Embedding runtime verification post-deployment

for real-time health management of safety-critical systems

by

Brian Christopher Schwinkendorf Kempa

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Aerospace Engineering

Program of Study Committee:
Kristin Yvonne Rozier, Major Professor
Peng Wei
Gianfranco Ciardo

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

ACKNOWLEDGMENTS

Without the efforts of many people, this would not have been possible. At the risk of momentarily forgetting someone, I would like to take this opportunity to thank a few people for their assistance.

First, my family, who never let me forget what I was working for.

Thanks to Prof. Holland and my colleagues at the Aerospace Corporation Microsatellite Systems Division who convinced me to continue on into graduate school.

I would like to thank Josh Wallin and Pei Zhang, the intersections of our research answered more questions than I knew we had.

I am grateful to my committee for their guidance and advice through this process.

Thanks to Zach and Austin for late night work sessions and occasional reminders about the rules of writing composition.

For the emotional and gastronomical support from Carley and Stella, I am forever in your debt.

My fellow basement dwellers, thanks for reassurances about grad school life; even when we were all only trying to convince ourselves.

Additionally, the staff at the local coffee shops – especially the opening crews: we know each other too well at this point, but it was worth it.

Finally, my sincerest appreciation to my advisor Prof. Rozier who took a curious undergrad, made him a PhD student and sent him to NASA to fix a space robot. I wouldn't have gotten to this point without your guidance, and I can't wait to tackle what's next.

ABSTRACT

As cyber-physical systems increase in both complexity and criticality, formal methods have gained traction for design-time verification of safety properties. A lightweight formal method, runtime verification (RV), embeds checks necessary for safety-critical system health management; however, these techniques have been slow to appear in practice despite repeated calls by both industry and academia to leverage them. Additionally, the state-of-the-art in RV lacks a best practice approach when a deployed system requires increased flexibility due to a change in mission, or in response to an emergent condition not accounted for at design time.

Human-robot interaction necessitates stringent safety guarantees to protect humans sharing the workspace, particularly in hazardous environments. For example, Robonaut2 (R2) developed an emergent fault while deployed to the International Space Station. Possibly-inaccurate actuator readings trigger the R2 safety system, preventing further motion of a joint until a ground-control operator determines the root-cause and initiates proper corrective action. Operator time is scarce and expensive; when waiting, R2 is an obstacle instead of an asset.

We adapt the Realizable, Responsive, Unobtrusive Unit (R2U2) RV framework for resource-constrained environments. We retrofit the R2 motor controller, embedding R2U2 within the remaining resources of the Field-Programmable Gate Array (FPGA) controlling the joint actuator. We add online, stream-based, real-time system health monitoring in a provably unobtrusive way that does not interfere with the control of the joint. We design and embed formal temporal logic specifications that disambiguate the emergent faults and enable automated corrective actions. We overview the challenges and techniques for formally specifying behaviors of an existing command and data bus. We present our specification debugging, validation, and refinement steps. We demonstrate success in the Robonaut2 case study, then detail effective techniques and lessons learned from adding RV with real-time fault disambiguation under the constraints of a deployed system.

CHAPTER 1. INTRODUCTION

1.1 Motivation

Software systems have become prevalent in systems that blur traditional delineations between physical and digital processes. From the rapid innovation of automated vehicles, to the presence of connected "Internet of Things" (IoT) technology in the microcontrollers of common consumer goods, the barriers between software errors and physical malfunction have grown imperceptibly thin. Modern engineering now concerns itself with *cyber-physical systems*, wherein these tightly knit domains are treated holistically. Unfortunately, our processes for the design, verification and control of these systems is lacking. [1]

In "Safeware", Leveson details how increasing complexity and prevalence of software systems have lead to increased accidents and defects. Leveson explores how engineering techniques used to handle risk and failure are not sufficient in software systems, and how in some cases their use can exacerbate the inadequacies. An enumerated list of myths about software systems and analysis of the failures of traditional approaches are supplemented with real-world events where the most unlikely accidents were caused by supposedly safe systems. [2] One of the main identified sources of this regression in engineering integrity comes form the ever increasing complexity of these systems paired with their continued abstraction and accelerated rate of development. Because of this, software glitches are a component of daily life that is accepted at a rate that would never pass muster for physical systems.

On the other front, advances in automation technology has lead to a renaissance in robotics, increased reliance on complex systems for health-care, and the explosion of unmanned vehicles like drones and cars [3]. This new breed of safety-critical systems are in double jeopardy from the previously mentioned deficiencies in software system safety analysis and the higher-level at which these systems are designed and configured[2].

The field of *formal methods* has seen success in securing cyber-physical systems, but adoption has been limited by the high cost and diminished speed of development for programs implementing formal techniques from the clean-sheet design. In addition, the shortage of skilled practitioners further constrains even projects willing and able to pay the cost.

In response to the inflexibility (and often, intractability) of traditional formal methods like model checking and automated theorem proving, *lightweight formal methods* were developed that trade away guarantees of existing formal techniques to gain advantages in specific use-cases. In 2001 the practice of trading-off the complete reasoning over all potential system executions for reasoning over a singular system trace was defined at a new workshop, eponymously named *runtime verification* (RV). [4]

Runtime verification based monitors exemplify qualities necessary for safety-critical system health management, and can even provide some of the desirable capabilities of learning based systems without the non-determinism or unjustifiable 'reasoning' [5]. The foresight at design-time is also finite; RV can be applied to contingencies that no amount of off-line analysis can address. However, these techniques have been slow to appear in practice despite repeated calls by both industry and academia to leverage them [3], [5]–[7].

Runtime verification provides an ideal platform for *system-health management* (SHM), which expands on traditional caution and warning systems. Simple logging and monitoring systems are limited to recording the state of the system, but for CPS diagnostics a list of parameters out of nominal range may not be sufficient to determine root cause or suggest corrective action. A SHM solution provides insight and awareness of the system behavior. Knowledge of the interconnected nature of the system and behavior over time is leveraged to contextualize anomalies, provide chain-of-failure tracking, and report the status of system requirements.

Using RV to build a SHM platform provides a trifecta of benefits: increased autonomy, assisting operators in complex scenarios and replacing them when intervention is not feasible; improved robustness, without undercutting traditional safety systems; and incorporation of temporal context, without a large increase in data generation.

1.2 Project Background

In the fall of 2017, the Laboratory for Temporal Logic at Iowa State University was approached by the Robotics Systems Technology Branch at NASA Johnson Space Center (JSC). They had identified a reoccurring fault in a robotic system and believed the *Realizable, Responsive, Unobtrusive Unit* (R2U2) framework could be adapted for detection, without invalidating existing safety systems. The following January I embedded within the team at JSC to learn the system architecture, design requirements, and fault behavior directly from the developers and operators. Upon returning to Iowa State, work continued using test hardware loaned by JSC.

1.2.1 Robonaut2

Robonaut2 is a dexterous humanoid robot that has been deployed to the International Space Station since 2012 [8]. Originally only a torso with a head and arms, it was upgraded with legs in 2014. These legs are comprised of serial elastic actuators wherein a torsional spring is placed within the joint connecting the motor and the load [9]. Using a high-fidelity mass-model of the robot and precise measurement of joint displacement, the system can command motion with specified impedance, maintaining precision while tightly bounding force applied by the system on the environment [10]. This enables the system to undertake tasks with a near-human level of dexterity while remaining safe when in confined spaces with astronauts.

Upon deployment, a new failure mode was observed in these joints wherein an *Absolute Position Sensor* (APS) may initialize to an incorrect value, biased approximately 60 ° from truth. Currently, mitigation requires a human operator: either ground support, subject to signal coverage; or by an astronaut whose time is limited, expensive, and difficult to schedule on short notice. Triggering automatic corrective action without compromising existing safety guarantees would increase the availability and resilience of the system.

1.2.2 Absolute Position Sensor

The series elastic joints in Robonaut2's legs use two Netzer APSs[1] each, one on each side of the torsional spring, to instrument both position and – through displacement – joint force. The Netzer Electric Encoder™APS uses a modulated electric field reflected by a dielectric rotor to generate a signal directly proportional to the sin and cos of the rotation angle.

The sensor has a "coarse" positioning channel with 3 cycles per revolution and is further subdivided by the "fine" position channel with 64 cycles per revolution. The number of cycles per revolution in each channel is selected such that they do not have a common denominator, and therefore each angle of the mechanism has an unambiguous combination of the four output values ('coarse' sin and cos as well as 'fine' sin and cos.) At initialization, both the coarse and fine channels are read to determine the absolute angular position of the sensor without movement. Afterwards, only the fine channel is tracked to provide updated position information.

After considerable testing and deployment to the International Space Station, a new failure mode was observed in these sensors. During initialization, an incorrect value was reported, with a measurement bias of $\frac{2\pi}{3} \approx 2.1$ rad observed. Based on the bias of the erroneous value, the source of the fault is suspected to be incorrect selection of the coarse channel position by the APS. Since after initialization only the fine channel is tracked, if this error occurs the sensor will read erroneously until re-initialized.

1.2.3 Robonaut2 Safety System

To comply with exacting safety standards on human spaceflight vessels, Robonaut2 has a multi-layered, two-fault tolerant safety system and several operational capabilities to reduce risk to its surroundings [11]. At the lowest level of this architecture, the embedded motor controllers run 4 nested control loops at 5 kHz each. One of the control loops tracks joint torque and computes the current state by measuring the spring displacement [10]. When an actuator is brought out of safe mode, a cascade of invariant, parameter, and system health checks are made. In the fault condition,

[1] Model: DS-90-64-SH-0 http://netzerprecision.com/products/ds-90/

these checks fail due to perceived high torque loading. While this system works as intended, the detected offset is well beyond the hard-stop of the joint, yet this is not treated as spurious error. Because of this, manual override of the motion control system is required, making recovery from this state difficult and expensive.

1.2.4 Constraints

Inherent in the challenge of adding supplemental health management to an existing system is the limited available resources and lack of flexibility in the system architecture. The motor controllers are composed of a paired *MicroController Unit* (MCU) and *Field Programmable Gate Array* (FPGA) where the MCU runs the high-speed nested control loops and the FPGA handles connectivity between the robot control bus, *Robonet*, and components of the joint. Only a subset of the full information collected by the motor controller is passed to Robonet. To increase the amount of data available for diagnostic reasoning, and to minimize the additional information to be passed up the control bus, the Robonaut2 team requested that our fault disambiguation solution run on the joint controller directly. Also, to maintain the certification for the control loops on the MCU, we were further restricted to using only the motor controller FPGA. The resources on the FPGA are limited to what unallocated fabric remains, and the addition of our monitor cannot impact the timing of the existing system.

An additional challenge was that the fault was only recently replicated in a controlled setting. At the beginning of this project, only limited debug information from the ISS Robonaut2 was available. Consequently, the initial fault specification was derived from a plain-language description of the Robonaut2 team's diagnosis of the underlying fault mechanics.

1.2.5 Goal

Disambiguating which APS is faulting, enables the proper mitigative action to trigger automatically – increasing system robustness without invalidating the guarantees of the safety system. We use runtime verification to construct a system health management platform hosted on the spare

computational resources in the embedded joint. Using the history of the sensor values, the faulting APS is identified.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 reviews the current state-of-the-art of online runtime verification as well as published efforts closest in objective to this effort. Work on the Robonaut2 fault case is split between chapter 3, which details the addition of a RV system health monitor to the Robonaut2 platform, and chapter 4 which details the design of a specification for the monitoring system embedded in the previous chapter. Chapter 5 describes the results of the R2 case study, including the final R2U2 configuration used. Finally, chapter 6 discusses the results and contributions, and explores future work.

