SOFTWARE REQUIREMENTS SPECIFICATION

for

Control System Analysis Framework (CSAF)

Version 1.0

September 17, 2020

Contents

Revision History

DISCLAIMER: This is a rough draft. The document contains many errors, and the architecture is subject to change.

1 Introduction

1.1 Purpose

CSAF is a middleware environment for describing, simulating, and analyzing closed loop dynamical systems. A CSAF component's primary interface is zeroMQ subscribe and publish system components, which communicate input/output of a lump abstracted component as ROSmsgs. It allows component creation using common system representations, especially for specifying controllers. The architecture is supportive of systems to be simulated with diverse components, agnostic to the languages and platforms from which they were implemented.

1.2 Document Conventions

Being middleware, the interfaces are described in the serialization formats. The features relate to integration tasks that are likely to be employed by the user.

Figures are linked in blue to their corresponding content. Acronyms are documented and linked to a glossary at the end of the document, labeled in blue. Hyperlinks are labeled in purple.

Code snippets, pseudocode and message are written in fixed width font. See Figure 1.1 for example.

```
def main():
"""Example Code Snippet
"""
        return 0
```

Figure 1.1: Example Code Snippet

Math expressions are used throughout the document. They float as equations, and can be referenced. See Equation 1.1 for example.

$$\mathcal{F}_t\{g\}(f) = \int_{-\infty}^{\infty} \exp\left(-i2\pi ft\right) g(t)dt \tag{1.1}$$

1.3 Intended Audience and Reading Suggestions

This document is designed for planning CSAF features and its use in the Assured Autonomy project. Relevant team members are intended to read, review, and request changes to the planned architecture and scope. The material is intended to benefit the CSAF contributors and control designers using the project for the Assured Autonomy project, who will use the components, interfaces and components.

1.4 Project Scope

Its primary use is modeling systems that utilize shields with learning enable controllers (LECs). CSAF subsystems accommodate a variety of controller representation; for example, some analyses require explicit knowledge of the fuzzy inference being used. Rather than treating everything a black box, there is value in knowing whether a controller utilizes a linear control law, is produced from a network architecture, and so on.

Another use of CSAF is in multi-loop systems, where the control vector gets transformed before entering a plant. For the Ground Collision Avoidance System challenge problem being considered, this separates problems of maneuverability and attitude control. Because of this, exogenic and environmental description is pertinent. Path planning, collision avoidance and environmental sensing are all areas that CSAF will handle.

Some of the system description might be done in the eDSL CoPilot 3, which outputs a monitor existing as a shared object library. As such, some of the components might be present only as binaries, which are utilized for monitoring/controlling. Further, the platform is for projects where controllers are delivered by other projects, and so agnosticism to language/library/platform exists.

1.5 References

This document uses the IEEE 830 SRS standard, using a LaTeX template provided on GitHub.

2 Overall Description

2.1 Product Perspective

CSAF serves the role of a middleware for closed loop system simulations. A user starts with a model that they have for a system, with well-defined inputs, outputs and states. Shown in Figure 2.1, they will proceed to create a CSAF pub/sub component. The package comes with an API to make common system descriptions easy to implement, and has a wrappers if the system is a black box. The component is described using a small configuration file, and can be interacted with using a subscriber and a publisher. The pub/sub pair is for temporal input and output—input can be specified at a given time in order to receive output at the specified time.

These CSAF components can be composed together to create a closed loop system. Figure 2.2 shows the blocks composed under a system analyzer. In the scope of controls research, analyses can be created and performed on the modular CSAF components.

2.2 Product Functions

A user is able to

1. add model specifics of a dynamical system to create a compatible component
2. compose controllers together to make closed loop systems
3. simulate a closed loop systems efficiently
4. design custom analyses, and is supported with libraries and documentation

