

On Integrating Runtime Verification in F' Architectures

By

Roman Riesen

Bachelor Thesis

Software Engineering Group

Department of Computer Science

Philosophisch-Naturwissenschaftliche Fakultät

Universität Bern

September 2022

Supervision:

Dr. PD Christos Tsigkanos

Prof. Dr. Timo Kehler

Abstract

Reliability and safety requirements are especially pronounced in the case of aerospace, robotic, and other critical systems. Space software, in particular, has historically enjoyed prime applications of formal methods. Such software is carefully designed, implemented, and verified against requirements before launch and deployment. Runtime verification is a flight-proven, lightweight method based on observing information from a system while in operation, and identifying if observed behaviors satisfy or violate certain requirements. It scales well in complex systems, and verdicts of monitored requirements may also be used for control decisions throughout the system operation. However, applications of runtime verification within space software have largely been residing at the implementation level.

In this thesis, we investigate the development of runtime verification support at the architectural level. We specifically target JPL's F', a framework enabling rapid development and deployment of spaceflight and other embedded software applications. We follow a principled approach, with the overall goal to enable existing F' systems to integrate monitoring of requirements expressed as temporal logic statements in a reusable manner. We illustrate runtime verification over a rocket model, and outline design decisions as well as supporting implementation adopting Metric Temporal Logic for specification and Reelay monitors for verification.

Acknowledgements

I wish to thank Dr. Tsigkanos for his ever-present enthusiasm and support as well as Prof. Kehrer for all the accommodations one could hope for. Further, I am grateful to my family for all their support.

Contents

1	Introduction	1
1.1	Problem Setting	1
1.2	Framework Support for Runtime Verification	2
1.3	Contribution	2
1.4	Related Work	2
2	Background	4
2.1	Motivating Example	4
2.2	A walk through F'	5
2.2.1	Workflow	5
2.2.2	Operating System Abstractions	6
2.2.3	Components	6
2.2.4	Ports	6
2.2.5	Ground Station	7
2.2.6	FPP	7
2.2.7	SysML and MagicDraw	9
2.3	Temporal Logic	9
2.3.1	LTL	10
2.3.2	MTL	10
2.3.3	Runtime Verification	10
2.3.4	Reelay	11
2.3.5	RYE-format	11
2.3.6	C++	12
3	Runtime Verification for F'	14
3.1	Monitor Adapter Pattern	14
3.2	ReelaySensor	14
3.3	ReelayMonitor	16
3.4	Design Rationale	18
3.5	Limitations	19
3.6	Towards A Generic Sensor	19
3.7	Alternative designs	20
4	Implementation & Deployment	22
4.1	Rocket Simulation component	22
4.2	Build system	23
4.3	Workflow	25
4.4	Workflow Example	25
4.5	Workflow Example Outputs	28
5	Discussion	29
5.1	Conclusions and Outlook	29

List of Figures

1	The height of the rocket over time as plotted by the F' ground system (introduced later)	5
2	Multiplication Error Event	7
3	A Pull-Based, Static ReelaySensor Component Diagram	15
4	ReelayMonitor Component Diagram	16
5	ReelayGenericSensor Component Diagram	20
6	Generic Sensor Example	21
7	RocketSimulator Component Diagram	22
8	Some values of the rocket plotted (acceleration, burnrate, remaining fuel) over time)	23
9	Speed and speed difference of the rocket	24
10	Rocket Simulation Topology	26
11	Screenshot of the ground system Events overview showing all properties passing, with the logging of positive verdicts enabled	28
12	Event stream when not all properties positively verified	29

1 Introduction

1.1 Problem Setting

Growing demand for software-heavy systems of growing complexity and working in growing scopes leads to an increase of software practitioners requiring accessible ways of achieving dependability. The requirements of reliability and safety are especially pronounced in the case of aerospace systems, robotics, and other critical systems as software errors might prove devastating to the missions in question.

Formal verification is the proving of the absence of incorrect behavior. Often this is achieved by model checking, which is traditionally done by exhaustively proving or checking that all states of a program are permitted by a specification. Model checking of a whole system is often either infeasible or too costly. Deductive formal verification approaches in which mathematical proofs are constructed to show the correctness of the program require expertise and are also very work intensive. And whilst more lightweight deductive proof techniques such as dependent typing have made big strides they are still far from production ready.

Testing may be relatively cheap but the guarantees provided are weak as the absence of insecure states cannot be proven and they rely heavily on an engineer's ability to notice potential edge-cases. Rare events might go ignored or unrecognized. Whilst fuzzy-testing does help in regards to rare-events testing it still lacks the hardness of the guarantees given by formal methods.

Runtime verification (RV) can offer a tradeoff between testing and formal verification. RV does not require a model. But it still allows for a very declarative way to define the required properties and check them upon a trace of system events occurring at system operation. Because not the entire state of the program is verified RV can also be used in bigger projects as the verification time scales super-linearly in model checking.

Advantages are further that hardware faults may be detected by the same properties that are used to verify software correctness and thus control sequences may be aborted before damage is done without much additional code. This holds too for the detection of potentially malicious commands or unforeseen emergent behavior. The verdicts of the monitored properties can also be used in the normal control flow decisions as the temporal logic can be quite a bit more expressive than ad-hoc detection of event sequences. Thus even if model-checking is possible one might still desire runtime verification.

Another important issue regarding the growth in software-heavy embedded systems is the ability for code reuse and maintainability. As the hardware can be limiting, commonly used abstractions of the wider software-engineering world are often avoided in favor of performance. This is where software architecture and supporting frameworks can be leveraged to great effect. Many tools try to use compile-time guarantees and constructs to simplify development and provide abstractions that allow the decoupling and encapsulation of components without incurring runtime costs. These two architectural goals enable teams to work more independently on different parts of the system, facilitate testing of the components themselves, and increase their reusability.

Creating abstractions to improve programmer productivity without strongly affecting runtime performance has been an ongoing concern and goal of software engineering since its beginning.

F' is a framework and architecture that, as we will see, fulfills these requirements.

1.2 Framework Support for Runtime Verification

The falling costs for bringing payloads to low earth orbit have resulted in a market for lower-cost space systems, such as so-called cubesats [37]. This development did lead to hardware and software that allows smaller and less well-funded teams to develop space systems. The interest in these low-cost space systems is both scientific and industrial.

In the following paragraphs we will introduce the tools used in the thesis and justify our choices.

One of these software systems is F', which is an architecture and framework developed by NASA's JPL for rapid development of flight software, but it can and has been used in general embedded systems as well. The architecture that F' encourages is one based on reusable components and connections thereof.

There are - as mentioned - already a great many uses of runtime verification in space systems. Reelay is a header-only library for runtime verification, which allows for online monitoring of temporal logic properties. It is fundamentally not dissimilar to a lot of other runtime verification frameworks that are commonly used by teams building critical systems.

Whilst there are fast and ready-to-use libraries providing RV based on temporal logic such as R2U2 [29] there seemingly has not been any work on providing reusable components for runtime verification. We believe the that advantages of such would align strongly with the use cases of the F' framework and fit well with its architecture.

Since the goal of F' is enabling engineers to quickly produce software for small-to-medium sized embedded systems having an easily reusable component for runtime verification of temporal properties is desired. There is work underway to easily integrate the SPIN model checker [13]. But as we reasoned in the previous chapter this would not fully remove the need or desire to support runtime verification. This thesis aims to produce such an RV component.

1.3 Contribution

The combination of tools and created software artifacts presented in this thesis have the following purposes and goals:

- A principled approach to enable existing F' systems to easily integrate monitoring of temporal logic statements
- Support engineers to use a declarative style to specify Instrumentation to monitor and the recognize of sequences of states
- Using F' enables both ease of deployment and moving verification monitors to a different physical chip from information creating/gathering chips, if performance concerns arise.

In the taxonomy of software engineering research laid out by [34] this thesis is part of the development and extension phase for runtime verification in F'.

1.4 Related Work

We could not find previous work that deals with the architectural aspects of reusable runtime verification. If it has it was mostly about achieving high-performance in an embedded context

and thus there is much focus on generating monitors in either hardware specification language or very fast C [30] [3] [[33]][38]. Or on improving reliability in the context of web services [19] [31]. Neither of which is fitting for our target of relatively high-performance embedded systems, such as those using ARM cortex chips [ArmCortexM4Processor] or similar.

In regards to implementing Runtime Verification in flight systems with small-to-medium scope, there is some work done by students [23] and older work by bigger teams [21]. So we focus here on alternatives to the individual tools used in this thesis, namely alternative flight-software or embedded-systems frameworks or runtime verification libraries.

Reelay, which is adopted in this thesis as the underlying verification engine is similar to R2U2 [33]. Thanks to its inclusion of bayesian networks TL properties can be more involved whilst still allowing to conclude the origin of the error.

If one were to use MagicDraw [7] with the SysML plug-in one can generate F' topologies in a diagramming software. Therefore this approach might superficially be a similar experience to using Simulink [12]. Both approaches are closed-source in nature and might thus not be applicable or desirable under certain circumstances. And using Simulink to generate F' Components might still require significant engineering effort, but might be an avenue worth exploring in future research. One downside of both these graphical approaches is the lack of suitability to commonly used version control systems of the underlying files.

Copilot is a language that enables hard-realtime verification and allows the formulation of specifications using a domain specific language based on haskell [3].

There are many more tools not explicitly listed here but which might be worth one's consideration for FSW development [18]. Many of the other solutions seem to lack the popularity and momentum of F'. And many of the tools developed by NASA can be seen as predecessors to F' but lack the cohesion and tools it provides.