CHAPTER 2. RELATED WORK

2.1 System Health Management and Cyber-Physical Systems

Partially a spin-off of the *Integrated Vehicle Health Management* (IVHM) programs of the 1990s and early 2000's[12], *Software Health Management* (SWHM) concerns the "automated detection, diagnosis, prediction, and mitigation of adverse events" [13] and was identified as a separate field in 2011. Quickly, relevant techniques were adopted from the related fields of *Fault Detection Isolation Recovery* (FDIR), *Verification and Validation* (V&V), and *Integrated System Health Management* (ISHM). [5]

Highly integrated application like spacecraft [14] took the SWHM concept full-circle as a health management solution for the whole vehicle system. Feeding back in on this cycle, there are now calls for IVHM programs to integrate the reasoning techniques championed by the SWHM movement. [15] All these closely-related lines of inquiry outline the general case of SHM. As CPS become increasingly complex, the various domain specif flavors of health management (like IVHM, FDIR, and ISHM) are converging. [16]

2.2 Runtime Verification

For constructing SHM systems from RV systems, only a subset of the wider field is applicable. To meet the needs of CPS SHM [5] we want RV that is *online* and *stream-based*. [17]

Online runtime verifiers operate alongside the systems they are verifying. To those outside the formal methods community, this would be the intuitive definition of RV itself, but a majority of the work in RV has been on offline methods the look at the logs of a system execution retroactively. [18]

Stream-based RV produces output, or verdicts, on the specifications throughout the run of the system, in opposition to a single judgment at the end of the system execution. [19]

2.2.1 Specification Language

The language used to specify the behavior being verified will determine the ease of formalizing the system requirements, and what implementations are available to monitor them. A verity of languages have been developed for use in RV, but most are related in some way to Linear Temporal Logic. [20]

In 1977, Amir Pnueli introduced Linear Temporal Logic (LTL) for use in formal method verification of computer programs[21]. LTL takes the basics of propositional logic and adds operators to contextualize time. This enabled the description of properties for formal methods. In formal verification, two common types of properties are safety (something bad won‘t happen), and liveness (the goal will eventually be reached) [22]. Without the vocabulary to describe the propositions over time, these basic system concepts could not be expressed.

LTL logic of timelines, and since operational concepts are also described with timelines, LTL and it's derivative languages are well suited for mission-based specification for vehicle systems and CPS. [1]

2.3 Prior Work

Most RV systems are designed to be incorporated into a system from design-time and therefore have inflexible requirements on the target system. [23] Overall, various components of the requirements of this project have been seen in isolation – but never combined in the manner require for use on R2.

Kane et al applied RV as a SHM solution to an autonomous truck, and utilized a bus observer rather than instrumented code – however, the monitor was software, not hardware based. [24] LOLA is a stream-based RV monitor designed for mission systems, but is also a software runtime. [25]

A majority of the RV tools available target software systems [26] and require modification of the source to produce an instrumented binary. RV monitoring in hardware has been demonstrated by several tools [27], [28], however they all require resynthesis of the monitor when changing specification. This makes the flight certification a prohibitive burden for their deployment.

R2U2 was designed from the ground-up for unobtrusive observation and SHM of unmanned flight systems from with hardware of software implementations available. [29], [30] The combination of flight-heritage and runtime configurable specification assist in meeting the specific demands of the R2 project.

CHAPTER 3. EMBEDDING RUNTIME VERIFICATION

3.1 Runtime Verification for System Health Management

The fault's suspected mechanism of action and testimonial from operators who had worked with the problem before indicated that the temporal context of the system was vita to selecting the correct corrective action. By comparing the system in fault-state to it's last known working configuration a heath monitor can use this temporal context to identify the erroneous sensor. Our approach embeds a runtime monitor to reason over the trace of the system bus in real time with temporal logic. By capturing the causal relations between the signals, we can relate the relative changes in sensor agreement to disambiguate the fault.

3.1.1 R2U2

The *Realizable, Responsive, Unobtrusive Unit* (R2U2) is an online, real-time, stream based RV framework providing SHM to CPS like autonomous aircraft. [29], [30]

R2U2's eponymous design goals coincide with the constraints of a robotics system with strong safety guarantees:

Realizable – Proven framework with adaptable implementations in hardware and software

Responsive – Online, real-time, stream-based runtime verification

Unobtrusive – No interference with host system operation, certification or timing

Unit – Modular and self-contained

3.1.1.1 Features

R2U2 has several layers of reasoning available; however for this project, only two are required: signal processing, and temporal observation.

The signal processing layer contains facilities for interpretation, conditioning and comparison of arbitrary attached signals. The key component of this layer is the boolean checker, used to convert the incoming signals into boolean variable for use by the 2nd layer.

The observer layer uses the boolean variables produced by the signal processing layer as inputs to a tree of temporal observers capable of evaluating temporal logical specifications in a number of logics. Logical specification is coved in chapter 4. The observation tree is stored in a run-time configurable program memory and can therefore be modified without resynthesizing the FPGA. This eases certification and allows a small realization of R2U2 to be used for multiple purposes throughout a mission by reconfiguration.

3.1.1.2 Output

R2U2 has two output modes, synchronous and asynchronous. In synchronous mode R2U2 returns a *verdict* about each specification it monitors at every time step. A verdict denotes that at the listed time, the temporal logic specification was either true, false, or is unknown due to insufficient information.

In asynchronous mode, R2U2 returns all specification verdicts eagerly, including early failure or acceptance[29]. This is the mode used for the R2 project as we only need to be notified when our fault conditions are tripped, but when they are we want to be informed as soon as possible.

3.1.1.3 Nomenclature

R2U2 is a general framework with multiple implementations. An *implementation* of R2U2 is the source of an R2U2 reasoning engine and may be written in a software or hardware language like C or VHDL. An implementation is combined with the *design-time configuration* which consist of additional information needed by the implementation to be hosted on a given platform. Common design-time configuration options include compiler settings for software implementations or vendor dependent intrinsics for hardware implementations. Combining the design-time configuration with the implementation, a runnable R2U2 engine for a platform, called a *unit*, can be generated. R2U2

units are runtime reconfigurable and the *runtime configuration* is the set of specification settings to be reasoned over. There might be different configuration sets for various mission phases or system modes contained in the runtime configuration. All together, the design-time and runtime configuration, combined with an implementation constitute a *deployment* of R2U2 and contain the requisite information to reproduce an identical monitoring system.

A specific deployment of R2U2 may have some otherwise runtime configurable options hard-coded at design time depending on the platform needs and certification requirements.

3.2 Target Platform Architecture

In accordance with the requirements of the Robonaut2 team, any solution must be hosted at the embedded joint controller. Each leg actuator is locally controlled by a MC-30 motor controller that connects to the Robonet command & control bus. High level planning is done by the flight computers, which do not have hard real-time deadlines. The resulting command is written to the Robonet controller which acts as a gateway into the real-time domain of the robot.

Robonet uses a request-reply communications link and a schedule to synchronize a shared memory space[31]. The Robonet head-node reads data values from the network of motor controllers, and writes data back to them, on a round-robin cycle. Under this scheme, each embedded joint controller only needs storage space to mirror memory values it reads or writes, ignoring most of the shared address space entirely. The resulting system ends up functionally equivalent to a pub-sub key-value store with real-time guarantees.

The MC-30 motor controllers consists of an Actel ©ProASIC3L FPGA[1] and a Texas Instrument ©Hercules [2] ARM Cortex-R4F Microcontroller. The FPGA model is targeted at the aerospace/defense by the vendor with selling points of relatively high radiation tolerance (though, it is not a "rad hard" product) and "instant-on" (no programming delay on power-on) operation. The microcontroller used is designed for safety-critical operation with features like error corrected memory and two cores running the same code for error detection.

[1] Model: M1A3P1000L
[2] Model: TMS570LS

The existing flight configuration routes all sensors, actuator control, and communications through the FPGA while the Microcontroller runs a high-rate model-based control algorithm[31]. In nominal operation, the Robonet IP on the FPGA receives a write to a memory register used by the MCU. Depending on the command received, the MCU can provide position, velocity, or impedance control of the actuator. Sensor values, preprocessed by the FPGA, are provided to the MCU as parameters for the control calculations. The resulting actuator command is passed back through the FPGA to the motor control circuitry on the MC-30.

The nature of the certification process dictates that R2U2 be resident on the FPGA. This is an ideal location; the FPGA acts at a nexus for all the information in the MC-30 controller. Figure 3.1 shows the initial desired architecture in this system. Using a serial debug port attached to the FPGA, specifications can be dynamically loaded on R2U2 and the returned verdicts captured for analysis. Since the FPGA acts as a nexus for all of the actuator's sensors, the observer can tie directly into the available data – unobtrusively.

In traditional design-cycles, an FPGA must be selected before the exact size required is known. Similar to the iterative process of sizing microcontroller memory, intentional over-provisioning is used to mitigate the risk of exhausting all the available resources later in the design process. If a product goes into mass-production, the size of the FPGA may be adjusted after the design is frozen to reduce the unit cost, but on low volume projects like Robonaut2 the re-work of hardware elements and validation required to switch components outweigh the benefits. The end result is spare FPGA logic fabric on most embedded systems that we hope to leverage for the new SHM.

3.3 Embedding Challenges

3.3.1 Realizable: Resource Utilization

By definition, utilizing the spare resources in an existing design is a constrained writing exercise. To begin adapting R2U2 to the Robonaut2 platform, the resource usage of the current flight code was analyzed. For reproducibility, Appendix A lists the build procedure.

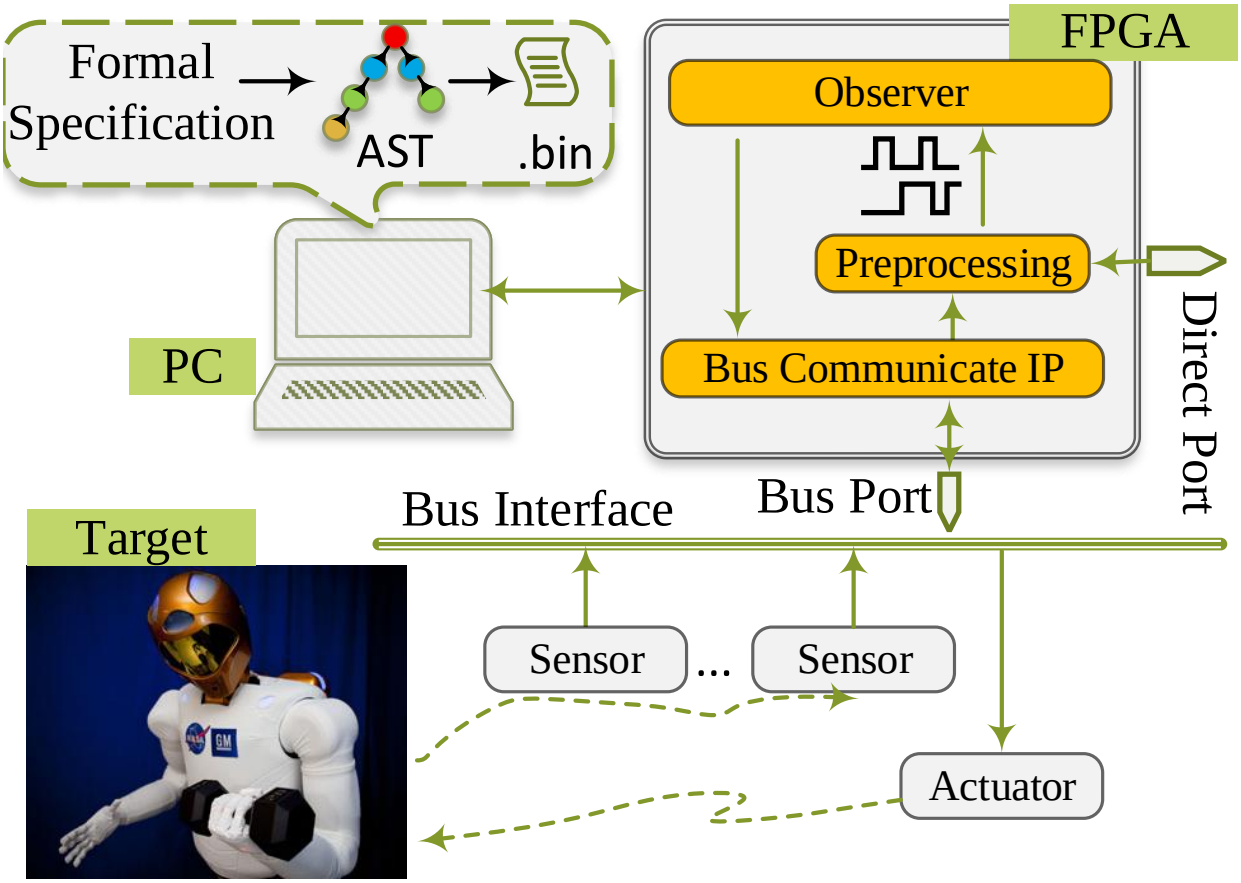


Figure 3.1 Embedded Diagram

The current version of the vendor synthesis software[3] was installed and used to synthesize the current version [4] of the flight software with the relevant communications IP [5]. Table 3.1 lists the consumable resources on the FPGA and the usage rates by the base flight code. Some values of note: the "CORE" element is the general logic fabric of the FPGA, and about half of it is used by the base design; the existing NASA flight code utilizes the only Phased Locked Loop (PLL) so R2U2 will need to tie into an existing clock-line; finally, only 2 of the 32 available RAM blocks are utilized – this becomes critical for embedding R2U2.

