Enhancing Trust in Autonomous Systems without Verifying Software

Joseph A. Stamenkovich

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Cameron D. Patterson, Chair

Bert Huang

Walid Saad

May 10, 2019

Blacksburg, Virginia

Keywords: Autonomy, Runtime Verification, FPGA, Monitor, Formal Methods, UAS, UAV

Copyright 2019, Joseph A. Stamenkovich

Enhancing Trust in Autonomous Systems without Verifying Software

Joseph A. Stamenkovich

(ABSTRACT)

The complexity of the software behind autonomous systems is rapidly growing, as are the applications of what they can do. It is not unusual for the lines of code to reach the millions, which adds to the verification challenge. The machine learning algorithms involved are often "black boxes" where the precise workings are not known by the developer applying them, and their behavior is undefined when encountering an untrained scenario. With so much code, the possibility of bugs or malicious code is considerable. An approach is developed to monitor and possibly override the behavior of autonomous systems independent of the software controlling them. Application-isolated safety monitors are implemented in configurable hardware to ensure that the behavior of an autonomous system is limited to what is intended. The sensor inputs may be shared with the software, but the output from the monitors is only engaged when the system violates its prescribed behavior. For each specific rule the system is expected to follow, a monitor is present processing the relevant sensor information. The behavior is defined in linear temporal logic (LTL) and the associated monitors are implemented in a field programmable gate array (FPGA). An off-the-shelf drone is used to demonstrate the effectiveness of the monitors without any physical modifications to the drone. Upon detection of a violation, appropriate corrective actions are persistently enforced on the autonomous system.

Enhancing Trust in Autonomous Systems without Verifying Software

Joseph A. Stamenkovich

(GENERAL AUDIENCE ABSTRACT)

Autonomous systems are surprisingly vulnerable, not just from malicious hackers, but from design errors and oversights. The lines of code required can quickly climb into the millions, and the artificial decision algorithms can be inscrutable and fully dependent upon the information they are trained on. These factors cause the verification of the core software running our autonomous cars, drones, and everything else to be prohibitively difficult by traditional means. Independent safety monitors are implemented to provide internal oversight for these autonomous systems. A semi-automatic design process efficiently creates error-free monitors from safety rules drones need to follow. These monitors remain separate and isolated from the software typically controlling the system, but use the same sensor information. They are embedded in the circuitry and act as their own small, task-specific processors watching to make sure a particular rule is not violated; otherwise, they take control of the system and force corrective behavior. The monitors are added to a consumer off-the-shelf (COTS) drone to demonstrate their effectiveness. For every rule monitored, an override is triggered when they are violated. Their effectiveness depends on reliable sensor information as with any electronic component, and the completeness of the rules detailing these monitors.

Acknowledgments

I would first like to thank Dr. Cameron Patterson for his advising on research, academics, career planning, and life in general. This would not have been possible without his continuous and genuine support. I would also like to thank Dr. Bert Huang and Dr. Walid Saad, professors whom I respect and greatly appreciate being readers of this thesis.

I would like to thank Lakshman Maalolan for his contributions to the work.

I thank my parents for their supportive feedback and proofreading, as well as understanding the work necessary during visits with family and even helping set up flight tests.

Thanks to GE Global Research for their support and collaboration with Jerry Lopez, Joel Markham, and Liling Ren.

I also thank my fiancée for supporting me through the struggles of research and graduate school, and dealing with drone parts and sensors around the apartment.

This research was supported by the NSF Center for Unmanned Aircraft Systems (C-UAS).

Contents

List of Figures ix

List of Tables xii

1 Introduction 1

1.1 Contributions . . . 3

1.2 Thesis Organization . . . 4

2 Background 6

2.1 General to the Design . . . 6

2.1.1 Runtime Verification . . . 6

2.1.2 Complex Functions . . . 7

2.1.3 Field Programmable Gate Arrays . . . 9

2.1.4 Universal Asynchronous Receiver/Transmitter . . . 10

2.2 Specific to the Implementation . . . 11

2.2.1 Quartus Design Suite . . . 11

2.2.2 ArduPilot . . . 13

2.2.3 QGroundControl . . . 13

2.2.4 MAVLink . . . 14

2.2.5 NMEA GPS 15

2.2.6 Intel Aero Quadcopter 15

2.2.7 Programming Without a JTAG Interface 18

2.3 Previous and Related Work 18

2.3.1 Runtime Verification for Real-time Embedded Systems 18

2.3.2 Monitors in FPGA Hardware 19

2.3.3 UAS Security and Safety Activity 20

2.3.4 Other Autonomous Safety Monitoring Techniques 20

3 High-level Design 22

3.1 ASTM International Standard F3269-17 23

3.2 Design Choices 26

3.3 Linear Temporal Logic 27

3.4 Automata 29

3.5 High-Level Synthesis 30

3.6 Model Checking 30

4 Implementation 33

4.1 Virtual Cage 33

4.1.1 Integer GPS Values 35

4.1.2 Nonaligned Virtual Cages 35

4.2 Monitor Synthesis 36
4.2.1 LTL Specifications to Abstract Automata 36
4.2.2 Abstract Automata to C Code 40
4.2.3 C Code To Parallel Hardware 43
4.3 Monitor Implementation Analysis 45
4.4 Pedigreed Component Architecture 47
4.5 Optimizations 48
4.5.1 Double Buffering 48
4.5.2 UART FIFO Buffers 50
4.5.3 Interrupt Routine Processing 51
4.5.4 Tightly Coupled Memory 55
4.6 Monitor Block Integration 57
4.7 Recovery Control Function Interjection 62

5 Evaluation **64**
5.1 Results 64
5.1.1 Indoor Flight 65
5.1.2 Unbounded Autonomous Flight 65
5.1.3 Bounded Manual Flight 67
5.2 Evaluation 68

5.3 Limitations 70

6 Conclusions **74**

6.1 Future Work 76

Bibliography **78**

List of Figures

1.1 Autonomous system domains 2

2.1 Some examples of complex functions 8

2.2 A generic FPGA architecture 9

2.3 Two UARTs communicating 10

2.4 Overview of the Quartus design suite 11

2.5 The steps required to map an FPGA design on the physical FPGA 12

2.6 A typical view of the QGC application 14

2.7 The Intel Aero drone, with optional indoor position sensor mounted on top . 16

2.8 The Aero compute board before adding monitors 17

2.9 The Aero compute board with the added monitor system shown 17

3.1 The overall design flow 22

3.2 Adding safety monitors to a UAS 24

3.3 Monitor event timing 25

3.4 Interactive Spot tool used to translate the LTL into Büchi automata 29

3.5 The basic idea behind model checking 31

3.6 Model checking closing the verification loop 31

4.1 A rendition of a simple virtual cage imposed on a drone 34

4.2 The virtual cage imposed inside of the netted Virginia Tech Drone Park and the rotation required to fit the shape 37

4.3 Formalizing what it means to leave the cage 38

4.4 AP values arising from different flight scenarios 40

4.5 Büchi automaton for the *within_cage_max_alt* specification 40

4.6 Handshake signals for an HLS-generated hardware block 44

4.7 Monitor analysis process 46

4.8 MAX 10 FPGA hardware resource types 48

4.9 I/O soft processor and monitor blocks 49

4.10 Quartus's Platform Designer creating the Nios II soft processor and connecting its peripherals 50

4.11 The fixed receive and transmit locations with swapping data buffers 51

4.12 The default UART architecture 52

4.13 The added FIFO components to the UARTs 52

4.14 The separate ISR design avoided that may be necessary under different circumstances 53

4.15 The one ISR design used 54

4.16 The default internal interrupt controller 54

4.17 The vectored interrupt controller mapping the processor directly to a particular ISR 55

4.18 The schematic of a single memory 56
4.19 The schematic of separate tightly coupled memories 56
4.20 Comparison of the ISR and context switching overhead optimizations 57
4.21 Monitor wrapper registers 59
4.22 Processor interface to the monitor wrapper 60
4.23 Overview of main and the ISR interacting with the double buffer communication scheme. 62
4.24 High-level view of the RCF being inserted into the MAVLink data stream . . 63
5.1 A rendition of an indoor flight test with the MarvelMind ultrasound sensors 66
5.2 Triggering the land RCF 67
5.3 A test succeeding in the Drone Park 68
5.4 Event sequence comparisons 71

List of Tables

2.1 The packet breakdown of a MAVLink message 15

2.2 U-blox support of NMEA GGA 16

4.1 SystemVerilog Assertions applied to the *within_cage_max_alt* monitor's hardware implementation 45

5.1 Resource utilization for the MAX 10M08 FPGA 69

5.2 Average execution time of software and hardware modules 70

List of Abbreviations

AP Atomic Proposition

ARV Autonomous Research Vehicle

COTS Consumer off-the-Shelf

FIFO First-in, First-out

FPGA Field Programmable Gate Array

GCS Ground Control Station

GPS Global Positioning System

HDL Hardware Description Language

HLS High-Level Synthesis

I/O Input and Output

IP Intellectual Property

LTL Linear Temporal Logic

LUT Lookup Table

pLTL Past-time LTL

QGC QGroundControl

RCF Recovery Control Function

RTA Runtime Assurance

STL Signal Temporal Logic

TCM Tightly Coupled Memory

UART Universal Asynchronous Receiver/Transmitter

UAS Unmanned Aerial System(s)

UTM Unmanned Aircraft System Traffic Management

VIC Vectored Interrupt Controller

Chapter 1

Introduction

We are at the forefront of an autonomous revolution as the technology shifts to much more sophisticated and safety-critical applications like our cars, drones, and other vehicles. Figure 1.1 shows some examples of emerging autonomous systems, and many applications are nearing deployment.

There is something interesting about driving late at night with opposing traffic in the adjacent lane, or boarding a red-eye flight more exhausted than you've ever been and not thinking twice about the pilot being in good enough health to fly safely. There is inherent trust that everyone we encounter is going to do what they are supposed to do and not jeopardize our own safety, despite knowing this is not always the case. Will we ever trust autonomous vehicles like we do other humans? Even more so than humans? Their use is already becoming more and more prevalent in society.

Iceland-based company Aha has begun using drones to deliver burgers across inconvenient rivers [8], but they're exploiting obstacle-free routes and safe airspace with no collision avoidance or even any imaging systems. Global positioning system (GPS) coordinates are programmed for the destination and off it goes. A simple system on a good start, but it is easy to see how these types of systems can quickly become problematic. Airport interference, like the one with Newark just this year [12], are becoming regular news stories. Infrastructure inspection is a widespread desired application for UAS, but currently, they cannot fly beyond visual line of sight of a human overseer because the risk of currently allowing them to do so

is too great [46]. Criminals are even exploiting the vulnerabilities of drones and the lack of infrastructure to support them by smuggling drugs [15]. The Mid-Atlantic Aviation Partnership is a Virginia Tech-affiliated, FAA-designated test site for addressing these concerns [9], including food delivery experiments in Blacksburg to assess the viability and vulnerabilities of personal drone delivery [6].

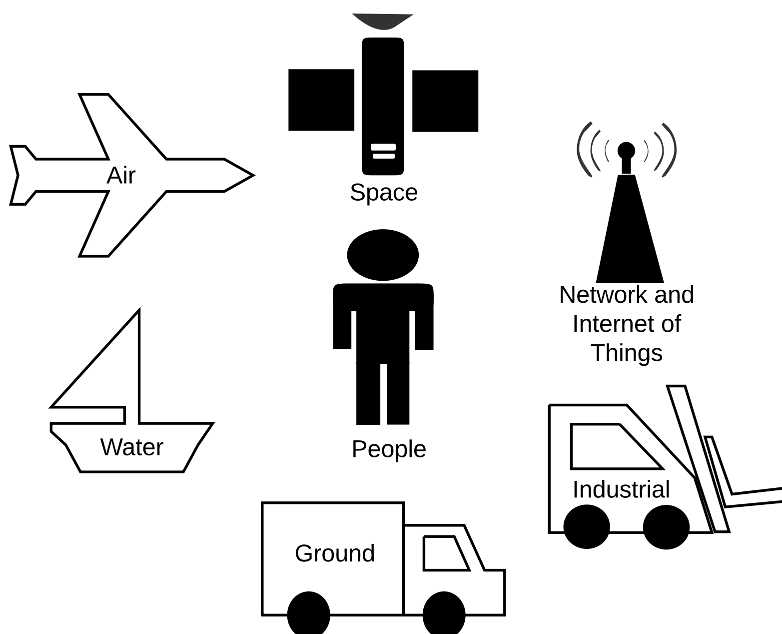

Figure 1.1: Autonomous system domains

An interesting and alarming aspect of using autonomy in these applications is the need to constantly make decisions. This raises the question of how to verify that the correct decision will be made in every scenario. Are there bugs in the code? What if there is a situation not encountered during training? Could a hacker take control of the vehicle? It is acknowledged that traditional means of software verification has struggled to satisfy these concerns. Formal analysis is unlikely since it took 25 person-years to establish the functional correctness of a 7000-line microkernel [27]. Even if one managed to prove that an embedded system's code is functionally correct, this result could be invalidated by an approved or malicious code update somewhere in the code stack. Yet, avionics technicians download software and

database updates from MyBoeingFleet.com to the latest Boeing aircraft at the gate over Wi-Fi using their laptop [5].

The alternative is using runtime verification techniques to confirm the active system adheres to formally stated correct behaviors [56]. These behaviors may include time-based propositions checked with automata rather than just basic logic needed for simple assertions. Normally runtime verification is used in a software or hardware debugging context, adding code to verify that the implementation does not deviate from specifications. However, this code is typically removed prior to deployment because of overheads. If malware is a concern, the verification code is not an effective countermeasure anyway since it could simply be disabled.

An approach is therefore proposed that retains monitors while avoiding software overheads and vulnerabilities. FPGAs permit system inputs (physical sensor data and commands received over a network) and outputs (actuator commands and system status reported over a network) to be monitored with configurable hardware. FPGAs allow efficient parallel implementation of all monitors which greatly reduces the overhead of consistency checking with formal specifications. The configurable hardware can be inaccessible and transparent to software, including operating systems or hypervisors.

1.1 Contributions

Several particular instances, mainly serving as proof of concept demonstrations of this approach, are developed over the course of the research. The approach is applied to an Intel Aero COTS drone, modifying the on-board FPGA. However, any autonomous system could have been selected for a demonstration. Monitors are successfully implemented to impose a virtual cage on the drone with both indoor ultrasonic positioning and GPS. The virtual cage

demonstration is an arbitrary choice to illustrate that the design is viable even on a basic platform with limited resources. In practice, a large collection of diverse monitors (e.g. flight plan adherence, envelope protection, traffic and obstacle avoidance actions, etc.) would be applied on commercial drones. The monitors are synthesized with formally captured behavioral rules and are added to the system in isolation from, and with no changes to, the software stack. They add little latency and can support an arbitrary number of concurrent monitors and recovery functions. The author's specific contributions are outlined below:

1. The hardware/software co-design to integrate and facilitate the monitors into the greater UAS system to physically implement the monitor design on the drone.
2. Including a high-performance and resource-constrained soft-processor with the control algorithms for creating and enforcing the safety behaviors of the UAS.
3. High-speed packet processing of autopilot software and flight controller communications.
4. Sensor processing while providing solutions for working with real-world environments and complications.
5. Creating and enforcing the safety behavior on a UAS.

The hardware monitors generated from LTL with model checking applied were contributed by others.

1.2 Thesis Organization

The remaining chapters of this thesis are organized as follows: Chapter 2 provides background knowledge and an overview of prior work. Chapter 3 provides an overview of the design, design decisions, and monitor transformations. Hardware and software implementa-

tion details and optimizations are presented in Chapter 4. Chapter 5 evaluates the approach before Chapter 6 concludes the thesis.

Chapter 2

Background

Autonomous systems are any device, or collection of devices, capable of completing a task or tasks without the help of outside intervention. A sub-class of autonomous systems that not only serves as a good example, but also perhaps the most interesting and applicable for the implementation detailed in this thesis, is autonomous vehicles. Self-operating cars, watercraft, and aircraft are all autonomous systems that are controlled by complex functions. A COTS drone is selected for the demonstration as autonomous aircraft are particularly safety critical and need enhanced trust. This chapter covers previous work done on this topic, as well as some supplementary information that will be useful in later sections.

2.1 General to the Design

2.1.1 Runtime Verification

Verification of hardware and software systems is nothing new, but what is termed runtime verification was developed in 2001 by what is now a company under the same name [57]. A more traditional practice is called static analysis. Static analysis is analyzing the software source code without executing it. Other techniques include the placement of analysis code before deployment to try and detect faults, or to detect bugs that went unnoticed after deployment prompting the release of patches to fix the issues [61]. This code is removed prior

to deployment because it usually degrades performance. Additional hazard analysis techniques also exist such as System-Theoretic Process Analysis, which is used to identify where unsafe actions may occur and capture flaws in the requirements to eradicate them in the design process [62]. Fault tree analysis is another technique which traces through the logic in a system to understand how a system may fail ahead of time [47]. Runtime verification is a different technique from the ones mentioned here by verifying that the code or system is performing to specifications while it is running in real time as part of the deployed product. "Runtime" in this context means this technique is actively and continuously providing verification alongside other system components as they operate in their intended conditions. This type of verification is selected for the safety monitors to continually provide assurances despite unforeseen issues or changes to the system as a whole.

2.1.2 Complex Functions

Complex functions are commonly referred to throughout this thesis. From a general perspective, they are functions or algorithms that are hard to verify or predict the result for every possible scenario. Figure 2.1 provides a few examples of complex functions. The recent rise in popularity of machine learning algorithms is of particular interest. Basically, a machine learning algorithm is trained on input data and learns how to adjust itself based on its output. In some cases, exactly what is going on inside the algorithm besides the input and output (I/O) is not precisely known, and this is why they may be referred to as "black boxes." Another concern is if they encounter a scenario in the real world that they were not trained on, they may act unpredictably. However, they are powerful tools for making decisions and in many cases faster and more accurate than a human can by noticing patterns and hidden features in data that humans cannot detect, so they have many useful applications for autonomous systems. Another complex function broad category is the non-

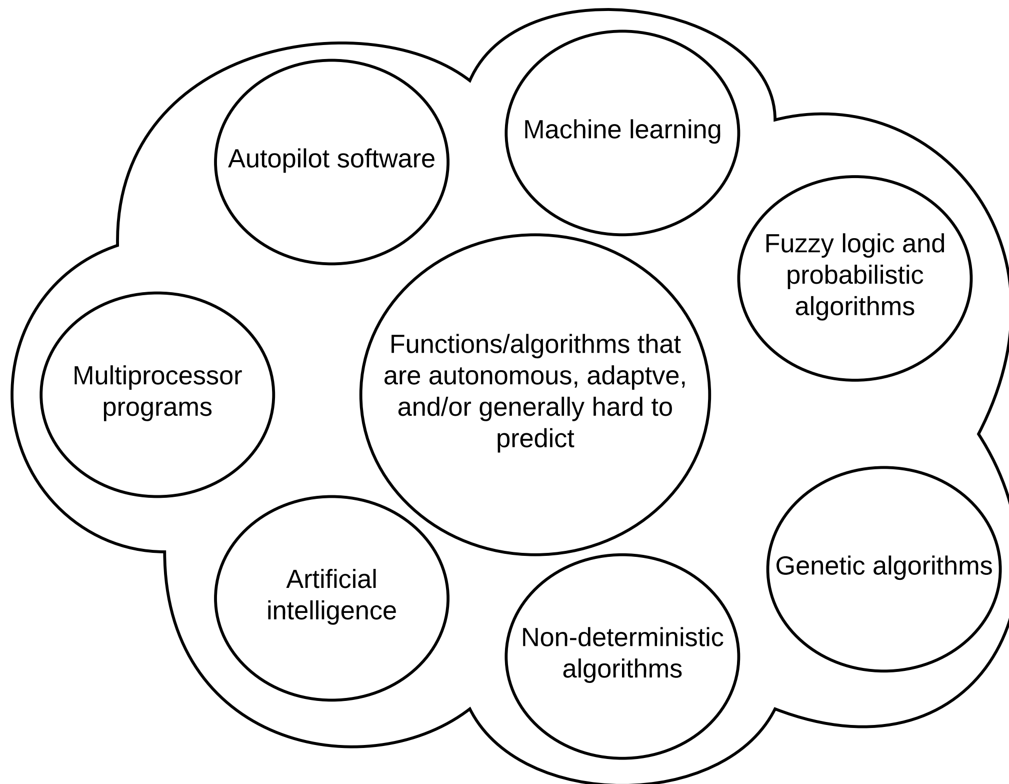

Figure 2.1: Some examples of complex functions

deterministic functions, which covers several of the examples. A non-deterministic function or algorithm is one that may have different outputs with different executions of the same code. Others may have high sensitivity to hard-to-measure conditions exacerbated by sensor noise, or have a certain probability of producing one output over another. It is difficult to guarantee a particular result with these types of functions. A practical example related to drones and autonomous aircraft is the Airborne Collision Avoidance System X developed by Lincoln Labs for the FAA [54]. It has been a proven system for effectively reducing collisions between aircraft, but its probabilistic models would classify it as a complex function. By keeping these algorithms and functions the complex functions category, the approach presented in this thesis requires no modifications to them.

