in an exit block is illegal.

2.6.2 State Machine DSL

The `StateMachine` configuration language is a high-level, imperative DSL designed to be instantly approachable by non-programmers. A state machine is programmed in one file. The file is divided into sections that begin with a bracketed name like `[foo]`. There are three section types: the state vector section, the local variables section, and a state section.

2.6.2.1 State Vector Section

The state vector section begins with `[state_vector]`. This section lists elements from the flight software application's `StateVector` config that the state machine may access. This includes the element's type, name, and optional annotations. An annotation `@read_only` makes an element read-only to the state machine. An annotation `@alias <alias name>` gives an element an optional alias. The state vector section requires two special elements:

1. A `U64` element with alias `G` that stores a global monotonic time value updated by external code. The unit of time and timestep size are application-defined.
2. A `U32` element with alias `S` that the state machine automatically writes the ID of the current state to. This has no functional impact on the state machine but gives external code insight into the current state.

These special elements are implicitly read-only. Figure 2.5 shows an example state vector section.

```
[state_vector]
U64  time         @alias G
U32  state        @alias S
F64  tankPressure @read_only
bool valveOpen
```

Figure 2.5: Example state machine DSL state vector section

The initial values of state vector elements are indeterminate from the perspective of the state machine programmer. The elements are updated concurrently to the state machine by external code like a `Task` or even another `StateMachine`.

2.6.2.2 Local Variables Section

The local variables section begins with `[local]`. This optional section contains definitions of private variables that exist locally within the state machine. Local variables must be explicitly assigned initial values in this section. Initial value expressions are evaluated at state machine compile time. Figure 2.6 shows an example local variables section.

```
[local]
F64 pi                = 3.14159 @read_only
U16 P_WarnThreshold  = 800
U16 P_AbortThreshold = P_WarnThreshold + 25
U64 counter           = 0
```

Figure 2.6: Example state machine DSL local section

2.6.2.3 State Sections and Language Syntax

A section other than `[state_vector]` and `[local]` defines a state with the name in brackets. A state section is subdivided into its entry, step, and exit blocks by labels `.entry`, `.step`, and `.exit`. Each label is followed by code for the corresponding block. A block that is omitted or contains no code is a no-op. Block code may use the following syntaxes:

- `x = y` evaluates expression `y` and assigns it to variable `x`
  - `x` may be a non-read-only state vector element or local variable
- `if x: y` executes statement `y` if expression `x` is true
  - The `if` keyword is optional
- `if x { y }` executes statement or statements `y` if expression `x` is true
  - The `if` keyword is optional
  - Statements inside the brackets are separated by newlines
  - May contain nested conditionals
- An `if` statement may optionally be followed by an `else:` or `else {}` branch that executes if the `if` branch is not taken
- `-> x` triggers a state transition to the state named `x`
- `T` is a built-in, read-only `U64` variable that stores time elapsed in the current state as computed using the global time element `G`
  - If `G=a` on the first step in the state and `G=b` on the current step, then `T=b-a`
  - `T=0` on the first step in a state

The language supports the same arithmetic, logical, and relational operators as most

programming languages. Parentheses may be used to override precedence. Figure 2.7 shows an example state section.

```
[TankVentControl]
.entry
    counter = 0
.step
    if tankPressure > 775 {
        valveOpen = true
        counter = counter + 1
    }
    else: valveOpen = false
    not (500 < tankPressure < P_AbortThreshold): -> Abort
    counter > 1000 and T > 4000: -> Idle
.exit
    valveOpen = true
```

Figure 2.7: Example state machine DSL state section

2.6.2.4 Type System

The state machine DSL is statically typed. Type conversion is performed implicitly according to the following rules:

1. All terms in an expression are intermediately converted to `F64` when evaluating the expression
2. Converting an `F64` to an integral type saturates the converted value at the integral type's numeric limits
3. NaN converted to anything is 0 or `false`

If an expression is used in a conditional, the resulting `F64` is converted to `bool`. If an expression is the right-hand side of an assignment statement, the resulting `F64` is converted to the type of the left-hand side variable before assignment.