[3]Microsemi Libero SoC Version: 11.8 SP3 (02/2018) on Windows 10 Pro 64-bit (build: 1803)
[4]Repository: robonaut-2-embedded/turbo_fpga commit: 28748dc3862d0be2bf71fa60ebf336974eca6389
[5]Repository: robonaut-2-embedded/robonet_ip commit: 33ea33f44ce839a9e6ce0951c157ab1707bef228

Table 3.1 MC-30 FPGA Resource Usage — As-Flown

Element	**Used**	**Total**	**% Used**
CORE	12582	24576	(51.20%)
IO (W/ clocks)	70	300	(23.33%)
Differential IO	0	74	(0.00%)
GLOBAL (Chip+Quadrant)	6	18	(33.33%)
PLL	1	1	(100.00%)
RAM/FIFO	2	32	(6.25%)
Low Static ICC	0	1	(0.00%)
FlashROM	1	1	(100.00%)
User JTAG	0	1	(0.00%)

The first test performed was to add the R2U2 hardware description into the project and resynthesize. After an unusually long delay, the tool chain reported failure to place and route the logic network on the selected part. The culprit was Block-RAM (BRAM) usage. BRAM is an FPGA resource used for data storage and the R2U2 implementation required much more than was available. To compensate for the shortage, the synthesis tool attempted to build storage units out of other logic resources. This is not an efficient conversion, and was the reason for the long processing time.

Looking at the architecture in Figure 3.2 of R2U2 on an FPGA explains the memory requirements. To enable dynamic reconfiguration of the observer without resynthesis – or recertification – of the hardware, R2U2 makes use of BRAM for instruction memory, variable memory and a variety of queues. This design provides the flexibility to change the observer specification at run-time without re-synthesizing the FPGA by reloading the instruction memory.

The required BRAM size for an R2U2 unit is a function of the max formula size, queue depth, and the time stamp width. Practically, these correlate to: how many specifications you can monitor; how complex the specifications can be; and over how long a period of time the specifications can reasons over. The exact impacts of these factors is complex but deterministic and has motivated work in the design-time calculation of precise resource needs, or the capability available for a given BRAM size[6].

[6]Pei Zhang, et. al. “Formal Specification Encoding Under Platform Resource Constraints” Under submission

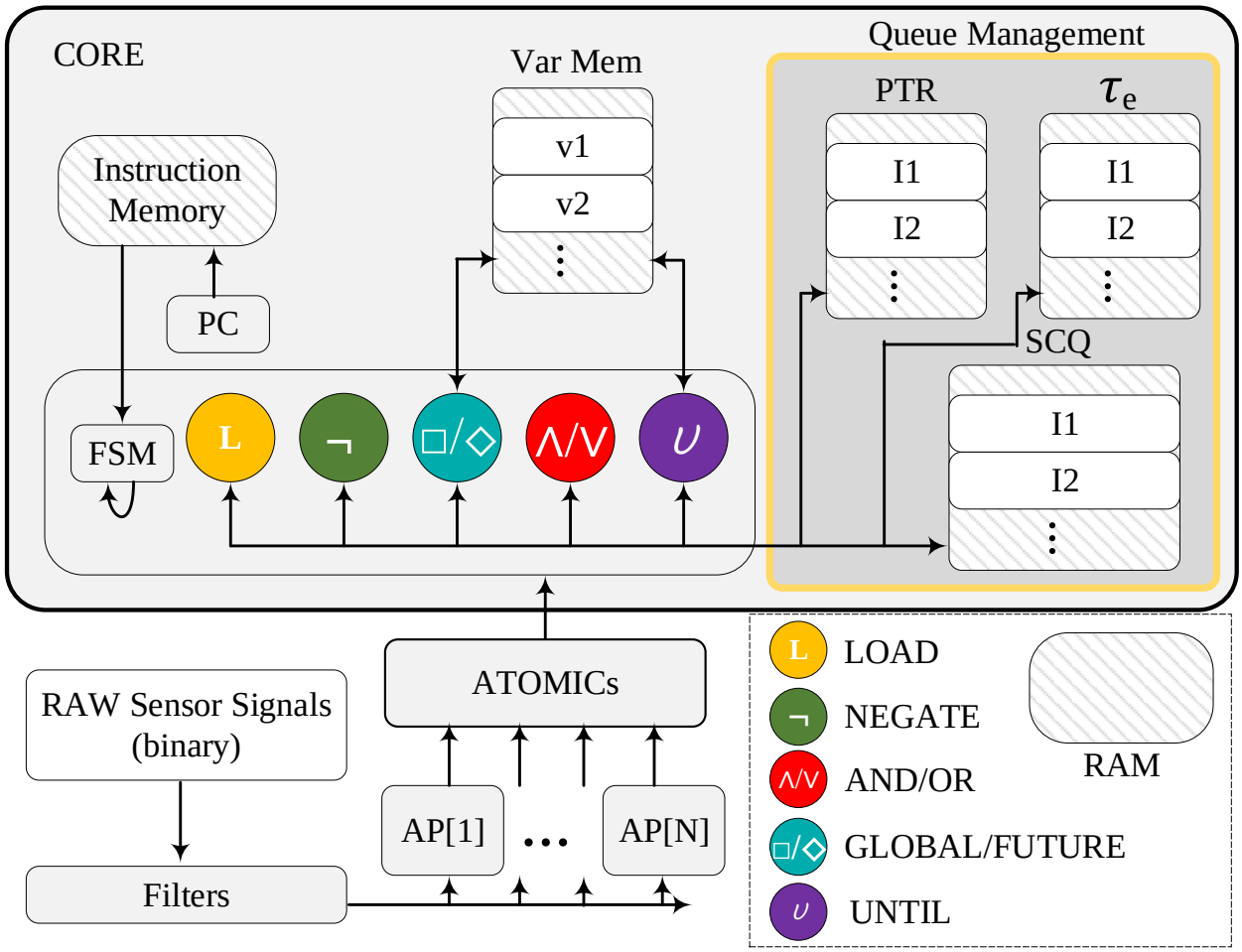


Figure 3.2 R2U2 FPGA Architecture

Using expectations from the R2 team about the time-scale over which they want to reason, a suitable queue depth setting was calculated. Consulting with the R2 team, a conservative estimate of the timestamps length required and the known number of free BRAMS was used to reduce the design space. This resizing was done primarily by expert judgment in this case as the initial goal was only to get a correctly synthesized R2U2 unit on the MC-30 FPGA.

To test the resource requirements of various R2U2 design parameters, a synthesis project without the MC-30 driver was built. In Table 3.2, the core logic and BRAM usage of the stand-alone MC-30 driver and BRAM-reduced R2U2 implementations are expressed in terms of percentage of chip

resources. The final column is an estimate of the requirements of the combined design as ideally they would sit side-by-side on the FPGA without impacting each-other.

Table 3.2 MC-30 and R2U2 Independent FPGA Resource Usage

Element	**MC-30**	**R2U2**	**R2U2 + MC-30 (Est.)**
CORE	(51.20%)	(36.32%)	(88.20%)
RAM/FIFO	(6.25%)	(78.13%)	(84.38%)

A final synthesis run of the combined R2U2/MC-30 hardware design yields the resource utilization in Table 3.3. The BRAM usage was purely additive as expected, however the core resource usage was below estimate. This is likely due to optimizations made by the synthesis tool-chain, however it also indicates that the R2U2 observer is more tightly integrated with the MC-30 structures. While it may be possible to force the synthesis to separate the designs more strongly, the resulting design was retained. Because it is impossible to fully separate R2U2 from the MC-30 driver due to the level at which they are connected, validation and verification of a more segmented design would be just as rigorous as a (slightly) more integrated one. While this satisfies the requirements for this project, it is entirely possible to further segment R2U2 in application that require it.

Table 3.3 MC-30 FPGA Resource Usage — With BRAM Reduced R2U2

Element	**Used**	**Total**	**% Used**
CORE	19614	24576	(79.81%)
IO (W/ clocks)	71	300	(23.67%)
Differential IO	0	74	(0.00%)
GLOBAL (Chip+Quadrant)	6	18	(33.33%)
PLL	1	1	(100.00%)
RAM/FIFO	27	32	(84.38%)
Low Static ICC	0	1	(0.00%)
FlashROM	1	1	(100.00%)
User JTAG	0	1	(0.00%)

With the tuning of the memory usage parameters, R2U2 is fully realizable on the space resources of the FPGA. However, the unobtrusive and responsive claims must be validated.

3.3.2 Responsive & Unobtrusive: Timing Compliance

In keeping with the R2U2 tenet of *unobtrusiveness*, the addition of the monitor must not impact the timing of the existing system. For FPGA designs, timing analysis is the measure of how long is needed for the outputs to change after then inputs change. While a number of factors go into timing analysis of FPGAs, modern synthesis tools are designed to allow the setup times to be violated first if all requirements cannot be satisfied. The value of the setup time as an intentional design flaw is that it can be corrected by adjusting the clock rate. Where as a hold-path or clock skew timing error will prevent an FPGA design from ever working correctly, a setup time violation will come with a satisfiable clock rate. For example, the timing analysis for a design attempting to run on a 10 MHz clock might indicate that a setup time violation occurred, and that it would require 9.5 MHz cycles to meet timing. In this case, the designer could reduce the clock rate to 9.5 MHz and the design would work as intended. If the required clock is rate fixed by some design parameter such as a communications bus or mission requirement, then the design would need to be reworked to ease the propagation delay responsible for the long setup time.

The results of a timing analysis for the MC-30/R2U2 design is shown in Table 3.4. This design runs R2U2 of the 10 MHz clock and instruments the MC-30 sensors as described in section 3.3.3. In the table, each row is one of the main clock lines in the design. For each clock, there is a requested and an estimated clock rate. As long as the estimated clock rate is higher than the requested clock rate, the design should function as expected. Unfortunately in the current configuration, sCLK_50 cannot be satisfied. The max rate the timing analysis anticipates the 50 MHz clock line to be capable of satisfying is only 32.8 MHz.

Table 3.4 MC-30 and R2U2 FPGA Timing Slack

Starting Clock	**Req. Freq.**	**Est. Freq.**
sCLK_10	10.0 MHz	17.6 MHz
sCLK_50	50.0 MHz	32.8 MHz
sCLK_50_90d	50.0 MHz	65.9 MHz

The impact on the 50 MHz clock is alarming but confusing. R2U2 is tied to the 10 MHz clock (the clock-rate of the Robonet bus) and therefore should have little to no impact on the 50 Mhz domain. Further investigation into the dynamics of the MC-30 clock domain discovered that even the unmodified flight code is unable to make timing. Table 3.5 shows the clock slack in the original flight code. Since this analysis would need to have been satisfied before flight, this result is surprising.

Table 3.5 MC-30 Flight Code FPGA Timing Slack

Starting Clock	**Req. Freq.**	**Est. Freq.**
sCLK_10	10.0 MHz	22.5 MHz
sCLK_50	50.0 MHz	48.0 MHz
sCLK_50_90d	50.0 MHz	65.9 MHz

In looking for an explanation for this discrepancy, the only difference from the flight build found was the usage of a different version of the synthesis software. The vendor no longer offers that older version of the tool-chain, and no longer issues the expiring and node-locked licenses, so it was not possible to confirm with a build on the original version. This calls into question the validity of both the original and the current timing analysis.

Investigation of the version differences revealed that a set of false path annotations were used by the flight code build and were being ignored by the updated build tool. The signal paths through the FPGA used to compute the timing compliance are detected automatically. If a signal path is possible, but will never occur in operation it is a false path. For FPGA designs like the sensor interfaces and Robonet communications node, it is common for many of these false paths to exist as the possible inputs are limited by the external devices or protocols. If one of these false paths is very long or slow, it can be the source of a timing error in an otherwise compliment and functional design. The existing false path annotations were ignored for two reasons: the naming scheme used by the synthesis tool had changed, so it no longer referred to valid components of the design; and the syntax for the annotations themselves had changed so the tool was not attempting to read them. The existence of these annotations do support the theory that the timing analysis is incorrect and

false paths exist in the design. These would need to be located and marked to get an accurate timing report. This is further supported by the disparate clock impact that R2U2 had on the 50 MHz clock when the bulk of the unit's logic is in the 10 MHz clock domain.

Because of the doubt surrounding the timing analysis, a motor controller was loaded with the combined MC-30/R2U2 build and used to run a bench-top test rig. All tests passed and no anomalous bus behavior was detected. The decision to continue to the next phase of integration was made on this basis.