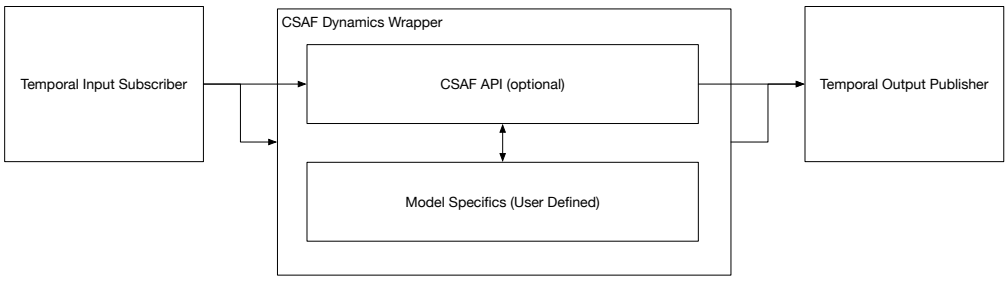

Figure 2.1: Structure of a CSAF Compatible Component

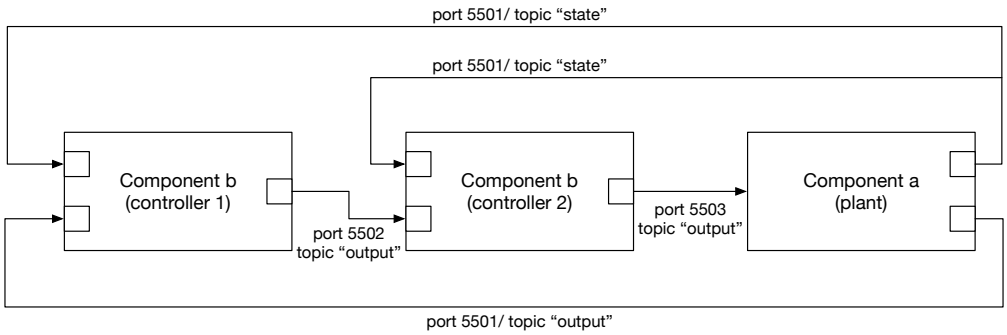

Figure 2.2: A closed loop system represented as pub/sub components. Control signals are broadcasted as a collection of messages with specific topics. The component processes the input, producing system output as a collection of messages with topics.

Figure 2.3 shows the user workflow for a systems analysis, from start to finish. All blocks in the control diagram must exist as components. Then, the components are composed together to achieve the controlled system. The system is ready to be simulated or analyzed.

2.3 User Classes and Characteristics

CSAF's user workflow is restricted to software developers, systems engineers and researchers. The users implementing components are required to have knowledge of programming and system theory. The workflow output is to provide results for a controls research project. As such, secondary users are project managers and reviewers.

2.4 Operating Environment

CSAF is implemented in modern python (3.5+), with specific packages for numerical/-controls computing. Also, it has optional package requirements, like Tensorflow and RosPy needed for specific features. Its primary use is to operate inside a debian based docker container for its delivery in the Assured Autonomy project, as well be shipped inside of a virtual machine. For full use, it will require a OS with CoPilot installed, needing Haskell and C toolchains.