The type conversion rules are designed to make the state machine DSL simpler and more intuitive to non-programmers by reducing the range of possible type errors. All values behave like real numbers, eliminating certain problems associated with integral types like signed-unsigned comparison and integer division. An integral that overflows or underflows saturates at the type's numeric limits rather than wrapping around, which may be safer for many applications.

A drawback of implicitly converting all types to `F64` is a loss of precision when converting from `I64` or `U64`. Because an `F64` has only 53 bits of precision, integers with magnitudes larger than 2^{53} may be approximated as `F64`. As 2^{53} is much larger than integers a flight software application typically needs to represent, this was deemed a non-issue.

2.6.2.5 Built-In Functions

The state machine DSL provides several built-in functions:

- `roll_avg(x, y)` – returns the rolling average of `x` over the last `y` steps
- `roll_median(x, y)` – rolling median
- `roll_min(x, y)` – rolling minimum
- `roll_max(x, y)` – rolling maximum
- `roll_range(x, y)` – rolling range

The value of `y` is evaluated at state machine compile time and fixes the rolling window size. `x` may be any expression. Function calls may be composed. These specific functions were chosen with sensor data processing in mind, for example, filtering noise by taking a rolling median of a sensor reading.

2.6.3 Use Cases

The `StateMachine` object is designed to implement state-based, timed control logic common in flight software. The following are potential use cases:

- Rocket engine ignition and shutdown sequencing
- Propellant feed or other fluid system control
- Parachute deployment sequencing
- Mission profile configuration
- Sensor calibration and taring

When developing flight software for a particular vehicle system, a flight software engineer could provide an expert on the system with a state machine DSL "stub" that defines only a state vector section containing the system's I/O channels. The system expert uses their domain expertise to express control logic for the system as a state machine. The flight software engineer then takes the system expert's state machine program and compiles it to flight software using the Config Library.

2.6.4 State Machine Testing

Once compiled, state machine DSL programs may be tested through regular software verification methods like unit testing and simulation. The Config Library also provides a language for writing simple state machine tests called "state scripts". A state script runs concurrently with a state machine, providing inputs in the form of variable assignments and making assertions about the system state at certain points in time. A state script also defines a stop condition that determines when the script ends. A state script "passes" if the stop condition is reached without any assertions

failing, otherwise the state script fails.

Syntax-wise, a state script input looks like a regular assignment statement. An assertion looks like an annotation `@assert` followed by a boolean expression. Inputs and assertions may reference any state vector elements or local variables used by the state machine under test. For testing purposes, an input is allowed to write an element or variable marked read-only in the state machine. Inputs and assertions must be placed in a conditional and state section that control when they execute. A special section `[all_states]` executes in every state. Another special section `[options]` specifies options like the timestep size `delta_t` (the amount by which `G` increases each step) and initial state `init_state`. A state script ends when a special annotation `@stop` is reached. Figure 2.8 shows an example state script.

```
[options]
delta_t    10
init_state TankVentControl

[all_states]
G == 0 {
    tankPressure = 600
    valveOpen    = false
}
valveOpen:  tankPressure = tankPressure - 1
!valveOpen: tankPressure = tankPressure + 1
G == 1000:  @stop

[TankVentControl]
true: @assert valveOpen == (tankPressure > 775)
```

Figure 2.8: Example state script

When a state script conditional executes on a particular state machine step, inputs under the conditional execute prior to the step and assertions execute after

the step. In this way, conditionals and inputs act like preconditions to a step, and assertions act like postconditions. A complete state script then acts like a test case that verifies state machine behavior under a sequence of conditions.

State scripts are a tool that can provide rapid feedback on the correctness of a state machine in the same way a programmer might “sanity check” a function they wrote by printing its output. State scripts are not intended for strong verification of flight software. The user runs a state script from a command-line interface (CLI) tool provided by the Framework.