Additional options exist for bringing the timing within bounds. The R2U2 implementation is currently written for readability and was originally targeted at a different FPGA family. Optimizations, both general and deployment specific, can easily be made to reduce the propagation delay. The current serial communications module used by the debug port is one of the largest drags on R2U2 timing constraints. This becomes unnecessary after the R2 unit is integrated with Robonet for configuration as well as verdict return. The Robonet implementation itself is also not optimized by the R2 team's own admission – it met timing and underutilized the available resources there was no previous motivation to do so. Also, we suspected there are false paths in the design that could be annotated for the timing analysis package to consider.

3.3.3 Instrumentation

During initial synthesis trials on the MC-30 FPGA, R2U2 did not connect to the Robonet bus in any way. This separation of concerns allowed focus on the realization of R2U2 in the constrained environment. These test variants of the SHM tool utilized the same debug serial port that loaded specifications to read data. By sending test specifications and data with known solutions, the R2U2 implementation can be tested for correctness while the impact on R2 is observed. With a working version of the MC-30 firmware and R2U2 successfully hosted simultaneously, a method of connecting the sensor values to the reasoning unit is required.

The initial plan was to tap the signals before they hit the preprocessing stages of the MC-30 FPGA design, but this was quickly abandoned despite the minimal impact on the existing

functionality. The format of the raw signals were wildly inconsistent and would have required the duplication of the same pre-processing steps, wasting valueable FPGA real-estate. By finding a tie-in point behind the battle-tested preprocessing stages, the values are consistently formatted, usefully conditioned, and helpful derived values like rate-of-change are computed.

Since all relevant sensors have their values synced over Robonet, the values are placed in a reserved memory location on the FPGA. The Robonet IP has existing facilities for arbitrated reading from this memory. The in-built Robonet arbitrated read will eliminate an entire class of concurrent memory access errors, without adding additional logic to handle this challenging application. A trade-off to this approach is that R2U2 loses access to raw signal information and can only select data that is synced to the flight-computer over Robonet. This removes the potential for extra visibility into the joint beyond what the other Robonet nodes can see. However, it is a win for system flexibility as the R2U2 unit will not need a custom synthesized tie-in to each joint peripheral sensor. The Robonet node approach also provides stronger guarantees about unobtrusiveness by leveraging an existing system.

3.4 Runtime Monitoring

The architecture of the final MC-30/R2U2 build is detailed in Figure 3.3. It is noteworthy that R2U2 sits beside the existing system, only intersecting at the shared memory and using a unused debug port for loading.

With this in place, reasoning over bus values can begin. The stacked traces in Figure 3.4 show the Robonet bus values above the output from the R2U2 reasoning engine. Robonet bus values for the joint encoder position are reported in radians by the red line, while the torque on the joint is reported in Newton-meters by the gold line. Over the course of the roughly 3.5 minute sample, the joint is commanded to move and hold at various radial positions while resistance to the motion is provided sporadicly by the operator.

The lower timeline of Figure 3.4 displays the verdicts of the specifications listed in Table 3.6. A deeper overview of R2U2 specifications are given in chapter 4 but the essential meaning of

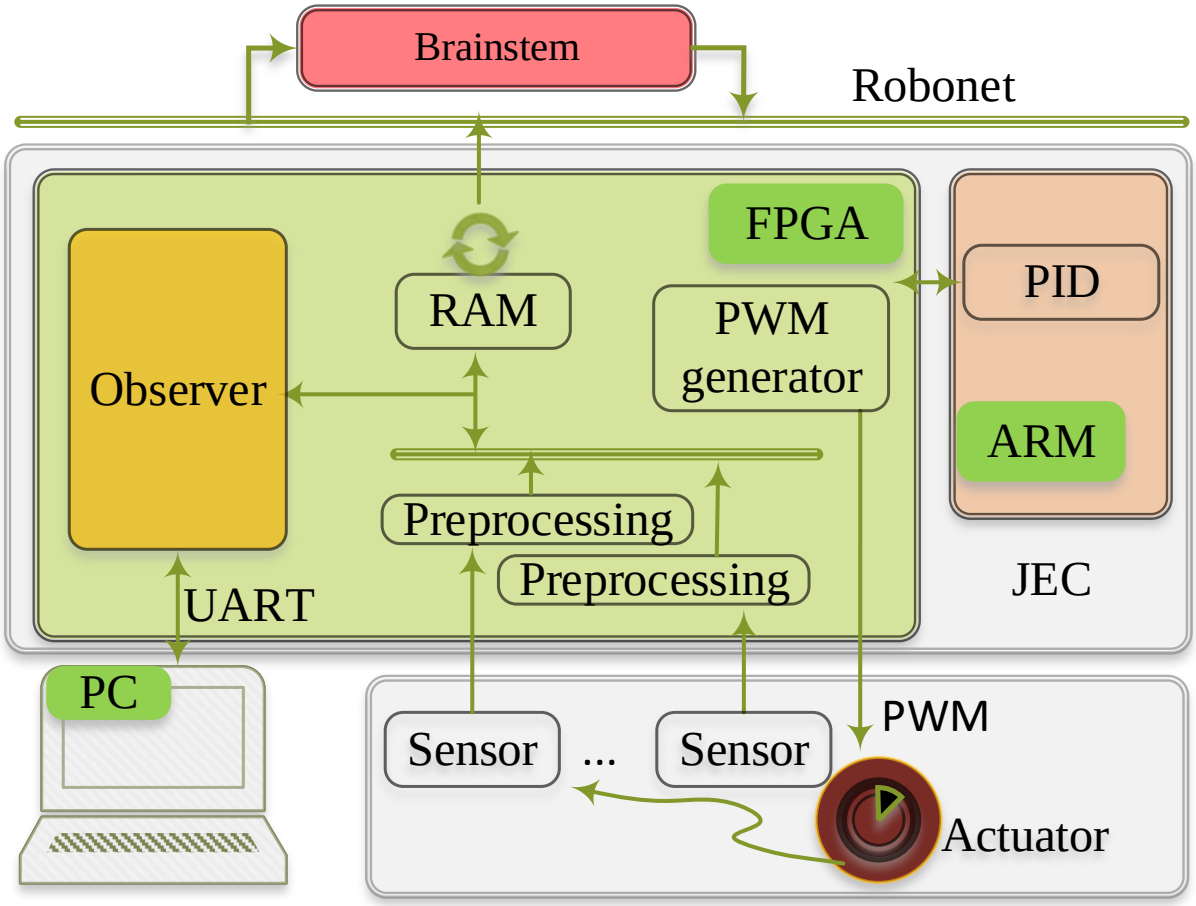


Figure 3.3 R2U2 FPGA Architecture

each sample is given here. The first two specifications, a_0 and a_1 are boolean checkers in the signal processing layer, determining if a value read from Robonet meets a arithmetic criteria. Specification a_2 is basic propositional logic on the previous booleans. Temporal logic is utilized in specifications a_3 and a_4. The diamond shape in a_3 is the eventually operator and with it's time-bound means that to be true, it's argument (a_2 in this case) must be true within the next 10 time steps. The final specification is significantly more complex and was only used to verify the operation of the until operator.

Table 3.6 Sample R2U2 on MC-30 specifications

Name	**Formula**	**Meaning**
a_0	EncoderPosition > 1	Encoder position above 1 radian
a_1	JointTorque < 2	Torque on joint below 2 Nm
a_2	$a_0 \wedge a_1$	a_0 and a_1 met simultaneously
a_3	$\Diamond_{[0,10]}(a_2)$	a_2 true at some point in the next 10 steps
a_4	$(a_3)\mathcal{U}_{[4,8]}(\Box_{[0,4]}\neg a_1)$	Between 4 and 8 steps from now, a_3 is true until a_1 is not

A video[7] is available demonstrating R2U2 running live on the R2 platform and reasoning over the joint state. The demo also showcases use of temporal operators and dynamic specification loading without stopping the robot.

[7] http://temporallogic.org/research/R2U2/R2U2-on-R2_demo.mp4

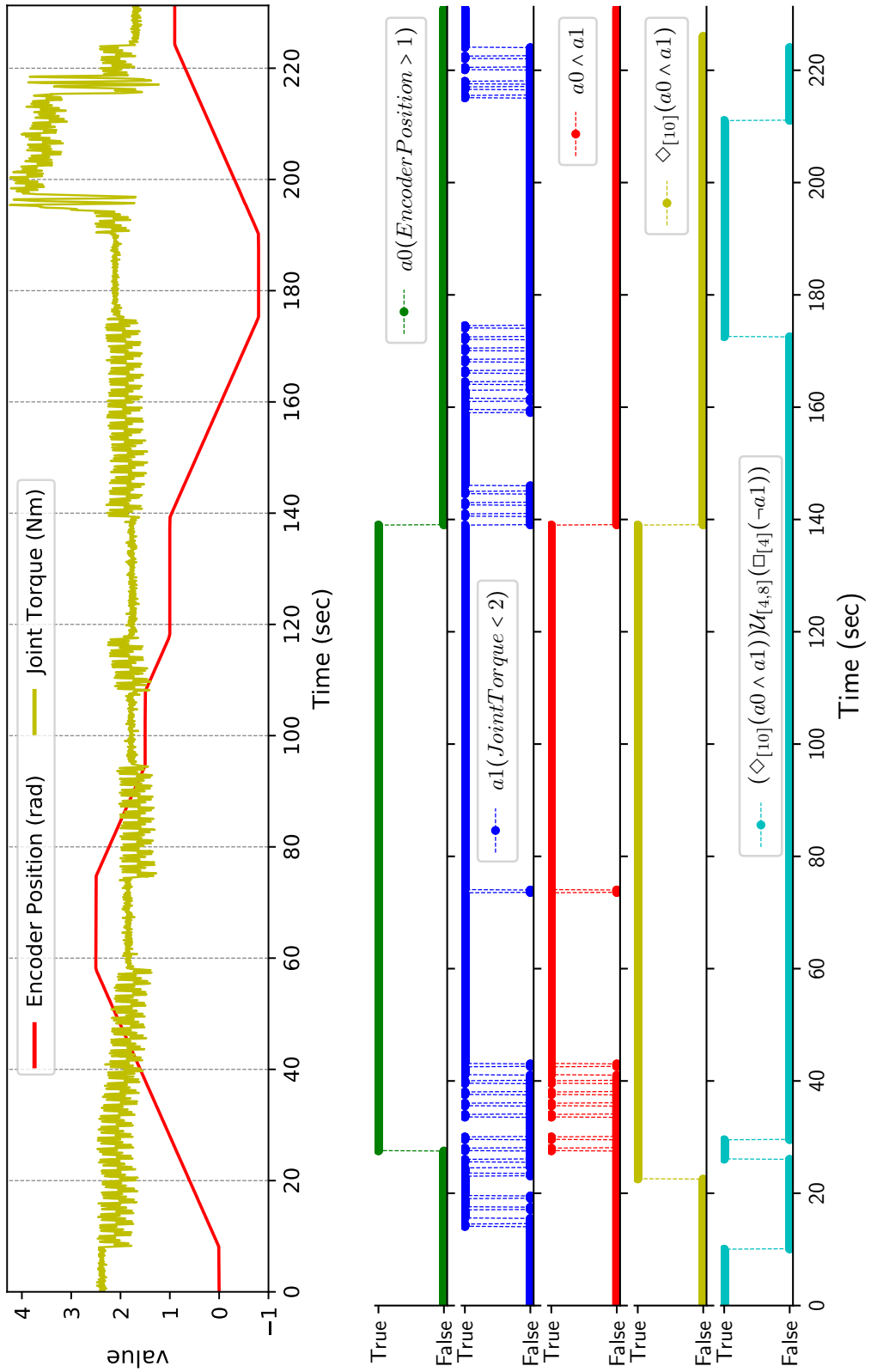


Figure 3.4 Sample R2U2 reasoning on Robonet data

CHAPTER 4. SPECIFICATION

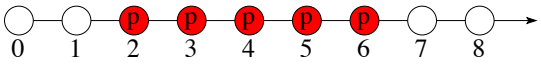
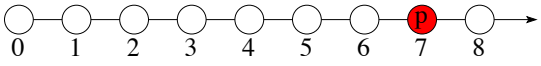
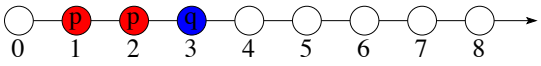
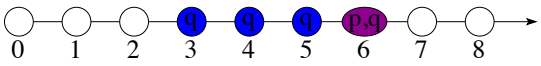
4.1 Specification Logics

The temporal observer trees in the R2U2 reasoning engine can be programed in several different logics. All of these logics are temporal – they encode the time varying relations between the signals received form the signal processing layer.[30] For this application, *Mission-time Linear Temporal Logic* (MLTL) is used as the specification language.