2 Background

2.1 Motivating Example

As an example application, we will consider a simple rocket model of which we will build an F' component, simulating possible sensor outputs. We then consider a few properties that one might verify in a real system. The numbers and details chosen are in line with typical student competition rockets such as in [23].

The simulation is based on the classic Tsiolkovsky rocket equation[8] with random error terms applied - simulating measurement errors or differences from the theoretical ideal of the system - so that the system may violate the defined properties. Pressure effects are ignored.

The rocket has an accelerometer measuring acceleration (a_m), burn rate (b), fuel weight (m_f), exhaust velocity (v_e), and weight of the empty rocket (m_0). We also assume there is a ground station system that can measure the velocity of the rocket (via the doppler effect). This velocity we will call (v_d). The stage of the rocket is one of "Prelaunch", "FirstStage", "SecondStage", "Descent", "Landed".

We also let E_a, E_d, ϵ_b be independently normally distributed random variables with mean 0 and standard deviation $\sigma_a, \sigma_d, \sigma_b$, respectively. The first two random variables represent the variance of measurement of the accelerometer and of the ground-to-rocket doppler-effect measurement. The last variable is the variance of the burn rate of the engine.

The Tsiolkovsky rocket equation formulates the force on the rocket at a point in time (t) by the change of mass (fuel) multiplied by the exhaust velocity multiplied by the current rocket mass:

$$F = ma = (m_0 - bt)(v_e b)$$

From this we derive a few more variables and introduce the simulated errors and inaccuracies:

- a_r , the actual rocket acceleration given by $\frac{v_e(b+\epsilon_b)}{m_0-bt}$, where t is time. The maximal burn time being $\frac{m_f}{b}$
- $a_m = a_r + E_a$
- v_I , the velocity calculated by integrating the accelerometer data

A few of the properties we might want to specify are:

In the scenario presented, a system designer may be interested in checking if certain requirements hold at runtime, when the rocket system is in operation. For our purposes we will consider the following properties:

- The stage must start at "Prelaunch" and after that increase in order
- 1 second after FirstStage the burnrate must be more than $1800kg/s$
- The difference between v_I and v_d should always be less than $1m/s^2$ when the stage is either FirstStage or SecondStage

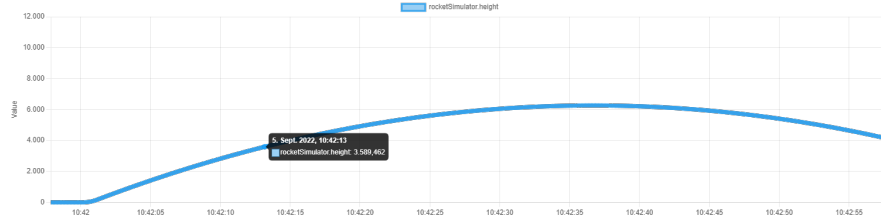


Figure 1: The height of the rocket over time as plotted by the F' ground system (introduced later)

2.2 A walk through F'

F' (F Prime) is an architecture, and framework in support of that architecture, which is being developed by NASA. Its stated goal is to enable component-driven rapid development and deployment of spaceflight and other embedded software applications [9]. F' takes a model-based approach to creating software; letting the developer express the high-level structure in a model which consists of a data structure and formal specification[25]. F' makes use of typed interfaces to provide compile-time guarantees and has built-in unit and integration testing facilities.

There exist a variety of flight-tested components already [4]. F' also offers operating-systems-like functionality and thus can be used cross-platform on both bare-metal and desktop systems which eases development further.

Successful real-world deployments of it exist, such as the Ingenuity mars helicopter [24].

What follows is a short introduction to the F' concepts necessary to understand the design and decisions in the rest of the thesis. For a more thorough introduction, there is an excellent overview by Bocchino et al [17]. There is a separate overview for the FPP modeling language (which was introduced in F' 3.0.0) by them as well [25].

2.2.1 Workflow

The main building blocks in F' are *components*, which have *ports*, which in turn are connected to each other by *connections* defined in *topology* definitions. The components are analogous to C++ classes; ports to their methods. Topologies do not have a one-to-one mapping to C++, they are both a way to structure bigger projects and a way to represent interaction.

The workflow of using F' is as follows:

1. Design a topology from the Requirements
2. Decide what existing components can be used and which have to be created
3. Specify the components the system requires (either in XML, FPP, or MagicDraw)
4. Specify how and what the components exchange via ports
5. From these specifications, C++ files are generated that already contain the necessary infrastructure and organization to quickly implement the business logic and receive commands.
6. Implement the C++ components
7. Generate and write a unit test for each component
8. Create a topology definition that instantiates components and connects these instances

9. Test the topology

2.2.2 Operating System Abstractions

By providing operating-systems-like functionality, such as thread management, synchronization primitives, and input and output, F' deployments can be run in a multitude of operating systems and bare-metal environments, easing development. However, the support of thread virtualization is considered experimental and should not be used in production [1].

2.2.3 Components

Components represent a physical or virtual part of the system. Such as sensors or logging systems. They provide encapsulation and thus should not communicate with other components, except by their connected *ports*.

The use of components is encouraged by the availability of tools for their creation, reuse, and testing. With the `fprime-util` suite of python tools, it is easy to create the C++ classes and corresponding tests from an FPP specification. This suite of tools also streamlines the building and running of components and deployments respectively.

Components are either **Active**, **Passive**, or **Queued**. This designation determines what synchronicity of invocations the component supports and how those are handled. **Active** and **Queued** components have buffers for their **input** ports. The former owns a thread on which invocations to its ports are processed. The latter needs the work to be done by an **Active** component. **Passive** components are exactly C++ classes and provide no queue or thread management. They are useful for data transformation between **Active** and **Queue** components. More on why we need this differentiation will be discussed in the next section.

As mentioned there are default components already available in F', such as the **ActiveLogger** which takes telemetry and events and logs them or converts them to telemetry, or the often used **RateGroupDriver**. This component can be thought of as a clock. its output is usually connected to an **ActiveRateGroup** component whose output in turn is used to schedule the code execution of components at a fixed rate. However, if the executed code takes longer than the rate, the next invocation will be delayed and a warning of high priority will be emitted. From component specifications, one can derive boilerplate for unit tests. This is done by creating a new CMake executable to which one adds the paths to the `.fpp` and `.cpp` component definition, as well as a `main` C++ file to drive the test. The derivation of that file is done by calling the `fprime-utils impl --ut` command that produces a test main function as well as the boilerplate for the component instantiation which is required by a test. Thus a lot of potential work is saved.

To ease testing and verification further there is an exploration [25] of directly incorporating the spin [13] model checker as well as a scenario-based testing specification system.

2.2.4 Ports

The interface of components is defined by their ports. Those are in turn defined by their type and their kind. The type of a port be zero or more F' or primitive types and can specify argument and return types. The kind of a port is either output or **input**.

Ports are also one of **synchronous/asynchronous/guarded**. These, in combination with the type of the component, determine how the work done by components on **input** port invocations is handled. **Passive** components may only have **synchronous** or **guarded** input ports as they lack a queue to store and process the invocation parameters. That would be required to later execute the invocation asynchronously.

Active and **Queued** component types must have at least either one **async** input port or **async** command.

Special types of ports represent the receiving of commands (from the ground station) as well as events and telemetry. Also, there are serial ports that can be connected with ports of any type.

Serial Ports can be connected to any type of port. The cost of this flexibility is the loss of compile-time safety of the interface type checking and the reconstruction of data from such a port is solely the programmer's responsibility. A side-effect of the architectural decision to only allow communication via ports is that it was relatively easy to create patches to ignore broken sensor input and replace it with something reasonable in the mars helicopter mission [10].

Event Ports are used to communicate notifications or warnings and they can contain data. For example:

```
event OVERFLOW_DETECTED(var1: U32, var2: U32) \
    severity activity high \
    format "Overflow detected: {}, {}!"
```

would declare a warning event of high severity and two values, and in the ground station log would be displayed as in 2.

f32multiplier.OVERFLOW_DETECTED	ACTIVITY_HI	Tried to multiply 65536 and 65537
---------------------------------	-------------	-----------------------------------

Figure 2: Multiplication Error Event

2.2.5 Ground Station

F' provides a ground data system (GDS) that allows the dispatching of commands and displaying of telemetry received. GDS supports creating a dashboard - a custom GUI layout - from predefined GUI components, which are intended to give a quick overview of the data available to the GDS from a running embedded system [5]. The GDS is implemented using web technologies and opens a web interface on the ground station computer thus it can run on the same computer as the F' deployment or on a separate one.

2.2.6 FPP

FPP is a modeling language that compiles to XML and C++ and has a more succinct syntax compared to writing the component and topology specifications by hand in either XML or C++[6]. Thus FPP furthers the main goals of rapid development of the F' ecosystem. Its parser also has better error reporting than the XML parser. Since version 3.0.0 F' comes with the FPP tools by default.

There is a plugin to support development inside Visual Studio Code [28], though it currently does not much other than code-highlighting ¹.

The syntax is c-like and sparse but expressive. For example, a component could be defined as follows (in e.g. `f32adder.fpp`):

```
@ a component for adding numbers
passive component F32Adder {
    constant port_nr = 2
    @ Input
    sync input port valueIn: [port_nr] F32
    @ Output
    output port valueOut: F32
    @ Telemetry
    telemetry port tlm

    # Telemetry channel definition
    telemetry VALUE : F32
}
```

The lines starting with `@` will be translated into comments in the generated C++ file whilst the lines starting with `#` are proper FPP comments and will be stripped.