2.1.3 Field Programmable Gate Arrays

As shown in Figure 2.2, an FPGA is an integrated circuit with a variety of hardware building blocks and interconnect that can be configured after fabrication to implement custom digital circuitry. Modern FPGAs include a heterogeneous mix of customizable memory, arithmetic, combinational, sequential, input/output, and specialized blocks. These components can be

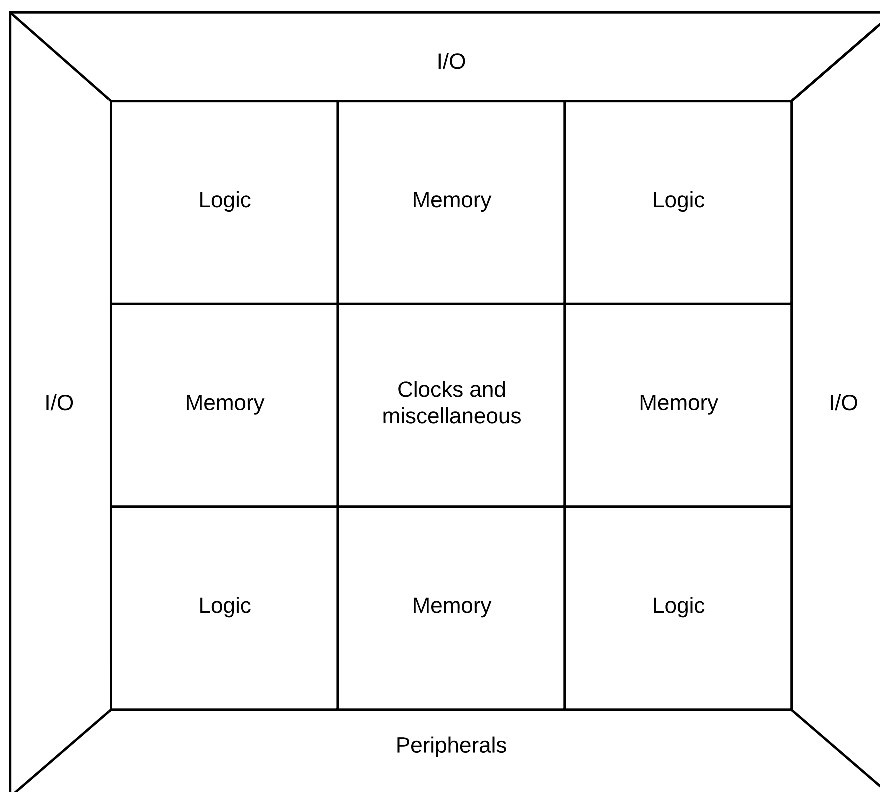

Figure 2.2: A generic FPGA architecture

assembled to create an instruction set processor referred to as a *soft processor* since it does not use immutable silicon resources like a typical processor. Specialized components are available to produce any multiple of an external oscillator's frequency. Other components include integer arithmetic units that can support any desired word width. The various memory components can be utilized for specific applications. Rather than share a single monolithic

memory, function units may have dedicated memories with the width and depth required. I/O components vary greatly from FPGA to FPGA but are essential for communicating with the outside world. Some are specialized, like an Ethernet or HDMI connection for example, and others are general purpose for configurable use. Boolean logic is typically implemented with logic elements consisting of lookup tables (LUTs) and flipflops.

2.1.4 Universal Asynchronous Receiver/Transmitter

A Universal Asynchronous Receiver/Transmitter (UART) is a serial communication device. Data is received and transmitted over one wire each between connected UARTs as shown in Figure 2.3. The rate at which data is transferred is the baud rate, and each data section of a packet is a byte. A start and stop bit are added to each packet so the UART knows when a packet is arriving and has finished. A parity bit is also frequently added to ensure packet integrity. These extra bits are removed by the receiving UART so just the relevant data may be easily accessed. The last packet received remains accessible in the UART just until the next packet, so data may be easily overwritten. To avoid this, interrupts are frequently used to alert connected systems to process the newly arrived packet. Similarly, interrupts are used to transmit data as quickly as possible by triggering one after each packet is transmitted.

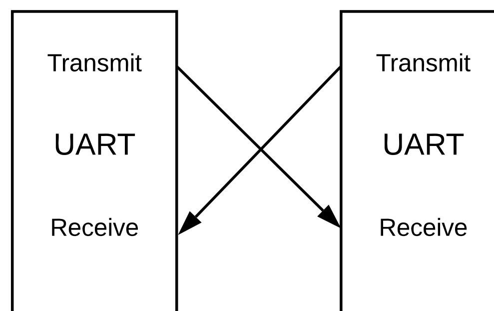

Figure 2.3: Two UARTs communicating

2.2 Specific to the Implementation

2.2.1 Quartus Design Suite

Quartus [14] is an Intel design suite for FPGA and programmable system-on-chip development. It allows each of their specific devices to be targeted with manual hardware description and auto-generation tools. Simulation and visualizations are also supported. Figure 2.4 shows the Platform Designer tool, which is a graphical environment that instantiates and connects soft processors and peripheral blocks. Software can then be created that targets created processors using the Software Build Tools in Eclipse, a common software development environment. Quartus also supports High-level synthesis (HLS) with its own custom

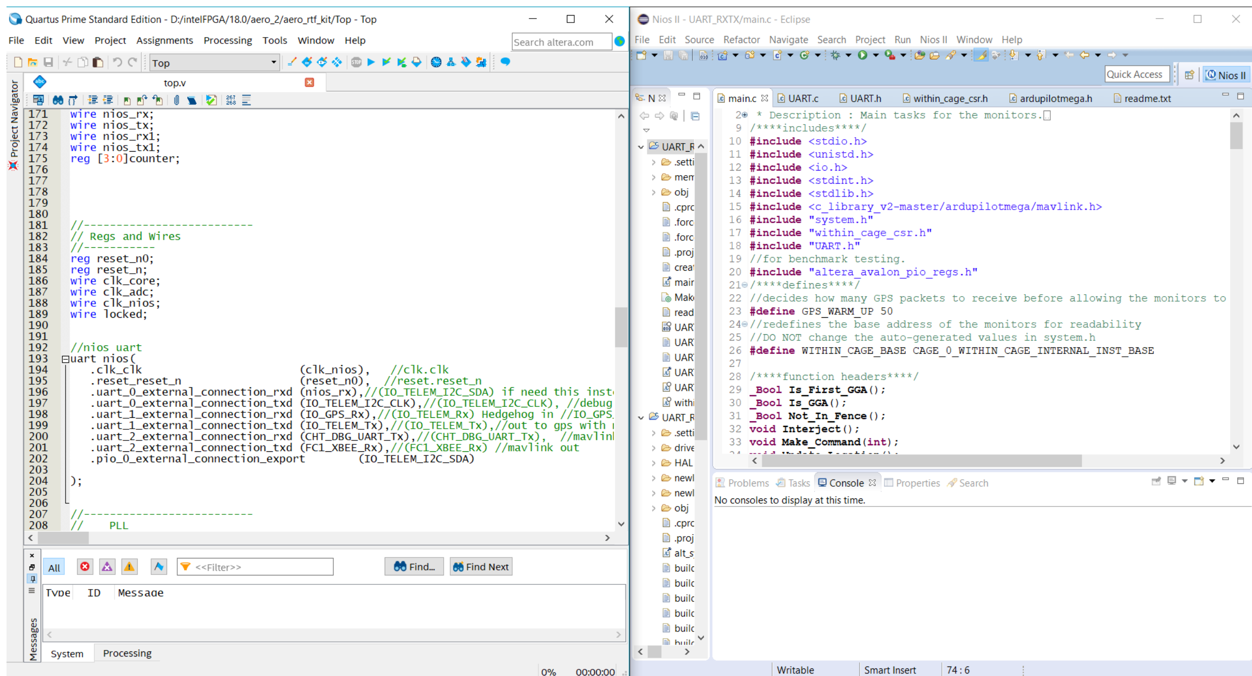

Figure 2.4: Overview of the Quartus design suite

tools. Custom intellectual property (IP) generated from the HLS tools can be instanced and connected manually with a hardware description language (HDL) or with Platform Designer as part of a soft processor. Tools are available to compile a design and target an FPGA [7].

Figure 2.5 shows these steps in order. First, pin planning is used to map I/O referenced in the design to physical pins on the FPGA. IP generation identifies and constructs any IP components used in the project, making sure they are compatible and up-to-date. Much like a software program, the analysis and elaboration steps check the design files for syntax or other errors. The synthesis step then optimizes the designed logic and maps it to actual logic block resources. The fitter finds the required resources on the specific FPGA being targeted, locates these components in a process called placing, and then connects them in a process called routing. Timing analysis verifies whether the design placed and routed by the fitter meets the timing constraints required for proper function. The power analyzer estimates overall power consumption. If either the timing or power constraints are not met, different compilation options can be used to try again, or adjustments to the design need to be made by the designer. Lastly, once a design is finalized the assembler packages it into an image to configure the FPGA. This is similar to a software assembler converting a high-level programming language into assembly language for the processor.

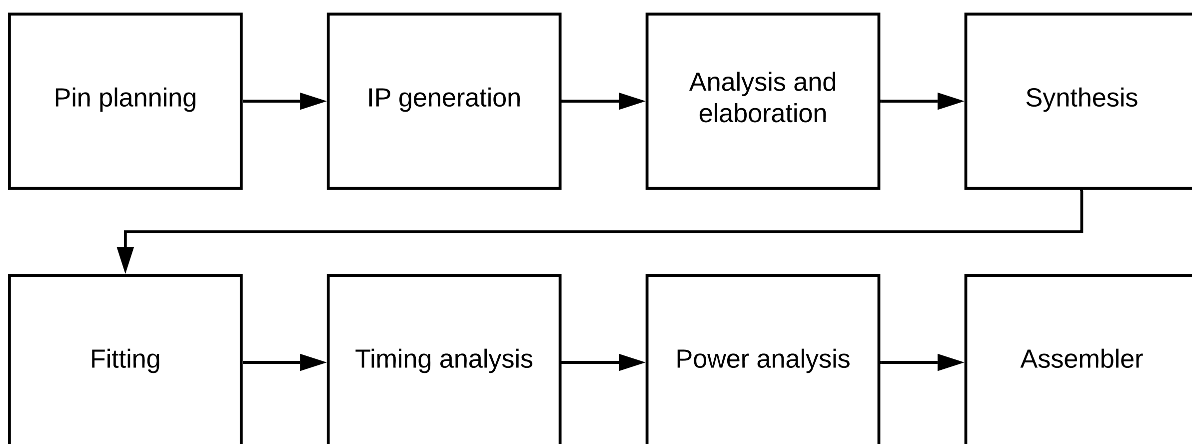

Figure 2.5: The steps required to map an FPGA design on the physical FPGA

2.2.2 ArduPilot

ArduPilot is an open source autopilot software used in this project and is considered the most tested and proven autopilot readily available [18]. It is unmodified in this work, and the end point of execution commands and flight functions for the drone used. PX4 is an alternative open source autopilot [19]. QGroundControl (QGC) communicates with ArduPilot through MAVLink commands, also detailed in this chapter. A wide array of information is processed by ArduPilot. Examples include flight commands, flight plans, status updates, location and attitude information, and sensor quality.

2.2.3 QGroundControl

QGC is a ground control station (GCS) that allows user interaction with a UAS. Parameters, status reports, camera feeds, and instructions can all be sent and received from this application. Figure 2.6 shows a typical interface of QGroundControl. Mobile app and personal computer versions are both available. It is through this interface that flight plans can be loaded and flight modes can be selected. They are received by a Wi-Fi connection on the drone and can then be processed by the autopilot and monitors. In return, the drone sends responses, acknowledgments, and specific messages that can be read and analyzed through QGC. Necessary parameters for different flight scenarios can also be configured through QGC. Examples are the speed of landing/takeoff, or the rate at which communication updates are received. Flights are recorded and can be played back in their entirety at a later time for analysis and confirmation of results.

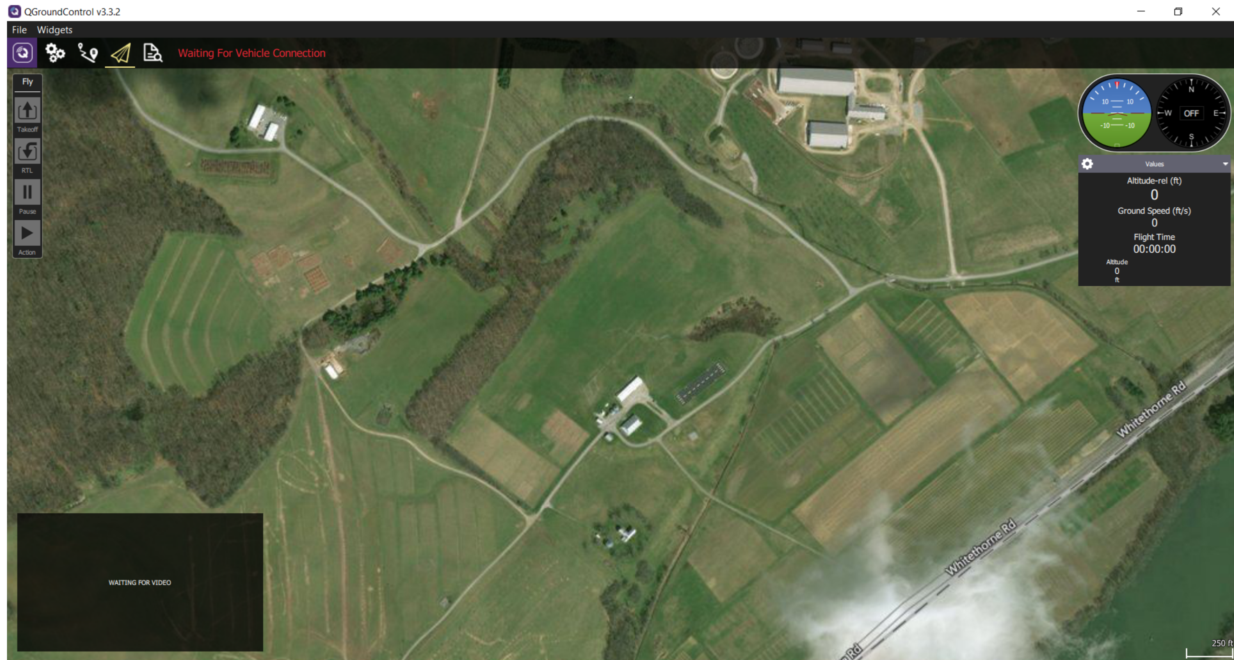

Figure 2.6: A typical view of the QGC application

2.2.4 MAVLink

MAVLink is a protocol used to communicate with drones, and between various components on the drone itself. It is very efficient, reliable, and portable [2]. The MAVLink channel is monitored in our approach, and our recovery control function (RCF) is a MAVLink command sent to the flight controller. Table 2.1 details the different fields a MAVLink packet should be packed into before sending. The `msgid` and `payload` are interpreted by the indicated `sysid` and `compid` to be distinct. The recipient of the message sends the appropriate response back to the system that generated the initial message. When messages are generated from the isolated monitor system, the autopilot assumes the command comes from the GCS and sends an acknowledgment which reflects the new state of the aircraft.

Data Type	Name	Description
uint8_t	magic	protocol magic marker
uint8_t	len	Length of payload
uint8_t	incompat_flags	flags that must be understood
uint8_t	compat_flags	flags that can be ignored if not understood
uint8_t	seq	Sequence of packet
uint8_t	sysid	ID of message sender system/aircraft
uint8_t	compid	ID of the message sender component
uint8_t	msgid 0:7	first 8 bits of the ID of the message
uint8_t	msgid 8:15	middle 8 bits of the ID of the message
uint8_t	msgid 16:23	middle 8 bits of the ID of the message
uint8_t	payload[max 255]	A maximum of 255 payload bytes
uint16_t	checksum	CRC
uint8_t	signature	ensure the link is tamper-proof (optional)

Table 2.1: The packet breakdown of a MAVLink message

2.2.5 NMEA GPS

NMEA is a human-readable GPS format consisting of ASCII characters. It includes several different message protocols with varying information. The protocol of particular interest is the GGA format and it is fully detailed in Table 2.2 [22]. The GPS coordinates and altitude information are extracted, processed, and used by the monitor system. Monitor systems can be designed to handle various other GPS and sensor formats. An alternative to NMEA GPS is the U-blox proprietary binary format supported with PX4. However, NMEA has the necessary information without having to write a custom binary parser. The GPS module available with many drones is a U-blox module with configuration settings to accept a variety of formats.

2.2.6 Intel Aero Quadcopter

The Intel Aero is a COTS, ready-to-fly drone permitting either manual or autonomous flight modes, and is shown in Figure 2.7. It is selected for the developer-targeted architecture

Field Num	Name	Unit	Format	Example	Description
0	$xxGGA	-	string	$GPGGA	GGA Message ID(xx = current Talker ID)
1	time	-	hhmmss.ss	092725.00	UTC time
2	lat	-	ddmm.mmmmm	4717.11399	Latitude (degrees and minutes)
3	NS	-	character	N	North/South indicator
4	long	-	dddmm.mmmmm	00833.91590	Longitude (degrees and minutes)
5	EW	-	character	E	East/West indicator
6	quality	-	digit	1	Quality indicator for position fix
7	numSV	-	numeric	08	Number of satellites used (rang:0-12)
8	HDOP	-	numeric	1.01	Horizontal Dilution of Precision
9	alt	m	numeric	499.6	Altitude above mean sea level
10	uAlt	-	character	M	Altitude in meters
11	sep	m	numeric	48.0	Geoid separation
12	uSep	-	character	M	Separation units: meters
13	diffAge	S	numeric	-	Age of differential corrections
14	diffStation	-	numeric	-	ID of station providing differential corrections
15	cs	-	hexadecimal	*5B	Checksum
16	<CR><LF>	-	character	-	Carriage return and line feed

Table 2.2: U-blox support of NMEA GGA

Figure 2.7: The Intel Aero drone, with optional indoor position sensor mounted on top

including the presence of a nearly unused MAX 10 FPGA between the compute board's Atom quad-core application processor and Arm-based autopilot, as shown in Figure 2.8. The Aero normally uses the FPGA to implement GPS connections, an analog-to-digital converter to monitor the battery voltage, a SPI interface, and a I2C bridge for the compass [41]. With all the relevant data and sensor information passing through the FPGA, monitors are

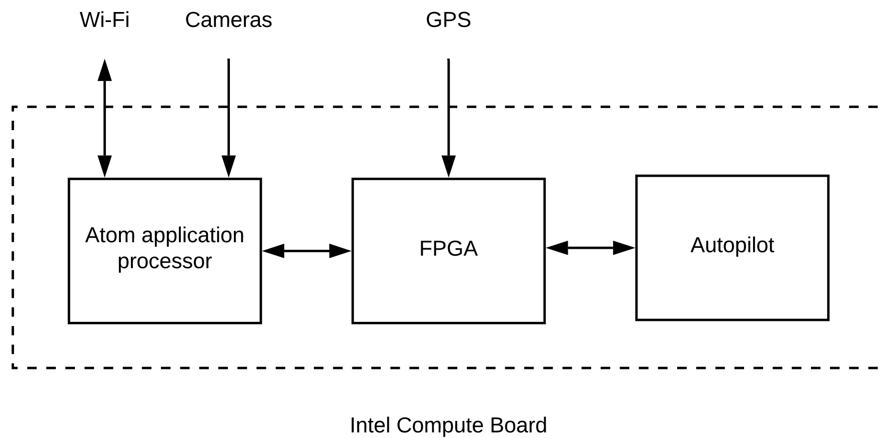

Figure 2.8: The Aero compute board before adding monitors

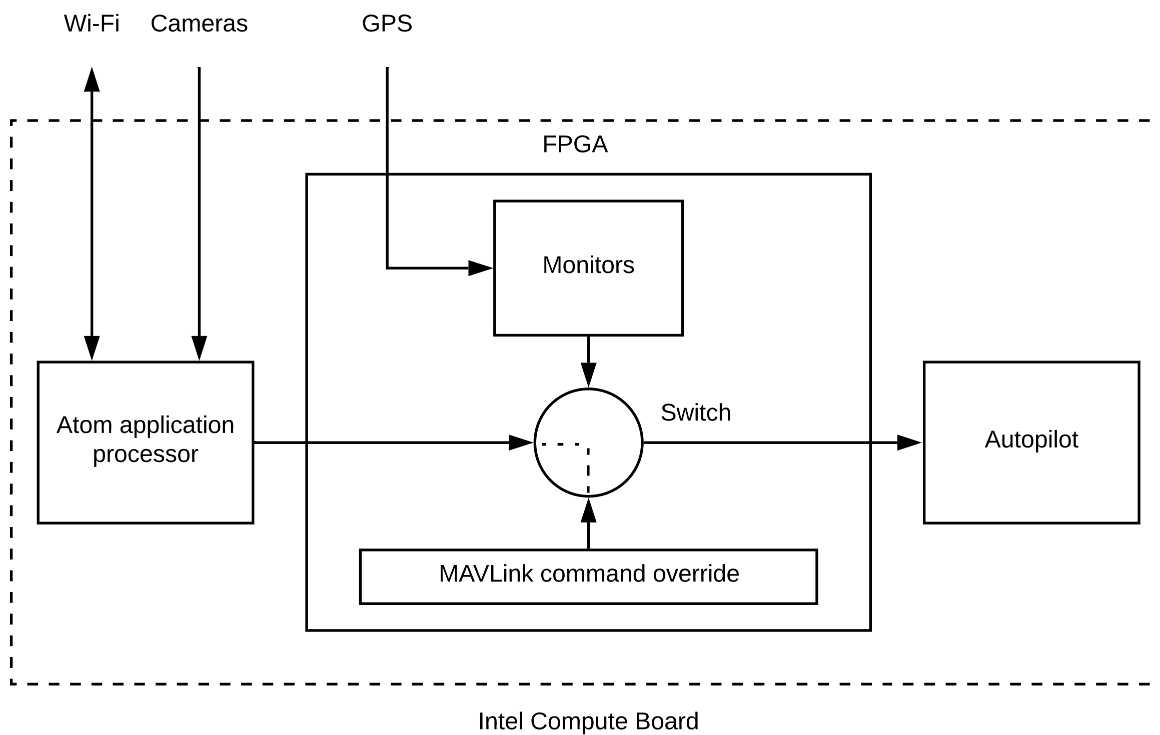

Figure 2.9: The Aero compute board with the added monitor system shown

seamlessly added to the system despite the small size of the FPGA. Figure 2.9 shows the added FPGA functionality. On platforms without the necessary components already present

on the system, hardware would have to be added. Flight functions can be preloaded on the Atom or transferred in real time from a ground station to the Atom over Wi-Fi. Common choices are QGroundControl [20] and Mission Planner [11]. The Aero supports running either PX4 [19] or ArduPilot [18] firmware as the autopilot. ArduPilot is selected for its ability to accept human-readable GPS data.