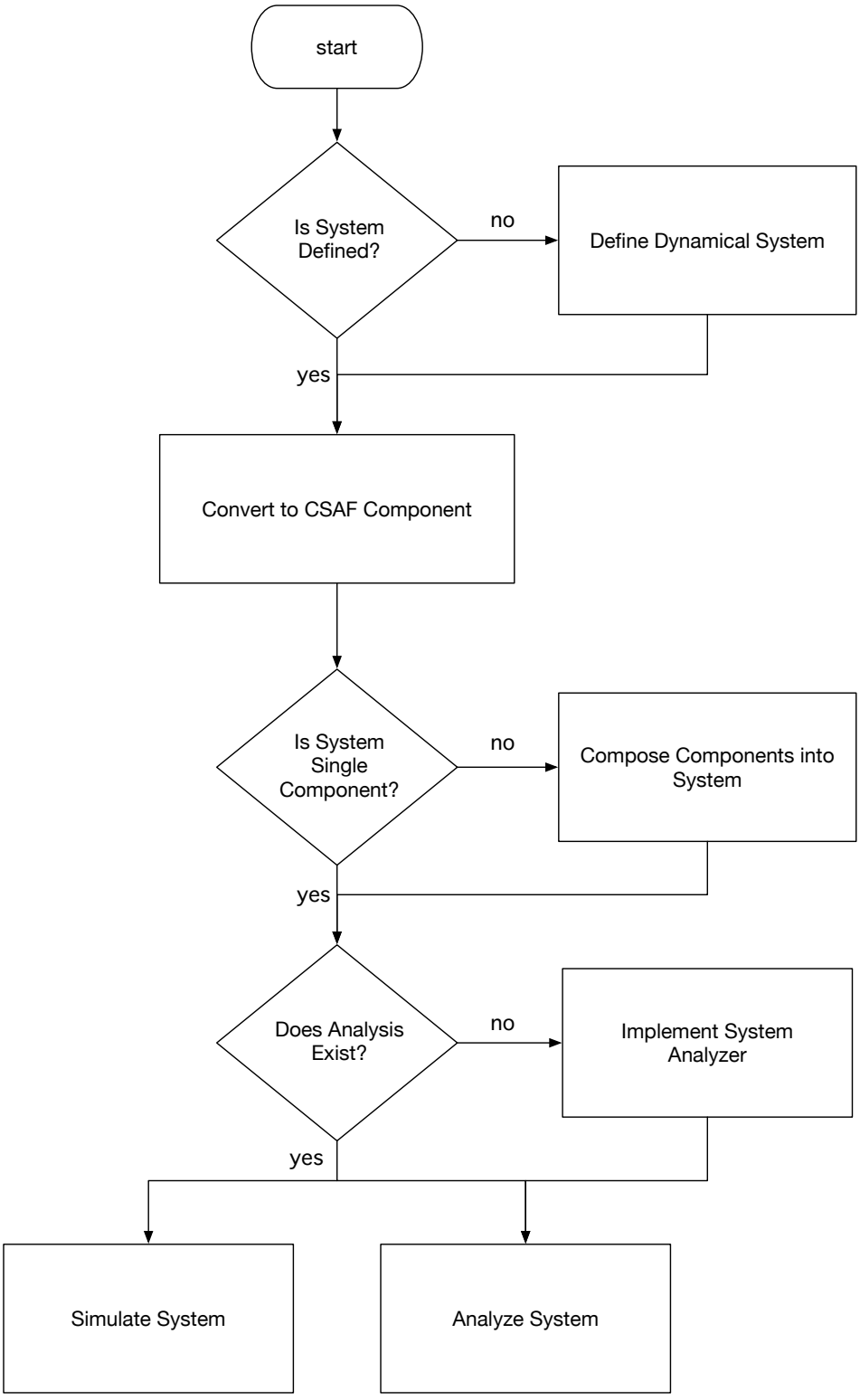

Figure 2.3: CSAF System Analysis User Workflow

2.5 Design and Implementation Constraints

CSAF is requested to support integration with deliverables from other teams in the AA project; it is required to create and accept ROS messages as a serialization method to be compatible with external systems. For visualization, the ability to communicate with FlightGear is requested to make visualization of aircraft systems possible.

2.6 User Documentation

As CSAF users will need detailed knowledge of its architecture and API to use it effectively, documentation is important. Its code repository contain a number of markdown files that outline the installation and setup. For object and function level documentation, the code uses an ample amount of Python docstrings, which can be used to autogenerate API documentation provided the code comments are suitably detailed.

API level documentation is not enough, and a user manual is available. This manual is conscience of task oriented documentation; the workflow is discussed and examples are provided for intended implementation use. Further, the use cases are also described in Jupyter notebooks, which provide an interactive platform to experiment with code.

2.7 Assumptions and Dependencies

CSAF's middleware approach can add substantial overhead to processing a closed loop system, as the serialization, transportation and deserialization over sockets can be much slower than doing everything in one application. The target system is assumed to have relatively low frequency sampling rates or short time span. If a multi-timescale system is added, the time domain simulation approach can be inefficient. If such a system were of interest in the AA project, new simulation methods may need to be investigated.

2.7.1 Limitations

No planned support for

1. closed loop continuous time controllers
2. interaction with fast/time critical systems

3 External Interface Requirements

3.1 User Interfaces

The CSAF package exists as middleware. It exists as a library, and no CLI or GUI tools are planned.

3.2 Software Interfaces

3.2.1 Mathematical Representations

A component is required to be a dynamical system in some form, a mathematical entity that has the ability to evolve in time. This means that a system assumes a function in a set $\left\{f^t\right\}_{t\in\mathbb{F}}$, where $\mathbb{F}$ is some set for time. For continuous time systems, this set can be the real numbers $\mathbb{R}$; for discrete, the integers $\mathbb{Z}$ suffices. For systems that use both discrete and continuous time, a relation between the two fields is necessary. A sampling frequency and sampling phase is used if the discrete time system is uniform time step. This general form, a system as a set of functions, is not conducive towards using in a software system, and another representation is desired.

Commonly, this set of functions can be characterized by a system of ordinary differential equations (ODEs) for continuous time or ordinary difference equations for discrete time. These systems of equations can be put into a form that permit a simple representation. For continuous time,

$$\begin{cases} \dot{x} = g(t, x, u; p) \\ f^t_{x_0} = h(., x(.), u; r) \end{cases} \tag{3.1}$$

This form separates the solution into two steps with respect to an initial condition x_0. First, all variables where the ODEs involve rates of change are grouped in a vector $x \in \mathbb{R}^N$, where it assumes an element in a state space. $u \in \mathbb{R}^M$ is an additional time varying vector, representing control input to the system. p and r are time independent parameters involved with the two functions. Then, another function h takes the space and returns the system configuration, evaluating one of the system functions. Similarly, a construction can be made for a discrete system,

$$\begin{cases} x_n = g(t, n-1, u; p) \\ f^n_{x_0} = h(., x(.), u; r) \end{cases}, \tag{3.2}$$

for the discrete time system. The evolution of the state x can be computed directly rather than integrating the system of equations defining the state differential. Further, time itself need not be directly expressed, an essential parameter for non-autonomous systems. The equivalence holds

$$\begin{bmatrix} \dot{x} \\ t \end{bmatrix} = \begin{bmatrix} g(t, x, u; p) \\ 1 \end{bmatrix} \iff \dot{x} = g(t, x, u; p). \tag{3.3}$$