4.1.1 Mission-time Linear Temporal Logic

A limitation of LTL, it always discusses infinite traces. To compensate, several derivative languages have been defined for use in specific applications. Reinbacher, Rozier, and Schumann introduced MLTL in 2014. [29] As a temporal logic targeting mission systems at runtime, MLTL adds bounds to all the temporal operators as seen in table 4.1.

Table 4.1 Mission-time Linear Temporal Logic[29]

Symbol	Operator	Timeline
$\Box_{[2,6]}p$	ALWAYS$_{[2,6]}$	
$\Diamond_{[0,7]}p$	EVENTUALLY$_{[0,7]}$	
$p\mathcal{U}_{[1,5]}q$	UNTIL$_{[1,5]}$	
$p\mathcal{R}_{[3,8]}q$	RELEASE$_{[3,8]}$	

4.2 Subsection Design

4.2.1 Components of an Observation Tree Specification

To create the run-time configuration for R2U2 on the MC-30, three layers of the unit require configuration:

Bus Values: Data values to read from Robonet

Signal Processing: Conversion of bus values to boolean values

Logic Observers: MLTL properties to evaluate

A critical design decision that is easily overlooked is the separation of concerns between the signal processing layer and the temporal reasoning layer of the R2U2 unit. As seen later in this case-study, this decision can have impacts on the required hardware resources, capabilities of the design and complexity.

4.2.2 A Structured Approach

Designing specifications for an existing system is inherently a constrained writing exercise wherein the constraints of the application naturally limit the design space. However, the design space is still large enough that trying to brainstorm entire designs directly for all but the simplest systems is infeasible. Using a structured approach allows the designers to be guided away from anti-patterns, reinforces good technique like strong separations of concerns, and eases collaboration by provide a vocabulary for reconciling different ideas.

Our approach follows the sequence:

1. Set measurable specification goals
2. Develop a strategy
3. Document assumptions
4. Work backwards from the top of the observer tree

5. Iterate, working from the bottom of the tree up

The principle behind this method is to guide designs toward maintaining separation of concerns between layers in R2U2, and encourage leveraging the strengths of each layer, without providing too rigid a structure as to inhibit creative problem solving. The specifics of each stage are discussed in detail in section 4.3 while showcasing how this method was used to develop an R2U2 runtime configuration for the Robonaut2 fault disambiguation case study.

4.3 Initial Specification

4.3.1 Defining Success

As an intermediate stage between the high-level project goals and the specification that runs on the system health monitor, a set of success criteria are constructed. These should function as measurable targets for specification performance.

For disambiguating the Robonaut2 joint offset fault we set the following 3 goals:

1. Zero error: No false positives or negatives
2. Disambiguation between three system states:
    - APS1 is reporting false position
    - APS2 is reporting false position
    - No action, a unrelated fault occurred
3. Detect possible precursors to assist root cause identification

4.3.2 Strategy

Specification design progresses backwards though the R2U2 reasoning tree. Starting from the subject matter experts' description, a system health strategy is constructed as the plain-language representation of specification. For the first revision of the fault disambiguation specification, we

began with the strategy: *If the differences between APSs are larger than 1 radian, then the APS that disagrees with the encoder by more than 0.01 radian is at fault.*

4.3.3 Assumptions

With this initial concept, but before designing a health monitor configuration, it is critical that all assumptions be explicitly documented. For this specification we identified our assumptions as:

1. Agreement with the encoder value implies correct APS2 position reporting
2. Agreement between any 2 position sources implies the minority opinion is incorrect (sensor voting)

After confirming these assumptions with the Robonaut2 team, the next stage of specification is the construction of the temporal observer tree.

4.3.4 Temporal Observer Tree

Decomposing the temporal and propositional logic of specification strategy, we elected to have three top-level booleans to represent the thee potential states of the system as described in the specification goals. The three formula to be checked can then be defined as:

1. The APSs are within the 1 radian threshold
2. APS1 and the encoder are within the 0.01 radian threshold
3. APS2 and the encoder are within the 0.01 radian threshold

By inspection of the specification strategy, there is no obvious time component. While this could be encoded using only propositional logic over the R2U2 boolean values from the signal processing layer, we elected to add a temporal "debounce" with a series of "globally for 3 steps" operators where processed bus values are used.

4.3.5 Signal Processing

With an initial logical structure in place, the signal processing layer design maps values from the system bus to boolean checkers. From the specification strategy, we identified three outside values required by the formula: APS1 position, APS2 position, and encoder position. Specifically, boolean checkers for the range between APS1 and APS2, as well as between each APS and the encoder are required by the specification.

4.3.6 Design Iteration

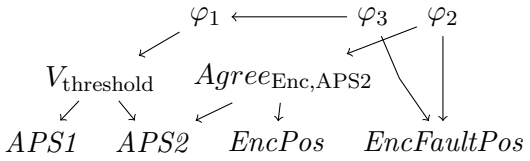
With the required bus values identified, the hardware implementation team was informed of the requested inputs. While examining the fault descriptions provided by the Robonaut2 team, references to a "encoder fault position" bus signal were found repeatedly. This system bus variable already acts as a boolean, active when APS1 and the encoder disagree on joint position. The availability of this signal from the system makes our boolean checker for APS1 and encoder position agreement redundant. With the addition of this 4th bus value and the elimination of one component of signal processing the complete R2U2 configuration for this specification can be written as shown in table 4.2.

4.4 Validation

After development and implementation of the initial specification, a terrestrial R2 unit developed the same fault as the unit on the ISS. With direct access to the system, the operator recorded traces of nominal, faulty, and recovery behavior. These traces provide a validation case of the proposed specification, however they are not recordings of the Robonet bus. The captured data was recorded from the Robot Operating System (ROS) bus that acts as a message-passing layer between the three redundant flight computers [31].

The traces were provided in a ROS message archive format called a bag file. To feed them through the R2U2 unit on the MC-30, the data stream first requires conversion. Using ROS

Table 4.2 Fault disambiguation specification – revision 1

<table>
<tr><td colspan="2">R2U2 Configuration</td></tr>
<tr><td>Bus Values</td><td>Temporal Formulas</td></tr>
<tr><td>APS1: Position [rad]
APS2: Position [rad]
EncPos: Position [rad]
EncFaultPos: Encoder Fault [bool]</td><td>$\varphi_1 = \square_{[0,3]}(V_{\text{threshold}})$
$\varphi_2 = \text{FaultEncPos} \wedge \square_{[0,3]}(Agree_{\text{Enc,APS2}}) \rightarrow \text{APS1}_{\text{Wrong}}$
$\varphi_3 = \varphi_1 \vee !\text{FaultEncPos} \rightarrow \text{APS2}_{\text{Wrong}}$
Observer Tree</td></tr>
<tr><td>Signal Processing
$V_{\text{threshold}}$ $= \lvert\text{APS1} - \text{APS2}\rvert > 1$ rad
$Agree_{\text{Enc,APS2}}$ $= \lvert\text{APS2} - \text{EncPos}\rvert > 0.01$ rad</td><td> 
</td></tr>
</table>

provided utilities, the relevant messages were extracted, aligned by time-stamp and repacked into an HDF5[1] file, the format used by the Robonet record and playback system.

This data trace is still not perfectly representative of what an embedded R2U2 monitor would observe during a fault, but using the R2U2 ability to run this *synthetic data* three traces were examined; one in which no fault occurs as a control (fig. 4.1), one in which the robot experiences the fault (fig. 4.2), and one in which the operator has unsuccessfully attempted recovery (fig. 4.3).

In Figures 4.1-4.3 the top timeline shows the position values of the encoder (red), APS1 (blue, labeled motor), and APS2 (yellow, labeled joint) in radians. The lower timeline shows the R2U2 verdicts of the two relevant specifications. Figure 4.1 displays nominal operation of R2 through a range of movement. Notice how tightly matched the three position indicators are. In Figure 4.2 the APS fault occurs at the 43 second mark. We see the expected 60 degree shift in APS position, and the $V_{\text{threshold}}$ specification correctly flags that our fault of interest occurred. However, the encoder jumps to an implausible 998 radians, making the encoder agreement test invalid. Further, in Figure 4.3 the operator attempts to correct the problem. While it may appear successful, the

[1] Hierarchical Data Format 5 https://www.hdfgroup.org/

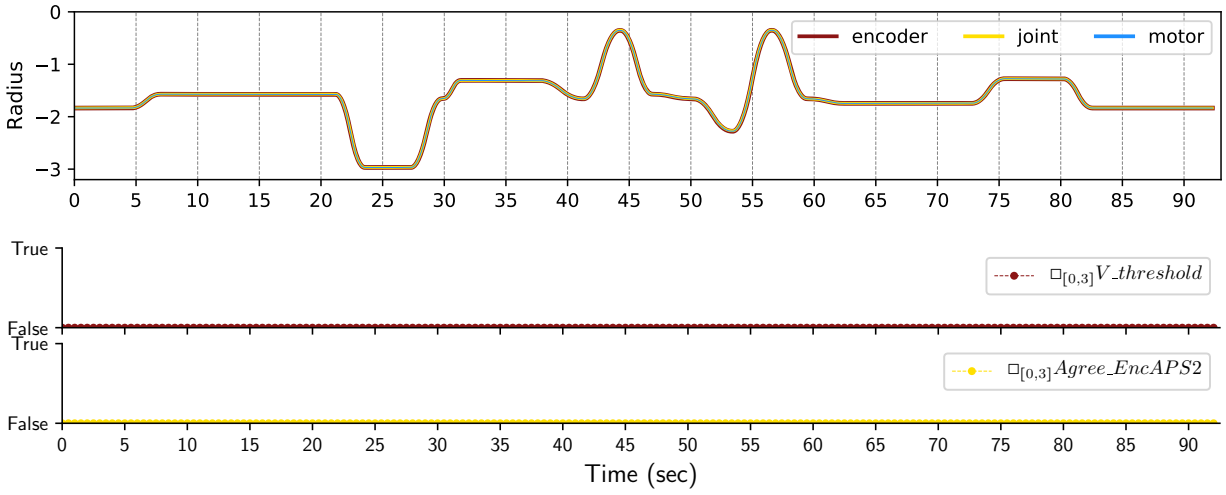


Figure 4.1 Ground R2: No Fault

difference between the three sensors after time 19 is still too wide to appease the safety system and release the joint. At first, this appears to challenge the $V_{\text{threshold}}$ fault detection specification, but this is technically a different failure mode and beyond the scope of what the disambiguation goal is trying to accomplish.

This data revealed a implicit assumption of the first specification, it was assumed that encoder values froze during the fault. The discontinuous behavior of the encoder was not anticipated and invalidates the basis of the original specification. An important caveat, this discontinuity could be an artifact of the ROS bus during the fault and may not be present on the Robonet bus R2U2 is reading.

4.4.1 Specification Revision

To correct for the invalidated assumptions, an adjustment can either be made to the temporal reasoning or the signal processing layer of R2U2. Temporally, a discontinuity in the encoder (logically captured similarly to the the existing detection for APS discontinuity) can be identified and the value of the encoder prior to the jump can be used. Alternatively, this behavior can be considered a matter of input conditioning.