2.2.7 Programming Without a JTAG Interface

Although the embedded FPGA is convenient, there were development complications because no JTAG interface is provided. JTAG allows both direct FPGA programming and debugging with a host computer running Quartus. It is also required for hardware-in-the-loop simulation. Quartus supports the creation of a `.jam` file which packages the information necessary to program an FPGA without a JTAG. However, the system as a whole needs to run a script to process this file, requiring the Atom application processor to be involved in the process. Furthermore, the software running on the soft processor needs to be manually loaded into memory before creating the `.jam` file.

2.3 Previous and Related Work

2.3.1 Runtime Verification for Real-time Embedded Systems

The use of runtime verification to enhance the correctness and safety of real-time embedded systems has been investigated by many researchers. Bartocci et al. provide a comprehensive overview of runtime verification covering different specification languages, types of monitors, instrumentation techniques, and monitorability [31]. Multiple runtime enforcement and healing techniques and ways to use them to detect, prevent and react to failures are

discussed in [38]. Instrumentation conveys necessary information about the monitored system to the monitors. Cassar et al. discuss instrumentation techniques—broadly classified as online and offline instrumentation—in detail and analyze the existing monitoring tools [35]. *Invited* is runtime monitoring for vehicles that uses signal temporal logic and the Breach [36] monitoring tool [21].

2.3.2 Monitors in FPGA Hardware

Implementing monitors in FPGA hardware is a technique previously adopted by several others [42, 53, 55, 58, 60]. Pellizzoni et al. proposed a framework, BusMOP, used to synthesize hardware monitors. Monitors are synthesized into an FPGA and the device is plugged into a PCI bus to monitor transactions between different COTS peripherals [55]. In [60], formal specifications expressed in past-time LTL (pLTL) formulas are synthesized into hardware monitors used for runtime verification of embedded software in a system-on-programmable-chip which is instrumented to send information to the hardware monitor. A system-on-programmable-chip is an FPGA with a processor, peripherals, and interfaces. A procedure for the synthesis of hardware monitors from signal temporal logic (STL) assertions is presented in [42], and the viability of the approach for mixed and digital signal systems is demonstrated with examples. Nguyen et al. put forth a framework for synthesizing hardware monitors from STL assertions for assessing correctness and robustness of automotive systems-of-systems emulated on hardware. Similar to our approach, the monitors are synthesized using HLS [53]. In [43], the authors discuss a monitoring algorithm, EgMon, used to monitor an autonomous research vehicle (ARV) comprised of COTS components by passively observing the system's broadcast buses. The monitor is implemented on an ARM development board and evaluated against the logs obtained from the robustness test of their ARV. The utility of runtime monitoring at different phases of electronic product development in

the automotive industry is well-depicted in [59].

2.3.3 UAS Security and Safety Activity

UAS security and safety are active and critical areas of research. R2U2 is a monitoring framework which can detect and diagnose security threats to UAS but cannot prevent or mitigate attacks [58]. The hardware monitor runs its check on the GPS, ground control inputs, and critical data obtained from the flight software (sensor data, actuator data and flight software status). The trustworthiness of the approach is demonstrated by evaluating R2U2 on a NASA DragonEye UAS in a lab environment. Afman et al. propose a mechanical-based safety system to design a safe-to-crash UAS, in addition to discussing existing hardware and software safety measures [28]. Coupled with runtime verification, such designs will improve UAS trust. Majd and Troubitsyna develop runtime collision avoidance and route optimization algorithms for swarms of drones [49]. ATEVV is a divide-and-conquer verification approach for UAS starting from a formal language [39]. Safeguard is an independent on-board system used specifically for high integrity geofencing of UAS [40]. Similar to our system in that it is isolated from other UAS components, it addresses the need for quality sensors, is formally verified, and imposes boundaries with similar logic.

2.3.4 Other Autonomous Safety Monitoring Techniques

Many techniques have been explored for addressing safety concerns with autonomous systems. The Autonomous Vehicle Control Module Strategy points out that there is no specific method for assuring safety levels of autonomous vehicles and suggests integrating an independent system during the design of the architecture [51]. The $/A^2CPS$ is an autonomous supervision and control system that executes resilient actions at runtime to assist in avoid-

ing autonomous vehicle collisions. Khouri emphasizes the importance of developing efficient safety systems for machine learning and artificial intelligence-based autonomy [44]. The Safety Monitoring Framework developed by M. Machin et al. also uses formal verification techniques to automatically synthesize safety components implemented in software and begins with hazard analysis [48]. Mallozzi addresses machine learning verification issues by splitting the decision-making process between those algorithms and a predictive runtime hardware monitoring system for the safety-critical requirements [50].

Chapter 3

High-level Design

Before anything should be implemented, the objectives and constraints must be clearly identified. Background on the tools and techniques that address these objectives and constraints are also detailed in this chapter. Section 3.1 describes an abstract architecture adopted in this work. As a possible requirement for high reliability UAS, the architecture is intended to augment existing systems. Figure 3.1 details the implementation design process as a whole. The goal is to have as much automation as possible to reduce the potential of human error

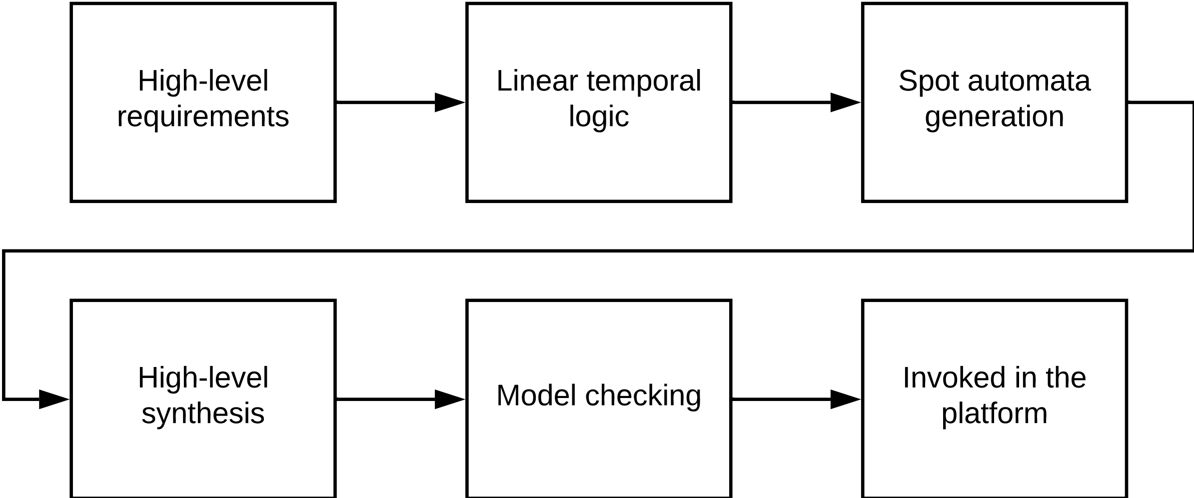

Figure 3.1: The overall design flow

and create as streamlined a development process as possible. All that is necessary to get started is high-level requirements. A 3D virtual cage serves as a high-level requirement example. The requirements are then formally captured. As Section 3.3 details, linear temporal

logic (LTL) is an appropriate choice for this application. The LTL formulas are converted to Büchi automata, which are continuously running state machines that accept infinite input sequences. The automata are translated to C code. High-level synthesis converts the C code to monitors implemented directly in digital hardware. The monitor hardware implementations are then validated using model checking.

3.1 ASTM International Standard F3269-17

ASTM F3269-17 defines best practices to *Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions* [30]. The primary goal of the document is to recommend a runtime assurance (RTA) architecture to verify complex functions implemented on a UAS that cannot be assured through traditional methods. There is a distinction between *non-pedigreed* and *pedigreed* components. A non-pedigreed component is software or hardware that has not been or cannot be proven to always exhibit a defined behavior with an acceptable level of certainty. A pedigreed component is the counterpart, namely a component that can be proven to always follow a prescribed behavior to the required level of certainty. Using isolated and pedigreed safety monitors to confirm the integrity of non-pedigreed, complex functions has the advantage of not constraining how the non-pedigreed components are implemented.

Figure 3.2 shows how the pedigreed monitors provide an external layer of oversight for the complex functions directing the flight controller, which is a specific vehicle management system responsible for producing physical actions from control commands. Non-pedigreed components transmit flight functions through the isolated component to the flight controller. If sensor data indicates a deviation from correct and safe behavior, the monitors activate the switch that routes the RCF to the flight controller rather than the normal flight function

source. The sensor data source and flight controller are required to be pedigreed through the imposition of other standards and practices. RCFs may vary from situation to situation, and several may exist on a particular system. A priority system is required if multiple RCFs may be triggered simultaneously,

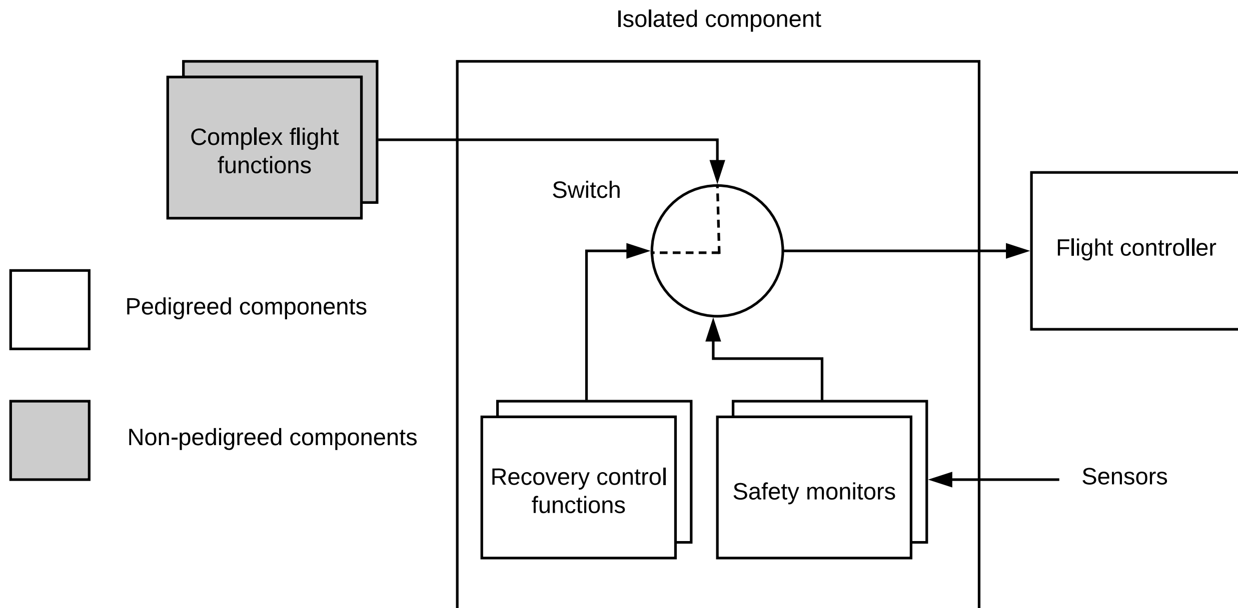

Figure 3.2: Adding safety monitors to a UAS

Figure 3.3 illustrates the monitor and RCF timing events. The RTA safety monitor cycle begins with a new set of UAS sensor data. Some steps are required regardless of monitor outputs, with additional time required if an RCF is invoked. Concurrent execution can reduce the execution time as the number of monitors is increased so that all processing completes before new sensor data are available. For data required by both the monitors and other UAS components, the sensor input communication channel can be shared instead of passing through the monitor system in order to reduce control loop latency.

The RTA must fulfill a variety of requirements:

1. A priority-based RCF hierarchy needs to be established.

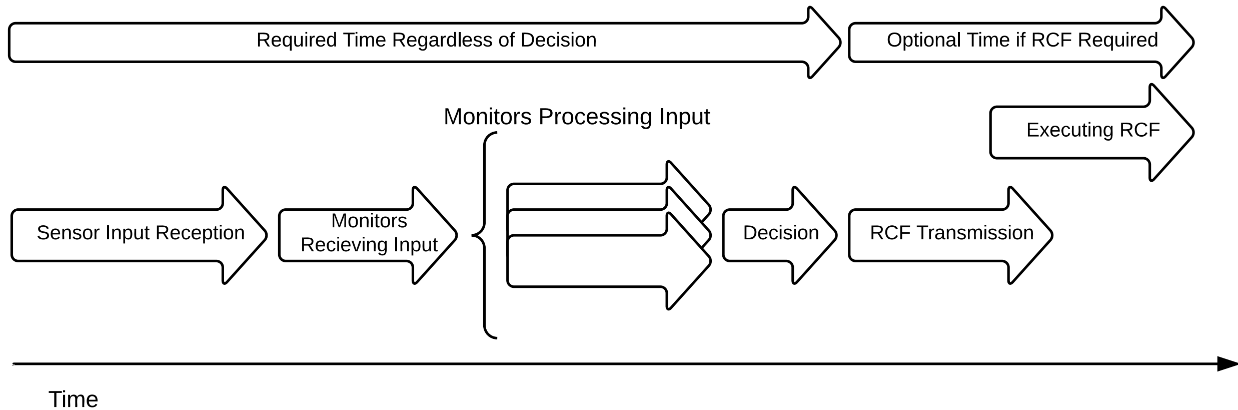

Figure 3.3: Monitor event timing

2. Complex functions must be separate from the safety monitors, RCFs, and switch.

3. An operational risk assessment must determine the likelihood and severity of the failures leading to an RCF.

4. An initialization protocol is needed to verify that individual components of the RTA safety monitor architecture are functioning properly.

5. The sensor types, performance, accuracy, precision, and frequency required by the safety monitors needs to be determined.

6. Safety monitors must be able to cope with failures of these inputs.

7. Complex functions must send the vehicle control commands to the RTA switch at the frequency needed for proper vehicle management.

8. RCFs must be designed to constrain the vehicle within predetermined limits.

9. The RTA switch must allow a continuous flow of commands to the vehicle management system even while switching between complex functions and RCFs.

10. The safety monitor must invoke the RCF in time to keep the vehicle operating within predetermined limits.

11. Each RCF may be enforced temporarily or until the vehicle completes a mission or mission phase.

12. Safety monitors must consider the confidence and quality of the sensor data when making a decision.

In summary, monitors and RCFs need isolation from the non-pedigreed components, very low latency, robustness, and correctness assurance. Throughout the document, these requirements will be referred to as "requirement (x)."

3.2 Design Choices

An integral part of the approach presented is designing the monitors in hardware. Most importantly, using independent hardware allows complete isolation from all existing components allowing the fulfillment of requirement (2). Software would have to execute on hardware, which in turn would have to be verified independently anyway. Hardware designs also inherently support parallelism, providing an advantage where autonomous systems would be running many monitors. In general, hardware implementations tend to be faster than the equivalent software implementations with less abstraction from the physical electronic components. To promote flexibility and easy configuration, an FPGA as detailed in Section 2.1.3 is used to implement the hardware instead of a fixed silicon circuit which would require refabrication with each change or addition. An FPGA's ability to implement a soft processor also provides a suitable means of generating RCFs, switching logic, and sensor I/O by being able to run its own software and connect to peripherals. The soft processor also allows for the creation of an RCF in the same format as created in the complex functions for seamless replacement when the recovery is initiated. Software executing on a soft processor

in the FPGA is simple and easy to verify, and can remain in the pedigreed classification.

In summary, the main goals of the approach are:

1. Guarantee that an autonomous system will be constrained to its requirements.
2. Provide these guarantees without dependencies on software or networks.
3. Continue to provide guarantees regardless of operator error, buggy code, malicious attacks, unprepared artificial intelligence, biased learning algorithms, etc.
4. Be as strictly additive and as minimally disruptive as possible.
5. Have minimal delay and resource use.
6. Implement the design in accordance with ASTM. Standard F3269-17's RTA architecture with pedigreed components.
7. Interpret pedigreed as formally synthesized and analyzed.

3.3 Linear Temporal Logic

LTL is a formal specification language that is used to specify and verify properties with respect to time. It consists of Boolean-valued atomic propositions (APs) and operators, as well as temporal operators representing properties in the future. This is an expansion of propositional logic, which defines present states. Examples of propositional statements include:

$a \wedge b$	a and b ,
$a \vee b$	a or b ,
$a \Rightarrow b$	a implies b ,
$\neg a$	not a.

While propositional logic allows formulas to be defined from time-invariant APs connected with Boolean operators, event sequencing and timing are essential aspects of reactive systems. These need to be included in descriptions of correct behavior. LTL expands on this with time modeled discretely as an infinite sequence of states. These operators include:

$X a$	proposition a is true in the *neXt* state,
$F a$	proposition a will be true at some *Future* time,
$G a$	proposition a will be *Globally* true in the future,
$a\, U\, b$	proposition a will hold true *Until* b becomes true.

LTL is used to formally specify the possible states of a UAS given its speed and position. EPITA's Spot tool [37] is used to translate the LTL formulas to automata, with the graphical web interface shown in Figure 3.4 to highlight how the tool works and some of the options available. The Spot tool's open source code may also be downloaded, compiled, and integrated into an overall flow. Generating runtime verification monitors from LTL and timed LTL (TLTL) specifications is extensively studied by Bauer et al. where they also demonstrate the feasibility of their methodology [32].

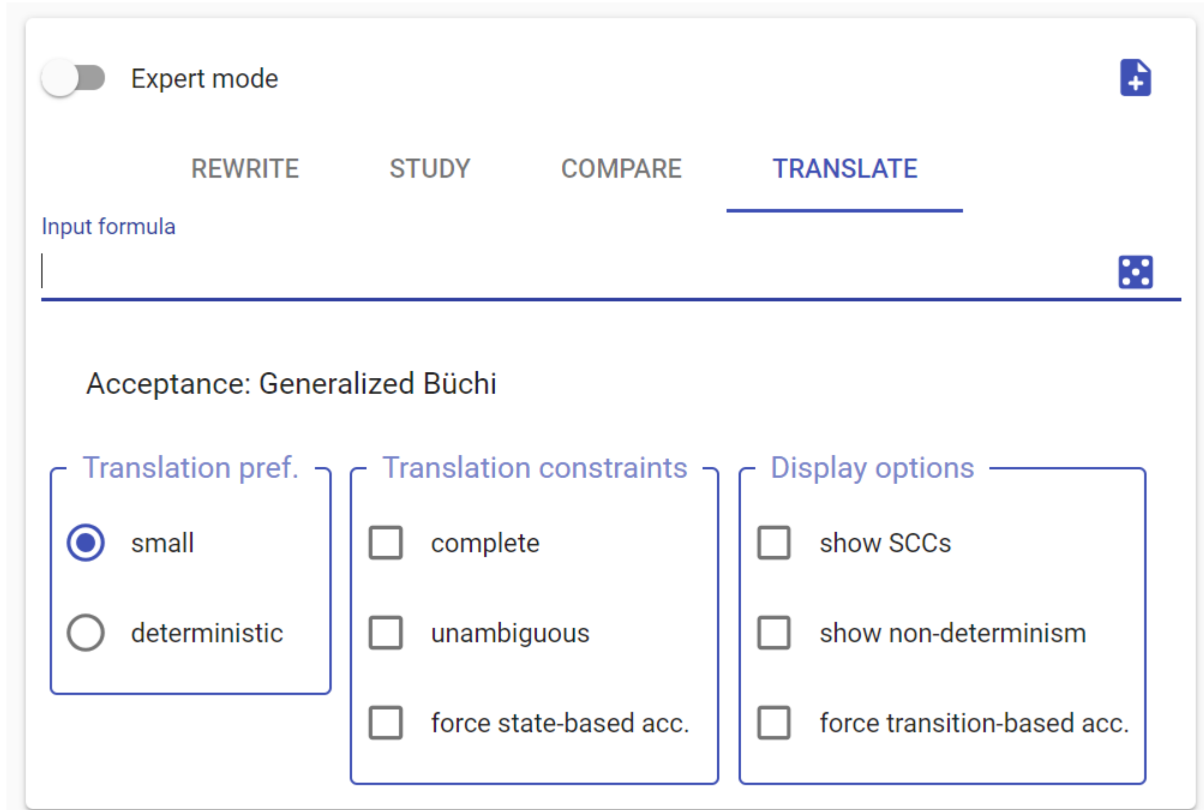

Figure 3.4: Interactive Spot tool used to translate the LTL into Büchi automata

3.4 Automata

In the simplest of terms, an automaton is a state machine reacting to a series of inputs. A state machine transitions between different states that may define state-specific actions. They accept finite strings from a formally defined language. A Büchi automaton is similar to a state machine, but can recognize the same infinite language as an LTL formula [34]. What distinguishes Büchi automatons specifically from other types is it extends the finite state machine to accept an infinite amount of input that is accepted if a final state is visited infinitely often. Rather than remaining in the accepting state, the automaton may be reset by returning to the initial state. Furthermore, Büchi automata support model checking.