2.7 Analog and Digital I/O Tasks

The Core Library provides two `Task` implementations `AnalogIOTask` and `DigitalIOTask` for reading and writing hardware I/O pins. These tasks use the `AnalogIO` and `DigitalIO` PAL interfaces, which are designed to interface with simple analog and digital sensors and effectors like pressure transducers, thermocouples, valves, and pyrotechnic igniters. `AnalogIOTask` deals with analog values (e.g., voltage or current) as `F64` state vector elements, and `DigitalIOTask` deals with digital values as `bool` state vector elements.

Onboard devices are likely to change during a rocket's development, so the Config Library supports creating analog and digital I/O tasks from config files. Each line of these config files defines an I/O channel as a direction, pin number, and state vector element name. An input channel reads the pin value into the element, and an output channel writes the element value to the pin. Figure 2.9 shows example I/O config files.

```
# Analog I/O task config
# Direction  Pin number  Element
# ---------  ----------  -------
  i          0           pressureSensor1Reading
  i          1           pressureSensor2Reading
```

```
# Digital I/O task config
# Direction  Pin number  Element
# ---------  ----------  -------
  o          0           valve1Position
  o          1           valve2Position
  i          2           limitSwitch1
```

Figure 2.9: Example I/O task configs. Text following a pound symbol is a comment.

Chapter 3

Implementation

This chapter covers the implementation of the main Framework APIs and state machine DSL.

3.1 State Vector

A `StateVector` object acts like a lookup table for elements and regions. The user provides the name of an element or region in the state vector config file and gets back a pointer to an element or region object. These objects act as handles to the memory underlying the state vector. An element handle is used to read and write the value of a single element, while a region handle is used to read and write a span of elements as raw, untyped memory. Regions are designed for dealing with state vector data in bulk, e.g., logging state to a file. Regions may also be used as implicitly-serialized network messages—this was done for the TREL Halcyon rocket, which uses a similar state vector API and shares data between flight computers by synchronizing regions over the network [3].

A state vector is laid out contiguously in memory so that regions or the entire state vector can be read and written more efficiently. The user can obtain a region handle spanning the entire state vector by looking up a name well-known to the API. Routing all state vector access through handle objects makes it simpler to implement

state vector thread safety. When locking is specified in a state vector config file, all handles share a spinlock. A spinlock was chosen because state vector critical sections are simple memory accesses and much shorter than a context switch.

3.2 Task and Executor

The user inherits from the abstract class `Task` to implement their application business logic. A task must implement abstract methods that define one-time initialization logic and business logic when the task is enabled. An optional method defines business logic when the task is safed and does nothing by default. A task is constructed with a `U8` state vector element that controls its mode. When the task steps, it reads the value of the mode element and calls the corresponding method, or no method if the task is disabled. A task may optionally be constructed without a mode element, in which case the task is always enabled.

The Core Library provides two `Executor` implementations for synchronous task execution on bare-metal and real-time platforms. These executors are implemented using the `Clock` and `Thread` PAL interfaces. The user may implement their own executor by inheriting from `Executor`. The `Executor` interface does not constrain the underlying scheduling algorithm in any way, only that it must schedule `Task`s. Flight software applications typically prefer synchronous, real-time scheduling in the interest of determinism, but implementing asynchronous, fair, etc. schedulers using the `Executor` interface is also possible.

3.3 State Machine

A `StateMachine` object represents the entry, step, and exit blocks of each state as code trees. Execution of a block starts at the code tree root and recurses down the tree, exploring branches and executing statements as conditionals are evaluated. Execution of the tree stops when all branches are explored or a state transition statement executes. Expressions such as conditionals or assignment statements are further decomposed into trees of operators, constants, variables, and function calls.