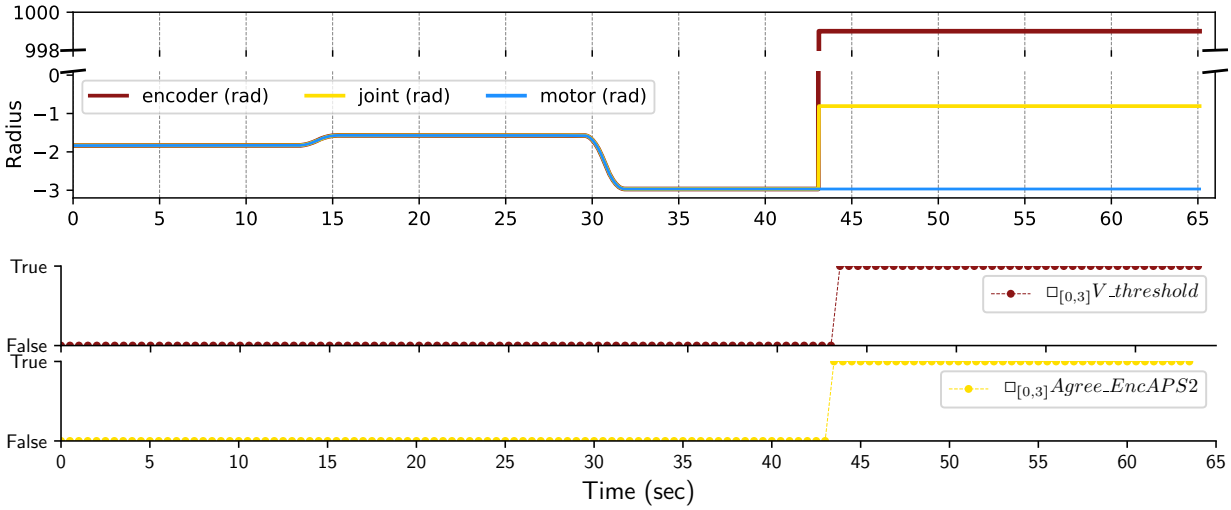


Figure 4.2 Ground R2: APS Fault

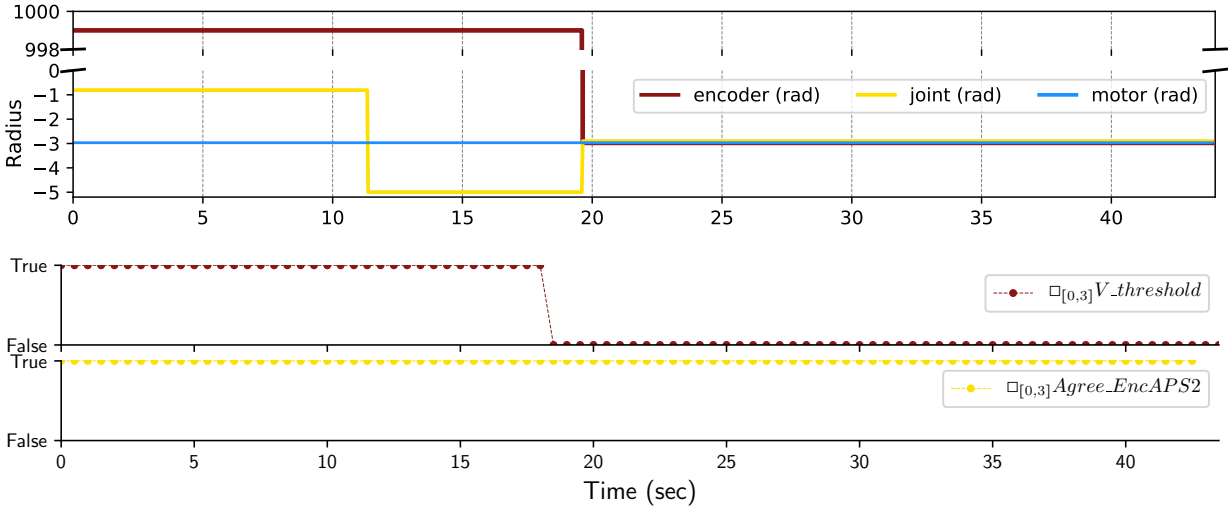


Figure 4.3 Ground R2: Failed Corrective Action

4.4.1.1 Signal Processing Approach

Initially, using the signal processing layer of R2U2 was explored as a solution to the detection of the discontinuity. While this approach was, itself, simple – the additional complexity placed on the runtime observer was prohibitive. The invalid encoder readings while in the fault condition would require the previous value of the system to be compared after the fault had been detected. Downsides of this approach include the additional signal processing capability is outside the run-time reconfigurability of the signal processing layer and would then require resynthesis. Additionally, several of the proposed approaches in the brainstorming session would involve introducing delays that would confuse the meaning of the time steps in the temporal observers.

4.4.1.2 Temporal Reasoning Approach

Alternatively, a nearly pure temporal logic approach is available. The trade-off is that it requites many small, but simple, formulas. With other tools, this would be a higher cost than the single processing approach. However, with R2U2 we can encode these entirely in the run-time configuration of the unit without synthesizing the FPGA design or using additional resources for the signal processing.

The temporal approach plays into the strengths of the R2U2 unit, and was selected for the revised specification. We now present our solution to the R2 case study.

CHAPTER 5. CASE STUDY

5.1 Specification

Following the design approach detailed in section 4.2.2, the following R2U2 specification was constructed for use in disambiguating the APS offset fault in Robonaut2.

5.1.1 Defining Success

Our goals remain unchanged from the original described in section 4.3.1:

1. Zero error: No false positives or negatives
2. Disambiguation between three system states:
    - APS1 is reporting false position
    - APS2 is reporting false position
    - No action, a unrelated fault occurred
3. Detect possible precursors to assist root cause identification

5.1.2 Strategy

The new strategy leverages the unexpected encoder behavior that invalidated the initial specification. Using the encoder as a marker for the fault occurring, a mechanism for selecting the fault APS is required. Since a large movement in value is indicative of a faulting sensor, comparison of the incorrect APS position before and after the fault will display a discontinuity.

Further reducing these thoughts to the main tenets of the strategy, we have:

- If there is a sudden, large change in reported position by the encoder, then the fault is occurring

- If there is a sudden, large change in reported position by an APS, then the APS that jumped is at fault

Forming the plain-language description of our strategy, we began the revised specification attempt with the strategy: *If there is a sudden, large jump in the encoder and an APS's position report, the APS that jumped is at fault.*

5.1.3 Assumptions

The previous assumptions being proven invalid, the disqualifying behavior is taken as the invariants for the new specification, namely:

- A significantly large discontinuity in the data is the signature of the fault
- Only the fault APS 'moves' in the fault case

5.1.4 Temporal Observer Tree

The encoder jumps to a value that will never occur in normal operation, this can easily be capture and used as a flag that the fault occurred. Let e be the proposition that the encoder is out-of-range (say, greater than 100):

$$e = \text{EncoderPosition} > 100$$

To express the comparison of the APS value before and after fault, we need to reason over this exact time boundary. Use a "globally in one time step" (equivalent to "next time") temporal operator with the encoder out of range in the later time but not the earlier:

$$\neg e \wedge \diamond_{[1]} e$$

This expression is true at the time immediately before the fault occurs.

Next, express that an APS jumps drastically over this range. As a placeholder let a_n represent APS1 being in some localized area. We can then make a statement such as:

$$(a_1 \wedge \neg e) \wedge \diamond_{[1]} (\neg a_1 \wedge e)$$

Which encodes the idea that at a given time, the encoder is in range and APS1 is reading in some local area "a_1", and that in one time steps the encoder will be out-of-range and APS1 will not be in that local area.

This would imply that APS1 is faulting:

$$(a_1 \wedge \neg e) \wedge \diamond_{[1]}(\neg a_1 \wedge e) \rightarrow \text{APS1}_{\text{Fault}}$$

We can use b_1 representing APS2 being close to some position to express the same thing for APS2:

$$(b_1 \wedge \neg e) \wedge \diamond_{[1]}(\neg b_1 \wedge e) \rightarrow \text{APS2}_{\text{Fault}}$$

If we repeat this with $(a_1, a_2, \ldots a_n)$ and $(b_1, b_2, \ldots b_n)$ with correctly sized regions for the propositions, we can catch APS jumps of a desired size.

5.1.5 Signal Processing

The only inputs required are the position readings for the encoder and two APS. The necessary boolean checkers need to provide the a_n and b_n variables of an appropriate size to catch the jumps.

To size the jump detection properly, it needs to be ensured that the APS cannot move more than a minimum amount without moving to a different numbered region. Since the APS has been seen to jump 60 degrees during the fault case, it would be a reasonable first approximation to set the minimal jump size to 60 degrees. Doing so would require that no section is larger than 60 degrees, and since the APS works on a range of -180 degrees to 180 degrees, that would require 6 even spaced regions, $(a_1, a_2, \ldots, a_6)$ and $(b_1, b_2, \ldots, b_6)$ each with a corresponding formula like above.

5.1.6 Design Iteration

During iteration an initial concern was the behavior at crossover, when the system is naturally crossing one of these region boundaries. Since the fault only occurs when arming a parked actuator, travel is not a concern. Also, nominal travel across a boundary will not be accompanied with an encoder range error, further preventing false positives.

To add some debounce against a potential spurious signal, the "globally" operator was expanded to require the fault state to exist for two consecutive time steps.

Let a_n and b_n indicate APS1 and APS2 respectively, as reporting a position in the n^{th} of 6 equally sized regions of the joint travel range. And let e indicate the encoder position being beyond the physical extents of the joint. For each $n \in [0, 6]$:

$$(a_n \wedge \neg e) \wedge \diamond_{[1,2]}(\neg a_n \wedge e) \rightarrow \text{APS1}_{\text{Fault}}$$

$$(b_n \wedge \neg e) \wedge \diamond_{[1,2]}(\neg b_n \wedge e) \rightarrow \text{APS2}_{\text{Fault}}$$

5.2 Verification

Before being run against the collected trace data used in section 4.4, a SMT[1] solver with an MLTL extension was used as a verification step to ensure the produced formula are not tautologies, and can be falsified. In other words, both the formula and their negation were checked for satisfiability. The tool output is provided in Appendix B. With the formula passing these checks and functioning correctly on the sample system trace, the process is complete.

[1]Satisfiability Modulo Theories http://smtlib.cs.uiowa.edu/

CHAPTER 6. CONCLUSION

6.1 Discussion

While the case study successfully accomplished the project goals, some open questions remain.

The timing analysis requires more investigation by JSC, but R2U2 on the MC-30 is believed to remain unobtrusive. While it would be preferable to have an outright analysis stating that the addition of R2U2 does not cause timing concerns, the results have called into question the validity of the analysis itself. It would be reasonable to expect future exercises in retro-fitting SHM units to require more work in the validation of unobtrusiveness than in embedding the unit itself.

The final specification does work successfully on both operator descriptions of the event, and on the traces provided by the one faulting ground. However, there is only one actual system trace. Having been synthesized form a different bus, caution must be exercised when declaring the universality of this solution. Points of particular concern include: is the faulting ground R2 behaving identically to R2 on the ISS, are there inconsistent behaviors in the fault case ,and are any features of the trace a product of the conversion to the ROS bus? In short, we are concerned about over-fitting. That being said, the candidate specification is correct to the limit of our available information.

Given the run-time reconfigurability of the R2U2 SHM unit, the MC-30/R2U2 deployment can be certified, installed, and loaded with the candidate specification. Then, the output of the monitor can be observed during the next several faults as a passive observer only. This will either validate the specification, or highlight shortcomings that can be utilized to improve the specification. After validation on R2, the specification can be used for automating corrective action.

6.2 Future Work

With a demonstrated R2U2 implementation utilizing the spare FPGA fabric, several potential extensions become possible. With R2U2 units installed on multiple joint controllers, an exploration into networked SHM nodes opens up several exciting possibilities. Several networked units could allow system-level SHM by aggregating output from individual embedded R2U2 units. In one potential setup, a head R2U2 unit could utilize verdicts from other embedded units to reason over total system health. In addition, a specification can be written across several units – dividing the temporal observation tree among the units and enabling a distributed approach to system-wide SHM. In a similar vein, a software implementation of R2U2 could be run on the flight computer, providing this system level reasoning with information only available on the other side of the real-time barrier, or as a secondary SHM system. The added value in running both software and hardware implementations, is they utilize different internal architectures – reducing the potential for common-cause or common-mode failures [30]. With the development of MTLT benchmarks[1], the reasoning capacity of this salvaged "spare logic" can be measured.

Work continues on improvements to R2U2 implementations themselves, and other applications for the framework. Integration of R2U2 with schedule planners and prognostic systems would greatly increase the capability for autonomous action and planning. Leveraging RV allows the power of formal techniques to augment existing systems without the massive computing power required by model-checking approaches. To further facilitate these integrations, exploration into methods to further reduce the resources needed by R2U2 could unlock new applications and allow embedding in more systems.

Overall, this line of inquiry will continue to pursue answers to open questions in the application of RV for SHM, such as: How do we efficiently incorporate temporal aware monitors in existing systems? How do we automate corrective action and use RV for more than monitoring? How do we

[1] J. Wallin, K.Y. Rozier. "Generating System-Agnostic Runtime Verification Benchmarks from MLTL Formulas via SAT" Under submission

extract specifications from existing sources, make specification easy for the domain expert without a background in formal methods, and glean insight on fault precursors from functional SHM units?

REFERENCES

[1] X. He, Z. Dong, and Y. Fu, "A Systematic Approach for Developing Cyber Physical Systems," en, Jul. 2018, pp. 456–505. DOI: 10.18293/SEKE2018-004. [Online]. Available: http://ksiresearchorg.ipage.com/seke/seke18paper/seke18paper_4.pdf (visited on 01/28/2019).

[2] N. Leveson, *SafeWare : system safety and computers / Nancy G. Leveson.* eng. Reading, Mass.: Addison-Wesley, 1995, ISBN: 0201119722.

[3] A. Desai, T. Dreossi, and S. A. Seshia, "Combining Model Checking and Runtime Verification for Safe Robotics," en, in *Runtime Verification*, S. Lahiri and G. Reger, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2017, pp. 172–189, ISBN: 978-3-319-67531-2.