3.5 High-Level Synthesis

HLS transforms C code into digital circuits rather than processor instructions as with software [52], and is used to generate the automata and switching logic within the monitor. Direct hardware implementation prevents common errors or attacks such as buffer overflows and stack corruption since monolithic memory and stacks are not used. Rather, physically distinct and private memories are allocated to arrays and buffers, and dedicated registers store function arguments and return values. This can improve performance and enable parallelism since a single, shared memory is often the computing bottleneck. Furthermore, the problem can be addressed from a higher level of abstraction than with an HDL. Clock signals, reset signals, denoting explicit wires, and other low level particulars that decrease productivity and are prone to errors are omitted. The full C language may be used except for library functions providing system calls and dynamic memory management.

To workaround C's sequential semantics, HLS normally adds parallelism by unrolling and/or pipelining inner `for` loops containing the calculations used in vector, matrix, and reduction operations. However, task parallelism is required (concurrent execution of monitor functions such as the virtual cage's five monitors) rather than data parallelism. Effective use of HLS requires the software structure and interfaces to mirror the desired hardware structure.

3.6 Model Checking

Model checking is a technique used to provide assurances beyond traditional testing techniques. The critical requirement for model checking is formally capturing a system model and its required properties. In the case of this approach, the specification is captured with LTL, and the model is an automata. Furthermore, model checking tends to be better suited

for hardware instead of software because of software's greater likelihood of state explosion with higher levels of abstraction and complexity. A major advantage over traditional testing is that model checking is able to perform an exhaustive analysis through what is essentially a graph search by exploring all possible state transitions. A test engineer does not have to worry about possibly forgetting a situation to check for, or even situations that nobody has even considered. Figure 3.5 shows the application of a model checker.

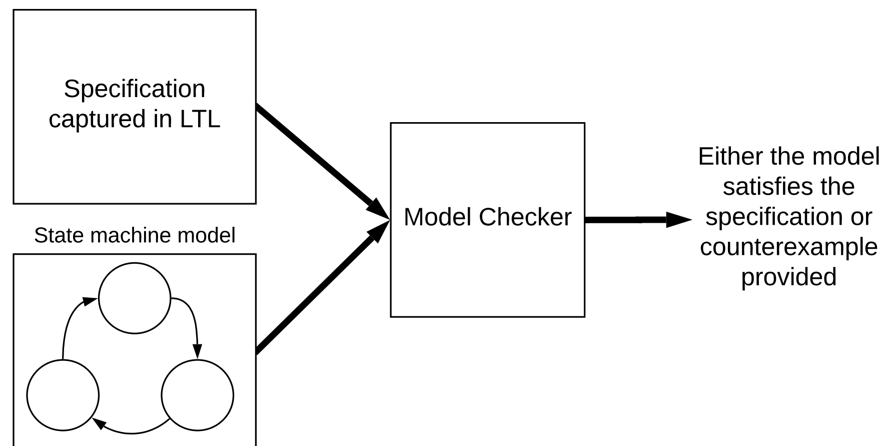

Figure 3.5: The basic idea behind model checking

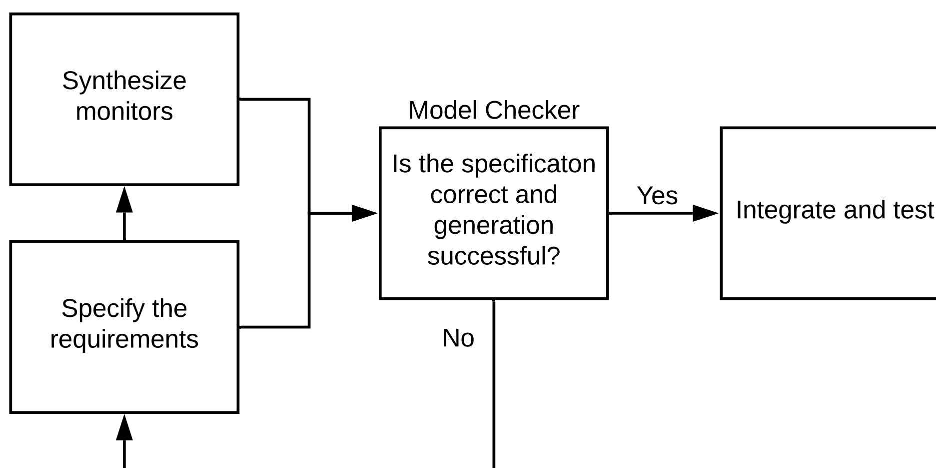

Figure 3.6: Model checking closing the verification loop

When model checking returns a negative result it will also provide a counterexample that can be used to better understand the results. With the use of a formal language, the results of model checking can be directly traced back and correlated to the specification. Model checking is applied to the hardware monitors generated with HLS. As shown in Figure 3.6, it provides a closed-loop formal verification process before any testing. This provides strong assurance that the implementation satisfies certain properties.

Chapter 4

Implementation

This chapter covers the implementation and integration of the high-level design strategy provided in Chapter 3. The design chain detailed is applied to the Intel Aero COTS drone discussed in Section 2.2.6 where hardware and software are designed simultaneously on the existing MAX10 FPGA. Challenges and contribution details are covered. Design alternatives are also considered. A virtual cage detailed in Section 4.1 is used to demonstrate the design process. A detailed look at the requirements to fulfill the virtual cage guides each of the steps and provides context to the abstract ideas. When a design choice targets a requirement from Section 3.1, it is referred to by the appropriate number.

4.1 Virtual Cage

Understanding the virtual cage is needed to appreciate the implementation. The concept is selected for its simplicity, dependency on only one sensor, clarity of the results, and practicality. As an extension, many other monitors can be implemented completely unrelated to the virtual cage monitors. A virtual cage is much like a physical drone cage, but instead of containing a drone physically with collisions, it contains drones electronically. A rendition of the concept is shown in Figure 4.1. The drone is contained by switching to an RCF when its trajectory would put the drone outside the cage in the near future.

A virtual cage is closely related to geofencing, but there are distinctions. Geofencing is

a software database that when enabled, will only allow the drone to fly in the areas not excluded by the database. The virtual cage yields the same behavior but is implemented by the physical circuitry on the FPGA. The monitors are not using software-based geofencing in any way to demonstrate the implementation. This is important because the software running the geofencing databases, and even the databases themselves are generally considered non-pedigreed, and therefore susceptible to bugs and hackers. Although addressing this problem in particular is not the goal of this thesis, it does serve as an example of a particular problem that can be addressed. To efficiently define a virtual cage, there is an independent monitor for five of the six fences. In other words, the maximum latitude, longitude, and altitude are three distinct monitors, and the minimum latitude and longitude are two more distinct monitors. Minimum altitude is excluding because it is just the ground. The monitors are aware of the drone's location from the NMEA GPS data described in Section 2.2.5.

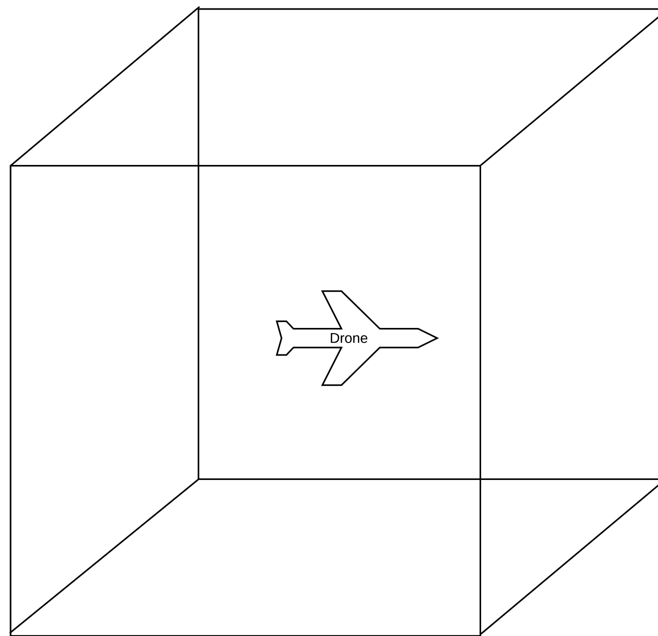

Figure 4.1: A rendition of a simple virtual cage imposed on a drone

4.1.1 Integer GPS Values

The limited resources available on the FPGA prevented the use of floating point variables and arithmetic. This is of particular concern since the NMEA GPS data is received as degrees, minutes, and decimal minutes. To circumvent the problem of having to compare decimal minutes and use floating point, a custom unit is created using the first five fractional minutes with a precision of minutes/10^5. Each value is computed by:

$$(deg \times 6000000) + (min \times 100000) + (frac.\ min) \tag{4.1}$$

This precision is accurate to approximately a few inches. The GPS coordinates are converted to integer format in the soft processor by parsing the individual components, converting to 32-bit integers, multiplying by integer constants, and then adding the terms together. The result is input to the monitors. The cage must also be defined with the integer coordinates so there may be a direct comparison.

4.1.2 Nonaligned Virtual Cages

To keep the initial implementation simple and clear to illustrate the design process, the virtual cage is ideally aligned with the cardinal directions north, east, south, and west to correspond with the maximum and minimum latitude and longitude. However, nonaligned cages can also be accommodated without requiring trigonometric calculations. An example is accommodating Virginia Tech's Drone Park as shown in Figure 4.2. The goal is to define a virtual cage inside of the physical cage to safeguard against drones hitting the net. The physical cage has a northwest alignment. To compensate, the cage is rotated 45 degrees after it is defined. Then to have a consistent comparison, GPS packet coordinates are also

rotated 45 degrees around the center of the cage before being input to the monitors. Without floating point available, trigonometric functions cannot be used. The cosine and sine of 45 degrees is therefore approximated with integers as 724/1024, which is equivalent to the third decimal. Since the distances are so small, the radius of the earth was also excluded and it is treated as a 2D transformation. The equations applied on the incoming sensor data are then as follows with latitude abbreviated as *lat* and longitude as *long*:

$$lat_rotate = (lat - lat_center) \times 724/1024 \tag{4.2}$$

$$long_rotate = (long - long_center) \times 724/1024 \tag{4.3}$$

$$new_lat = lat_center + lat_rotate - long_rotate \tag{4.4}$$

$$new_long = long_center + lat_rotate + long_rotate \tag{4.5}$$

4.2 Monitor Synthesis

The architecture described in Section 3.1 supports an arbitrary number of monitors synthesized from correctness specifications captured in LTL. To avoid potential errors arising from manual implementation, the monitors are generated using the steps described in Sections 4.2.1, 4.2.2, and 4.2.3.

4.2.1 LTL Specifications to Abstract Automata

Formal specifications need to be captured in a language with formal syntax and semantics. As discussed in Chapter 3, LTL is the appropriate choice. Before jumping to the LTL

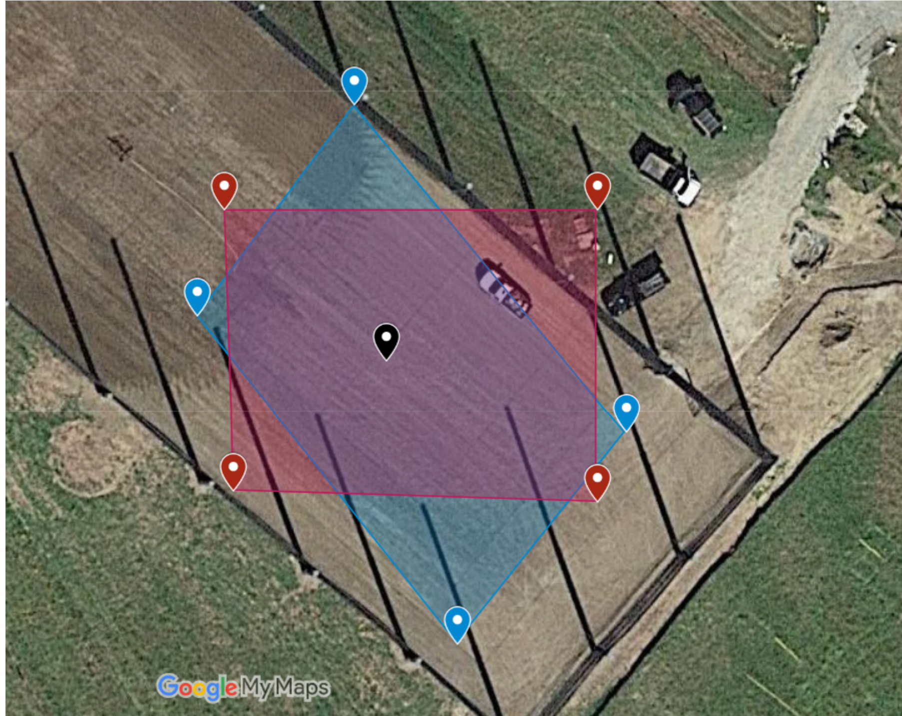

Figure 4.2: The virtual cage imposed inside of the netted Virginia Tech Drone Park and the rotation required to fit the shape

however, a clear high-level requirement needs to be established. For a virtual cage the requirement can be expressed informally as "the drone needs to stay in the cage", but this needs to be formalized. A good question to address is what does it mean to be leaving the cage? Figure 4.3 depicts a scenario where the drone is leaving the cage with real-world values considered to help in formalizing this condition. This is the most important step for implementing the monitors in accordance with the requirements and objectives.

Requirements (3), (5), (6), (8), and (10) are of particular interest when designing the LTL formula. These emphasize the importance of considering the sensors available and their performance when designing the safety monitors. In our case, the relevant sensor is the GPS (or ultrasound in the early implementations) with both position and speed data being received in one-second intervals. It's clear that if the drone is near the cage edge (referred to as the slow zone), and approaching that edge, then it needs to be able to slow down until

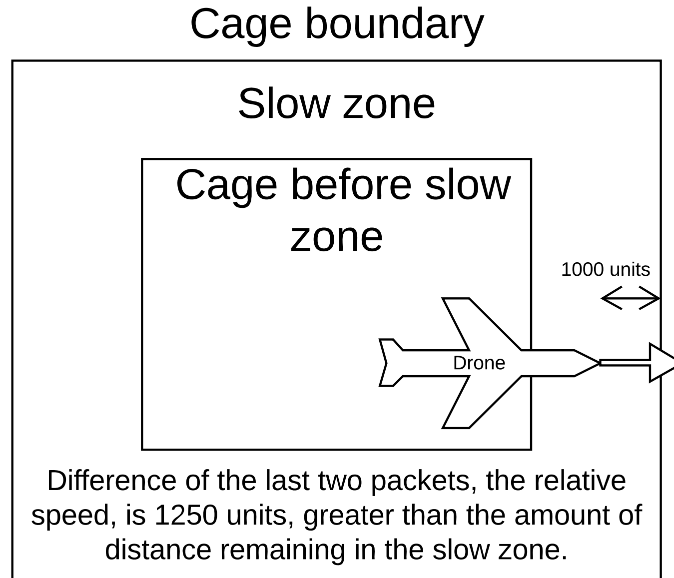

Figure 4.3: Formalizing what it means to leave the cage

it stops in order to not breach the cage. The reason why the scenario in Figure 4.3 is one where the drone will breach the cage is because its speed of 1250 units/second is greater than the distance remaining in the cage after a second. This if-then relationship discussed is referred to as an implication, similar to a programming language statement `if a then b`. In LTL this is expressed as:

$$a \Rightarrow b \tag{4.6}$$

Therefore, correct behavior for a UAS flying inside a virtual cage can be partially captured with the LTL formula

$$(nearing_fence \wedge in_slow_zone) \Rightarrow (can_slowdown\, U\, stopped) \tag{4.7}$$

In other words, if the UAS is approaching a particular fence and enters the fence's slowdown zone, then it should decelerate in proportion to the remaining distance from the fence until

it stops. This expresses the desired approximate actions when a UAS moves towards a fence. There is a distinct fence and monitor for the maximum altitude, minimum latitude, maximum latitude, minimum longitude, and maximum longitude. The APs have a different binding for each fence. For example, the AP values for the maximum altitude monitor are determined from the UAS vertical speed (ver_speed) and altitude (alt):

$$
\begin{aligned}
nearing_fence &= (ver_speed > 0) \wedge (alt > MID_ALT), \\
in_slow_zone &= (alt > UP_SLOWDOWN_START), \\
can_slowdown &= (ver_speed \propto (MAX_ALT - alt)), \\
stopped &= (ver_speed = 0).
\end{aligned}
$$

Momentum must be taken into account since a UAS cannot stop instantaneously. On the other hand, a UAS should be allowed to move parallel to a fence without decelerating even if it is within that fence's slowdown zone. The distinctions between nearing the fence and being in the slowdown zone, as well as between slowing down and stopping, provide more detailed scrutiny of the UAS behavior. Additional behaviors could be constrained with extra formulas rather than complicate a single formula. Each of these scenarios with the corresponding propositional values are illustrated in Figure 4.4. A catch-all assertion could require the UAS to always be inside the cage in case of loopholes in the other formulas.

LTL formulas can be converted to Büchi automata which are an infinite input extension to finite automata used in model checking [45]. The Büchi automaton corresponding to Equation 4.7 is generated by the EPITA Spot tool [37] and shown in Figure 4.5. Valid event sequences are displayed; any other event sequences are invalid.

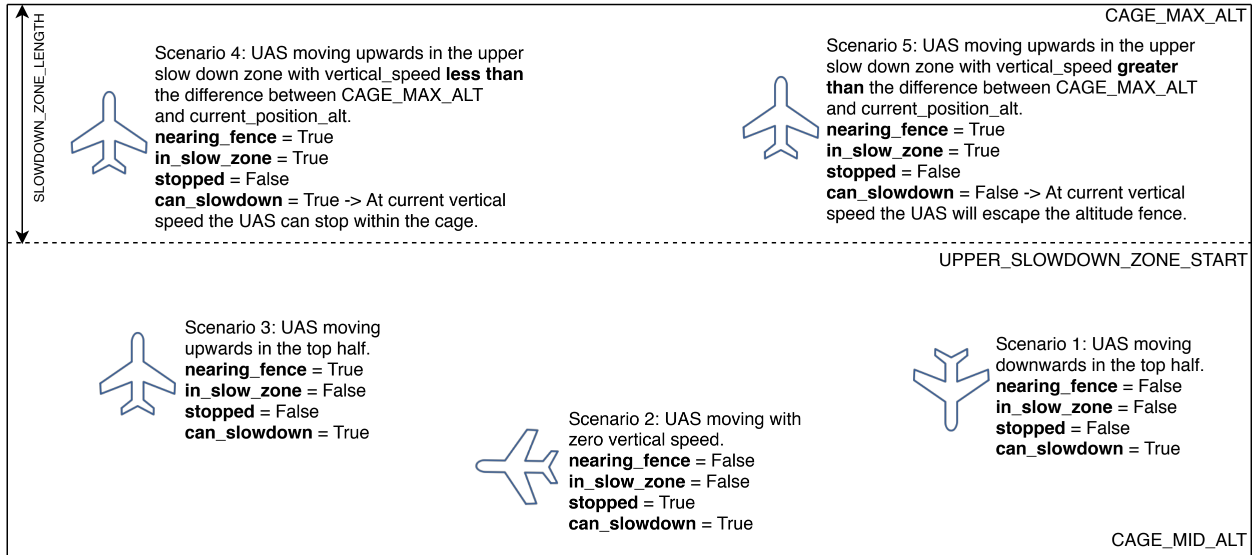

Figure 4.4: AP values arising from different flight scenarios

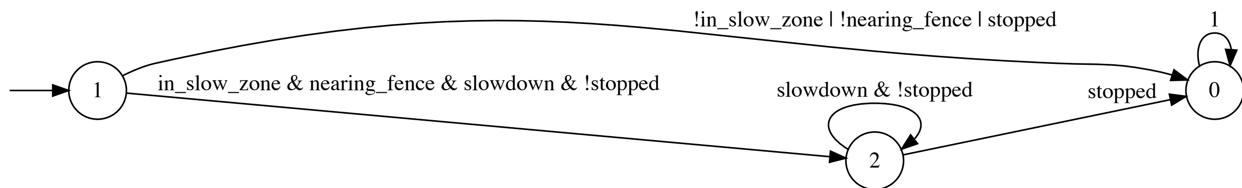

Figure 4.5: Büchi automaton for the *within_cage_max_alt* specification

4.2.2 Abstract Automata to C Code

An intermediate translation to C is easier than a direct translation to an HDL such as VHDL or Verilog, especially for defining override calculations and algorithms. Unlike HDLs, there is no need for explicit clock or reset signals. The function in Listing 4.1 is invoked once per control cycle and is generated semi-automatically. First, a Python script shown in Algorithm 1 generates the function skeleton and state machine code from a textual representation of the *within_cage_max_alt* automaton shown in Figure 4.5. State information is retained in a local static `state` variable. Function arguments provide the sensor values needed to define the APs. The Python script adds declarations of the APs appearing in Equation 4.7, although initialization code must be manually entered. Any undefined automata transitions

cause the `state` variable to have the `ALARM_STATE` value. If this occurs then an RCF (such as forcing the UAS to hover or land) may be invoked.