The component in turn would be instantiated and named `f32adder` like this (usually a separate fpp file named `instances.fpp` is used):

```
instance f32adder: F32Adder base id 0xE00
```

the `base id` is used as an offset for the telemetry channels.

The instance could then be connected to other instances in a topology definition file (e.g. `topology.fpp`):

```
connections f32example {
    f32generator1.valueOut -> f32adder.valueIn[0]
    f32generator2.valueOut -> f32adder.valueIn[1]
    f32adder.valueOut -> f32consumer
}
```

where `f32generator1`, `f32generator2`, and `f32consumer` are assumed to be instantiated somewhere else.

The final steps of the workflow are adding the `.fpp` files of the components and other type or port definitions to the `cmake` build system. Afterward one can then use the `fprime` tools to generate the boilerplate code for the topology and the component. In the generated `.cpp` and `.hpp` files, one can then define what happens exactly when another component *invokes* (or calls) a port (by implementing the corresponding method).

¹But it does it well and even sets the parts of `.fpp` files that contain C++ correctly in C++

2.2.7 SysML and MagicDraw

This is a combination of tools that allows for the graphical drawing of F' topologies. It was unavailable to us during the writing of this thesis thus we cannot relate how well the Reelay component integrates with that environment - or the environment in general. But there seems to be criticism to this combination of tools by experienced F' practitioners [25], highlighting the non-trivial increase in complexity and the lack of ability to generate the ground system dictionary, which contains the data specifying what commands, parameters and telemetry channels are available (this is done automatically by the `fpp-compile` tool thus detail it is not mentioned otherwise). For the above reasons we decided to not elaborate on the impact of our components onto this toolchain.

2.3 Temporal Logic

Whilst the history of combining temporal properties with what we today call propositional logic and the resulting questions of valuations of sentences have their roots in ancient Greece (namely Aristotle and Diodorus Cronus in the third- to second-century B.C.E., e.g. the *master argument* of the latter [32]), we will focus on the more formal work developed in the middle of the 20th (C.E) century.

We call any logic that can express temporal behavior *temporal logic*. In the narrow sense, these have their roots in modern analytical philosophy [22] and later found many applications in formal verification and as mentioned in runtime verification. Initially, these were *branching time* temporal logics, which view time as a tree structure where the future is one path of many on that tree and the past is known and is thus a linear graph. As algorithms for determining satisfiability of linear complexity in formula size as well as model size exist for CTL they have become prevalent in model checking [26]. But CTL suffers from a lack of expressivity (later remedied by CTL*)[22] such that formulas in CTL can be exponentially bigger than LTL.

Much later on TLA⁺ was developed by Leslie Lamport, which builds on top of first order logic and set theory, thus being much more expressive than the other logics treated here. This system is widely used in industry [15].

2.3.1 LTL

Linear Temporal Logic is an extension of classic propositional logic by operators representing a notion of time. This is achieved by having a sequence of assignments for the propositional variables, W_i ($\forall i \in \mathbb{D}$) where \mathbb{D} is the time domain, which has a total order and can either be dense² or discrete. As the name implies this kind of logic considers the future to be fixed in the sense of there existing only one branch that can be taken, with the past being a line (the path taken so far)[22].

Some common operators are defined as follows for the variable p at time point j on a discrete time domain:

$\bigcirc p_j$	Next	p_{j+1} must be true (only exists in discrete time)
$\Diamond p_j$	Eventually	$\exists i \geq j$: such that p_i holds
$\Box p_j$	Henceforth	$\exists i \geq j$: such that $\forall k \geq i$: p_k holds
$p_j U p_k$	Until	if $\exists k : p_k$ then $\forall i \leq k$: p_i else \perp
Gp_j	Globally	$\forall i \in \mathbb{D} p_j$ holds

2.3.2 MTL

Metric Temporal Logic is similar to LTL with an addition of a metric for time, enabling the expression of durations. The operators thus take some form of duration specification, usually in the form of an interval, we write $\text{op}[t0:t1]$ with $t0, t1 \in \mathbb{D}$ for the metric version of LTL operator op , which is only checked on the interval from time point $t0$ to $t1$. Not all LTL operators have a reasonable metric version, in the above table \bigcirc and G can not have such time restrictions as they already imply their interval (e.g. $\bigcirc p_j$ from $t0 = j + 1$ to $t1 = j + 1$ and Gp_j from $t0 = \inf \mathbb{D}$ to $t1 = \sup \mathbb{D}$).

2.3.3 Runtime Verification

The goal of runtime verification is to specify the desired behavior of a system and then check the adherence to that specification at runtime (online) or afterward (offline). We focus mainly on the former case. This is done by generating a *monitor* from the specifications which is then connected to the running system in such a way that it has access to all the data required to verify the specification. The monitor is run on a *trace* of a system. A trace is for, the purpose of our thesis, a totally ordered set of sets of variables and their values. From the specification, one derives *properties*, which describe a set of traces. A property is said to hold for a trace if the current trace is in the set defined by that property. The monitor checks if a given specification is satisfied on a trace. [16]

- *Monitorability* is the trait of a property to be determined from the underlying system. A trivial example of a property that is not monitorable in our rocket example is “the rocket’s angle may not exceed one degree for the first 90 seconds of the flight” because the data required is not part of the system.

²In the usual mathematical sense of *dense order*

- *Polarity* of a specification is whether it captures a state that is allowed or not allowed. For example, ” “The difference between the integrated speed and the speed measured by doppler effect exceeds 1m/s” has negative polarity as it specifies a state that is not allowed. We will assume positive polarity in this thesis.

2.3.4 Reelay

Reelay is a library that provides runtime monitors for past metric temporal logic (pmtl) [35]. It enables the efficient monitoring of dense time metric logic by using a non-pointy (based on durations) definition. One of the reasons for the achieved performance is the omission of future-time operators as they were introduced in LTL. Thus only statements about the past are possible.

The Reelay monitor can either be discrete or dense. In the discrete case the total order mentioned previously is the only notion of time available. But in the dense case a `time` variable is expected to be part of the trace. The value of that variable may correspond to physical units such as milliseconds (the unit we chose for our implementation). When using a dense monitor the output of the monitor isn’t a single boolean value but a set of results as the verdict may change multiple times between two measurement time points [36].

The pre-defined temporal operators are:

<code>pre {A}</code>	<code>Y {A}</code>	in discrete time: {A} was true at now-1, not available in dense time
<code>once {A}</code>	<code>P {A}</code>	Property {A} was true in at least one point in time ($t < \text{now}$)
<code>once[t0:t1] {A}</code>	<code>P[t0:t1] {A}</code>	Property {A} at least once true in between now-t0 and now-t1
<code>historically {A}</code>	<code>H {A}</code>	{A} true at all times in the past
<code>historically[t0:t1] {A}</code>	<code>H[t0:t1] {A}</code>	{A} true at all points between now-t0 and now-t1
<code>{A} since {B}</code>	<code>{A} S {B}</code>	true if {A} always true since {B}
<code>{A} since[t0:t1] {B}</code>	<code>{A} S[t0:t1] {B}</code>	true if {A} always true since {B} was true between now-t0 and now-t1

Where `t0`, `t1` are time bounds that can be omitted to get unbounded expressions and {A} and {B} are atoms. Now is the current time.

The monitoring is implemented using sequential networks as opposed to the more commonly used approaches of rewriting rules or büchi-automata.

2.3.5 RYE-format

RYE is the format used by Reelay for the specification of the properties to be verified [11]. Atoms evaluate to either `true` or `false` but they can be constructed more expressively than in propositional logic. The types of atoms allowed in the Rye are boolean literals, numerical

comparisons, and string equality. The operations of propositional logic `not`, `and`, `or`, and `->` behave as usual.

A few properties of our rocket example expressed in RYE are:

- P1: The stage must start at “Prelaunch” and after that increase in order

```
({stage: 'FirstStage'}    since not {stage: 'Prelaunch'})
or ({stage: 'SecondStage'} since not {stage: 'FirstStage'})
or ({stage: 'Descent'}    since not {stage: 'SecondStage'})
or ({stage: 'Landed'}     since not {stage: 'Descent'})
```

- P2: 1 second after ignition the burnrate must be more than 1800kg/s

```
{burnrate > 1800} since[0:1000] {stage: 'FirstStage'}
```

- P3: The difference between v_I and v_d should always be less than 1m/s^2 during `FirstStage` and `SecondStage`. Because Reelay formulas do not support arithmetic operations we have to provide the absolute difference manually, we'll call it v_Δ , which we can then use:

```
({stage: 'FirstStage'} or {stage: 'SecondStage'}) -> {v_Delta < 10}
```

For the verification to happen we of course need some data. Reelay has chosen the ubiquitous JSON-format for that purpose. The properties above could be evaluated upon a trace - a sequence of events occurring at runtime. in practice, and recalling our rocket example, a JSON sequence of events would have elements of the following form:

```
{
  "time": 577495296, "burnrate": 2000, "v_I": 702.41217,
  "v_Delta": 5.256469, "stage": "FirstStage"
}
```

Note that if the monitor is chosen to be discrete-time, the time field is not needed and the temporal intervals correspond to the number of events passed.

Related to events are two features of Reelay [14]:

- *Condensing* refers to the ability of reelay to reuse events if only the time changes. This saves memory costs and decreases the number of events that need to be looked at when verifying a property.
- *Persistence* refers to the fact that if an event is missing some values compared to a previous event the values of the earlier event are reused.