However, for the sake of implementation, it is advantageous to express the time parameter directly. To use this form, all CSAF components require two functions,

$$\begin{aligned} g(., ., .; p) &: \mathbb{F} \times \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}^N \\ h(., ., .; r) &: \mathbb{F} \times \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}^D \end{aligned}. \tag{3.4}$$

For continuous time solution of a single component, the function $g(., ., u(.); p)$ is of a form that common ODE solvers can evaluate. The parameters r and p don't necessarily need to be elements of a vector space, and can be described using other structures.

Linear System Representation

A linear system can take the form,

$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx + Du \end{cases}, \tag{3.5}$$

where x is in the state space and u is in the input space. A, B, C, and D are matrices representing linear transformations.

For a controlled system, the forcing input u can be the result of a control policy. A control policy itself can be dependent of the state vector x, as is the case for a state space controller. A linear controller follows a linear control policy,

$$u = K(x_d - x) = -Ke, \tag{3.6}$$

where K is some linear operator and $e = x - x_d$ represents an error signal from a desired state x_d. This representation is commonly used, and controller design methods like LQR and H_∞ methods, produce an operator K that optimizes some notion of cost. From this view, it is clear that the controller is itself a linear system, possessing no state and an input vector e. To match the linear representation in Equation 3.5,

$$\begin{cases} \dot{w} = \mathbf{0}w + e \\ u = -Ke + Gw \end{cases} \tag{3.7}$$

The effect of w is to integrate e with respect to time. This system can represent controllers with tracking properties, like a PID controller.

Neural Controller Representation

Neural networks can be used to produce the functions g and h of a controller system, being trained beforehand and operating feedforward during use. In this, no distinction is made is made in the representation.

Fuzzy Logic Controller Representation

A fuzzy logic controller is a controller that uses a Fuzzy Logic System to determine its output. Generally, this involves taking the input and transforming them to linguistic variables $(x_i, x_2, ..., x_{n_i}) \in \mathbb{U}^{n_i}$. Then, an inference rule $\mathbb{R}^j$ can be applied from a rules base to produce a linguistic variable describing the controller output, being a non-linear mapping between two fuzzy sets $\mathbb{U} \to \mathbb{R}$.

$$\mathbb{R}^{(j)}: \text{ IF } x_1 \text{ is } A_1^j \text{ and } \ldots \text{ and } x_{ni} \text{ is } A_{ni}^j \text{ THEN } y \text{ is } B^j, j = 1, \ldots, n_j, \tag{3.8}$$

where A_i^j, B^j are fuzzy sets in $\mathbb{U}$ and $\mathbb{R}$. This set can be "defuzzified" into a crisp value $y \in \mathbb{R}$ to produce a control signal.

A controller adhering to this paradigm is stateless, and need only implement the output mapping h mentioned previously that performs the fuzzification, inference, and defuzzification. Besides this, the fuzzy sets that the input and output assume, the fuzzication/defuzzification startegies, and the inference rules employed, are necessary for the fuzzy logic controller representation.

3.2.2 Model Interfaces

A concept of a model is derived from the representations in 3.3 and 3.4. Figure 3.1 presents the IO relationships of a CSAF model. The function arguments serve as the input, and evaluations of g and h can be the output. A third output, labeled "info" is available to pass specific information about the model representation (such as the ones seen in 3.2.1 and 3.2.1). Note that the model does not use any time varying parameter as a state, requiring all such quantities having to be passed manually. The state of a dynamical system does not serve as a state represented by the programmatic object model. Time invariant parameters are states in the object and settable.

3.2.3 Component Interfaces

A component is an object that presents a model as an element of a publish/subscribe architecture, with it IO diagram presented in Figure 3.2. It subscribes to topics received by its input sockets. For normal inputs, the messages received are deserialized into a control signal that is stored into an input buffer. This buffer stores the necessary members to update a system represented by a CSAF model object. The model output is serialized into a collection of output messages, and then send over a single output socket.