Algorithm 1: Automata to C Translation Script

```
1  define a C enum with N+1 states where N is the number of states in the automata
2  declare a variable for each AP in the automata
3  output a switch statement
4  for each of the automata states do
5      output a case statement for the state
6      for each transition from the current state do
7          output an (else) if statement and the expression for the transition
8      end
9  end
10 output a default statement
```

Although the function code requires manual additions, all monitor automata code is automatically synthesized from the original LTL. Hence a state machine implementation detecting discrepancies between correct and observed behavior is generated directly from formal specifications. This is preferable to manually generating an abstract automata or its C implementation because of the added complexity and potential for oversights. For additional assurance, Section 4.3 applies model checking to the automaton's final hardware implementation without concern for state explosion.

Translating Büchi automata to C allows several extensions that cannot be expressed in LTL. As shown, APs may be defined using any C language statements and function calls applied to sensor variables. Although not applicable to this example, liveness can be checked by imposing a time limit to reach an accepting state. This provides much of what is offered by timed automata [29] without complicating the LTL. If a control cycle counter reaches zero before a particular state is reached, corrective action can be taken. Time bounds can be set on certain state transitions without adding extra states and transitions. These extensions still permit the use of existing tools to translate basic LTL formulas into Büchi automata.

Listing 4.1: C function for the *within_cage_max_alt* automaton

```
1  #define MID_ALT ((CAGE_MIN_ALT + CAGE_MAX_ALT) / 2)
2  #define UP_SLOWDOWN_START (CAGE_MAX_ALT - SLOW_ZONE_LENGTH)
3
4  bool within_cage_max_alt(int alt, int ver_speed)
5  {
6    enum States {STATE0, STATE1, STATE2, ALARM_STATE};
7    static enum States state = STATE1;
8
9    bool nearing_fence = ((ver_speed > 0) && (alt > MID_ALT));
10   bool in_slow_zone  = (alt > UP_SLOWDOWN_START);
11   bool can_slowdown  = (ver_speed <= (MAX_ALT - alt));
12   bool stopped       = (ver_speed == 0);
13
14   switch (state) {
15     case STATE0:
16     case STATE1:
17       if (!in_slow_zone || !nearing_fence || stopped)
18         state = STATE0;
19       else if (in_slow_zone && nearing_fence && can_slowdown && !stopped)
20         state = STATE2;
21       else
22         state = ALARM_STATE;
23       break;
24     case STATE2:
25       if (stopped)
26         state = STATE0;
27       else if (can_slowdown && !stopped)
28         state = STATE2;
29       else
30         state = ALARM_STATE;
31       break;
32     default:
33       state = ALARM_STATE;
34       break;
35   }
36
37   return (state != ALARM_STATE);
38 }
```

4.2.3 C Code To Parallel Hardware

It is important that an HLS algorithm captures the desired hardware structure to be synthesized. This is the primary reason to: (1) isolate each monitor in a function such as Listing 4.1's `within_cage_max_alt()`; (2) explicitly pass sensor values as arguments so that the monitors may be consulted every control cycle; and (3) minimize the function's code complexity in order to reduce the I/O latency. A large number of simple monitors provides more opportunities for parallelism than a small number of more complex monitors.

Listing 4.2: C function for the *within_cage* monitor wrapper

```
1  bool within_cage(int alt, int ver_speed,
2                   int lat, int lat_speed,
3                   int lon, int lon_speed)
4  {
5    bool max_alt_check = within_cage_max_alt(alt, ver_speed);
6    bool min_lat_check = within_cage_min_lat(lat, lat_speed);
7    bool max_lat_check = within_cage_max_lat(lat, lat_speed);
8    bool min_lon_check = within_cage_min_lon(lon, lon_speed);
9    bool max_lon_check = within_cage_max_lon(lon, lon_speed);
10
11   return (max_alt_check &&
12           min_lat_check && max_lat_check &&
13           min_lon_check && max_lon_check);
14 }
```

HLS semantics permit a sequence of functions to be executed concurrently if there are no dependencies. Therefore, HLS is also applied to a wrapper C function that invokes all monitors in order to execute them in parallel. The wrapper C function shown in Listing 4.2 accepts the six arguments of altitude, vertical speed, latitude, latitude speed, longitude, and longitude speed. The `main()` routine extracts and prepares these arguments from the relevant sensor data. It then invokes the five fence monitors and returns whether all monitors return `true`. If one returns `false`, the RCF is triggered.

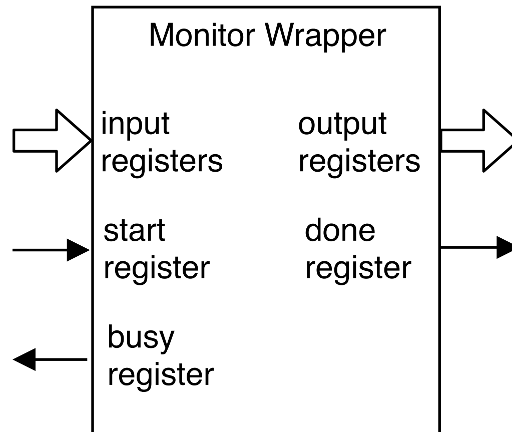

Figure 4.6: Handshake signals for an HLS-generated hardware block

The Intel HLS compiler [24] generates a hardware block for the wrapper function with the interface shown in Figure 4.6. The wrapper hardware module appears to a processor as a bus slave component [25] with the function's arguments, return value, and handshake registers mapped to consecutive memory addresses. Unlike HDL, the clock, reset, and handshake signals are not explicit in the C code. If `busy` is not asserted, arguments can be written to the input registers prior to asserting `start`. Return values can be read from the output registers when `done` is asserted. Figure 4.3 shows the actual Verilog interface that is automatically generated by the HLS process.

Listing 4.3: Auto-generated Verilog interface

```
1  within_cage_internal within_cage_internal_inst (
2      .clock              (clock),               //clock.clk
3      .resetn             (resetn),              //reset.reset_n
4      .done_irq           (done_irq),            //irq.irq
5      .avs_cra_read       (avs_cra_read),        //avs_cra.read
6      .avs_cra_write      (avs_cra_write),       //.write
7      .avs_cra_address    (avs_cra_address),     //.address
8      .avs_cra_writedata  (avs_cra_writedata),   //.writedata
9      .avs_cra_byteenable (avs_cra_byteenable),  //.byteenable
10     .avs_cra_readdata   (avs_cra_readdata)     //.readdata
11   );
```

4.3 Monitor Implementation Analysis

ID	Description	SystemVerilog Assertion	Result
S1	LTL formula (Equation 4.7)	(*nearing_fence* & *in_slow_zone*) $\mid\rightarrow$ (*can_slowdown* until *stopped*)	Failure
S2	Monitor always detects when the UAS is about to leave the cage	(*current_state* != ALARM & !*can_slowdown*) $\mid\rightarrow$ (!*within_cage_max_alt* & *next_state* == ALARM)	Success
S3	Monitor does not trigger a false alarm	(*current_state* != ALARM & (*can_slowdown* ‖ *stopped*)) $\mid\rightarrow$ *within_cage_max_alt*	Success
S4	Monitor remains in alarm state once it enters that state	(*current_state* == ALARM) $\mid\rightarrow$ (!*within_cage_max_alt* & *next_state* == ALARM)	Success
S5	UAS cannot escape the cage if it is not nearing the fence	(*current_state* != ALARM & !*nearing_fence*) $\mid\rightarrow$ *within_cage_max_alt*	Success
S6	UAS cannot escape the cage if it is not in the slow zone	(*current_state* != ALARM & !*in_slow_zone*) $\mid\rightarrow$ *within_cage_max_alt*	Success

Table 4.1: SystemVerilog Assertions applied to the *within_cage_max_alt* monitor's hardware implementation

Synthesized monitors are analyzed using a model checking tool to ensure they satisfy formally captured behaviors. Model checking is facilitated by the explicit state machine structure of the monitor code. While conventional testing can only check for correct behavior during particular executions, model checking can ensure these behaviors across all possible executions. Model checking is performed on the HDL code generated by the HLS rather than the C code because it is closer to the final implementation. Since most HDL model checking tools support Verilog or SystemVerilog, the HLS generated VHDL code is converted to Verilog code using the `vhd2vl` tool [26]. This Verilog code is converted to SystemVerilog code with the addition of SystemVerilog Assertions (SVAs). These conversions were dictated by the

tools available for our specific platform.

The monitor analysis steps are depicted in Figure 4.7. The SystemVerilog code is input to the EBMC tool [23] which can perform both bounded and unbounded model checking. There are two different analyses of interest: (1) verify whether the synthesized HDL code satisfies Equation 4.7; and (2) verify whether the LTL specifications are correctly stated. The second analysis is important since behaviors may be under-specified or over-specified. Table 4.1 describes the assertions both informally and formally, and whether the assertion is satisfied by the *within_cage_max_alt* monitor's hardware implementation.

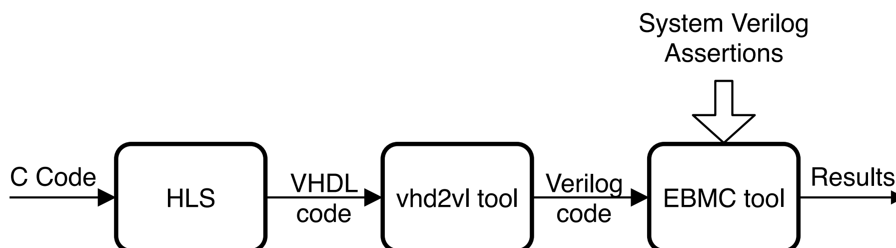

Figure 4.7: Monitor analysis process

The first case is analyzed by translating Equation 4.7 to SVA *S1*. The EBMC tool generates an expected counterexample for this assertion: UAS is in the slowdown zone, heading towards the fence at a speed with which it cannot slow down and stop within the fence. This scenario causes a transition to the monitor's `ALARM` state which triggers a "land" RCF.

The second case is analyzed by submitting different logical queries to the EBMC tool. SVAs *S2*–*S6* are not derived from the LTL formulae and independently capture the expected behaviors of the monitor. For example, *S2* checks whether (1) the monitor always detects if the UAS cannot slow down and stop within the cage, and (2) the monitor enters the `ALARM` state when this situation occurs. Validation of this assertion confirms there is no scenario in which the monitor does not detect if the UAS is about to escape the cage. Monitor trust is enhanced by confirmation of all assertions.

4.4 Pedigreed Component Architecture

The Nios II soft processor architecture [3] is available on Intel FPGAs. Figure 4.8 shows the base resource types available on the MAX 10 FPGA [16, 17]. The phase-locked loop takes the 25 MHz external clock signal as an input and produces a 10 MHz clock for the ADC, a 50 MHz clock for the SPI interface, and a 100 MHz clock for the Nios II processor. A large number of I/O pins available allow for communication with numerous other subsystems, including the GPS [22], ultrasonic [10], and MAVLink [2] interfaces used by the safety monitors. Even though the FPGA functionality may be altered at design time, it cannot be changed at runtime. The programming interface can even be physically disabled, effectively turning the FPGA into a fixed-function ASIC. The soft processor has a simple 32-bit RISC organization with performance options such as a five-stage pipeline.

Figure 4.9 shows the individual components and their connections. The Nios II processor handles the I/O interrupts, invokes the monitors, and initiates the RCF. There are separate UART communication channels for the GPS data, the only sensor information necessary for the demonstration, and the MAVLink protocol between the Atom application processor and the flight controller. The soft processor does not interact with any of the existing components shown at the bottom of Figure 4.9. Since the number of monitors is limited by the FPGA resources available, area-efficient monitors are selected to fit within the Aero's small FPGA. On the other hand, larger FPGA devices exist with two orders of magnitude more logic elements. Instantiating a pipelined processor uses 58% more LUTs and 27% more flipflops than the monitors. Figure 4.10 shows the user interface for creating the pedigreed architecture with the monitor system highlighted in blue. ArduPilot is considered a pedigreed component, as defined in Section 3.1.

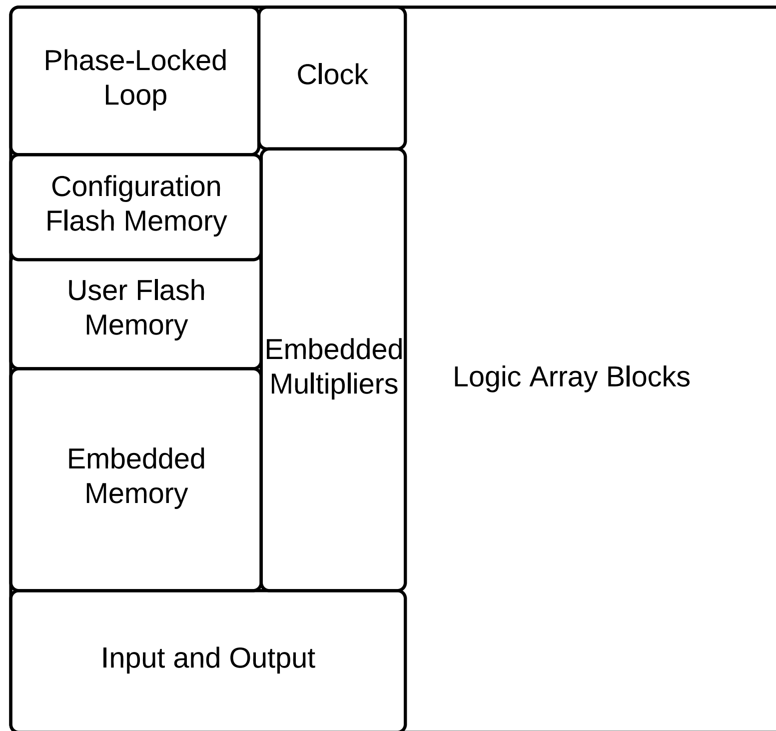

Figure 4.8: MAX 10 FPGA hardware resource types

4.5 Optimizations

Performance-oriented hardware and software optimizations were a large part of the implementation effort. They can be divided into the following sub-categories.

4.5.1 Double Buffering

A double buffer scheme allows concurrent processing of consecutive packets by the `main()` and interrupt tasks [1]. This double buffer scheme is an alternative to FIFOs between the interrupt service routine (ISR) and read/write tasks which require copying between the FIFOs and arrays used to construct or parse a packet. A circular buffer is unsuitable because variable length MAVLink packets could wrap around and hence not occupy contiguous memory.

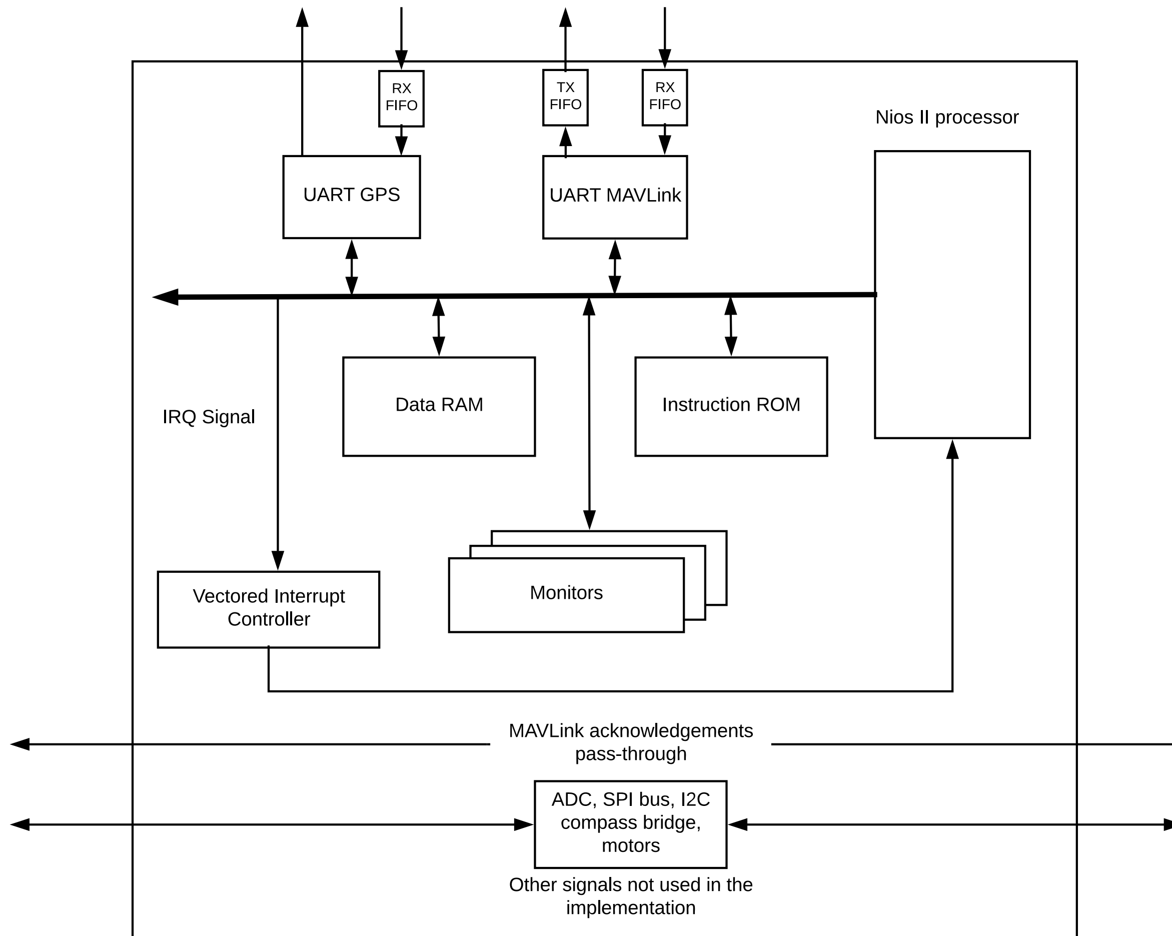

Figure 4.9: I/O soft processor and monitor blocks

Twice as much memory is required compared to a circular buffer, but the memory is present and available on the FPGA.

All packets are either a fixed size or have a length field in the packet header. Once this size has been reached by both the `main()` and interrupt tasks, their buffer pointers are simply swapped to avoid buffer copy and clearing overheads. While a single ring buffer is sufficient for byte transfers, a buffer pair is more appropriate for MAVLink's variable-sized packets subject to transmission errors and flushing. With a double buffer, one buffer is always receiving bytes and one is always transmitting bytes for each UART channel. There are fixed physical locations that always receive and transmit, but the buffers holding the data

Use	Connections	Name	Description	Export	Clock
✓		⊞ **CLK_100**	Clock Source		***exported***
✓		⊞ **UART_2**	FIFOed UART (RS-232 serial port)13.1		**CLK_100**
✓		⊞ **UART_1**	FIFOed UART (RS-232 serial port)13.1		**CLK_100**
✓		⊞ **UART_0**	UART (RS-232 Serial Port) Intel FPGA IP		**CLK_100**
✓		⊞ **pio_0**	PIO (Parallel I/O) Intel FPGA IP		**CLK_100**
✓		⊟ **NIOS2**	Nios II Processor		
		clk	Clock Input	*Double*	**CLK_100**
		reset	Reset Input	*Double*	[clk]
		data_master	Avalon Memory Mapped Master	*Double*	[clk]
		instruction_master	Avalon Memory Mapped Master	*Double*	[clk]
		tightly_coupled_data_master_0	Avalon Memory Mapped Master	*Double*	[clk]
		tightly_coupled_instruction_master_0	Avalon Memory Mapped Master	*Double*	[clk]
		interrupt_controller_in	Avalon Streaming Sink	*Double*	[clk]
		custom_instruction_master	Custom Instruction Master	*Double*	
✓		⊟ **cage_0**	within_cage		
		avs_cra	Avalon Memory Mapped Slave	*Double*	[clock]
		clock	Clock Input	*Double*	**CLK_100**
		irq	Interrupt Sender	*Double*	[clock]
		reset	Reset Input	*Double*	[clock]
✓		⊞ **onchip_memory_instruction**	On-Chip Memory (RAM or ROM) Intel FPGA IP		***multiple***
✓		⊞ **onchip_memory_data**	On-Chip Memory (RAM or ROM) Intel FPGA IP		**CLK_100**
✓		⊞ **vic_0**	Vectored Interrupt Controller		**CLK_100**

Figure 4.10: Quartus's Platform Designer creating the Nios II soft processor and connecting its peripherals

that are in these locations are constantly swapping. This is shown in Figure 4.11. Hence there is always time to process new data without delaying the transmission of processed data. Once a full packet has been reached by both the producer and consumer, the pointers swap and allow the used data to be overwritten in the producer while information currently being used by the `main()` routine can be simultaneously transmitted by the consumer without interference. For each packet produced, one will be consumed, as long as there is new data resulting in the need for a swap.