A function call is represented by an expression tree node that carries all the logic and data needed to evaluate the function. For built-in functions like `roll_avg()`, this includes memory to store rolling windows. A `StateMachine` is aware of rolling windows used in its blocks and will update these windows prior to each state machine step. A step goes as follows:

1. Update the state elapsed time element `T`
2. Update rolling windows
3. If this is the first step in the current state, execute the entry block
4. If the entry block did not trigger a state transition, execute the step block
5. If a state transition was triggered, execute the exit block

If a state transition is triggered within a step, the step ends with the exit block executing. The next step will be the first step in the new state. A state may transition to itself, which effectively restarts the state. Figure 3.1 illustrates the logic flow of a state machine step.

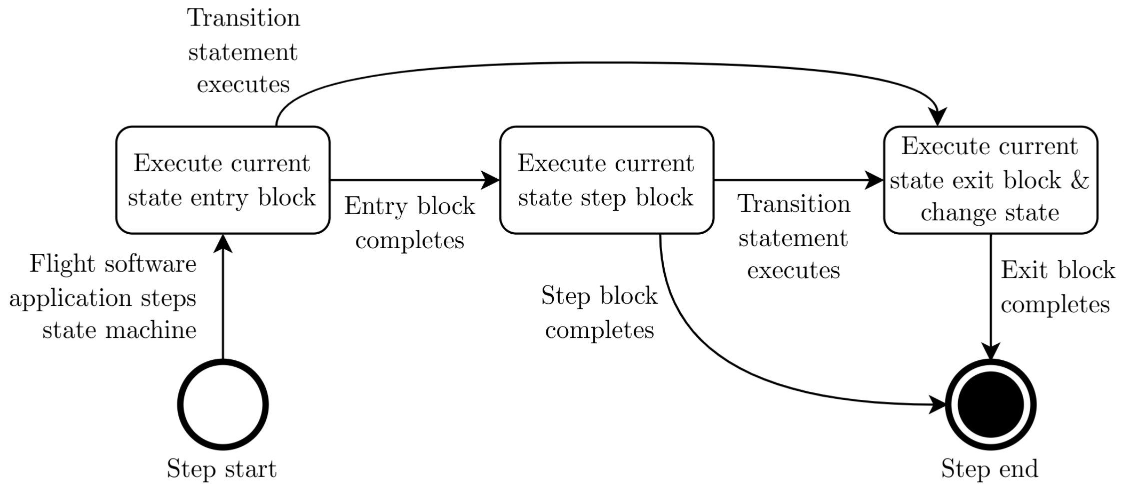

Figure 3.1: State machine step logic flow

3.4 Config Library

Config files have the same high-level compilation process: an input file is tokenized, the tokens are parsed into an abstract syntax tree (AST), and the AST is compiled to a Core Library object in-memory. Autocoding is an optional stage of compilation that generates a C++ source code file from the object produced by the compiler. The source code file defines a function that can be compiled (with a C++ compiler) and called to reconstruct the configured Core Library object. Figure 3.2 illustrates the config file compilation process.

The object produced by the compiler has an internal structure of abstract objects. For example, a `StateMachine` contains expression trees of abstract objects that represent different expression terms. These objects expose runtime type information and private data that allow the autocoder stage to "decompile" them into

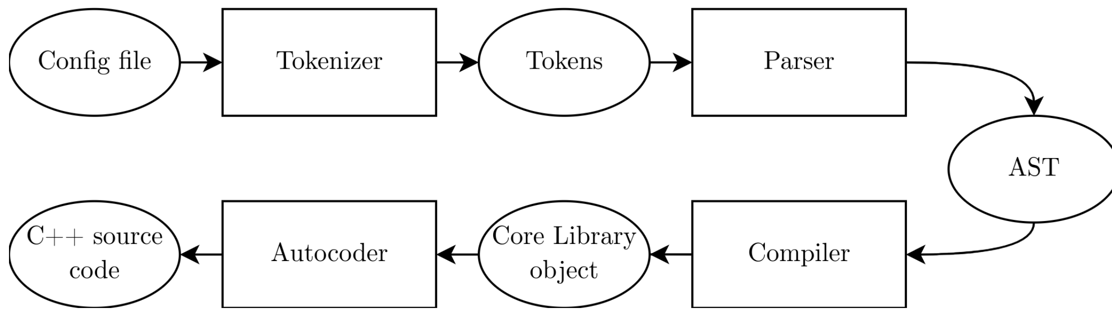

Figure 3.2: Config file compilation process

C++ code that instantiates them. Config Library compilers do not optimize configs, but if optimizations were implemented at the compiler stage, these optimizations would propagate into autocode without modifying the autocoder stage.