2.3.6 C++

Because space systems that are targeted by F' generally strive for using as few resources as possible there are a few concessions to be made in the use of C++ features. As both RAM and ROM that are radiation-hardened cost magnitudes more than their terrestrial counterparts, features that increase the binary size (such as templates) are discouraged. Further, the use of the F' functions over `std`-library equivalents is encouraged for efficiency and bare-metal compatibility.

In the implementation, we tried to stick to the style and limited feature set suggested by [2]. Though there were quite a few concessions made in the name of quicker development time and

ergonomics, which due to Reelay not being written with these strict limitations in mind should not be too impactful a decision on either runtime performance or compile time.

3 Runtime Verification for F'

In a top-down approach, we will now present how the Reelay monitors were mapped to the concepts of F' and explain some of the design decisions and alternatives. For ease of understanding, we will assume there to be a component **PhysicalSensor** representing some sort of data generator, the output of which we wish to verify with Runtime Verification. Following our example this component will interchangeably be called **RocketSimulator** in diagrams and later sections

3.1 Monitor Adapter Pattern

One of the fundamental design challenges with using an off-the-shelve RV-library, like Reelay, is that to verify anything we first need to create inputs in a format that the library requires. In the case of Reelay that is - as mentioned - the JSON-format. As overhead is a major design concern of F' the more natural expression of such data therein would be binary streams, which are used liberally by component connections (the F' **serial** ports) and connections to external physical systems (using standard protocols like GPIO, UART). Thus there needs to be some sort of translation from the data-streams of sensors or F' provided components to the JSON-format required by Reelay. These generated events should then be validated at a fixed, and relatively high rate for the RV output to be useful.

One of the goals of the provided components is easy integration with existing components. Thereto we split the creation of events and the monitoring thereof into two components, called them **ReelaySensor** and **ReelayMonitor**. **ReelaySensor** creates the **ReelayEvents** from data gathered from **PhysicalSensor**. These events are then sent to the **ReelayMonitor** which is the actual verifier and has an output port for the verdicts. This decouples the data transformation from the data verification, which in turn makes it possible to have different implementations of **ReelaySensor** whilst fully reusing the **ReelayMonitor**. The **ReelaySensor** is therefore similar to the *Adapter* in an adapter pattern [20]. The **ReelaySensor** components can also have ports to forward the incoming data to other components. This is useful if the **ReelaySensor** is used as part of a larger, existing system and one wishes to create Reelay events to send to a monitor, but have the data generated still be available as if connected to the original destination. One may use one or more instances of each component in one topology and use the output of one instance of the **ReelaySensor** as the input of another instance. This allows for the mixing of multiple approaches to the sensor components as later illustrated.

3.2 ReelaySensor

Before diving into the details of both **ReelaySensor** and **ReelayMonitor** we will first give a quick justification of the diagram layout we have chosen to represent components. It is similar to the ubiquitous UML class diagram. But as we have seen previously F' components don't necessarily map one-to-one onto classes of object-oriented programming. Thus our diagrams have three categories of members instead of the typical two. The first one contains member variables and does specify their visibility: using + for public and - for private. The second category contains ports and subcategories noted in bold, such as **input** and **output**. The third, and last category contains the commands of the component. We note that the diagrams have neither an exact correspondence to the FPP definitions nor the C++ implementations and are not exhaustive. They aim to serve as illustration aids and are intended to be interpreted informally. Thus we also

omit some types as they can get quite verbose and are not relevant to the understanding of the design, or simplify port types by giving the underlying type instead of the full port type, e.g. if a port type is `ReelayEventPort (event : ReelayEvent)` we will write `ReelayEvent` as a type.

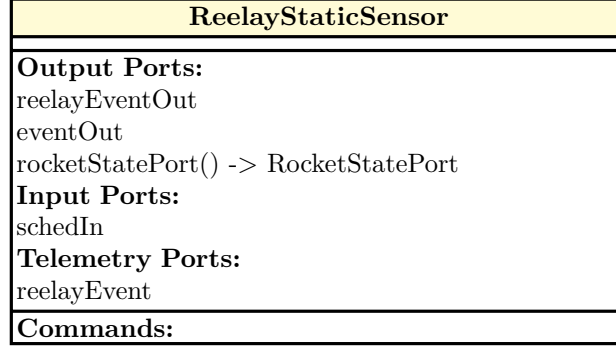


Figure 3: A Pull-Based, Static ReelaySensor Component Diagram

The `ReelaySensor` is most naturally represented by a `queued` component, which emits events on a schedule set by a `rateGroupDriver`. These events contain the most up-to-date data of all the actual sensors for which this `ReelaySensor` instance is responsible. But as both `queued` and `active` components add quite significant overhead to the complexity of the system, their use is discouraged on bare-metal. We will use a `passive` component to maximize the usefulness of our provided components. Further, thanks to *persistence* of the monitors provided by Reelay we don't have to accumulate the values in the `ReelaySensor` and can simply emit events even if they might not contain all the values.

In regards to the `ReelaySensor` there are two approaches discussed and one implemented in the finalized topology example:

1. Having the input ports be of type `serial` and having a port that then specifies the data types expected on that port, we call this `ReelayGenericSensor`.
2. Having the input ports be statically typed, we call this one `ReelayStaticSensor`.

Version (1) has the advantage of being more reusable at the cost of being type-safe. It is less safe because a discrepancy between the topology definition and the runtime setting of the port types can lead to garbage data being sent to the `ReelayMonitor`. Also one has to ignore the warnings about ports not being connected if there are fewer input ports than the specified number as there is no documented way to parametrically specify the number of ports of a component instantiation in FPP (or XML). This approach will be discussed later on in 3.6. The main issue with the `ReelayStaticSensor` is that one has to create a new F' component for every kind of `PhysicalSensor` one wants to use because the connections need matching port types on both ends. Therefore this version requires a new type of sensor for each kind of input one wishes to monitor, but it offers compile-time safety.

There is also the question of which component has the output port and which one has the input port. If the `ReelaySensor` has the input port we call it 'push based' as the component representing the physical sensor will have to invoke the port for data to travel to the `ReelaySensor`.

On the other hand, if the port on the **ReelaySensor** is an output port that has a return value, declared in FPP along the lines of `output port get_data() -> SensorData`, then we call it pull-based as the **ReelaySensor** invokes the port to get the data. If the physical sensor has many more changes than we wish to forward to the monitor we can use the pull-based approach, however, if we wish to have every single change visible at the monitor we can use the push-based approach. Regarding these two design decisions we have the following two-by-two matrix:

Pull Based	Push Based	
(2)		Typed
	(1)	'Runtime Typed'

There is a third, somewhat trivial approach - generating the **ReelayEvents** directly on the **PhysicalSensor** component, but as this is entirely deployment specific, we will not show it here. This approach is fine if the project intends to use the provided Reelay components from the beginning but it could be quite cumbersome to retrofit the corresponding ports into existing components. It also violates the single responsibility principle to an extent as the **PhysicalSensor** already has the purpose of providing data and transforming that data could be seen as orthogonal to that.

3.3 ReelayMonitor

ReelayMonitor
<ul style="list-style-type: none"> - properties - log_positive_verdicts : bool - monitor_options - monitors
Input Ports: eventIn : [5] string newProperty : ReelayProperty removeProperty : U32 time get timeGet Output Ports: verdictOut : ReelayVerdict verdictOutBool : bool C++ only methods: - add_property_impl(string, id, bool) + add_property(string, id)
Events: NO_PROPERTY_REGISTERED() INVALID_PROPERTY_STRING (property) INVALID_EVENT_STRING (reelay_event) NEW_PROPERTY_REGISTERED() NEGATIVE_VERDICT (property) ALL_PROPERTIES_POSITIVE()

Figure 4: ReelayMonitor Component Diagram

This component contains the monitors provided by the Reelay library and manages the verification of properties. The verdict of which is then sent onwards to consumers, such as telemetry, components that are part of the control system, or by way of a `ReelaySensor` to other `ReelayMonitor` instances.

The `ReelayMonitor` C++ implementation has a method `add_property` which registers the formulas to be verified. This can be used at runtime or at instantiation time like this:

```
instance reelayMonitor: Ref.ReelayMonitor base id 0xE000\
{
  phase Fpp.ToCpp.Phases.configComponents ""
    reelayMonitor.add_property("{stage: 'FirstStage'} \
      since not {stage: 'Prelaunch'}}", 0);
  ""
}
```

The `add_property_impl` method takes a string containing a Reelay formula and an id which can be used to delete the corresponding property (the management of the ids is up to the user). The method takes the property provided as a string and constructs a dense-time Reelay monitor from it. The third argument of type `bool`, `added_at_runtime`, is set to `true` if the property has been added by the `newProperty` port implementation and is `false` when called from `add_property`, which is the method used by users at instantiation. The `ReelayMonitor` does emit a fatal-priority event if the registered formula is invalid but only if `added_at_runtime` is `false`. If the formula was added at runtime the `ReelayMonitor` will simply ignore it and log a high-priority error. This is done to avoid the system crashing, but to make it obvious as close to compile-time as possible that one of the properties defined is invalid.

The Reelay monitors used in our implementation are densely timed because the rate of verification is given from outside the component and might change at runtime. Therefore events need to have a `time` variable. If that is missing from the JSON event to the `ReelayMonitor` that component will add one in the unit of milliseconds as given by the special time port `timeGet`. Because the dense time behaviour reelay would require stepping through each increase of the time value when updating a monitor we use dense time behaviour as that is more efficient when there are large jumps (around 100 steps) in the numerical value of time [14]. And since we want milliseconds as our base unit, to enable a great range of update rates, large jumps in time are not uncommon, when choosing update rates of 10Hz or less.