4.5.2 UART FIFO Buffers

Large hardware FIFO buffers added to the UARTs enable the default 57.6k baud rate to be increased to 921k baud by buffering entire packets in hardware before being processed by software interrupt handlers. These require 265% more LUTS, 246% more flipflops, and

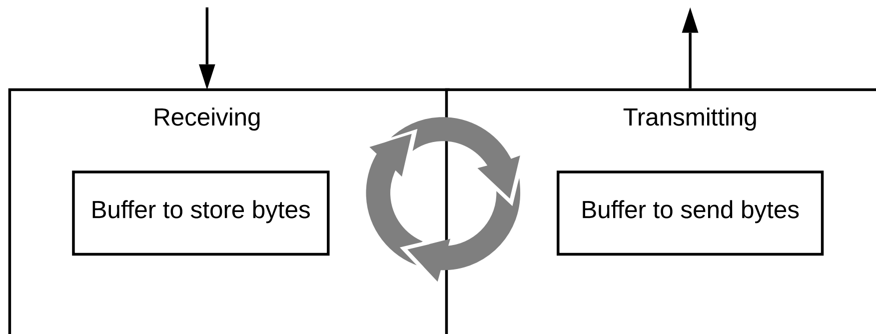

Figure 4.11: The fixed receive and transmit locations with swapping data buffers

8k memory bits compared to standard, memory-less UARTs with hardware buffering for just a few bytes. Figure 4.12 shows these default UARTs. By allowing bytes to queue up as shown in Figure 4.13, much faster baud rates are possible because the software has the full time between packets to process the entire packet instead of the small amount of time between individual bytes to process each byte. With the standard UART, if a second byte is received before the first one is processed, the first is overwritten. This is problematic in most situations since a complete packet needs to be received in the proper order. Furthermore, the FIFO buffers complement having one large ISR by preventing the arrival of another byte at the UART before the ISR has finished. Specialized IP blocks are available for easy incorporation using Platform Designer.

4.5.3 Interrupt Routine Processing

There are two separate tasks that timeshare the soft processor, but there is only space for one processor on the FPGA. `Main()` is the primary task, but it is affected by a heavy interrupt load when MAVLink packets are received and sent. The ISR can be considered a separate thread to `main()` in the sense that it competes for time, instruction calls, and resources. Vectoring all UART interrupts to the same interrupt routine allows all pending

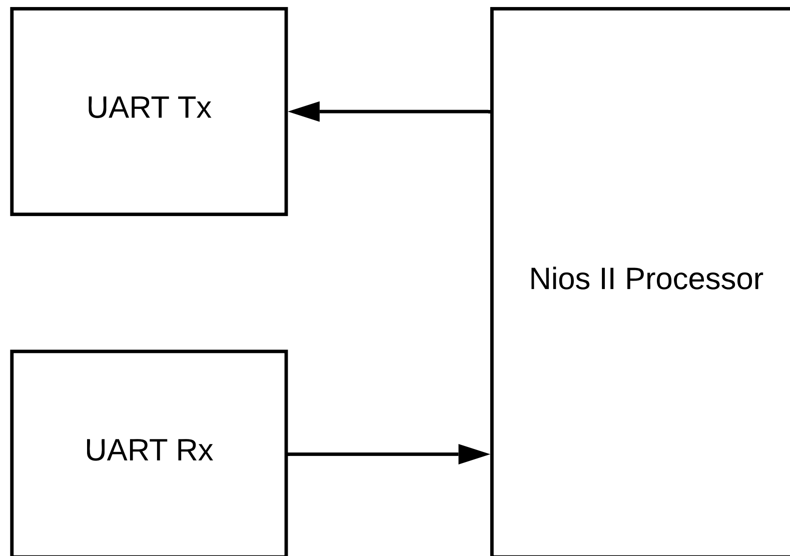

Figure 4.12: The default UART architecture

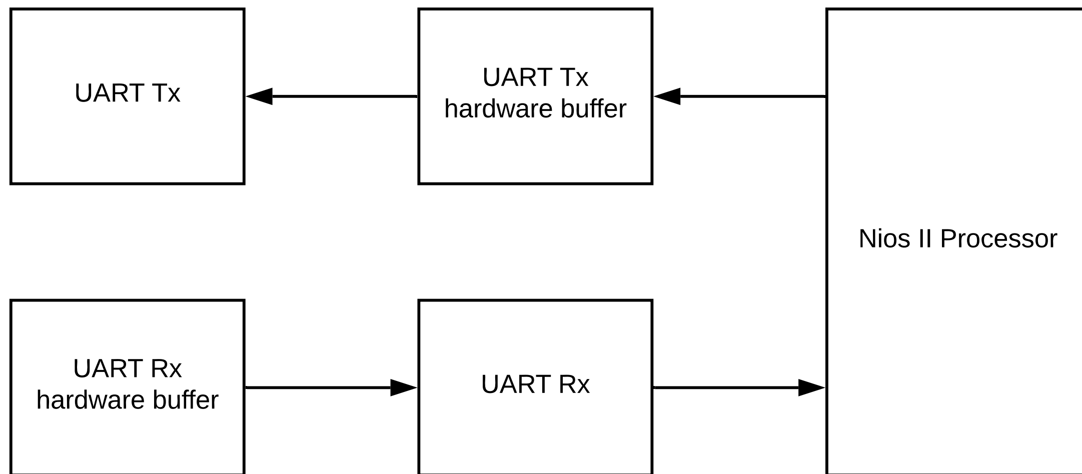

Figure 4.13: The added FIFO components to the UARTs

interrupts to be checked before returning from the interrupt handler. This optimization reduces instruction memory requirements by having fewer independent functions. The separate structure is shown in Figure 4.14 and the one routine structure is shown in Figure 4.15. This dramatically reduces the context switching overhead as well. Instead of each byte of

each communication channel independently interrupting `main()` on every transmit and receive, every pending interrupt can be processed on the same context switch. The best case scenario would be each transmit and receive is ready during the same switch, and the worst case is only one is ready. This would be the best case for each one having their own routine. Combining routines compliments the FIFO buffers as each communication line's buffer can all be processed simultaneously and repeatedly when full.

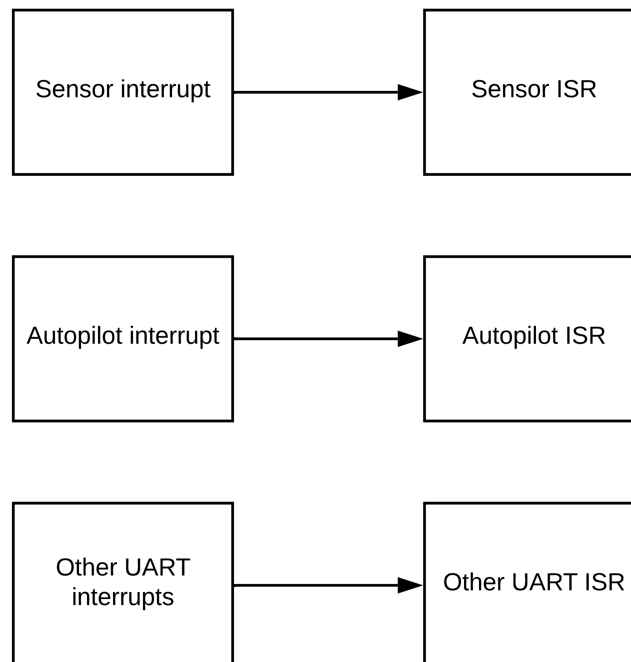

Figure 4.14: The separate ISR design avoided that may be necessary under different circumstances

Interrupt context switching overhead is further reduced by an external vectored interrupt controller (VIC) that causes the processor to switch to a shadow register set [1]. Shadow registers are a separate set of registers that eliminate the timing overheads of context switching by using the unused high address lines in already existing memory. This hardware optimization is an alternative to software overheads of `main()` and the ISR sharing the same registers and takes full advantage of the double buffers. The difference between the executions is

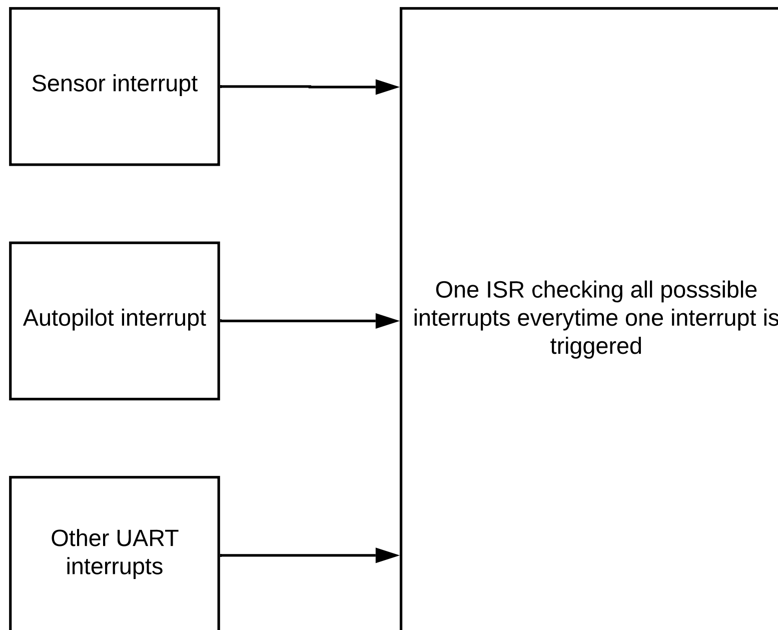

Figure 4.15: The one ISR design used

shown in Figure 4.16 and 4.17. A VIC can provide performance four to five times better than the default internal controller [13]. By using the single ISR design with tightly coupled memory, the VIC performance benefits are not as dramatic, but for future work where this is not possible, it would be more significant. However, there is still an additional 18% time reduction from 10823 ns to 8913 ns for processing interrupts and switching context back to `main()`.

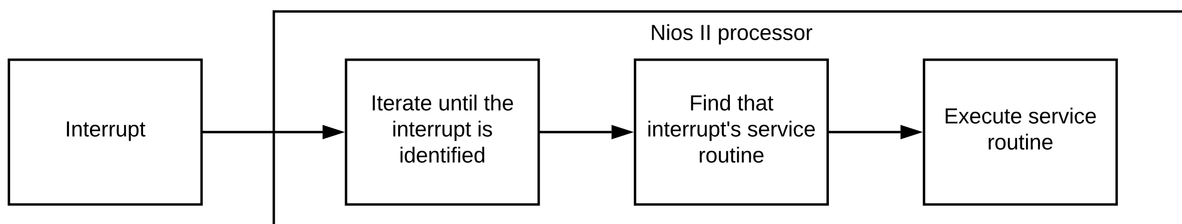

Figure 4.16: The default internal interrupt controller

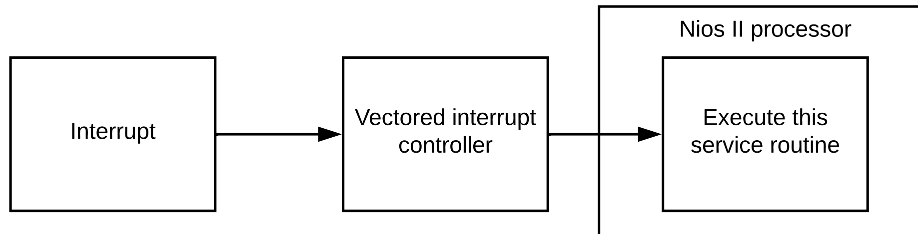

Figure 4.17: The vectored interrupt controller mapping the processor directly to a particular ISR

4.5.4 Tightly Coupled Memory

Tightly coupled data and instruction memories enable concurrent instruction and data access with cache-level performance without the time overheads of caching. Cache misses take time to process. This reduces latency for the large buffers read and written by `main()` and the ISR. By better partitioning and organizing memory, performance is significantly increased without additional logic elements. The time for the ISR to execute, including context switching, is reduced by 66% from 10823 ns to 3635 ns. When combined with the VIC, the time is further reduced to 2896 ns. The reduction is not fully additive because the direct access to the ISR the VIC provides is not as dramatic with the ISR already placed optimally in memory by using TCM. Figure 4.20 summarizes these optimizations. This is preferable to complicated software algorithms to do the memory management.

Figure 4.18 shows the hardware structure with one memory and Figure 4.19 shows the structure with tightly coupled memories. With a single memory, instruction and data access have to compete over the same bus. With independent ports, instruction fetches and data transfers can occur simultaneously. The original data connection is also connected to the instruction memory to support debugging such as setting instruction breakpoints. However, this is precluded by the lack of a JTAG interface. With all the instructions being tightly

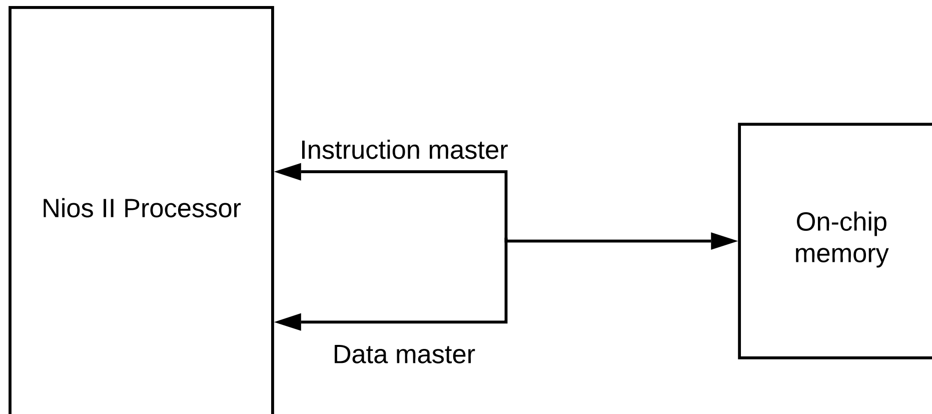

Figure 4.18: The schematic of a single memory

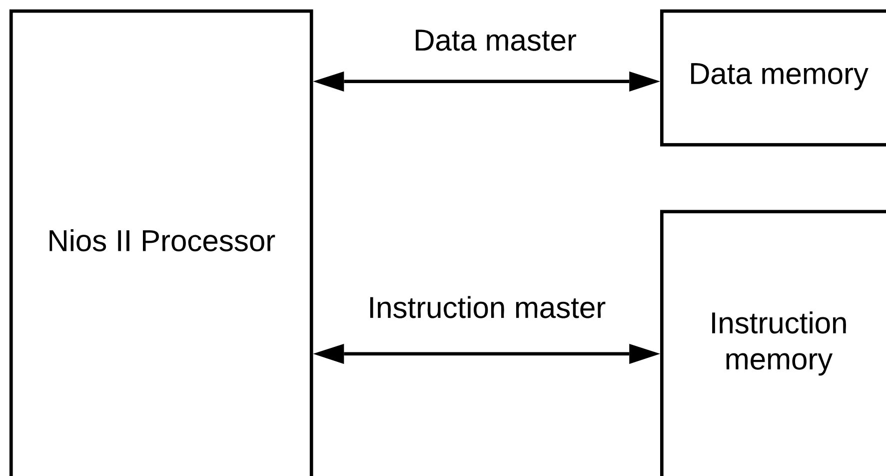

Figure 4.19: The schematic of separate tightly coupled memories

coupled, the original instruction port is left unconnected. The term "tightly coupled" refers to placing everything in an optimal order, similar to how an array in software is efficient for consecutive access. Tightly coupled memories are not typically used on more complex systems because they do not have the dynamic ability to satisfy time-varying memory usage, which is better handled by caches. It works well on a small real-time system because the same simple code is running on every iteration. Before downloading the software into the `.jam` file, the separate memories need to be allocated in the linker with the appropriate instructions

or data. There can be a different data TCM for each critical task, which increases overall memory usage. With only small amounts of data necessary and one large ISR, just one data TCM was used in this approach.

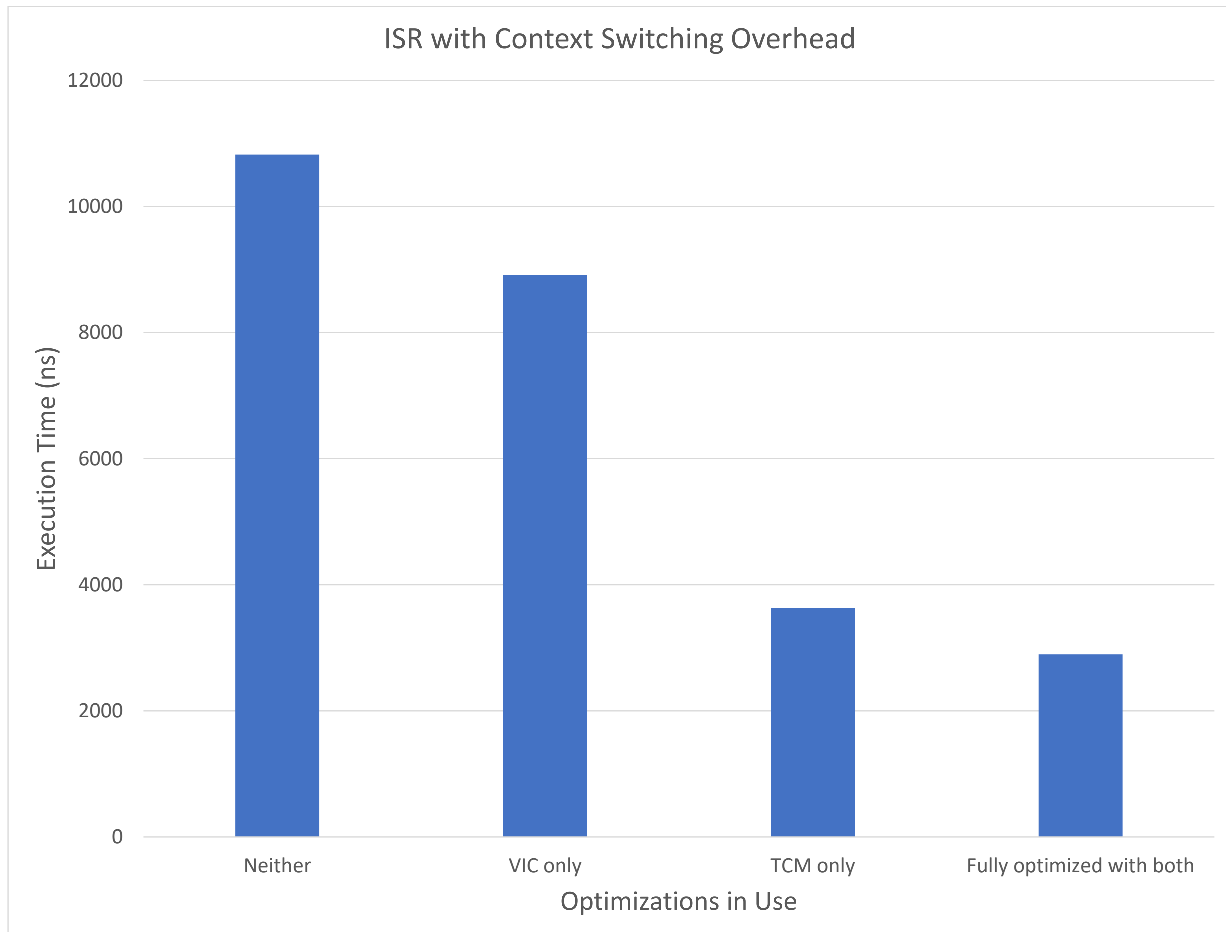

Figure 4.20: Comparison of the ISR and context switching overhead optimizations

4.6 Monitor Block Integration

Before using GPS, Marvelmind ultrasound sensors were used as the position location system for indoor applications [10]. The receiver was connected to unused pins on the Aero. The

transmitters are then mounted around the flight environment to actively track the position of the drone. They are also a practical alternative to much more expensive camera-based indoor position tracking systems. A highlight of the design is that regardless of the position sensing used, the software remains the same.

Algorithm 2 summarizes the Nios II processor's cyclic executive code structure. Initialization

Algorithm 2: main() task

```
1  initialize communication
2  make recovery commands
3  verify sensor data
4  while forever do
5      if new sensor data then
6          load sensor data into the monitors
7          start monitors
8          wait for monitors to finish
9          if any monitor indicates a violation then
10             stop the producer from receiving new data
11             overwrite the producer with the RCF
12             if full packet received and transmitted then
13                 if buffers are ready to swap then
14                     swap buffers
15                 end
16             end
17         end
18     end
19 end
```

includes enabling UART communication channels, connecting them to an interrupt service routine, and ensuring that a GPS lock has been obtained in accordance with F3269-17 requirement (4) discussed in Section 3.1. The Nios II code is part of a pedigreed component since it initiates RCFs in response to monitor anomaly detection. RCF precomputations help to ensure compliance with requirements (10) and (11) by minimizing latency and having it ready to go when needed. The main loop verifies the integrity of the data prior to sending it to the monitors, which satisfies requirements (6) and (12). Relevant sensor data are written

to the monitor wrapper registers shown in Figure 4.21, followed by a write to the `start` register. The location and speed value for each register is sent individually in no particular order. All data registers need to be updated before the monitors begin processing so an accurate trajectory model can be computed. These registers can only be accessed by the Nios II processor and are therefore isolated from non-pedigreed components, in accordance with requirement (2).