A Framework user has two options when using the Config Library to create a Core Library object:

1. Link the Config Library to the flight software application and invoke the config compiler from the application at runtime, skipping the autocoder and using the object produced by the compiler
2. Prior to compiling the flight software application, generate autocode and call the autocode from application source code to create the object

Option 1 allows greater runtime configurability but has a larger memory footprint and causes the flight software application to inherit the Config Library's dependency on the C++ Standard Library.[1] Option 2 is more suitable for platforms that lack a

[1]Using the C++ Standard Library in flight software may be undesirable in applications with strict safety requirements that disallow exceptions, third-party code, etc.

C++ Standard Library or have limited memory available, but the configuration is fixed at compile time.

3.4.1 State Vector Compiler

The state vector compiler processes a state vector config file into a `StateVector` object. The `StateVector` and supporting data like element handles and backing memory are allocated on the heap. Their lifetimes are managed by a `StateVectorAssembly` object produced by the compiler stage—this is the object returned to the user. The autocoder stage takes the `StateVector` and generates a C++ header and source file defining a function that produces an identical `StateVector`. An autocoded `StateVector` resides entirely in static memory.

State vector compiler errors are fatal and immediately stop compilation. Errors are accompanied by a human-readable message that includes the location of the error in the user's config file. Example errors include an unrecognized element type, duplicate element or region name, and an empty region.

3.4.2 State Machine Compiler

The state machine compiler works in much the same way as the state vector compiler. The compiler stage produces a `StateMachineAssembly` object that owns the compiled `StateMachine` and all supporting data such as code trees. This data is allocated on the heap, whereas an autocoded `StateMachine` resides entirely in static memory. Example errors checked by the state machine compiler include:

1. Transitioning to an undefined state or transitioning in an exit block
2. `roll_*()` function call with a non-integer window size

3. Reusing a state vector element name for a local variable
4. Writing a read-only variable

A compiled `StateMachine` is guaranteed to run successfully. The only errors possible at runtime are user logic errors.

State machine local variables are implemented using a `StateVector`. The state machine compiler builds an intermediate state vector config from the user's local variable definitions and then invokes the state vector compiler to compile it. The resulting `StateVectorAssembly` becomes part of the `StateMachineAssembly` produced by the state machine compiler.

3.4.3 State Script Compiler

The state script compiler reuses parts of the state machine compiler to compile the expressions and statements in a state script. As state scripts are not config files, the compiler has no autocoder stage. The compiler stage produces an object that can be used to run the state script and get a human-readable report of the results. The user runs state scripts from the Framework CLI tool and does not directly interact with the state script compiler. State scripts have several unique errors that the compiler checks for, such as unreachable statements after a `@stop` annotation.

3.4.4 State Machine Runtime Reconfiguration

Linking the Config Library to a flight software application makes it possible to reconfigure a state machine at runtime by hot-swapping it with a newly compiled state machine. The process might go as follows:

1. On application startup, a `StateMachine` is compiled from a DSL file. The state machine has one or more "safe" states that can be safely interrupted.
2. A user modifies the DSL file at runtime. While the state machine is in a safe state, a command is sent to the application to reconfigure the state machine. This command includes the initial state for the new state machine.
3. The application sets the specified initial state by writing the appropriate state vector element, deletes the old state machine, and then recompiles it from the modified DSL file.

Because state machines operate on an application-global state vector, hot-swapping state machines can effectively replace portions of application logic at runtime without affecting other parts of the application or bringing it offline. However, there are other challenges to be solved here, such as ensuring that state machine recompilation latency does not disrupt other critical application tasks. The kind of logic that can be reconfigured at runtime is also limited by the features of the state machine DSL.