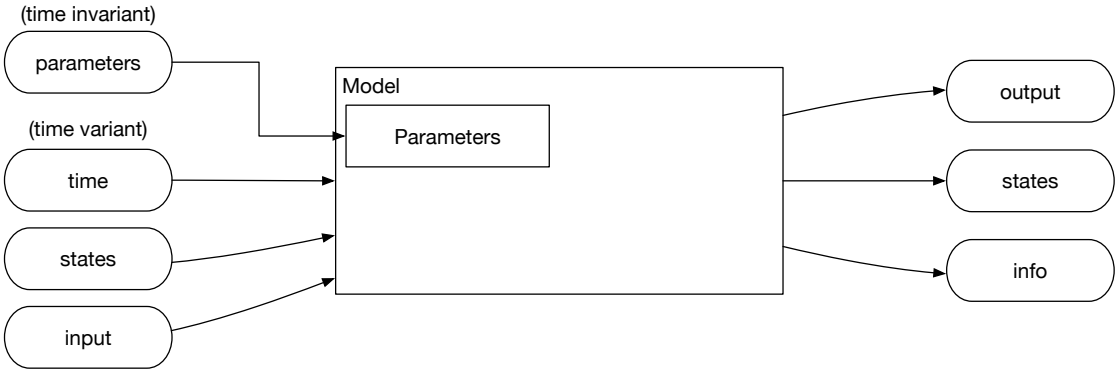

Figure 3.1: CSAF Model IO. A model requires time, state, and input to be passed in order to collect output. A model can have associated parameters that can be passed in once and set.

An events socket is also required (input c); messages can be sent in order to configure the component internals. These actions are

1. reset the buffers to default
2. configure buffer defaults
3. enable debugging
4. pause/resume the component
5. update the component
6. delete the component, unbinding/disconnecting any ports

3.2.4 System Description/Initialization

Once valid components are made, they can be composed together to make systems. This composition can be described in a TOML file (Code Block 3.3).

3.3 Communications Interfaces

Analyzers and simulators will interact with the system components via 0MQ, using sockets. 0MQ provides CSAF with a variety of levels to communicate with components, whether in-process, inter-process or across TCP and multi-cast. The component does hold state, and the system needs to be properly initialized to avoid staleness.

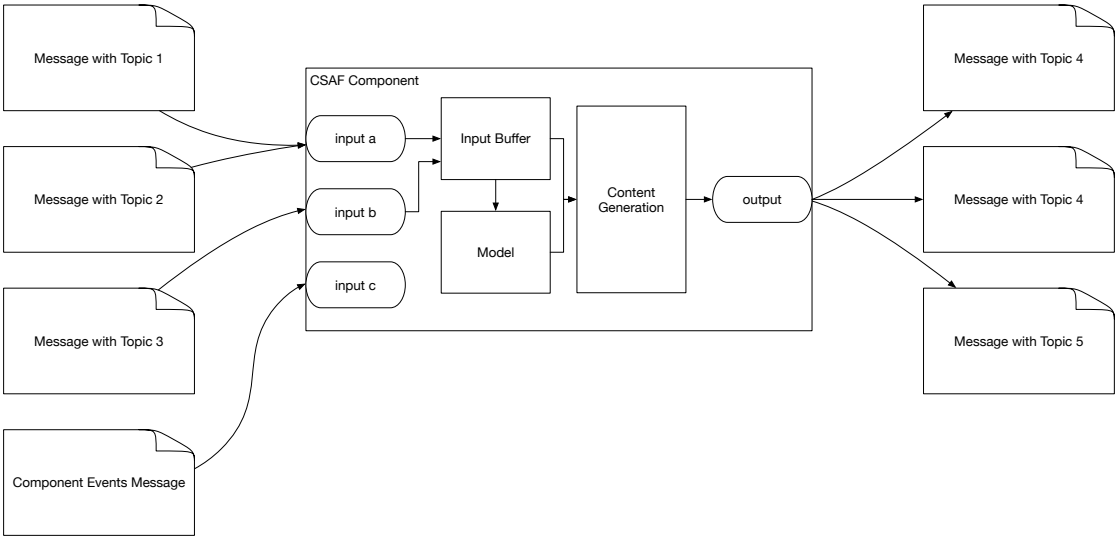

Figure 3.2: CSAF Component IO. For input, sockets receive a message with specific topics. The output is a single socket, producing messages with different topics.

3.3.1 Message Contents

The temporal messages are sent by every CSAF component that represent vectors. The message type is derived from the ROSmsgs serialization format. First, the the CSAF version is transmitted to avoid incompatibility between components made with different versions. Second, system time is included to ensure that components evaluate correctly. Next is the vector. These vectors are enumerated under a header, rather than transmitted as a contiguous array. Figure 3.5 shows an example message.