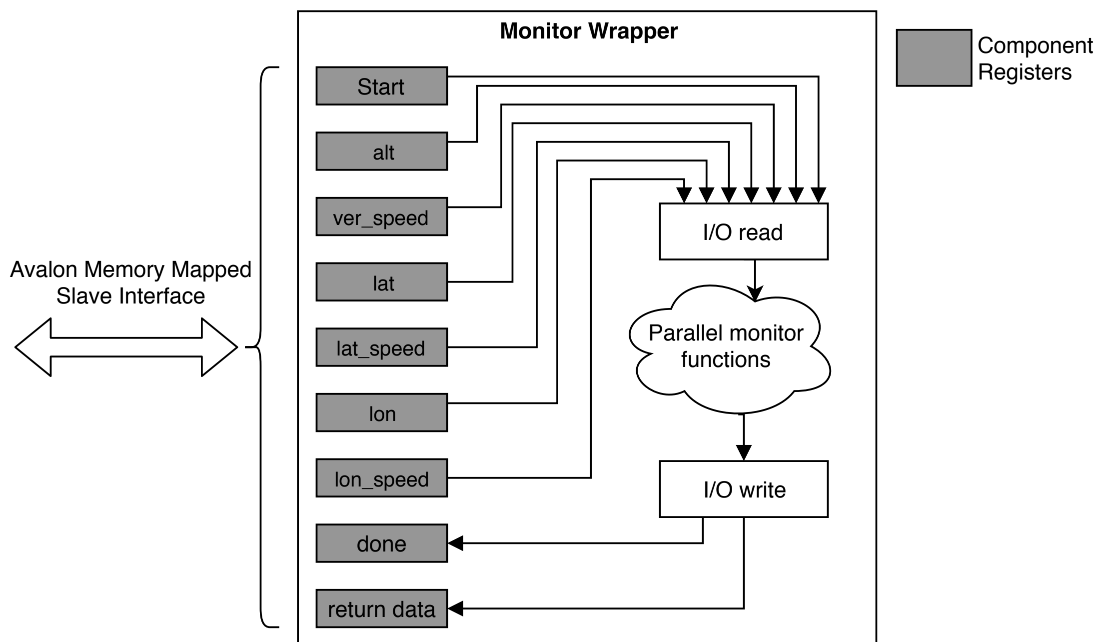

Figure 4.21: Monitor wrapper registers

As shown in Figure 4.22, the soft processor treats the monitor wrapper as an Avalon bus peripheral with a conventional software/hardware interface consisting of memory mapped data, control, and status registers. Avalon is the name of the component interfacing and interconnecting system Intel uses for their FPGA designs [4]. The wrapper signals completion via a `done` bit in a status register. Next, the overall monitor outcome may be read from a data register. Using a hardware wrapper for monitors utilizing the same sensor data and RCF achieves the lowest latency by permitting a hardware-implemented logical AND reduction to the five monitor outputs that are computed at approximately the same time. If the system

is within the predefined limits as determined by the monitors analyzing the current sensor data, the `main()` loop repeats the overall cycle with new sensor data. If the system is not within the predefined limits, the appropriate RCF is selected and triggered in a manner that satisfies F3269-17 requirements (1), (8) and (9).

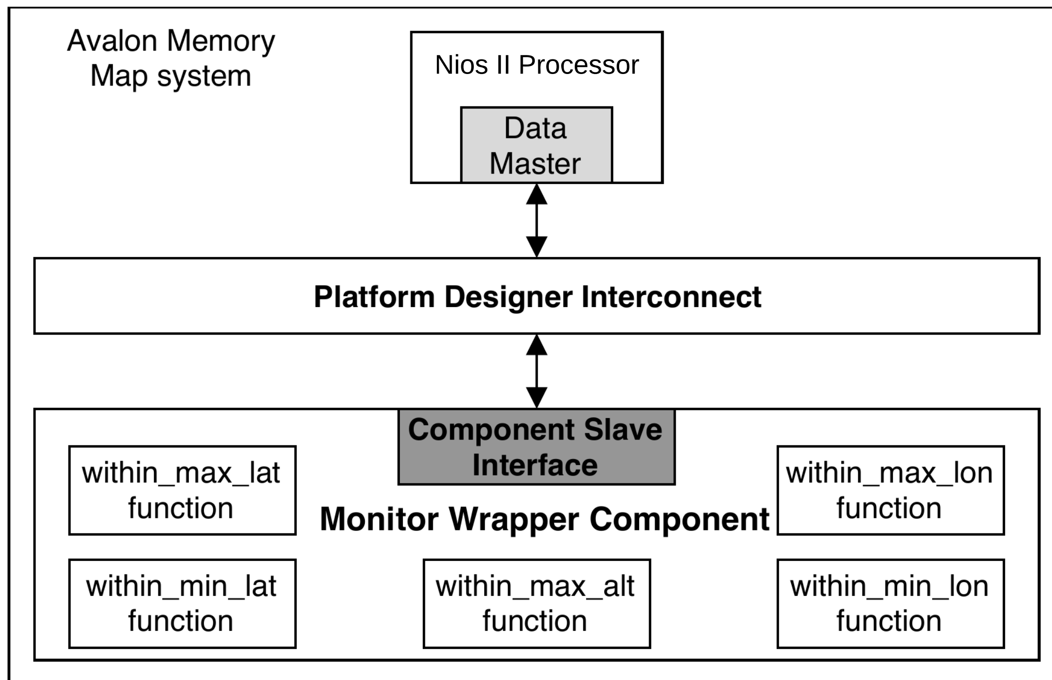

Figure 4.22: Processor interface to the monitor wrapper

Processing sensor and complex function data is a critical role for the Nios II processor, with latency of particular concern. Optimizations detailed in Section 4.5 focused on minimizing interrupt latency and maximizing UART baud rates. This reduces timing disturbances on communication between the complex functions and flight controller.

Algorithm 3 shows the interrupt task actions. With one ISR, any individual interrupt can start the execution for this routine which will service all outstanding interrupts sequentially. As mentioned in Section 4.5.1, a double buffer scheme is used with transmit and receive buffers for each UART channel. A counter keeps track of how many bytes have been received

Algorithm 3: Interrupt handler task

```
1  if any interrupt triggered then
2      for each communication channel do
3          if pending data to receive then
4              add byte to receive buffer
5          end
6          if pending data to transmit then
7              remove byte from transmit buffer
8          end
9          if full packet received and transmitted then
10             if buffers are ready to swap then
11                 swap buffers
12             end
13         end
14     end
15 end
```

for the current packet. The packets can be of variable sizes, but the position of the byte containing the size of the packet is fixed. When the buffer index reaches this location, the counter knows how many bytes to accept before being ready to swap. Similarly, when transmitting, the size is read first so the routine knows how many bytes to send before being ready to swap. When both are ready, the swap happens. Since the transmit interrupt is triggered when it is ready to send another byte, it is temporarily disabled if it finishes before the receive buffer to keep it from running unnecessarily. It is also temporarily disabled during a swap to prevent partial transmissions. Swapping resets the size of the receive buffer and both counters to the beginning. The routine is modular with respect to each UART and can scale to an arbitrary number.

4.7 Recovery Control Function Interjection

With the proper pedigreed integration, the RCF switch is capable of being activated. Afterward, a seamless switch returns to the double buffer scheme. Figure 4.23 shows an overview

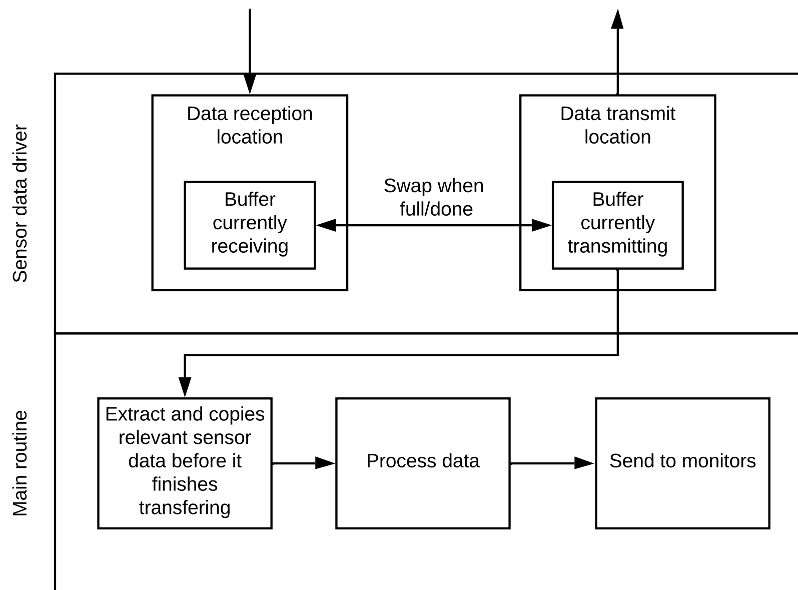

Figure 4.23: Overview of main and the ISR interacting with the double buffer communication scheme.

of the double buffer scheme in action. Both tasks must be finished receiving or transmitting their respective packets for the swap to occur. When an interjection is triggered, data reception is stopped and replaced by the RCF. To curtail commands triggering the override and fulfill requirement (7), the RCF overwrites the receive buffer instead of the currently transmitting buffer to still allow for a continuous flow of commands to the flight controller. Loading a complete RCF packet into the receive buffer makes it ready to swap with the transmit buffer. If the transmit buffer is overwritten instead, a partially completed packet would be sent leaving the system in an undefined state while also wasting time receiving data that is overwritten. By checking to swap after loading the receive buffer with the RCF,

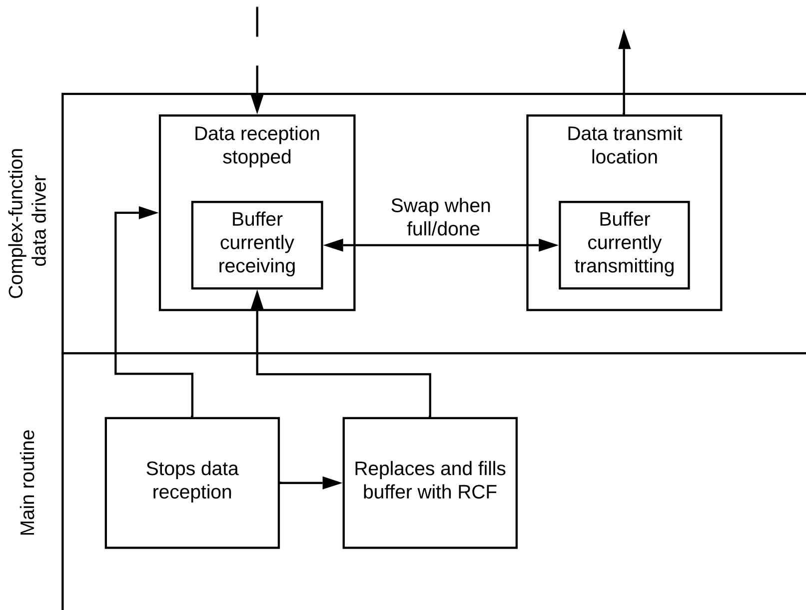

Figure 4.24: High-level view of the RCF being inserted into the MAVLink data stream

the last packet transmission is allowed to complete. Figure 4.24 provides a high-level view of the RCF being inserted.

Chapter 5

Evaluation

5.1 Results

Several flight scenarios are tested, with the following three scenarios highlighted. Certain conditions are consistent across all scenarios:

1. To prevent the drone from leaving the cage between sensor packets, and to provide enough time to land before leaving the cage, a trigger zone is defined within the cage borders.

2. If the monitors determine that the drone's current speed could result in a cage breach before the next GPS sensor packet, a land is triggered.

3. Expensive operations such as division are avoided by calculating speed in terms of the difference between consecutive location updates.

4. The trigger zone is sufficiently wide so that the drone may not breach the cage before the next GPS packet even at its maximum horizontal speed.

5. When the speed defined above is greater than the distance remaining to the edge of the cage, the drone is forced to land.

6. A faster drone may need a greater sensor sampling frequency or a larger trigger zone.

5.1.1 Indoor Flight

The Marvelmind ultrasound system allows continuous indoor tracking with two centimeter resolution. At this precision, the size of the drone must be considered beyond the centralized sensor. The sensors must be carefully aligned and oriented to properly communicate indoor position information. When the receiver is mounted, the outdoor GPS sensor is disabled to prevent system confusion. The ultrasound sensor's consistency and precision allowed for a constrained testing environment. The techniques in every implementation started from the refinements made with indoor testing for its controlled and easily reproducible environment. The RCF is successfully triggered to protect the UAS from making contact with the physical walls of the testing location on each axis. Since the hard walls are unforgiving, the boundary of the cage needs to be far enough away to prevent inertial drift collisions. Altitude is set relative to the floor of the test site, and the sensors are generally placed evenly around the site as depicted in Figure 5.1.

5.1.2 Unbounded Autonomous Flight

Switching to GPS brought its own complications. During boot-up, the GPS produces several blank packets that can potentially destabilize the system from the very beginning. To compensate for this, and to satisfy requirement (4), an initial warm-up period is enforced to let the GPS module stabilize before use. The monitors invoke a "land" RCF when attempting to autonomously fly the UAS outside a 100×100 foot lateral by 50 foot vertical virtual cage defined at a local outdoor test facility. Altitude is set as additional elevation above sea level from take-off. After placing the drone at the center of the cage, it is necessary to wait for a GPS lock although this is normally done before any flight. An additional monitor could be defined to prevent takeoff before the GPS lock is achieved.

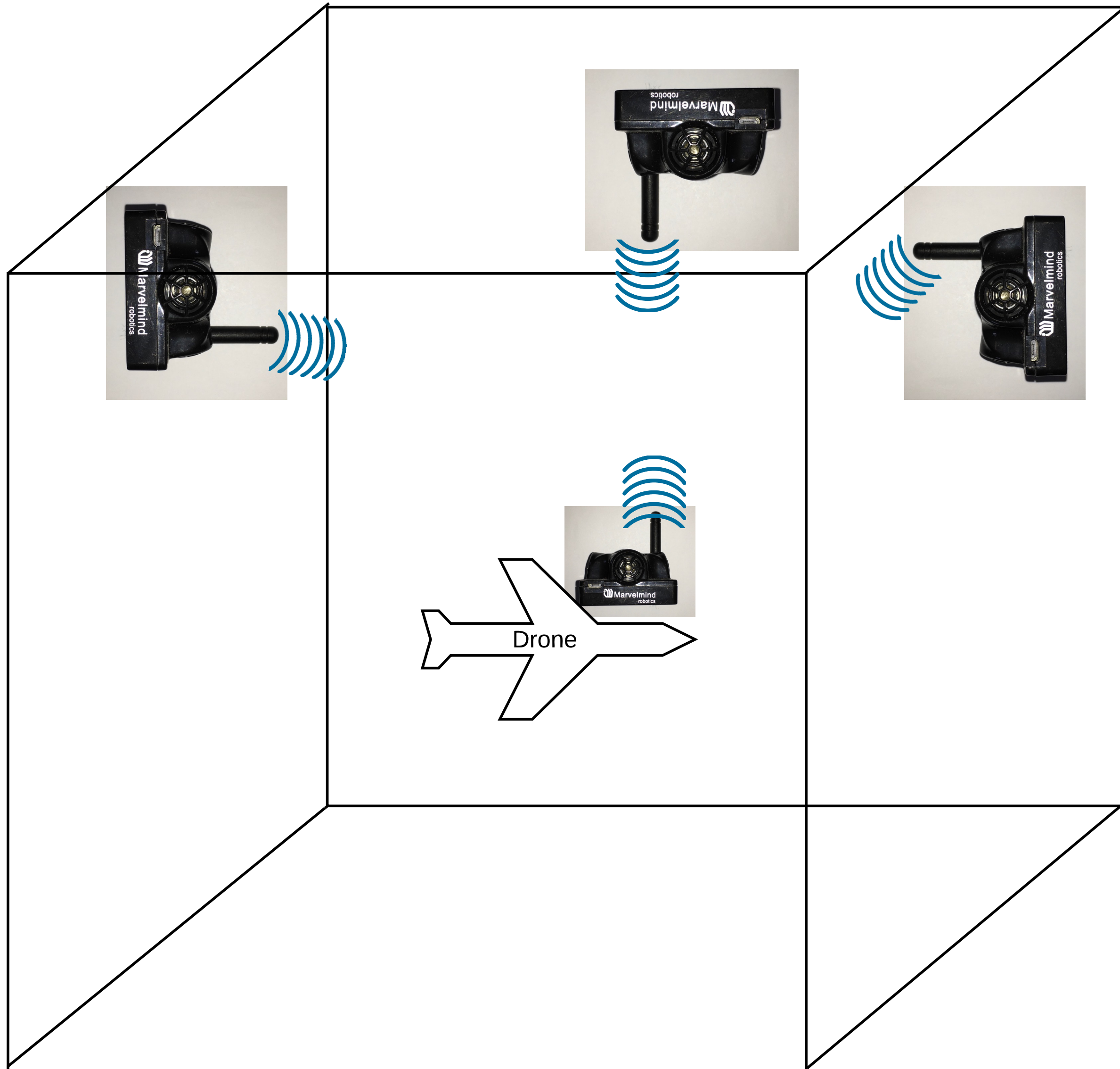

Figure 5.1: A rendition of an indoor flight test with the MarvelMind ultrasound sensors

GPS data accuracy and frequency limit the precision of the cage boundaries. Figure 5.2 shows a flight track from an autonomous flight plan with the RCF initiated at the cage boundary shown by the red line. The UAS first climbs to 35 feet, remaining under the altitude limit of 50 feet. It then proceeds to waypoint 2 that is 31.3 feet from the takeoff position. The drone tries to reach waypoint 3 (70 feet from the start) but is forced to land

near the 50 foot fence. Inertia and drone flight mechanics may occasionally result in the drone landing slightly outside the cage. Glitches in GPS data or loss of GPS quality may also trigger the RCF regardless of position in accordance with requirements (6), (7), and (12).

Figure 5.2: Triggering the land RCF

5.1.3 Bounded Manual Flight

Virginia Tech's Drone Park is roughly the size of a football field with an 85 foot ceiling. The Park actively protects the nearby airport and outside world from a UAS but can damage a UAS if it were to collide with the netting. A virtual cage is successfully implemented to protect drones from making contact with the physical netted barrier. Figure 5.3 shows this happening. RCF's are successfully triggered when a manual pilot attempts to exit the cage or approach too fast towards the netting. It was this test where the geometric transformations described in Section 4.1.2 were essential. Ultrasound sensors could have been mounted on the netting supports and used to navigate the drone, but GPS was selected with all the considerations described previously.

Figure 5.3: A test succeeding in the Drone Park

5.2 Evaluation

Table 5.1 shows the FPGA resources available and those used to implement preexisting interfaces and the RTA consisting of the Nios II soft processor and monitor hardware blocks. The processor and monitors use the same 100 MHz clock. No additional I/O pins are required by the RTA. The Nios II processor consumed roughly 33% of the logic elements, the monitors consumed 23%, and the UARTs consumed 18%.

Packet-sized hardware FIFOs and the other optimizations described in Section 4.4 allow 921k baud over the MAVLink channel, the standard baud rate used by the PX4 flight controller, and an unexpected result for a 100 MHz soft processor. Prior to the optimizations, communication was limited to 57.6k baud, which is supported by ArduPilot. The interrupt

Resource	Available	Used	Usage
Lookup table	8064	5799	72%
Flipflop	8064	3436	43%
Embedded SRAM (Kb)	49	38	78%
User flash memory (Kb)	1376	0	0%
General purpose I/O pin	112	34	30%
Embedded multiplier	48	6	13%
Clock phased-locked loop	1	1	100%

Table 5.1: Resource utilization for the MAX 10M08 FPGA

service routine originally consumed 3% of the processor cycles, which is decreased to 1.5% after the optimizations.

Table 5.2 details the average execution time required by various software and hardware modules. A different design was considered initially. The first idea for a set of monitors included just one for each axis. However, it was quickly realized that these required much more logic to discern which direction the UAS was moving. Consequently, they would also be slower to run. So for the actual implementations, different ways of executing the monitors are explored. The last four rows compare the different means of implementing the monitors. The sequential, overlapped, and parallel schemes differ primarily in the software/hardware interface, with sequential corresponding to a blocking interface to each monitor, overlapped corresponding to a non-blocking protocol, and parallel using hardware control circuitry to enable true concurrent monitor execution.

Wrapping the five hardware monitors in a hardware block resulted in the fastest execution. It may seem counterintuitive that the software execution outperformed the sequential hardware execution, especially after discussing how speed is an advantage of the hardware monitors. However, this is a circumstance specific to the demonstration which is kept computationally simple. Software/hardware interfacing overheads reduce the speedup arising from hardware implementation. The sequential hardware execution did worse than the soft-

ware execution because it has numerous read and write bus accesses compared to just a few simple conditional statements in the software execution.

Figure 5.4 compares the different execution sequences. Although the software also has its own overheads with executing functions and making a decision, it is clear that the communication with the hardware monitors is responsible for making the non-parallel schemes slower than the software for simple logic cases with a just a few monitors. An important observation is that despite these overheads, the parallel execution still outperformed the software. Furthermore, the parallel timing would be constant or grow very slowly in accordance with the worst-case monitor under the same sensor set. As a best-case, the software execution would grow at a constant rate as a function of the number of monitors.