This idea essentially turns a Framework application into a miniature operating system, where state machines are processes and the state vector is a mechanism for inter-process communication. The same effect could be achieved using the target platform's actual operating system and processes, but an all-in-one-process approach using Framework state machines may be preferable for certain applications and platforms, such as embedded operating systems with limited multiprocess support. The all-in-one-process approach may also be preferable for its simplicity.

Runtime reconfiguration is pertinent to flight software that requires experimental tuning and cannot conveniently be taken offline mid-experiment. For exam-

ple, a liquid-propellant rocket may have a complex fluid system that must be characterized through testing. Safely taking the software offline for modification would require detanking and then repressurizing the fluid system once the modifications are complete, a process that could take hours, especially for high-pressure or cryogenic systems. In contrast, modifying the control logic (e.g., tweaking an abort condition that was too conservative) in the middle of testing would allow faster iteration of the software by avoiding unnecessary system duty cycles.

3.5 Framework CLI Tool

The Framework provides a CLI tool for validating config files, generating autocode from config files, and running state scripts. This tool is part of the workflow for developing in the state machine DSL. The tool may also be integrated into DevOps automation, e.g., by validating config files and generating autocode prior to a deployment. Figure 3.3 shows example CLI output from the state machine compiler. Figure 3.4 shows example CLI output from running a state script.

```
state machine config error @ tank.sm:18:9:
  | valveOpe = true
  | ^ unknown element ‘valveOpe‘
```

Figure 3.3: Example CLI state machine compiler output

```
state script ran for 101 steps
100 asserts passed
final state vector:
  G = 1000
  P_AbortThreshold = 825
  P_WarnThreshold = 800
  S = 1
  T = 1000
  counter = 0
  pi = 3.141590
  tankPressure = 701.000000
  valveOpen = false
```

Figure 3.4: Example CLI state script output

Chapter 4

Evaluation

This chapter describes how the Framework and state machine DSL were evaluated as solutions to the stated problem. The chapter also revisits the design goals outlined in Section 2.1 and identifies features of the Framework that address each goal.

4.1 State Machine DSL User Studies

The design of the state machine DSL was informed by user studies. Five user studies were made to determine whether the DSL was intuitive to non-programmers and practical for real rockets. Users were student engineers from the Texas Rocket Engineering Lab with backgrounds in fluid systems, propulsion, GNC[1], and flight software. Only one user was primarily a software engineer. Data collected during the studies included observations of how quickly the user learned the DSL, verbal feedback on the DSL's applicability to real rockets, and DSL feature requests.

Each user was given a short tutorial on the DSL and tasked with writing two programs in the language. For the first program, the user developed the mission profile of a simple rocket (similar to the one in Section 2.2) and tested it using a state script. For the second program, the user developed a state machine of their choice for

[1] GNC is not a domain specifically targeted by the state machine DSL. There are already widely-used tools like Simulink Coder for autocoding GNC flight software.

a real-world application they had previous experience with. Choices included fuel tank pressurization, chilling of cryogenic fluid lines, and an orbital rocket mission profile. Each user used the DSL for about an hour.

In all studies, the user was able to quickly learn the syntax and semantics of the DSL and successfully apply it to a problem. The studies supported that the DSL does not require significant programming experience and can effectively express types of logic common in flight software.

4.2 State Machine DSL Hardware-in-the-Loop Test

As an experiment, the state machine DSL was used to replace the main state machine of the Halcyon rocket being developed by the Texas Rocket Engineering Lab. The Halcyon state machine was reimplemented in the state machine DSL, compiled to C++ via the Config Library, and integrated back into the Halcyon flight software. The modified flight software was then tested in a hardware-in-the-loop (HIL) simulation of Halcyon completing a suborbital spaceflight.