Non-temporal aspects of the system need to need to be accessed, which is described in a configuration file associated with a component. The system name, parameters, representation identifier and solver name is included. Two booleans are used to determine if the system type—continuous, discrete or hybrid. Fields specific to the representation of the system is accessible. For example, if the system is a fuzzy controller, the inference table can be visible. As the components have a varying degree of transparency ("black box"), no headers are required and parameters are allowed to unchangeable.

A controller's representation and purpose influences its requirements. As such, much variability can exist in what is contained in the message. New headers and variables are allowed to appear in a component message, but the structure has to remain constant over time.

```
name = "inverted-pendulum"

# directory setup
codec_dir = "codec"
output_dir = "output"

# order to construct/evaluate components
evaluation_order = ["maneuver", "controller", "plant"]

# log setup
log_file = "inv_pendulum.log"
log_level = "info"

# port to broadcast events to the components
events_port = 6001

[components]

  [components.controller]
    # environment to run under
    run_command = "python3"

    # path to model executable
    process = "ipcontroller.py"

    # whether to print debug diagnostics
    debug = false

    # subscribe to topic of a component (component name, topic name)
    sub = [["plant", "states"], ["maneuver", "outputs"]]

    # port to publish
    pub = 5502

  [components.plant]
    run_command = "python3"
    process = "ipplant.py"
    debug = false
    sub = [["controller", "outputs"]]
    pub = 5501

  [components.maneuver]
    run_command = "python3"
    process = "ipmaneuver.py"
    debug = false
    sub = []
    pub = 5503
```


Figure 3.3: Example TOML File Describing Components and System Compositions of Components

```
uint32 version_major
uint32 version_minor

string topic

float64 time
```

Figure 3.4: Required ROSmsg CSAF Component Temporal Message Contents (Other Fields are Permitted)

```
system_name = "Inverted Pendulum Plant"
system_representation = "black box"
system_solver = "Euler"

sampling_frequency = 100

is_discrete = false
is_hybrid = false

[parameters]
  mm = 0.5          # Mass of the cart
  m = 0.2           # Mass of the pendulum
  b = 0.1           # coefficient of friction on cart
  ii = 0.006        # Mass Moment of Inertia on Pendulum
  g = 9.8           # Gravitational acceleration
  length = 0.3      # length of pendulum

[inputs]
  msgs = ["ipcontroller_output.msg"]

[topics]

  [topics.states]
    msg = "ipplant_state.msg"
    initial = [0.0, 0.01, 0.52 , -0.01]
```

Figure 3.5: CSAF Component Configuration Information)

4 System Features

This section organizes the functional requirements for CSAF . As the package serves as a library and middleware, it is organized by tasks commonly needed for system analysis.

4.1 [CSAF -01] Component Creation

The user is able to take common model description and put them into the CSAF environment.

4.1.1 Description and Priority

This is a high priority feature. For now, emphasis is put on the integration of a F-16 model. Also, a variety of controllers need to be converted to this middleware.

4.1.2 Stimulus/Response Sequences

The user

1. identifies their model to a common model description (dynamical, linear, fuzzy, neural, etc)
2. wraps their model to interfaces required by the CSAF middleware
3. tests the new component on the CSAF platform

4.1.3 Functional Requirements

1. REQ-1 the user can transform a system written in python into a CSAF component

4.2 [CSAF -02] Time Domain Simulation

The user is able to take a CSAF system and simulate it.

4.2.1 Description and Priority

This is a high priority feature.

4.2.2 Stimulus/Response Sequences

The user

1. has a valid system of components
2. specifies initial states for all components
3. specifies a time interval AND valid conditions for the system
4. run a simulator, and gets a time trace as an output

4.2.3 Functional Requirements

1. REQ-1 the user receives a time trace as an output
2. REQ-2 the user can view the system changing in a real-time dashboard

4.3 [CSAF -03] Component Composition

The user is able to compose components to create a controlled plant system.

4.3.1 Description and Priority

This is a high priority feature.

4.3.2 Stimulus/Response Sequences

The user

1. specifies input and output signal between components
2. checks that they have valid topology using a checker

4.3.3 Functional Requirements

1. REQ-1 the user can specify inner/outer loop systems
2. REQ-2 a checker ensures that the components are correct configured before further use
3. REQ-3 the user can concatenate, re-order and split signals
4. REQ-4 the user can utilize ROS messages between components

4.4 [CSAF -04] CoPilot Support

The user is able to describe monitors in copilot, and translate them to CSAF components automatically.

4.4.1 Description and Priority

This is a high priority feature.