Module	Implementation	Time (ns)
Initialize communication	software	6640
RCF precomputations	software	17220
Parse sensor data	software	70883
RCF interjection	software	6096
Double buffer swap	software	571
Interrupt service routine	software	2064
Timing measurement overhead	software	20
5 sequential monitor functions	software	898
5 sequential monitors	hardware	1512
5 overlapped monitors	hardware	1111
5 parallel monitors + wrapper	hardware	779

Table 5.2: Average execution time of software and hardware modules

5.3 Limitations

Several limitations exist with the current design:

1. The system is entirely dependent on the sensors available. While quality checks and

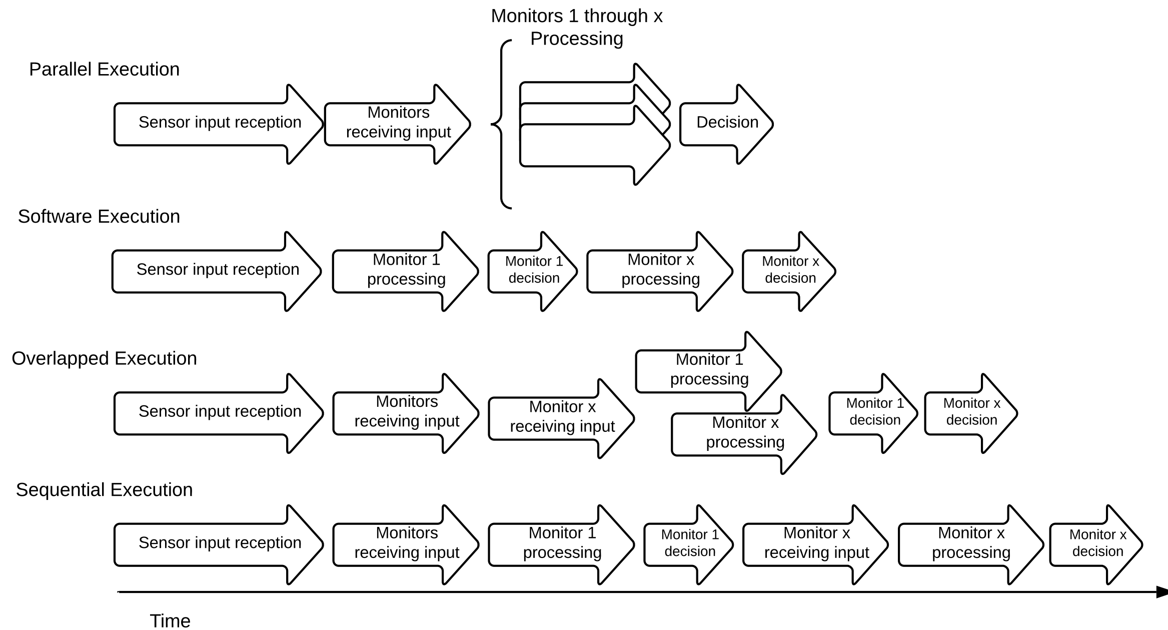

Figure 5.4: Event sequence comparisons

assurances are performed on the data, a compromised sensor could activate the RCF. Furthermore, conditions can only be monitored that have sensors available. A monitor based on pressure couldn't be enforced without a reliable barometer or other appropriate sensor, for example.

2. A large number of simple monitors provides more opportunities for parallelism than a small number of more complex monitors. To be able to take advantage of parallel hardware execution, a set of monitors must be dependent upon the same sensor data. For example, all of the monitors used in the virtual cage could execute simultaneously

because they use information from the same packet. A monitor dependent on battery life would have to execute independently on a cycle relative to when the battery information is updated.

3. Monitors executing simultaneously would still need to have their necessary information extracted from their sensor(s) packets sequentially. To prevent conflict with monitors on different and independent sensor cycles, separate processors can be used to process only their relevant types. An important distinction is that they will be fully independent and not processing the same information, therefore not requiring multiprocessor programming and remaining outside of the complex function classification. Any one of them could trigger their appropriate RCF without affecting the other.

4. Sensor data must be treated discreetly as a function of time under LTL formalization, but future states can be extrapolated and sensor frequencies can be increased to be as near to continuous as necessary.

5. An FPGA would have to be targeted with tools that support HLS, and with enough resources available to support all monitors desired. The autonomous system targeted would then also need to have the necessary communication channels accessible for successful integration of the monitoring system.

6. Additionally, the current design requires a new flash of the FPGA whenever a change to the monitor's parameters is required. This conveniently prevents tampering, but is not very practical for most autonomous systems.

7. Processing GPS data with the original ultrasound sensor FPGA configuration imposed its own limitations. The sensor processing software needs to be manually modified to handle the new data format. The sensor itself needs to be configured to output the desired information only.

8. Under the current data rates with the above changes, the FPGA failed to meet timing constraints. This would occasionally cause unpredictable behavior from the soft processor. Reducing the clock rate for the soft processor from 150 MHz to 100 MHz alleviated the problem. Using a faster speed grade FPGA would permit faster clock rates.

Chapter 6

Conclusions

Impressive behavioral algorithms continue to be developed for an ever-growing level of autonomy, but with increased behaviors comes increased complexity. Increasing integration into everyday society increases the potential for interactions to end poorly, and the anxiety from something going wrong could easily outweigh the benefits autonomy could bring. Consider something simple, like following road signs and lane markings. Society barely trusts a teenage human to do this, and establishing a relationship with computers to do this and more complex activities safely will take time. Moving forward, strengthening trust in these systems is crucial. The difficulty stems from the complexity required of these autonomous systems, challenging both conventional testing methodology and formal methods. It is difficult to envision the formal analysis of a modern UAS's entire software stack including device drivers, operating system, middleware, and application code. Where a human pilot requires years of training and hundreds of hours of practice, it would take much longer to verify to correctness of fully autonomous autopilot code with traditional means. ArduPilot alone accounts for around 700,000 lines of C++ code [18]. A suitable runtime verification method appears to be the only feasible solution. Concurrency dramatically increases the verification effort since the underlying abstraction can no longer be a simple sequential state machine, and frequent code updates may invalidate previous analysis. Following the guidance in ASTM F3269-17, changes to the system are strictly additive with runtime monitoring of the physical system state. The resultant RTA fulfills all requirements given in the standard.

How the monitors are implemented is critical to achieving the necessary isolation and low latency. Concerns about latency and monitor integrity are simultaneously resolved by implementing the monitors, RCFs, and switching logic in an isolated FPGA. Configurable hardware is conventionally used to reduce latencies and power. It is also used as a root of trust that removes all dependencies on application software correctness, reliability, and trustworthiness. An FPGA-implemented soft processor suits the RCFs and switching logic, while direct hardware implementation provides an efficient means of rendering parallel automata. Correctness, monitor parallelism, and productivity goals are met with a synthesis flow that translates from LTL to automata to C code to hardware. The use of HLS avoids the need to use HDLs, which are further down in the abstraction hierarchy. However, model checking may still be applied to the HDL implementation of the automata, allowing the use of mature hardware model checking tools and increasing confidence in the translation process. Analyzing the HDL is analogous to formally analyzing the assembly code output from a compiler.

Although this approach is applied to a quadcopter drone, it could easily apply to cars, planes, trains, or any system that wants to guarantee requirements independent of the software controlling the system or possible network interference. The end vision for this line of research is a world where the autonomous systems that will inevitably be involved in more facets of our lives than we can imagine, will also be trusted to such an extent that their integration will be more seamless than human operators. There are some things humans are inherently good at, but for other things our tools simply do them better, and autonomous systems will be a subset of those tools. It takes a dedicated human pilot a staggering amount of hours of training and practice flights to be trusted behind the yoke, and unlike mathematics or writing, their skills do not directly transfer to the next generation. Much like an athlete, it takes a lot of time and practice beyond just the theory of aerodynamics to

be a pilot. However, unlike an athlete, society needs trust and guarantees of our operators instead of excitement. Having trust be an inherent component of our autonomous systems is necessary to move this technology forward.

6.1 Future Work

The Intel Aero is an appropriate choice for a fast prototype demonstrating the successful implementation of virtual cage monitors, but the MAX 10 FPGA's resources are completely exhausted by them. Expanding to other platforms, like a Zynq UltraScale+ or Microsemi SmartFusion2, will allow for more interesting and practical systems of monitors to enforce additional behaviors. FPGAs with JTAG ports will allow hardware-in-the-loop simulation to safely experiment with the more sophisticated monitoring systems. The more advanced systems incorporate the ability to reconfigure the monitors without having to recompile and flash the FPGA design. Designs with over 50 monitors and several specialized soft processors are anticipated. Other enforceable conditions that will be explored include envelope protection, land conditions, takeoff conditions, communicating failure actions, power levels, military-focused applications, and anything that can be formally specified with LTL.

Although the virtual cage is just used as an example, it can help to address a recurrent UAS issue. Further work with those monitors includes allowing dynamic boundary changes as appropriate, deciding between legitimate and malicious changes, and permitting complex shapes and cages within cages to accommodate real-world flight plans. This would address the current time consuming FPGA re-flash limitation and could permit 4D cages by imposing 3D cages as a function of time as a UAS completes a flight plan.

Tools are in development by NASA for the auto-generation of correct LTL formulas from everyday English and could be incorporated into the tool chain to further increase the level

of automation and reduce the possibilities for human error. There is also potential for integration with an unmanned aircraft air traffic control system. NASA is developing an Unmanned Aircraft System Traffic Management (UTM) system [33]. To coordinate and have cooperative interaction with numerous UAS, UTM would require a continuous stream of communication, navigation, and surveillance. Formally defined monitors can serve a role in providing safety and separation assurances. Monitors can also be applied to other autonomous systems such as robots and cars.

Bibliography

[1] Improving Nios II ISR Performance. URL https://www.intel.com/content/www/us/en/programmable/documentation/mwh1416947091611/mwh1416946760505/mwh1416946900343.html.

[2] Introduction · MAVLink Developer Guide. URL https://mavlink.io/en/.

[3] Nios II Processor Reference Guide. URL https://www.intel.com/content/www/us/en/programmable/documentation/iga1420498949526.html.

[4] Avalon Interface Specifications Updated for Intel Quartus Prime Design Suite: 18.1. Technical report. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl{_}avalon{_}spec.pdf.

[5] MyBoeingFleet. URL https://www.myboeingfleet.com/ReverseProxy/Authentication.html.

[6] Project Wing partners with Virginia Tech to test delivery by unmanned aircraft | Virginia Tech Daily | Virginia Tech. URL https://vtnews.vt.edu/articles/2016/09/ictas-maapprojectwing.html.

[7] Intel Quartus Prime Pro Edition User Guide: Design Compilation. URL https://www.intel.com/content/www/us/en/programmable/documentation/zpr1513988353912.html.

[8] Are Delivery Drones Commercially Viable? Iceland Is About to Find Out - IEEE Spectrum. URL https://spectrum.ieee.org/robotics/drones/are-delivery-drones-commercially-viable-iceland-is-about-to-find-out.

[9] About MAAP | Mid-Atlantic Aviation Partnership | Virginia Tech. URL https://maap.ictas.vt.edu/About/about-us.html.

[10] Marvelmind Indoor Navigation System Operating manual v2019_02_05. Technical report. URL www.marvelmind.com.

[11] Mission Planner home. URL http://ardupilot.org/planner/.

[12] Newark Airport Traffic Is Briefly Halted After Drone Is Spotted - The New York Times. URL https://www.nytimes.com/2019/01/22/nyregion/drones-newark-airport-ground-stop.html.

[13] Embedded Peripherals IP User Guide. URL https://www.intel.com/content/www/us/en/programmable/documentation/sfo1400787952932.html{#}iga1401399659862.

[14] Intel Quartus Prime Software User Guides. URL https://www.intel.com/content/www/us/en/programmable/products/design-software/fpga-design/quartus-prime/user-guides.html.

[15] Illicit drone flights surge along U.S.-Mexico border as smugglers hunt for soft spots - The Washington Post. URL https://www.washingtonpost.com/world/national-security/illicit-drone-flights-surge-along-us-mexico-border-as-smugglers-hunt-for-soft-spots/2018/06/24/ea353d2a-70aa-11e8-bd50-b80389a4e569{_}story.html?noredirect=on{&}utm{_}term=.1a1e5f0b72d2.

[16] MAX 10 FPGA Device Architecture. Technical report, Intel, 2017. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/max-10/m10_architecture.pdf.

[17] Intel MAX 10 FPGA Device Overview, 2017. URL https://www.intel.com/content/www/us/en/programmable/documentation/myt1396938463674.html#myt1396939274982.

[18] ArduPilot Open Source Autopilot, 2018. URL http://ardupilot.org/.

[19] Open Source for Drones - PX4 Open Source Autopilot, 2018. URL https://px4.io/.

[20] QGC - QGroundControl - Drone Control, 2018. URL http://qgroundcontrol.com/.

[21] Runtime Monitoring for Safety of Intelligent Vehicles. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, jun 2018. ISBN 978-1-5386-4114-9. doi: 10.1109/DAC.2018.8465912. URL https://ieeexplore.ieee.org/document/8465912/.

[22] u-blox 8 / u-blox M8 Receiver Description Including Protocol Specification. Technical report, 2018. URL www.u-blox.com.

[23] Enhanced Bounded Model Checker - A model checker for hardware designs, 2019. URL http://www.cprover.org/ebmc/.

[24] Intel High Level Synthesis Compiler, 2019. URL https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[25] Avalon Interface Specifications, 2019. URL https://www.intel.com/content/www/us/en/programmable/documentation/nik1412467993397.html.

[26] Translate synthesizable VHDL into Verilog 2001, 2019. URL http://doolittle.icarus.com/~larry/vhd2vl/.

[27] Home | seL4, 2019. URL https://sel4.systems/.

[28] Juan-Pablo Afman, Laurent Ciarletta, Eric Feron, John Franklin, Thomas Gurriet, and Eric N. Johnson. Towards a new paradigm of UAV safety. *CoRR*, abs/1803.09026, 2018. URL http://arxiv.org/abs/1803.09026.

[29] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[30] ASTM International. Standard practice for methods to safely bound flight behavior of unmanned aircraft systems containing complex functions. pages 1–9, Sept 2018. doi: 10.1520/F3269-17.Copyright.

[31] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. *Introduction to Runtime Verification*, pages 1–33. Springer International Publishing, Cham, 2018. ISBN 978-3-319-75632-5. doi: 10.1007/978-3-319-75632-5_1. URL https://doi.org/10.1007/978-3-319-75632-5_1.

[32] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011. ISSN 1049-331X. doi: 10.1145/2000799.2000800. URL http://doi.acm.org/10.1145/2000799.2000800.

[33] Angela Boyle and Joseph Rios. NASA UTM, 2019. URL https://utm.arc.nasa.gov/index.shtml.

[34] J. Richard Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *The Collected Works of J. Richard Büchi*, pages 425–435. Springer New York, New York, NY, 1990. doi: 10.1007/978-1-4613-8928-6_23. URL http://link.springer.com/10.1007/978-1-4613-8928-6{_}23.

[35] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A survey of

runtime monitoring instrumentation techniques. *Electronic Proceedings in Theoretical Computer Science*, 254:15–28, 2017. doi: 10.4204/eptcs.254.2.

[36] Alexandre Donzé. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 167–170, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14295-6.

[37] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, October 2016. doi: 10.1007/978-3-319-46520-3_8.

[38] Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. *Runtime Failure Prevention and Reaction*, pages 103–134. Springer International Publishing, Cham, 2018. ISBN 978-3-319-75632-5. doi: 10.1007/978-3-319-75632-5_4. URL https://doi.org/10.1007/978-3-319-75632-5_4.

[39] Mazen Farhood, Craig Woolsey, Devaprakash Muniraj, Micah Fry, and Dany Jaoude. Autonomy Test & Evaluation, Verification & Validation (ATEVV): An Exploratory Case Study of Motion Planning and Control. In *Center for Unmanned Aircraft Systems*, College Station, 2019.

[40] Russell V. Gilabert, Evan T. Dill, Kelly J. Hayhurst, and Steven D. Young. SAFEGUARD: Progress and test results for a reliable independent on-board safety net for UAS. In *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, pages 1–9. IEEE, sep 2017. ISBN 978-1-5386-0365-9. doi: 10.1109/DASC.2017.8102087. URL http://ieeexplore.ieee.org/document/8102087/.

[41] Paul Guermonprez. Intel Aero Wiki, 2018. URL https://github.com/intel-aero/meta-intel-aero/wiki.

[42] S. Jakšić, E. Bartocci, R. Grosu, R. Kloibhofer, T. Nguyen, and D. Ničkovié. From signal temporal logic to FPGA monitors. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 218–227, Sep. 2015. doi: 10.1109/MEMCOD.2015.7340489.

[43] Aaron Kane, Omar Chowdhury, Anupam Datta, and Philip Koopman. A case study on runtime monitoring of an autonomous research vehicle (ARV) system. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification*, pages 102–117, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23820-3.

[44] Kamal Khouri. Keynote Abstract: Safety and Security at the Heart of Autonomous Driving. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pages 1–1. IEEE, mar 2018. ISBN 978-1-5386-7367-6. doi: 10.1109/EMC2.2018.00006. URL https://ieeexplore.ieee.org/document/8524012/.

[45] B. Khoussainov and A. Nerode. *Automata Theory and its Applications*. Progress in Computer Science and Applied Logic. Birkhäuser Boston, 2012. ISBN 9781461201717.

[46] Anders La Cour-Harbo. Quantifying risk of ground impact fatalities of power line inspection BVLOS flight with small unmanned aircraft. In *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1352–1360. IEEE, jun 2017. ISBN 978-1-5090-4495-5. doi: 10.1109/ICUAS.2017.7991323. URL http://ieeexplore.ieee.org/document/7991323/.

[47] W. S. Lee, D. L. Grosh, F. A. Tillman, and C. H. Lie. Fault Tree Analysis, Methods, and Applications A Review. *IEEE Transactions on Reliability*, R-34(3):194–203, aug

1985. ISSN 0018-9529. doi: 10.1109/TR.1985.5222114. URL http://ieeexplore.ieee.org/document/5222114/.

[48] Mathilde Machin, Jeremie Guiochet, Helene Waeselynck, Jean-Paul Blanquart, Matthieu Roy, and Lola Masson. SMOF: A Safety Monitoring Framework for Autonomous Systems. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 48(5):702–715, may 2018. doi: 10.1109/TSMC.2016.2633291. URL https://ieeexplore.ieee.org/document/7786826/.

[49] Amin Majd and Elena Troubitsyna. Integrating Safety-Aware Route Optimisation and Run-Time Safety Monitoring in Controlling Swarms of Drones. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 94–95. IEEE, oct 2017. ISBN 978-1-5386-2387-9. doi: 10.1109/ISSREW.2017.63. URL http://ieeexplore.ieee.org/document/8109263/.

[50] Piergiuseppe Mallozzi. Combining Machine-Learning with Invariants Assurance Techniques for Autonomous Systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 485–486. IEEE, may 2017. ISBN 978-1-5386-1589-8. doi: 10.1109/ICSE-C.2017.40. URL http://ieeexplore.ieee.org/document/7965396/.

[51] Caroline Bianca Santos Tancredi Molina, Jorge Rady de Almeida, Lucio F. Vismari, Rodrigo Ignacio R. Gonzalez, Jamil K. Naufal, and Joao Batista Camargo. Assuring Fully Autonomous Vehicles Safety by Design: The Autonomous Vehicle Control (AVC) Module Strategy. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 16–21. IEEE, jun 2017. ISBN 978-1-5386-2272-8. doi: 10.1109/DSN-W.2017.14. URL http://ieeexplore.ieee.org/document/8023692/.

[52] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016. ISSN 0278-0070. doi: 10.1109/TCAD.2015.2513673.

[53] Thang Nguyen, Ezio Bartocci, Dejan Ničković, Radu Grosu, Stefan Jaksic, and Konstantin Selyunin. The HARMONIA project: Hardware monitoring for automotive systems-of-systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 371–379, Cham, 2016. Springer International Publishing. ISBN 978-3-319-47169-3.

[54] Wesley (Lincoln Labs) Olson. Airborne Collison Avoidance System X. *Tech Notes*, (June), 2015.

[55] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In *2008 Real-Time Systems Symposium*, pages 481–491, Nov 2008. doi: 10.1109/RTSS.2008.43.

[56] G. Reger, Y. Falcone, and K. Havelund. *A Tutorial on Runtime Verification*, volume 34. IOS Press, 2013.

[57] Grigore Rosu. Runtime Verification - Brief Overview. URL https://runtimeverification.com/presentations/RV{_}Overview.pdf.

[58] Johann Schumann, Patrick Moosbrugger, and Kristin Y. Rozier. R2U2: Monitoring and diagnosis of security threats for unmanned aerial systems. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification*, pages 233–249, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23820-3.

[59] Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, and Radu Grosu. Applying runtime monitoring for automotive electronic development. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 462–469, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46982-9.

[60] D. Solet, J. Béchennec, M. Briday, S. Faucou, and S. Pillement. Hardware runtime verification of embedded software in SoPC. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–6, May 2016. doi: 10.1109/SIES.2016.7509425.

[61] Neeraj Suri (TU Darmstadt). Lecture, Topic: "Kicking and Fixing Software: The Fun & Science of Experimental Approaches", Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, Oct., 12, 2018.

[62] John Thomas. Systems Theoretic Process Analysis (STPA) Tutorial. Technical report, 2013. URL http://psas.scripts.mit.edu/home/wp-content/uploads/2014/03/Systems-Theoretic-Process-Analysis-STPA-v9-v2-san.pdf.