[4] K. Havelund and G. Rosu, "Preface: Volume 55, issue 2," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 287–288, 2001, RV'2001, Runtime Verification (in connection with CAV '01), ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(05)80577-8. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066105805778.

[5] A. N. Srivastava and J. Schumann, "Software health management: A necessity for safety critical systems," *Innovations in Systems and Software Engineering*, vol. 9, no. 4, pp. 219–233, Dec. 2013, ISSN: 1614-5054. DOI: 10.1007/s11334-013-0212-0. [Online]. Available: https://doi.org/10.1007/s11334-013-0212-0.

[6] K. Y. Rozier, "From simulation to runtime verification and back: Connecting single-run verification techniques," in *Proceedings of the Spring Simulation Conference (SpringSim)*, Tucson, AZ, USA: Society for Modeling & Simulation International, Apr. 2019, TBD. DOI: TBD. [Online]. Available: TBD.

[7] C. Torens, J. C. Dauer, and F. Adolf, "Towards Autonomy and Safety for Unmanned Aircraft Systems," en, in *Advances in Aeronautical Informatics: Technologies Towards Flight 4.0*, U. Durak, J. Becker, S. Hartmann, and N. S. Voros, Eds., Cham: Springer International Publishing, 2018, pp. 105–120, ISBN: 978-3-319-75058-3. DOI: 10.1007/978-3-319-75058-3_8. [Online]. Available: https://doi.org/10.1007/978-3-319-75058-3_8 (visited on 01/28/2019).

[8] M. A. Diftler, J. S. Mehling, M. E. Abdallah, N. A. Radford, L. B. Bridgwater, A. M. Sanders, R. S. Askew, D. M. Linn, J. D. Yamokoski, F. A. Permenter, B. K. Hargrave, R. Platt, R. T. Savely, and R. O. Ambrose, "Robonaut 2 - the first humanoid robot in space," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 2178–2183. DOI: 10.1109/ICRA.2011.5979830.

[9] G. A. Pratt and M. M. Williamson, "Series elastic actuators," in *Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots*, vol. 1, Aug. 1995, 399–406 vol.1. DOI: 10.1109/IROS.1995.525827.

[10] J. Badger, A. Hulse, R. Taylor, A. Curtis, D. Gooding, and A. Thackston, "Model-based robotic dynamic motion control for the robonaut 2 humanoid robot," in *2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, Oct. 2013, pp. 62–67. DOI: 10.1109/HUMANOIDS.2013.7029956.

[11] J.M.Badger, A.M.Hulse, and A.Thackston, "Advancing safe human-robot interactions with robonaut 2," in *Proceedings of the 12th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2014.

[12] S. Ofsthun, "Integrated vehicle health management for aerospace platforms," *IEEE Instrumentation Measurement Magazine*, vol. 5, no. 3, pp. 21–24, Sep. 2002, ISSN: 1094-6969. DOI: 10.1109/MIM.2002.1028368.

[13] A. N. Srivastava and J. Schumann, "The case for software health management," in *2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology*, Aug. 2011, pp. 3–9. DOI: 10.1109/SMC-IT.2011.14.

[14] J. Schumann, O. J. Mengshoel, and T. Mbaya, "Integrated software and sensor health management for small spacecraft," in *2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology*, Aug. 2011, pp. 77–84. DOI: 10.1109/SMC-IT.2011.25.

[15] C. M. Ezhilarasu, Z. Skaf, and I. K. Jennions, "The application of reasoning to aerospace integrated vehicle health management (ivhm): Challenges and opportunities," eng, *Progress in Aerospace Sciences*, vol. 105, pp. 60–73, 2019, ISSN: 0376-0421.

[16] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, "Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications," en, in *Lectures on Runtime Verification: Introductory and Advanced Topics*, ser. Lecture Notes in Computer Science, E. Bartocci and Y. Falcone, Eds., Cham: Springer International Publishing, 2018, pp. 135–175, ISBN: 978-3-319-75632-5. DOI: 10.1007/978-3-319-75632-5_5. [Online]. Available: https://doi.org/10.1007/978-3-319-75632-5_5 (visited on 01/28/2019).

[17] Y. Falcone, S. Krstić, G. Reger, and D. Traytel, "A Taxonomy for Classifying Runtime Verification Tools," en, in *Runtime Verification*, C. Colombo and M. Leucker, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2018, pp. 241–262, ISBN: 978-3-030-03769-7. DOI: 10.1007/978-3-030-03769-7_14.

[18] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07), vol. 78, no. 5, pp. 293–303, May 2009, ISSN: 1567-8326. DOI: 10.1016/j.jlap.2008.08.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1567832608000775 (visited on 02/08/2019).

[19] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to Runtime Verification," en, in *Lectures on Runtime Verification: Introductory and Advanced Topics*, ser. Lecture Notes in Computer Science, E. Bartocci and Y. Falcone, Eds., Cham: Springer International Publishing, 2018, pp. 1–33, ISBN: 978-3-319-75632-5. DOI: 10.1007/978-3-319-75632-5_1. [Online]. Available: https://doi.org/10.1007/978-3-319-75632-5_1 (visited on 02/08/2019).

[20] K. Havelund and G. Reger, "Runtime verification logics a language design perspective," in *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, L. Aceto, G. Bacci, G. Bacci, A. Ingólfsdóttir, A. Legay, and R. Mardare, Eds. Cham: Springer International Publishing, 2017, pp. 310–338, ISBN: 978-3-319-63121-9. DOI: 10.1007/978-3-319-63121-9_16. [Online]. Available: https://doi.org/10.1007/978-3-319-63121-9_16.

[21] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, Oct. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[22] G. J. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997, ISSN: 0098-5589. DOI: 10.1109/32.588521.

[23] A. Desai, T. Dreossi, and S. A. Seshia, "Combining Model Checking and Runtime Verification for Safe Robotics," en, in *Runtime Verification*, S. Lahiri and G. Reger, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2017, pp. 172–189, ISBN: 978-3-319-67531-2.

[24] A. Kane, O. Chowdhury, A. Datta, and P. Koopman, "A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System," en, in *Runtime Verification*, E. Bartocci and R. Majumdar, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 102–117, ISBN: 978-3-319-23820-3.

[25] S. Schirmer, "Runtime monitoring with lola," Master's thesis, Saarland University, Nov. 2016. [Online]. Available: https://elib.dlr.de/113126/.

[26] N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, Dec. 2004, ISSN: 0098-5589. DOI: 10.1109/TSE.2004.91.

[27] L. Convent, S. Hungerecker, T. Scheffel, M. Schmitz, D. Thoma, and A. Weiss, "Hardware-Based Runtime Verification with Embedded Tracing Units and Stream Processing," en, in *Runtime Verification*, C. Colombo and M. Leucker, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2018, pp. 43–63, ISBN: 978-3-030-03769-7.

[28] 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*, IEEE, 2008, pp. 481–491.

[29] T. Reinbacher, K. Y. Y. Rozier, and J. Schumann, "Temporal-logic based runtime observer pairs for system health management of real-time systems," in *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science (LNCS), vol. 8413, Springer-Verlag, Apr. 2014, pp. 357–372.

[30] J. Schumann, K. Y. Y. Rozier, T. Reinbacher, O. J. J. Mengshoel, T. Mbaya, and C. Ippolito, "Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems," *International Journal of Prognostics and Health Management (IJPHM)*, vol. 6, no. 1, pp. 1–27, Jun. 2015.

[31] J. Badger, D. Gooding, K. Ensley, K. Hambuchen, and A. Thackston, "Ros in space: A case study on robonaut 2," in *Robot Operating System (ROS): The Complete Reference (Volume 1)*, A. Koubaa, Ed. Cham: Springer International Publishing, 2016, pp. 343–373, ISBN: 978-3-319-26054-9. DOI: 10.1007/978-3-319-26054-9_13. [Online]. Available: https://doi.org/10.1007/978-3-319-26054-9%5C_13.

APPENDIX A. HARDWARE SYNTHESIS PROCEDURE

```
Robonaut 2 -R2U2 project:

Current Build:
  Build of current flight software using latest Microsemi Libero SoC
    Version: 11.8 SP3 (02/2018)
    OS: Windows 10 Pro 64-bit (build: 1803)

  Source code used:

    Turbo-Driver Project:
      Via fork: https://gitlab.com/LTL-AERO/turbo_fpga
      Branch: master
      Commit: 28748dc3862d0be2bf71fa60ebf336974eca6389

    Robonet:
      Via fork: https://gitlab.com/LTL-AERO/robonet_ip
      Branch: master
      Commit: 33ea33f44ce839a9e6ce0951c157ab1707bef228

  Modifications:

    - Directly vendor robonet in src/ip of turbo_fpga (replacing git submodule)
    - Modify turbo_fpga/src/gen_libero_project.py to run:
      * print from statement to function call if using python 3
      * Do not execute libero automatically
```

```
    * Save generate .tcl


Procedure:
  - Modify turbo_fpga/src/gen_libero_project.py as stated above
  - Run gen_libero_project.py to generate libero_project_gen.tcl
  - Open libero
  - Select menu: Project > Execute Script
  - Select generated tcl file and execute
  - Verify in design hierarchy that "TOP_R2CL_R3.vhd" is set as design root
  - Synthesize:
    * Right-click "turbo_fpga/src/constraint/MC-30_synplify.sdc"
      and select "Mark as used"
    * Right-click "Synthesize" and select "open interactively"
    * On the left-hand panel select the following - deselecting all others:
      - "FSM Compiler"
      - "Retiming"
    * Click Run and close when finished
  - Compile:
    * Right-click and select "Mark as used" for the following:
      - "turbo_fpga/src/constraint/MC-30_R2CL_R3.pdc"
      - "synthesis/TOP_R2CL_R3_sdc.sdc"
    * Right-click "compile" and select "open interactively"
    * Select "Programming file" and use the following options on the
      configuration screens that appear:
      - Compile Options: Leave defaults - click "Ok"
      - Layout Options: Leave defaults - click "Ok"
      - FlashPoint - Programming File Generator:
        * Check "FlashROM"
        * Browse for config file (turbo_fpga/src/ufc/c\*_n\*.ufc)
        * Click Next
        * Check all the 'Program page' checkboxes
```

```
  * Click Finish
- Generate Programming Files: Leave defaults - click "Generate"
```

APPENDIX B. SPECIFICAITON SATISFIABILITY

This technique works by flattening the temporal components of the MLTL formula so it can be solved with a standard SAT solver using SMT.