The difference between `verdictOut` and `verdictOutBool` is that the former supplies additional information about what property failed whilst the latter outputs `true` exactly if all properties registered on a monitor hold. The definition is as follows:

```
port ReelayVerdictOut (
  verdict : bool,
  property_id : U32
)
```

The `property_id` is meaningless

The monitor does the following on an `eventIn` port invocation from another component:

1. Check if the event contains valid JSON, if not a high-priority event `INVALID_EVENT_STRING` is sent out
2. If the JSON does not contain a `time` field, one is added based on the `getTime`. This time is in milliseconds.
3. Check all properties individually if they are satisfied by the current values. Satisfy means here that no value from the verdict-list returned by the relay monitor is `false`. If this is not the case a high-priority `NEGATIVE_VERDICT` event is sent out.
4. If all properties are satisfied a low-priority `ALL_POSITIVE` event may be sent out if `log_positive_verdicts` is true.

The reason we have five `eventIn` ports is due to limitations concerning the F' string capacity discussed later in 3.5.

3.4 Design Rationale

To communicate with a component one can either send it a command or invoke one of its ports. The first approach is mostly used for manually giving inputs to the system or high-level scripting by use of command sequences. The second way is used for communication inside the system. As we are designing components for reuse the balance to strike in regards to how much of a component is controlled by commands as opposed to input ports is non-obvious. The downside of having too many commands in pre-made components is making the command selection in the ground station cluttered and as the components are kept generic the command names might not be very useful to the issuer of commands, which may result in user error with possibly dangerous results. End-users can create components receiving commands and issuing the corresponding message to the provided components over ports easily, the main downside being more components and a more complex topology to maintain. In our view the downsides of generically named commands that may be confused by users outweigh the introduction of a slight increase in complexity, thus our provided components only use ports as inputs.

A similar issue exists in regards to communicating monitor verdicts from the `ReelayMonitor` to either other components or the GDS. One can have a telemetry port that sends the events directly to the ground station or an output port which communicates with other components. Relying only on the former approach has the downside that if such a verdict output port were not connected to a component communicating the violation of a property to the necessary places such errors might go entirely unnoticed during testing or simulation. This might prove catastrophic, later on. The downside of the former approach is that a lot of potentially unnecessary communication is created, resulting in wasted energy and a more busy ground station log. We used a hybrid approach of having both ports available. To avoid cluttering the logs the logging of positive verdicts can be turned off. But to avoid an error detected by RV going unnoticed the logging of a negative verdict is always logged with high priority. In doing so we assume that the properties of the system are expressed positively; that is if the result is `true` the system is assumed to work as intended in regards to the property covered by RV.

Whilst we thought about introducing a parameter to the property registration for a human-readable name (e.g. `speedDiffMaximum`) we decided against that because this can relatively easily be handled by an additional component managing the properties, therefore we don't need to impose the costs of those strings onto all users of the `ReelayMonitor` component. We use the dense-time behavior and do not provide an easy way to configure the options of the Reelay

monitor used. One of the reasons for that decision is that verification behavior might change in unexpected ways when changing to discrete-time behavior and the `time` variable might lose any physical meaning. Therefore we chose to hide the monitor setting details.

3.5 Limitations

As `F'` is intended to be usable on memory-limited hardware it generally avoids allocations at runtime thus the type `Fw::String` has a statically allocated buffer with a default capacity of 256 bytes (which can be adjusted by changing the macro declaration `FW_FIXED_LENGTH_STRING`). A result of this limitation is that one has to take care that the total size of one JSON event (sent from `ReelaySensor` to `ReelayMonitor`) fits into that size. The solution would have been to implement an abstract FPP-type JSON to pass the data between the sensor and monitor. `F'` requires anything sent over ports to inherit from the `Fw::Serializable` abstract class. This in turn means that the serialized maximal size of the serialized JSON object would have to be known at compile time. Therefore even this more elaborate approach would simply increase the limit (and decouple it from the `Fw::String`'s capacity, lowering the overhead added by increasing the limit of all strings used), not remove it entirely. But since the serialization may be in a binary format, the size of the JSON object could be reduced compared to the same object in a string format. On the other hand, because Reelay monitors themselves use the `nlohman` JSON library [lohmannJSONModern2022], which was not developed with the same restraint of runtime allocations in mind as `F'`, the monitors aren't suited to low-memory systems.

As a compromise, we rely on Reelay's persistence to remember all values it ever saw and provide more ports to which the JSON strings can be sent (`eventIn`). This automatically multiplies the total length of a verifiable event by the number of ports provided (five in our case).

3.6 Towards A Generic Sensor

As we have mentioned earlier, having a reusable, generic `ReelaySensor` would greatly ease the integration of our components into existing `F'` deployments. This section describes such a component, but due to challenges faced during the implementation, we have not been able to fully realize it. The analogous issue of having a user-definable way of declaring the type of an argument or member is solved in C++ by templates or inheritance, those are both not currently understood by FPP. But there exist `serial` ports which can be connected to any other type of port. In theory, this enables the creation of a generic Sensor, however, the details are not documented and we lacked the time to reverse engineer the underlying format, which would have been necessary to deserialize the data received by such serial ports. Thus this section has to be seen as an outline for future work.

An example instantiation of the generic Reelay sensor might look like this for receiving the data of our simulation component:

```
instance reelaySensor1: Ref.ReelayGenericSensor base id 0x0E200 \
{
  phase Fpp.ToCpp.Phases.configComponents ""
    reelaySensor1.set_data_channel_name(0, "v_I", Ref::ReelayDataType::F32);
    reelaySensor1.set_data_channel_name(1, "v_Delta", Ref::ReelayDataType::F32);
    reelaySensor1.set_data_channel_name(2, "burnrate", Ref::ReelayDataType::F32);
```

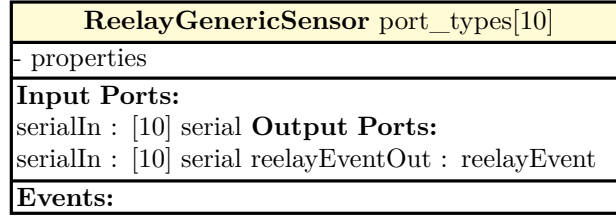



Figure 5: ReelayGenericSensor Component Diagram

```

reelaySensor1.set_data_channel_name(3, "stage", Ref::ReelayDataType::String);
reelaySensor1.set_data_channel_name(4, "height", Ref::ReelayDataType::F32);
"""
}

```

This sets up 5 serial ports, one for each data channel. The data channel names are used to identify the data in the JSON object sent to the monitor. The data type is used to deserialize the data received by the serial port. The data type is an enum containing the allowed types:

```

enum ReelayDataType {
    Bottom = 0,
    F32,
    Bool,
    String,
    ReelayEvent,
}
}

```

The bottom type is used to catch accidental default initializations of the data channel definitions. The **ReelayEvent** type is present to enable ReelaySensors to send events to other ReelaySensors, this might be useful if one wants to verify simple properties of a subset of a system's values with one monitor and more complex properties requiring the data of multiple sensors with another monitor, running at a lower rate.

The output serial ports can be used to simplify the integration into existing systems. The data received by the serial ports can be forwarded to other, already existing components (*Sensor Consumers*). As illustrated by figure 6, where the double dashed lines indicate **serial** port connections.

3.7 Alternative designs

As the properties can get quite long there might be a desire to work on files containing the specifications rather than on the individual strings thereof. This could at present be addressed by adding two components. A FileManager component instance would be responsible for the on-disk organization of the specification files and a provided component that loads the individual properties into the **ReelayMonitor** component from the FileManager.

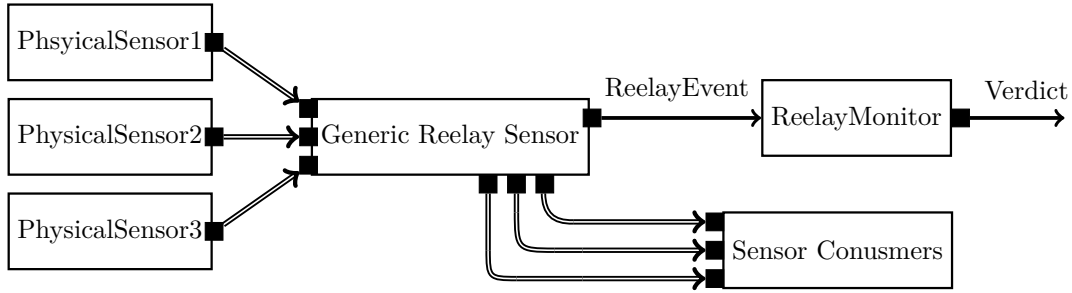


Figure 6: Generic Sensor Example

The monitor definition of the `ReelayMonitor` could be supplied at the FPP-component initialization by changing the constants used in the creation of the monitor. An instantiation of a dense-time monitor from the end-user perspective may be along the lines of:

```

instance reelayMonitor: Ref.ReelayMonitor base id 0xE000\
{
  phase Fpp.ToCpp.Phases.configConstants """
    enum {
      INPUT_T = U32,
      OUTPUT_T = U32,
      MONITOR_TYPE = ReelayMonitor::DiscreteTime
    }
    """
}

```

Whilst the performance overhead would be relatively minimal the complexity overhead for both the implementors (us) and the users was deemed too high, therefore we abstained from implementing this feature.