HIL is a technique for testing embedded systems by replacing the plant under control (in this case, a rocket) with a hardware emulation driven by a simulation of the plant. The Halcyon HIL consists of the actual flight computer and a software simulation of the rocket's dynamics and onboard systems. The HIL system converts simulation outputs to electrical signals that emulate the rocket's sensors and "trick" the flight computer into operating as if it were in flight. The flight computer responds with effector control signals that are read by the HIL system and factored back into the simulation in real time. This represents the highest fidelity form of flight software

testing possible without actually launching the rocket.

The new state machine orchestrated the simulated flight successfully, with the rocket reaching the target altitude and safely parachuting back to Earth. Responsibilities of the state machine included monitoring abort conditions and toggling flight software tasks on and off in response to specific events. The state machine DSL was well-suited for implementing this logic, which was essentially a mission profile for the rocket. Figure 4.1 shows a graph of the HIL test flight profile annotated with state transitions triggered by the state machine.

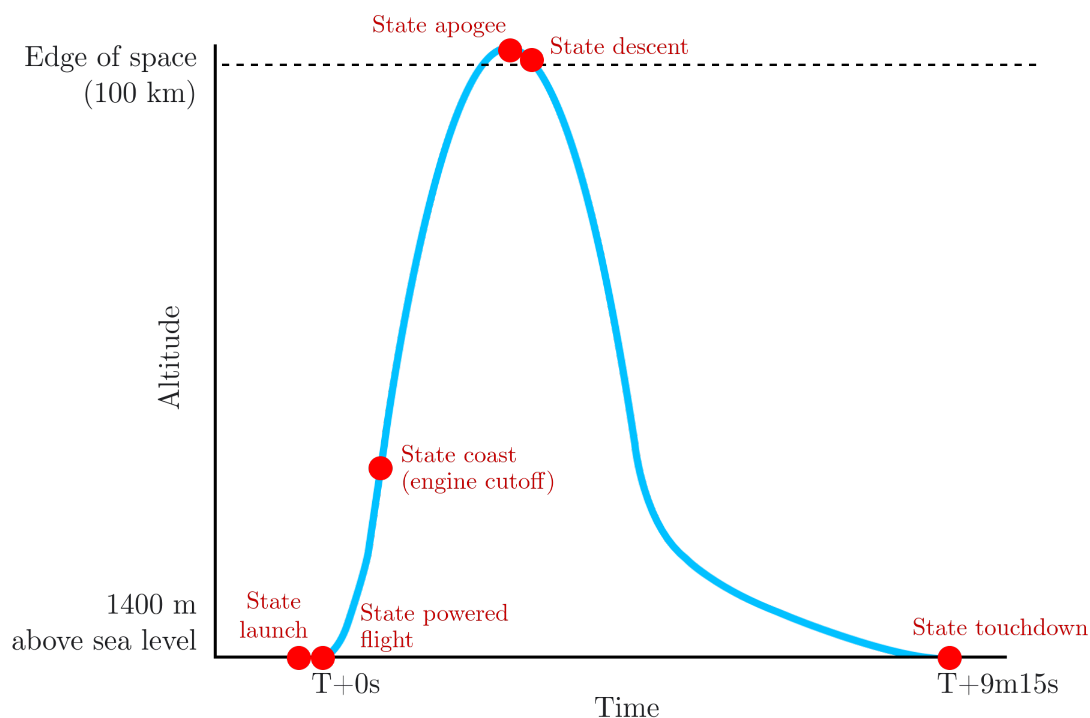

Figure 4.1: HIL test flight profile with state transitions. Circles mark points where a new state began.

4.3 Framework Design Goals Revisited

Design goal 1 was to provide portable interfaces for common flight software needs. These needs are met by the Core Library and PAL. Portability comes from the Platform Abstraction Layer and overall use of C++11.[2] Bare-metal applications (including those without heaps) can make use of the entire Core Library, while applications on operating systems can take advantage of the file system, heap, etc. by using the Config Library to increase runtime configurability of the Core Library.

Design goal 2 called for the ability to rapidly iterate flight software. This is addressed by the Config Library, which allows configuration of flight software from config files. Config files specifically target areas of flight software that are likely to change frequently: the state vector, device I/O, and state machines that control vehicle systems.