4.4.2 Stimulus/Response Sequences

The user

1. creates motions primitives in CoPilot
2. describes a monitor policy
3. invokes CSAF to wrap a component around the built monitor
4. associates controllers with the monitor to make a shield

4.4.3 Functional Requirements

1. REQ-1 supports CoPilot 3

4.5 [CSAF -05] Controller Monitoring and Shields

CSAF 's use case is to simulate and analyzes shields, where multiple controllers can be alternated via a monitoring policy. A monitor is a special controller component that is typically stateless and outputs a controller selection index. Also, the controllers generally accept the same input/output signatures, although that is not required.

4.5.1 Description and Priority

This is a high priority feature.

4.5.2 Stimulus/Response Sequences

The user

1. has multiple valid controllers and monitor
2. associates each controller with a position index for the monitor to select
3. configures the inputs and outputs of the monitor and all controllers

4.5.3 Functional Requirements

1. REQ-1 permits the use of recovery shields

4.6 [CSAF -06] Fuzzy Controller Representation

The user can implement controllers in fuzzy logic easily, and in a form that allows specialized fuzzy controller analysis. See Section 3.2.1 for the mathematical representation.

4.6.1 Description and Priority

This is a medium priority feature.

4.6.2 Stimulus/Response Sequences

The user

1. specifies the fuzzy variables
2. selects/implements a fuzzifier/defuzzifier
3. provides a rules inference table

4.6.3 Functional Requirements

1. REQ-1 can represent rules inference from numpy array, loaded from .npy file

4.7 [CSAF -07] Neural Controller Representation

The user can conveniently create neural controllers from neural networks described in common formats.See Section 3.2.1 for the mathematical representation.

4.7.1 Description and Priority

This is a medium priority feature.

4.7.2 Stimulus/Response Sequences

The user

1. trains a neural network
2. relates network IO to controller IO
3. the feedforward network is wrapped into a CSAF component

4.7.3 Functional Requirements

1. REQ-1 supports a network represented under TensorFlow 1.x

4.8 [CSAF -08] Time Trace Validation

The user can interact with a temporal data structure as output from a time domain simulation, checking properties.

4.8.1 Description and Priority

This is a medium priority feature.

4.8.2 Stimulus/Response Sequences

The user

1. received a time trace from simulation output
2. can check properties of the trace or collection of traces
3. can visualized variables inside the trace or collection of traces

4.8.3 Functional Requirements

None

5 Application Examples

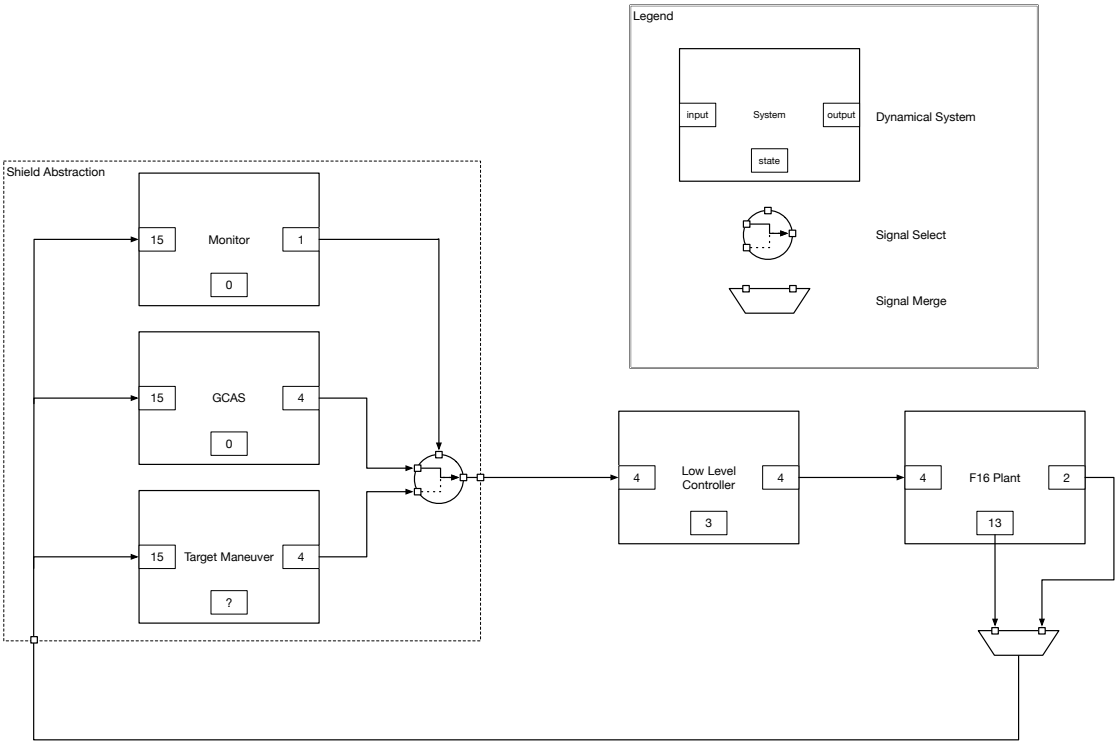

Figure 5.1: Example Shield Configuration for a Controlled F-16 Model Used in Demo