Another reasonable modification of the `ReelayMonitor` would have been to run the monitors at a fixed rate and not every time an `eventIn` port is invoked. This might reduce the overhead of the monitors, but would also reduce the usefulness of the monitors as they would not be able to react to events as they happen. In the worst case the verdict output would be delayed by the `rateGroup`'s schedule rate.

4 Implementation & Deployment

To show how the components interact and behave in an F' deployment we will show here the details of using the provided components to verify the properties of our rocket example.

4.1 Rocket Simulation component

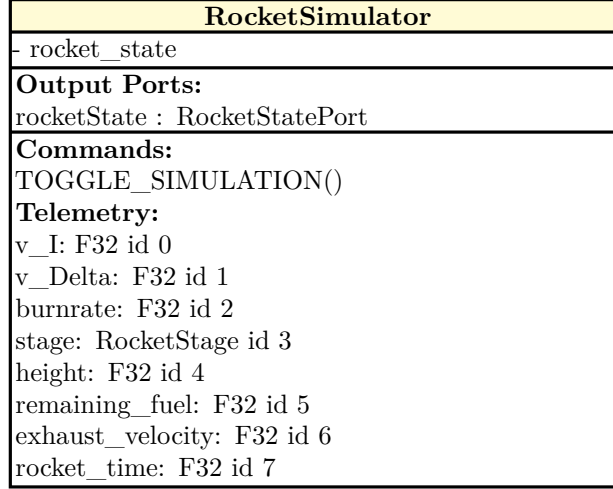


Figure 7: RocketSimulator Component Diagram

As mentioned in our case the only ‘physical sensor’ is the rocket simulation. This component is a naive implementation of the rocket model introduced earlier, with two stages. Once the first stage no longer has fuel for the next time step the second stage gets activated. Every time the scheduling port receives an event, the rocket simulation component updates the state of the rocket and invokes the output ports to send those to the connected component. The integration of the rocket equation is done by a left Riemann sum without regard to numerical stability. The purpose is solely to test the rest of the system. Care must be taken that the time step in the simulation corresponds to the time step of the to `schedIn` connected `rateGroup`. For simplicity this has to be done manually; if the update time value in `run1cycle` in `Top/Main.cpp` is the same numerical value as `rate_group_dt` the numerical time values correspond to milliseconds and the simulation runs in real-time. Then numerical interval values in the Reelay properties correspond to milliseconds. However the rocket time can be configured to move at an arbitrary rate, but care must be taken to change the interval definitions of the properties accordingly as the `ReelayMonitor`’s inserted time value is in milliseconds and the rocket simulation does not provide its internal time in our implementation. We have defined a macro, `BASE_SIMULATION_DELAY` in `RocketSimulator.hpp` to minimize the chance of confusion.

The `rocketState` output port is used to send the current state of the rocket to the `reelayRocketSensor` component when it is invoked (to enable a pull based sensor). It is defined by:

```
RocketStatePort (vI: F32, vDelta: F32, burnrate: F32, stage: RocketStage)
```

The telemetry ports are used to plot and observe the values from the simulator in the ground station system. There is a provided chart definition file called `rocket_charts.txt` in the `Ref` directory. Opening that from the **Charts** tab in the ground system will show plots of the height, acceleration, burnrate, etc. of the rocket. The large four large jumps correspond to the launch, the first stage burnout and the second stage ignition and burnout. The small jumps arise from the random terms from our model.



Figure 8: Some values of the rocket plotted (acceleration, burnrate, remaining fuel) over time)

The rocket model with normally distributed errors with mean 0 and $\sigma_a = 0.01$ and $\sigma_d = 0.5$ (same notation as in ??) has the following speeds and speed difference:

From 9 shows from top-to-bottom: v_I , v_Δ , v_d over time.

4.2 Build system

There is a non-trivial amount of CMake used by F' deployments to build deployments. This makes changes to the build settings - even for relatively simple tasks such as keeping debug symbols in the final binary - relatively challenging. Thus we will give a brief outline of how to adapt the CMake files for the deployment to include the Reelay components and their dependencies.

The F' build system is organized by heavy use of the CMake function `add_fprime_subdirectory(dir)`,

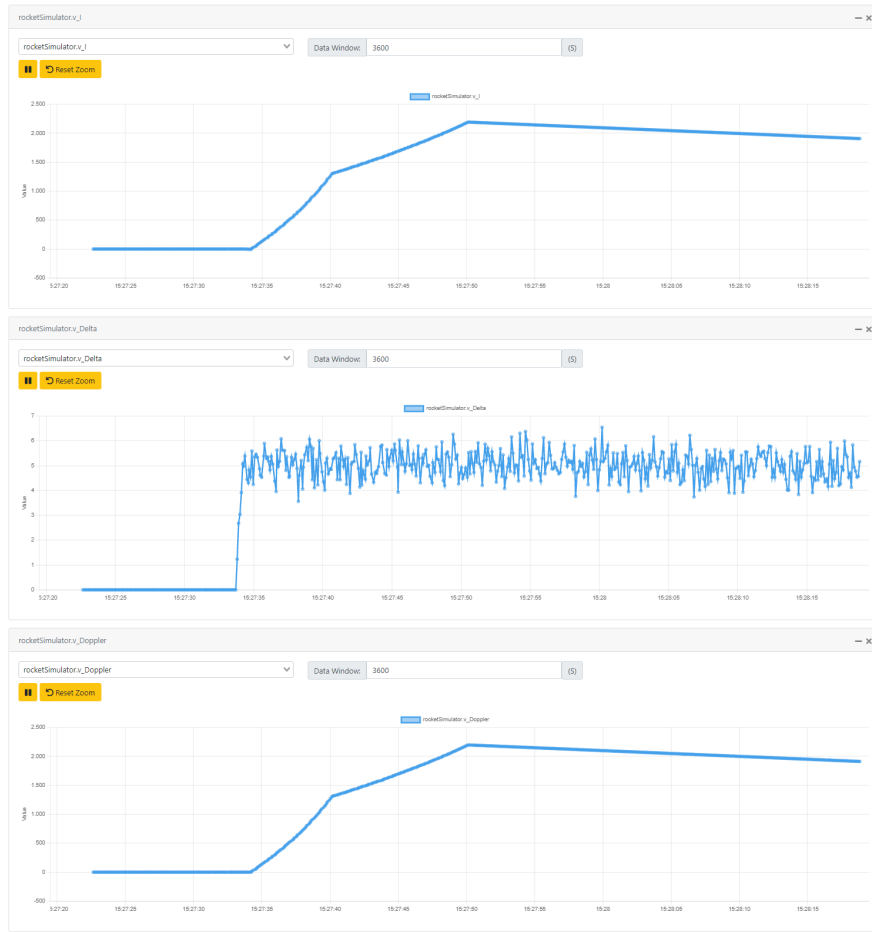


Figure 9: Speed and speed difference of the rocket

which takes the name of a directory and adds its content to the current build - handling F' specific things such as FPP component declarations and the test setups. Inside the specified directory the `add_fprime_subdirectory(dir)` function expects a `CMakeLists.txt` file containing usually a declaration of the `.fpp` and `.hpp` files to be included by this subdirectory and a call to register the current subdirectory as an fprime module (`register_fprime_module()`):

```
set(SOURCE_FILES
    "${CMAKE_CURRENT_LIST_DIR}/Component_specification.fpp"
    "${CMAKE_CURRENT_LIST_DIR}/Component_specification.hpp"
)

register_fprime_module()
```

There is a top-level `CMakeLists.txt` file that is responsible for including the nec-

essary modules. This is where a user of our components would have to add the line `add_fprime_subdirectory("${CMAKE_CURRENT_LIST_DIR}/ReelayComponents/")` where `ReelayComponents` is the directory which contains the `CMakeLists.txt` importing our provided components.

Reelay depends upon the *cudd* library for implementing the monitors [27]. Therefore we have to link any executables created by the build system with that library. First, we have to locate the library on the system of the user. This is achieved by using the CMake functions

```
add_library(CUDD_LIB STATIC IMPORTED GLOBAL)
find_library(CUDD_LIB NAMES libcudd)
```

in the `CMakeLists.txt` in `ReelayComponents` which, if one has installed *cudd*, will create a Cmake library referenced by the name `CUDD_LIB`. But then to link to the library we have to use the following in the highest-level `CMakeLists.txt` file (in the example this is the one located `example/Ref`):

```
target_link_libraries("${PROJECT_NAME}" PUBLIC "${CUDD_LIB}")
```

This is not great because it forces the user of our components to make two changes to that file. But due to lack of documentation, the complexity of the F' build system, and variable shadowing of the `${PROJECT_NAME}` in submodules we were unable to create a more elegant solution.

4.3 Workflow

The Workflow of monitoring an F' deployment can be summarised as follows:

1. Understand the system to be monitored and the properties to be verified
2. Specify the properties to be verified in natural language (e.g., the stages should increase monotonically).
3. Generate the Reelay formulas from the natural language specifications.
4. Determine where the necessary data originates and where it will be required; how many Monitors should there be? What kind of Reelay sensors should be used? Should the JSON events be created directly? Is there an opportunity to use temporal logic instead of ad-hoc methods for determining certain states?
5. Create the necessary components and connect them to satisfy the requirements in (4)
6. The result of (5) is a topology using our provided components to verify the specification from (2).
7. Run the deployment and observe the results.