Satisfiability

```
INPUT: (((((((A_10&(G[1,1] ((!A_10)&E0)))&(G[1,1] (G[1,1] ((!A_10)&E0))))->APS_A_Fault0)
    &(((A_20&(G[1,1] ((!A_20)&E0)))&(G[1,1] (G[1,1] ((!A_20)&E0))))->APS_A_Fault0))&(((
    A_20&(G[1,1] ((!A_30)&E0)))&(G[1,1] (G[1,1] ((!A_30)&E0))))->APS_A_Fault0))&(((B_10&(
    G[1,1] ((!B_10)&E0)))&(G[1,1] (G[1,1] ((!B_10)&E0))))->APS_B_Fault0))&(((B_20&(G[1,1]
     ((!B_20)&E0)))&(G[1,1] (G[1,1] ((!B_20)&E0))))->APS_B_Fault0))&(((B_30&(G[1,1] ((!
    B_30)&E0)))&(G[1,1] (G[1,1] ((!B_30)&E0))))->APS_B_Fault0))
NON-TEMPORAL: (((((((A_10&((!A_11)&E1))&((!A_12)&E2))->APS_A_Fault0)&(((A_20&((!A_21)&E1
    ))&((!A_22)&E2))->APS_A_Fault0))&(((A_20&((!A_31)&E1))&((!A_32)&E2))->APS_A_Fault0))
    &(((B_10&((!B_11)&E1))&((!B_12)&E2))->APS_B_Fault0))&(((B_20&((!B_21)&E1))&((!B_22)&
    E2))->APS_B_Fault0))&(((B_30&((!B_31)&E1))&((!B_32)&E2))->APS_B_Fault0))
TSEITIN: (((ts_1_0|(A_11|(!E1)))&(((!ts_1_0)|(!A_11))&((!ts_1_0)|E1)))&(((ts_2_0|((!A_10)
    |(!ts_1_0)))&(((!ts_2_0)|A_10)&((!ts_2_0)|ts_1_0)))&(((ts_3_0|(A_12|(!E2)))&(((!
    ts_3_0)|(!A_12))&((!ts_3_0)|E2)))&(((ts_4_0|((!ts_2_0)|(!ts_3_0)))&(((!ts_4_0)|ts_2_0
    )&((!ts_4_0)|ts_3_0)))&((((!ts_5_0)|((!ts_4_0)|APS_A_Fault0))&((ts_5_0|ts_4_0)&(
    ts_5_0|(!APS_A_Fault0))))&(((ts_6_0|(A_21|(!E1)))&(((!ts_6_0)|(!A_21))&((!ts_6_0)|E1)
    ))&(((ts_7_0|((!A_20)|(!ts_6_0)))&(((!ts_7_0)|A_20)&((!ts_7_0)|ts_6_0)))&(((ts_8_0|(
    A_22|(!E2)))&(((!ts_8_0)|(!A_22))&((!ts_8_0)|E2)))&(((ts_9_0|((!ts_7_0)|(!ts_8_0)))
    &(((!ts_9_0)|ts_7_0)&((!ts_9_0)|ts_8_0)))&((((!ts_10_0)|((!ts_9_0)|APS_A_Fault0))&((
    ts_10_0|ts_9_0)&(ts_10_0|(!APS_A_Fault0))))&(((ts_11_0|((!ts_5_0)|(!ts_10_0)))&(((!
    ts_11_0)|ts_5_0)&((!ts_11_0)|ts_10_0)))&(((ts_12_0|(A_31|(!E1)))&(((!ts_12_0)|(!A_31)
    )&((!ts_12_0)|E1)))&(((ts_13_0|((!A_20)|(!ts_12_0)))&(((!ts_13_0)|A_20)&((!ts_13_0)|
```

```
    ts_12_0)))&(((ts_14_0|(A_32|(!E2)))&(((!ts_14_0)|(!A_32))&((!ts_14_0)|E2)))&(((
    ts_15_0|((!ts_13_0)|(!ts_14_0)))&(((!ts_15_0)|ts_13_0)&((!ts_15_0)|ts_14_0)))&((((!
    ts_16_0)|((!ts_15_0)|APS_A_Fault0))&((ts_16_0|ts_15_0)&(ts_16_0|(!APS_A_Fault0))))
    &(((ts_17_0|((!ts_11_0)|(!ts_16_0)))&(((!ts_17_0)|ts_11_0)&((!ts_17_0)|ts_16_0)))&(((
    ts_18_0|(B_11|(!E1)))&(((!ts_18_0)|(!B_11))&((!ts_18_0)|E1)))&(((ts_19_0|((!B_10)|(!
    ts_18_0)))&(((!ts_19_0)|B_10)&((!ts_19_0)|ts_18_0)))&(((ts_20_0|(B_12|(!E2)))&(((!
    ts_20_0)|(!B_12))&((!ts_20_0)|E2)))&(((ts_21_0|((!ts_19_0)|(!ts_20_0)))&(((!ts_21_0)|
    ts_19_0)&((!ts_21_0)|ts_20_0)))&((((!ts_22_0)|((!ts_21_0)|APS_B_Fault0))&((ts_22_0|
    ts_21_0)&(ts_22_0|(!APS_B_Fault0))))&(((ts_23_0|((!ts_17_0)|(!ts_22_0)))&(((!ts_23_0)
    |ts_17_0)&((!ts_23_0)|ts_22_0)))&(((ts_24_0|(B_21|(!E1)))&(((!ts_24_0)|(!B_21))&((!
    ts_24_0)|E1)))&(((ts_25_0|((!B_20)|(!ts_24_0)))&(((!ts_25_0)|B_20)&((!ts_25_0)|
    ts_24_0)))&(((ts_26_0|(B_22|(!E2)))&(((!ts_26_0)|(!B_22))&((!ts_26_0)|E2)))&(((
    ts_27_0|((!ts_25_0)|(!ts_26_0)))&(((!ts_27_0)|ts_25_0)&((!ts_27_0)|ts_26_0)))&((((!
    ts_28_0)|((!ts_27_0)|APS_B_Fault0))&((ts_28_0|ts_27_0)&(ts_28_0|(!APS_B_Fault0))))
    &(((ts_29_0|((!ts_23_0)|(!ts_28_0)))&(((!ts_29_0)|ts_23_0)&((!ts_29_0)|ts_28_0)))&(((
    ts_30_0|(B_31|(!E1)))&(((!ts_30_0)|(!B_31))&((!ts_30_0)|E1)))&(((ts_31_0|((!B_30)|(!
    ts_30_0)))&(((!ts_31_0)|B_30)&((!ts_31_0)|ts_30_0)))&(((ts_32_0|(B_32|(!E2)))&(((!
    ts_32_0)|(!B_32))&((!ts_32_0)|E2)))&(((ts_33_0|((!ts_31_0)|(!ts_32_0)))&(((!ts_33_0)|
    ts_31_0)&((!ts_33_0)|ts_32_0)))&((((!ts_34_0)|((!ts_33_0)|APS_B_Fault0))&((ts_34_0|
    ts_33_0)&(ts_34_0|(!APS_B_Fault0))))&(((ts_35_0|((!ts_29_0)|(!ts_34_0)))&(((!ts_35_0)
    |ts_29_0)&((!ts_35_0)|ts_34_0)))&ts_35_0))))))))))))))))))))))))))))))))))))))))
ENCODING TIME: 0.013116s
RESULT: SATISFIABLE
SOLVE TIME: 0.000307s
```

Falsifiability

```
INPUT: (!(((((((((A_10&(G[1,1] ((!A_10)&E0)))&(G[1,1] (G[1,1] ((!A_10)&E0))))->
    APS_A_Fault0)&(((A_20&(G[1,1] ((!A_20)&E0)))&(G[1,1] (G[1,1] ((!A_20)&E0))))->
    APS_A_Fault0))&(((A_20&(G[1,1] ((!A_30)&E0)))&(G[1,1] (G[1,1] ((!A_30)&E0))))->
```

```
    APS_A_Fault0))&(((B_10&(G[1,1] ((!B_10)&E0)))&(G[1,1] (G[1,1] ((!B_10)&E0))))->
    APS_B_Fault0))&(((B_20&(G[1,1] ((!B_20)&E0)))&(G[1,1] (G[1,1] ((!B_20)&E0))))->
    APS_B_Fault0))&(((B_30&(G[1,1] ((!B_30)&E0)))&(G[1,1] (G[1,1] ((!B_30)&E0))))->
    APS_B_Fault0)))
NON-TEMPORAL: (!(((((((((A_10&((!A_11)&E1))&((!A_12)&E2))->APS_A_Fault0)&(((A_20&((!A_21)&
    E1))&((!A_22)&E2))->APS_A_Fault0))&(((A_20&((!A_31)&E1))&((!A_32)&E2))->APS_A_Fault0)
    )&(((B_10&((!B_11)&E1))&((!B_12)&E2))->APS_B_Fault0))&(((B_20&((!B_21)&E1))&((!B_22)&
    E2))->APS_B_Fault0))&(((B_30&((!B_31)&E1))&((!B_32)&E2))->APS_B_Fault0)))
TSEITIN: (((ts_1_0|(A_11|(!E1)))&(((!ts_1_0)|(!A_11))&((!ts_1_0)|E1)))&(((ts_2_0|((!A_10)
    |(!ts_1_0)))&(((!ts_2_0)|A_10)&((!ts_2_0)|ts_1_0)))&(((ts_3_0|(A_12|(!E2)))&(((!
    ts_3_0)|(!A_12))&((!ts_3_0)|E2)))&(((ts_4_0|((!ts_2_0)|(!ts_3_0)))&(((!ts_4_0)|ts_2_0
    )&((!ts_4_0)|ts_3_0)))&((((!ts_5_0)|((!ts_4_0)|APS_A_Fault0))&((ts_5_0|ts_4_0)&(
    ts_5_0|(!APS_A_Fault0))))&(((ts_6_0|(A_21|(!E1)))&(((!ts_6_0)|(!A_21))&((!ts_6_0)|E1)
    ))&(((ts_7_0|((!A_20)|(!ts_6_0)))&(((!ts_7_0)|A_20)&((!ts_7_0)|ts_6_0)))&(((ts_8_0|(
    A_22|(!E2)))&(((!ts_8_0)|(!A_22))&((!ts_8_0)|E2)))&(((ts_9_0|((!ts_7_0)|(!ts_8_0)))
    &(((!ts_9_0)|ts_7_0)&((!ts_9_0)|ts_8_0)))&((((!ts_10_0)|((!ts_9_0)|APS_A_Fault0))&((
    ts_10_0|ts_9_0)&(ts_10_0|(!APS_A_Fault0))))&(((ts_11_0|((!ts_5_0)|(!ts_10_0)))&(((!
    ts_11_0)|ts_5_0)&((!ts_11_0)|ts_10_0)))&(((ts_12_0|(A_31|(!E1)))&(((!ts_12_0)|(!A_31)
    )&((!ts_12_0)|E1)))&(((ts_13_0|((!A_20)|(!ts_12_0)))&(((!ts_13_0)|A_20)&((!ts_13_0)|
    ts_12_0)))&(((ts_14_0|(A_32|(!E2)))&(((!ts_14_0)|(!A_32))&((!ts_14_0)|E2)))&(((
    ts_15_0|((!ts_13_0)|(!ts_14_0)))&(((!ts_15_0)|ts_13_0)&((!ts_15_0)|ts_14_0)))&((((!
    ts_16_0)|((!ts_15_0)|APS_A_Fault0))&((ts_16_0|ts_15_0)&(ts_16_0|(!APS_A_Fault0))))
    &(((ts_17_0|((!ts_11_0)|(!ts_16_0)))&(((!ts_17_0)|ts_11_0)&((!ts_17_0)|ts_16_0)))&(((
    ts_18_0|(B_11|(!E1)))&(((!ts_18_0)|(!B_11))&((!ts_18_0)|E1)))&(((ts_19_0|((!B_10)|(!
    ts_18_0)))&(((!ts_19_0)|B_10)&((!ts_19_0)|ts_18_0)))&(((ts_20_0|(B_12|(!E2)))&(((!
    ts_20_0)|(!B_12))&((!ts_20_0)|E2)))&(((ts_21_0|((!ts_19_0)|(!ts_20_0)))&(((!ts_21_0)|
    ts_19_0)&((!ts_21_0)|ts_20_0)))&((((!ts_22_0)|((!ts_21_0)|APS_B_Fault0))&((ts_22_0|
    ts_21_0)&(ts_22_0|(!APS_B_Fault0))))&(((ts_23_0|((!ts_17_0)|(!ts_22_0)))&(((!ts_23_0)
    |ts_17_0)&((!ts_23_0)|ts_22_0)))&(((ts_24_0|(B_21|(!E1)))&(((!ts_24_0)|(!B_21))&((!
    ts_24_0)|E1)))&(((ts_25_0|((!B_20)|(!ts_24_0)))&(((!ts_25_0)|B_20)&((!ts_25_0)|
    ts_24_0)))&(((ts_26_0|(B_22|(!E2)))&(((!ts_26_0)|(!B_22))&((!ts_26_0)|E2)))&(((
```

```
    ts_27_0|((!ts_25_0)|(!ts_26_0)))&(((!ts_27_0)|ts_25_0)&((!ts_27_0)|ts_26_0)))&((((!
    ts_28_0)|((!ts_27_0)|APS_B_Fault0))&((ts_28_0|ts_27_0)&(ts_28_0|(!APS_B_Fault0))))
    &(((ts_29_0|((!ts_23_0)|(!ts_28_0)))&(((!ts_29_0)|ts_23_0)&((!ts_29_0)|ts_28_0)))&(((
    ts_30_0|(B_31|(!E1)))&(((!ts_30_0)|(!B_31))&((!ts_30_0)|E1)))&(((ts_31_0|((!B_30)|(!
    ts_30_0)))&(((!ts_31_0)|B_30)&((!ts_31_0)|ts_30_0)))&(((ts_32_0|(B_32|(!E2)))&(((!
    ts_32_0)|(!B_32))&((!ts_32_0)|E2)))&(((ts_33_0|((!ts_31_0)|(!ts_32_0)))&(((!ts_33_0)|
    ts_31_0)&((!ts_33_0)|ts_32_0)))&((((!ts_34_0)|((!ts_33_0)|APS_B_Fault0))&((ts_34_0|
    ts_33_0)&(ts_34_0|(!APS_B_Fault0))))&(((ts_35_0|((!ts_29_0)|(!ts_34_0)))&(((!ts_35_0)
    |ts_29_0)&((!ts_35_0)|ts_34_0)))&(((ts_36_0|(!ts_35_0))&((!ts_36_0)|ts_35_0))&ts_36_0
    ))))))))))))))))))))))))))))))))))))))))
ENCODING TIME: 0.012680s
RESULT: SATISFIABLE
SOLVE TIME: 0.000304s
```