5.1 F-16 Model

The flagship use of CSAF is a benchmark F16 fighting falcon model. It serves as an advanced model to develop offline and online control assurance schemes. The hope is to provide a benchmark to motivate better verification and analysis methods, working beyond models based on Dubins car dynamics, towards the sorts of models used in aerospace engineering. Roughly speaking, the dynamics are nonlinear, have about 10-20 dimensions (continuous state variables), and hybrid in the sense of discontinuous ODEs, but not with jumps in the state.

A demonstration was created to show how the controllers can be swapped and recompose to alter flight trajectories. Figure 5.1 shows an example control system for a Ground

Collision Avoidance System shield. The model and GCAS problem is implemented here and a theoretical perspective is offered in a paper [1]. Target maneuvers are shielded by ground collision avoidance and the inner loop is using the classical controller with this configuration. For the demo, the outer loop is stateless, while the inner loop has three states for tracking. All of the system blocks are directly transferable to CSAF components.

5.1.1 Example Messages

To show translation to components, the associated message with the pub/sub elements can be formulated.

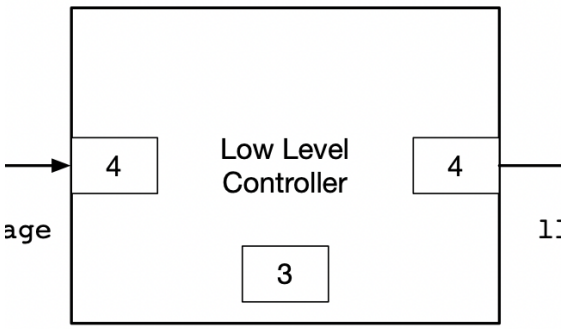

Figure 5.2: F-16 Controller

Temporal Message

state

```
uint32 version_major
uint32 version_minor

string topic

float64 time

float64 int1
float64 int2
float64 int3
```

output

```
uint32 version_major
uint32 version_minor
```

```
string topic

float64 time

float64 delta_e
float64 delta_a
float64 delta_r
float64 throttle
```

Component Configuration

```
system_name = "F16 Low Level Controller"
system_representation = "black box"
system_solver = "Euler"

sampling_frequency = 100

is_discrete = false
is_hybrid = false

[parameters]
  K_lqr = [ [-156.9,  -31.0,  -38.7, 0.0, 0.0, 0.0, 0.0, 0.0],
            [ 0.0, 0.0, 0.0, 30.5, -5.71, -9.31, -34.0, -10.7],
            [0.0, 0.0, 0.0,-22.7,-14.2, 6.74, -53.7]]
  xequil = [502.0, 0.039, 0.0, 0.0, 0.039, 0.0, 0.0, 0.0, 0.0,
                                  0.0, 0.0, 1000.0, 9.06]
  uequil = [0.139, -0.750, 0.0, 0.0]
  throttle_max = 1
  throttle_min = 0
  elevator_max = 25
  elevator_min = -25
  aileron_max = 21.5
  aileron_min = -21.5
  rudder_max = 30.0
  rudder_min = -30.0

[inputs]
  msgs = [ "f16plant_state.msg", "f16plant_output.msg",
                                  "autopilot_output.msg" ]

[topics]

  [topics.outputs]
```

```
  msg = "f16llc_output.msg"

[topics.states]
  msg = "f16llc_state.msg"
  initial = [ 0.0, 0.0, 0.0 ]
```

Bibliography

[1] P. Heidlauf, A. Collins, M. Bolender, and S. Bak, "Verification challenges in f-16 ground collision avoidance and other automated maneuvers." in *ARCH@ ADHS*, 2018, pp. 208–217.

Acronyms

0MQ zeroMQ. 5, 14

AA Assured Autonomy. 6, 8, 10

API Application Programming Interface. 7, 10

CSAF Control System Analysis Framework. 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 23, 24

eDSL Embedded Domain Specific Language. 6

FLS Fuzzy Logic System. 13

GCAS Ground Collision Avoidance System. 6, 23, 24

H_∞ H-infinity ("Hardy Space"). 12

IEEE Institute of Electrical and Electronics Engineers. 6

LEC Learning Enabled Controller. 6

LQR Linear Quadratic Regulator. 12

ODE Ordinary Differential Equation. 11, 12, 23

PID Proportional Integral Derivative. 12

ROS Robot Operating System. 5, 10, 15, 17, 19