4.4 Workflow Example

As there are a lot of technical details to get F' to interact with the ground station system. For sake of simplicity we assume that we take the `Ref` example deployment, which already implements these details and will extend it to contain the following topology:

We take the properties from the example deployment formulated in the RYE format directly from 2.3.5. With that and 2.1, we have steps (0), (1), (2) complete.

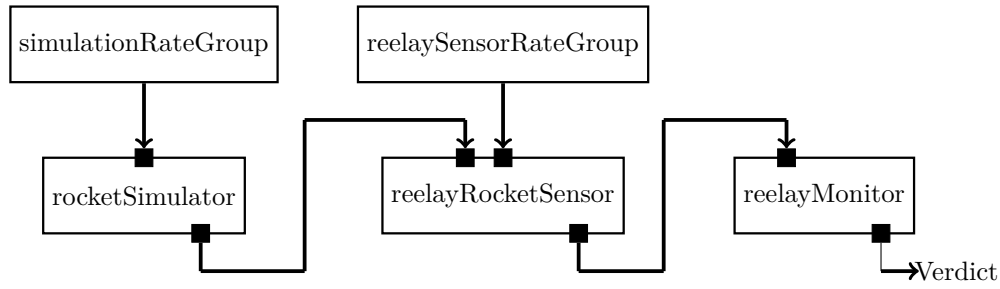


Figure 10: Rocket Simulation Topology

Now one needs to copy the the provided directory containing the Reelay components into /Ref/. Then add the necessary lines to the CMakeLists.txt mentioned in 4.2. Further add the following line to the Ref/CMakeLists.txt file to load the Reelay components:

```
add_fprime_subdirectory("${CMAKE_CURRENT_LIST_DIR}/Reelay/")
```

As we wish to demonstrate a minimum viable sensor and monitor setup we will have one ReelayMonitor and one ReelayStaticSimulator implemented for transforming the data of our rocket simulation into JSON-events. With that step (4) is complete. Because the Sensor is implementation specific we put it in the namespace Ref, just as the RocketSimulator.

To construct our topology and achieve step (5) we need to create instances of the components used so in the instances.fpp file in the /Ref/Top directory we add the following:

```

instance rocketSimulator: Ref.RocketSimulator base id 0xE400

instance reelayRocketSensor: Ref.ReelayRocketSensor base id 0x0E200

instance reelayMonitor: Reelay.ReelayMonitor base id 0xE000\
{
  phase Fpp.ToCpp.Phases.configComponents ""
  reelayMonitor.add_property("\
    ({stage: 'FirstStage'} since not {stage: 'Prelaunch'})\
    or ({stage: 'SecondStage'} since not {stage: 'FirstStage'})\
    or ({stage: 'Descent'} since not {stage: 'SecondStage'})\
    or ({stage: 'Landed'} since not {stage: 'Descent'})", 0);

  reelayMonitor.add_property("({stage: 'FirstStage'} or {stage: 'SecondStage'})\
    -> {v_Delta < 10}", 1);

  reelayMonitor.add_property("{burnrate > 1800} since[0:1000] {stage: 'FirstStage'", 2);
  ""
}

```

For simplicity again we assume the existence of two instances of `Svc.ActiveRateGroup` named `simulationRateGroup` and `reelaySensorRateGroup`.

Then in `topology.fpp` inside of `/Ref/Top` we add the following:

```
connections Monitoring {  
  
    connection simulationRateGroup.RateGroupMemberOut[0] -> rocketSimulator.schedIn  
    connection reelaySensorRateGroup.RateGroupMemberOut[0] -> reelayRocketSensor.schedIn  
  
    connection reelayRocketSensor.rocketState -> rocketSimulator.rocketState  
    connection reelayRocketSensor.reelayEvent -> reelayMonitor.eventIn  
}
```

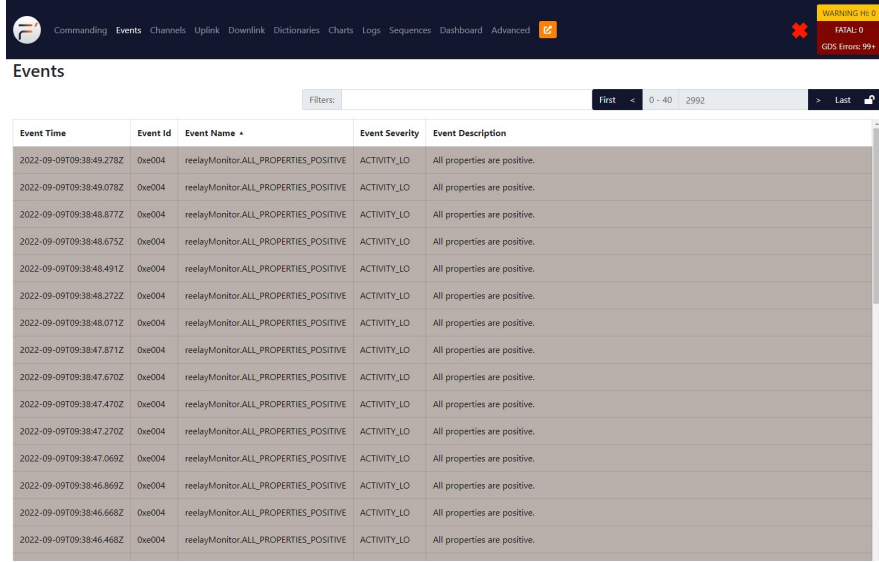
Finally we run `fprime-util generate` and `fprime-util build` to generate the necessary code and build the executable. The result is a binary that can be run by running `fprime-gds` in the `Ref` directory.

When starting `fprime-gds` the simulation is initially turned off so one can load the provided chart definitions to observe the simulation. When sending the command `TOGGLE_SIMULATION` the simulation is turned on and the rocket starts to move. Once the simulation is started it advances by time-steps `dt`. At every time step, new values for the rocket state are calculated.

To change the speed at which the simulation runs one needs to configure the cycle duration of the F' deployment. In our example deployment, this means changing the `run1cycle` function inside `Main.cpp` inside the `Top` folder.

4.5 Workflow Example Outputs

Figure 11 shows the event stream when the standard deviations of the simulation are chosen small, so the error never gets big enough to cause our properties to no longer be valid. So they all hold.



Event Time	Event Id	Event Name	Event Severity	Event Description
2022-09-09T09:38:49.278Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:49.078Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:48.877Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:48.675Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:48.491Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:48.272Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:48.071Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:47.871Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:47.670Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:47.470Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:47.270Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:47.069Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:46.869Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:46.668Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.
2022-09-09T09:38:46.468Z	One004	reelayMonitor.ALL_PROPERTIES_POSITIVE	ACTIVITY_LO	All properties are positive.

Figure 11: Screenshot of the ground system Events overview showing all properties passing, with the logging of positive verdicts enabled

Figure 12 shows the event stream when the standard deviations of the simulation are chosen large (here $\sigma_a = \sigma_b = 15$), the error gets big enough to cause our speed difference property (P3 from 2.3.5) to fail. The green events mark the simulation start command and the event informing us of the execution of the command.

Similarly for large enough values of σ_b the property P2 will fail. There is further information written to the log visible in the corresponding tab. For example the events being processed.

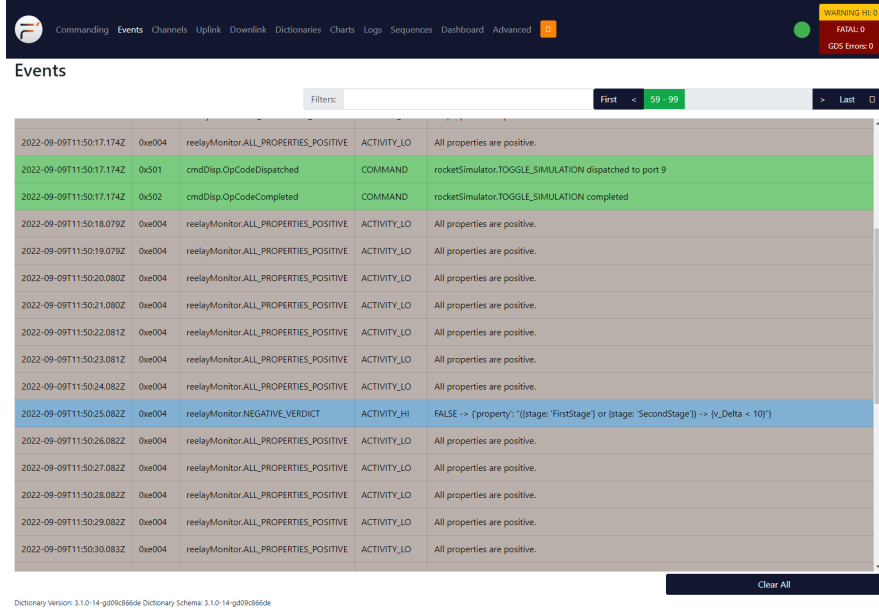


Figure 12: Event stream when not all properties positively verified

5 Discussion

As mentioned one of the issues with the monitor-design pattern in FPP is that there is no way for a user to define the type of data that is sent over the `serial` connections and have the compatibility of this declaration verified in regards to the type of data that is actually sent. This issue could be solved in stand-alone C++ using templated classes, but neither F' nor FPP have such a concept, though it would in our opinion greatly ease the creation of reusable components.

Whilst altering the provided components to use R2U2 was considered - and would be reasonably straightforward - it was deemed out-of-scope for this thesis. As mentioned in related_work R2U2 has some distinct advantages over Reelay but all the additional user functionality comes at the cost of complexity which is not coherent with the philosophy of rapid development present in F'. But an unmatched benefit of R2U2 is the ability to compile the monitors to Verilog and thus to FPGA's.