Design goal 3 demanded safety in flight software. Type conversion rules in the state machine DSL like downcast saturation are motivated by safety, as is the provision of a safe mode for tasks. Safety is also increased through other Framework implementation details not mentioned in this thesis like function error codes, non-use of exceptions, and precondition checking.

Design goal 4 asked that non-software domain experts be able to program their own control logic. The state machine DSL specifically caters to this. The DSL has a simple syntax and type system that make it instantly approachable by non-programmers. The DSL semantics make it suitable for expressing state-based control logic common in flight software.

[2] As opposed to later C++ standards, which not all embedded platform toolchains have adopted.

Chapter 5

Future Work

The Framework as presented in this thesis is functional enough to build complete flight software applications. This chapter covers a few opportunities for future work.

5.1 Improved State Machine DSL Toolchain

Before integrating a state machine into flight software, the programmer's only development tools are state scripts and the state machine compiler. There are improvements to these tools and additional tools that might aid state machine development. For instance, compile time static analysis would help catch certain logic errors like unreachable states and unused variables. A tool for generating state script skeletons that exercise all transitions in a state machine would help the user write stronger state scripts. Verification of particularly complex state machines might benefit from a debugger or statement coverage tool.

5.2 Network I/O and Entry Point Config Files

Additional config files would further streamline application development by reducing handwritten boilerplate. For example, the Framework PAL provides an interface for network sockets, but the user must develop any networking middleware themselves. A config file for network I/O tasks that defines network endpoints, channels, and

message formats would simplify network programming for distributed flight software applications. Additionally, the ability to autocode an application entry point with task and executor initializations would allow the user to focus on task business logic and not the glue code that runs it.

5.3 State Machine DLL for LabVIEW

The state machine DSL may be useful for real-time control applications beyond those onboard a rocket, for example, ground support equipment or engine test stands. NI LabVIEW is a popular software for controlling these systems via graphical programming and an ecosystem of plug-and-play data acquisition hardware. LabVIEW also supports runtime DLL calls, which may allow it to interface with the Framework state machine DSL if it were packaged into a DLL. The DLL would contain the state machine compiler, runtime, and functions for loading state machines, running them, and accessing the state vector. The DSL may be preferable to the LabVIEW graphical programming language for certain applications, but exploring this is outside the scope of this thesis.

Chapter 6

Conclusion

A rocket's onboard flight software represents the intersection of software engineering and many other technical domains. These domains may be especially numerous for rockets with liquid propulsion and 100+ km apogees, which are becoming more common at the collegiate level. This high multidisciplinarity, combined with a tendency for frequent design iteration, makes flight software development challenging. This thesis presented a C++ flight software framework that attempts to undercut these challenges for advanced collegiate rocketry projects. The framework caters to common flight software needs and supports config file formats for frequently-changed parts of flight software like device I/O. The framework also provides a high-level state machine DSL that simplifies software integration by allowing a non-software domain expert to program their own control logic, e.g., a propulsion engineer programming an engine ignition sequence. The state machine DSL was evaluated through user studies and a hardware-in-the-loop test in which the DSL took control of a simulated research rocket. The DSL was determined to be intuitive to non-programmers and effective at implementing common flight software logic like fluid system controls and mission profiles.

References

[1] Adam Aitoumeziane, Peter Eusebio, Conor Hayes, Vivek Ramachandran, Jamie Smith, Jayasurya Sridharan, Luke St. Regis, Mark Stephenson, Neil Tewksbury, Madeleine Tran, and Haonan Yang. Traveler iv apogee analysis. Technical report, University of Southern California, 2019.

[2] HeroX. Base 11 space challenge teams. `https://www.herox.com/spacechallenge/teams`, 2018. Accessed: 2022-4-23.

[3] Sarah Masimore. A distributed avionics software platform for a liquid-fueled rocket. Master's thesis, University of Texas at Austin, 2020.