As F' allows for components to be physically distributed accross space (and thus time) the naive notion of time used by our monitor might not always suffice. But we deemed this out-of-scope for this thesis.

5.1 Conclusions and Outlook

We have shown how to build components for runtime verification for F' on the basis of Reelay and showed different approaches of how to integrate such a component. We have also implemented a fully reusable monitor component and shown its usefulness by providing a small example. This demonstrates the feasibility of creating more advanced and more reusable components in F'

and the potential of the monitor-design pattern for runtime verification in a component based architecture.

We also motivated and outlined future work towards a reusable Reelay event translator, a generic **ReelaySensor**. But this work also indicates the need for a more general solution to the problem of compile time checked parametric-like types in FPP.

Further work is required to test the performance implications of our components on embedded systems as well as show the feasibility of using our components in a more complex F' deployment. Another avenue for exploration might be how to integrate another Runtime verification framework such as R2U2 using our Architectural approach.

We also hope that this thesis and the accompanying code will serve as a starting point for others to explore the possibilities of runtime verification in F' or for the development of reusable F' components in general.

The code can be found on [github](#) or the [university of bern gitlab](#). The **thesis_submission** branch contains the software as submitted for grading whilst **main** contains the most up-to-date work.

Erklärung

gemäss Art. 30 RSL Phil.-nat.18

Name/Vorname: Riesen Roman

Matrikelnummer: 17-914-284

Studiengang: Informatik

Bachelor ☒

Master ☐

Dissertation ☐

Titel der Arbeit: On Integrating Runtime Verification in F' Architectures

LeiterIn der Arbeit: Dr. PD Christos Tsigkanos
Prof. Dr. Timo Kehrler

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist. Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern 9.9.2022

Ort/Datum

Roman Riesen

Unterschrift

- [1] F´ On Baremetal and Multi-Core Systems. Retrieved August 10, 2022 from <https://nasa.github.io/fprime/v3.1.0/UsersGuide/dev/baremetal-multicore.html>
- [2] Code and Style Guidelines. Retrieved July 24, 2022 from <https://nasa.github.io/fprime/UsersGuide/dev/code-style.html>
- [3] Copilot: home. Retrieved September 9, 2022 from <https://copilot-language.github.io/>
- [4] F´ Features. Retrieved July 23, 2022 from <https://nasa.github.io/fprime/features.html>
- [5] GDS Dashboard Component Reference. Retrieved August 10, 2022 from <https://nasa.github.io/fprime/v3.1.0/UsersGuide/dev/gds-dashboard-reference.html>
- [6] Home · fprime-community/fpp Wiki. Retrieved July 23, 2022 from <https://github.com/fprime-community/fpp>
- [7] MagicDraw - CATIA - Dassault Systèmes®. Retrieved July 28, 2022 from <https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/>
- [8] A Man – and an equation – Rocket Science. Retrieved September 6, 2022 from <https://blogs.esa.int/rocketscience/2012/10/14/a-man-and-an-equation/>
- [9] A More Complete Introduction To F´. Retrieved July 23, 2022 from <https://nasa.github.io/fprime/UsersGuide/user/full-intro.html>
- [10] NASA’s Mars helicopter has a problem. This clever software trick could fix it | ZDNet. Retrieved July 29, 2022 from <https://www.zdnet.com/article/nasas-ingenuity-mars-helicopter-has-a-problem-this-clever-software-trick-could-fix-it/>
- [11] Rye Format - Reelay. Retrieved July 29, 2022 from <https://doganulus.github.io/reelay/rye/>
- [12] Simulink - Simulation and Model-Based Design. Retrieved July 28, 2022 from <https://www.mathworks.com/products/simulink.html>
- [13] Spin - Formal Verification. Retrieved August 1, 2022 from <https://spinroot.com/spin/whatispin.html>
- [14] Temporal Behaviors - Reelay. Retrieved September 6, 2022 from <https://doganulus.github.io/reelay/behaviors/>
- [15] Yamine Ait Ameur and Klaus-Dieter Schewe (Eds.). 2014. *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg. DOI:<https://doi.org/10.1007/978-3-662-43652-3>
- [16] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification: Introductory and Advanced Topics*, Ezio Bartocci and Yliès Falcone (eds.). Springer International Publishing, Cham, 1–33. DOI:https://doi.org/10.1007/978-3-319-75632-5_1
- [17] Robert Bocchino, Timothy Canham, Garth Watney, Leonard Reder, and Jeffrey Levison. F Prime: An Open-Source Framework for Small-Scale Flight Software Systems. 19.
- [18] Sonja Caldwell. 2021. 8.0 Small Spacecraft Avionics. Retrieved July 28, 2022 from <http://www.nasa.gov/smallsat-institute/sst-soa/small-spacecraft-avionics>

- [19] Tien-Dung Cao, Trung-Tien Phan-Quang, Patrick Felix, and Richard Castanet. 2010. Automated Runtime Verification for Web Services. In *2010 IEEE International Conference on Web Services*, 76–82. DOI:<https://doi.org/10.1109/ICWS.2010.19>
- [20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. AddisonWesley Professional.
- [21] A. Goldberg, K. Havelund, and C. McGann. 2005. Runtime verification for autonomous spacecraft software. In *2005 IEEE Aerospace Conference*, IEEE, Big Sky, MT, USA, 507–516. DOI:<https://doi.org/10.1109/AERO.2005.1559341>
- [22] Valentin Goranko and Antje Rumberg. 2022. Temporal Logic. In *The Stanford Encyclopedia of Philosophy* (Summer 2022), Edward N. Zalta (ed.). Metaphysics Research Lab, Stanford University. Retrieved August 6, 2022 from <https://plato.stanford.edu/archives/sum2022/entries/logic-temporal/>
- [23] Benjamin Hertz, Zachary Luppen, and Kristin Yvonne Rozier. 2021. Integrating Runtime Verification into a Sounding Rocket Control System. In *NASA Formal Methods*, Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz and Ivan Perez (eds.). Springer International Publishing, Cham, 151–159. DOI:https://doi.org/10.1007/978-3-030-76384-8_10
- [24] <https://www.jpl.nasa.gov>. Meet the Open-Source Software Powering NASA’s Ingenuity Mars Helicopter. Retrieved July 23, 2022 from <https://www.jpl.nasa.gov/news/meet-the-open-source-software-powering-nasas-ingenuity-mars-helicopter>
- [25] Robert L Bocchino Jr, Jeffrey W Levison, and Michael D Starch. FPP: A Modeling Language for F Prime. 15.
- [26] Joost-Pieter Katoen. 5 Complexity Considerations. 11.
- [27] Ivan Maidanski. 2022. *Ivmai/cudd*. Retrieved September 7, 2022 from <https://github.com/ivmai/cudd>
- [28] Daniel Martindale. 2022. *FPPTools*. Retrieved July 24, 2022 from <https://github.com/p hxe/fpptools>
- [29] Patrick Moosbrugger, Kristin Y. Rozier, and Johann Schumann. 2017. R2U2: Monitoring and diagnosis of security threats for unmanned aerial systems. *Form Methods Syst Des* 51, 1 (August 2017), 31–61. DOI:<https://doi.org/10.1007/s10703-017-0275-x>
- [30] Ahmed Nassar, Fadi J. Kurdahi, and Wael Elsharkasy. 2015. NUVA: Architectural support for runtime verification of parametric specifications over multicores. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 137–146. DOI:<https://doi.org/10.1109/CASES.2015.7324554>
- [31] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (March 2015), 66–73. DOI:<https://doi.org/10.1145/2699417>
- [32] Peter Øhrstrøm and Per Hasle. 2020. Future Contingents. In *The Stanford Encyclopedia of Philosophy* (Summer 2020), Edward N. Zalta (ed.). Metaphysics Research Lab, Stanford University. Retrieved August 9, 2022 from <https://plato.stanford.edu/archives/sum2020/entries/future-contingents/>

- [33] Johann Schumann, Patrick Moosbrugger, and Kristin Y. Rozier. 2016. Runtime Analysis with R2U2: A Tool Exhibition Report. In *Runtime Verification*, Yliès Falcone and César Sánchez (eds.). Springer International Publishing, Cham, 504–509. DOI:https://doi.org/10.1007/978-3-319-46982-9_35
- [34] Mary Shaw. What Makes Good Research in Software Engineering? 10.
- [35] Dogan Ulus. 2019. Online Monitoring of Metric Temporal Logic using Sequential Networks. Retrieved July 23, 2022 from <http://arxiv.org/abs/1901.00175>
- [36] Doğan Ulus. 2022. *Reelay Monitors*. Retrieved September 7, 2022 from https://github.com/doganulus/reelay/blob/bc30e86f888e903e0521c90637147e21d6080dce/docs/gs_cpp.md
- [37] Thyrso Villela, Cesar A. Costa, Alessandra M. Brandão, Fernando T. Bueno, and Rodrigo Leonardi. 2019. Towards the Thousandth CubeSat: A Statistical Overview. *International Journal of Aerospace Engineering* 2019, (January 2019), 1–13. DOI:<https://doi.org/10.1155/2019/5063145>
- [38] Xi Zheng, Christine Julien, Hongxu Chen, Rodion Podorozhny, and Franck Cassez. 2017. Real-Time Simulation Support for Runtime Verification of Cyber-Physical Systems. *ACM Trans. Embed. Comput. Syst.* 16, 4 (November 2017), 1–24. DOI:<https://doi.org/10.1145/3063382>