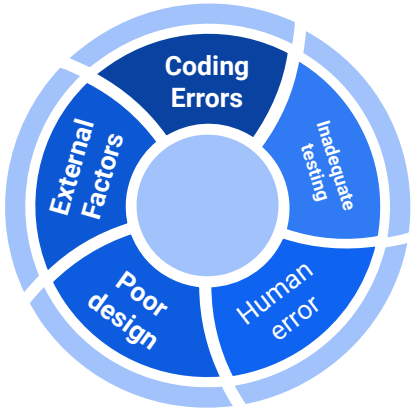
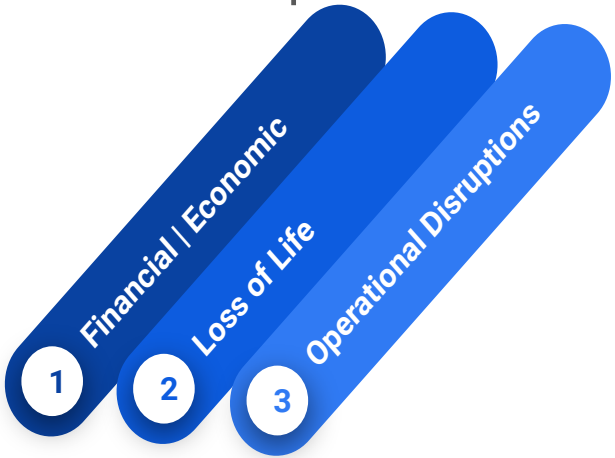
Software Failures

What

Why



Impact



Notable Software Failures and Their Impact

2024

2024 Crowdstrike Incident

CrowdStrike experienced a catastrophic IT outage caused by a faulty update. This update led to a critical logic error. The error triggered a Blue Screen of Death (BSOD) on millions of Windows machines, rendering them inoperable.

- Over 8.5 million devices affected.
- Economic losses over tens of billions of dollars.

2012

Knight Capital Group's trading loss

A software bug in the trading system of Knight Capital Group caused erratic trading behavior. The bug was introduced due to the deployment of untested software on the production system.

- lead to a $440 million loss within 45 minutes.

2010

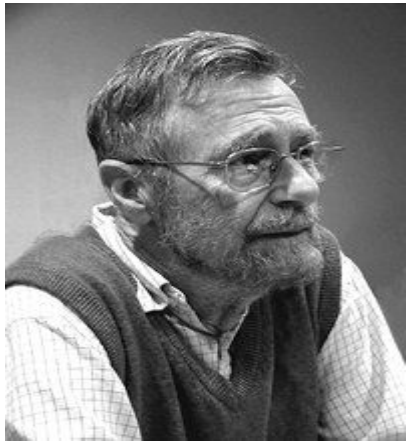
Toyota's Unintended Acceleration

Software defects in Toyota's Electronic Throttle Control System were linked to unintended acceleration incidents, leading to multiple accidents and fatalities.

- Massive recalls
- Over $1 billion in fines and settlements.
- Bodily harm to individuals involved in accidents,

How Can We Guarantee Software Reliability?

? Testing



"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." — Edsger W. Dijkstra

How Can We Guarantee Software Reliability?

Mathematics to the Rescue

What is Formal Program Verification?

Mathematical methods to prove the correctness of algorithms.

- guarantees correctness.
- reduces the risk of failure

Techniques usually involve:

- ➔ Theorem proving
- ➔ Model checking

Axiomatic Program Verification & Hoare's Logic

Axiomatic Program Verification (APV) is a formal method for proving the correctness of computer programs. The goal of APV is to establish, through mathematical proofs, that a program will function correctly according to its specifications.

Key Concept:

- **Formal Specifications**
- **Assertions**
- **Proof Obligations**

Axiomatic Program Verification & Hoare's Logic

Formal Specifications:

The Formal specifications define what the program is supposed to do. These specifications are expressed in logical assertions, typically written in a formal language. They describe the preconditions (assumptions before execution) and postconditions (expected outcomes after execution) of the program's operations.

Axiomatic Program Verification & Hoare's Logic

Assertions:

Assertions are logical statements placed at various points in the program to assert certain truths about the program's state at that point. The most common types of assertions in APV are:

- **Preconditions**: Conditions that must hold true before the execution of a program or a specific operation within it.
- **Postconditions**: Conditions that must hold true after the execution of a program or a specific operation.
- **Invariants**: Conditions that must hold true at certain points during the execution of a program, typically within loops.

Axiomatic Program Verification & Hoare's Logic

Proof Obligations

APV involves generating proof obligations, which are logical formulas that must be proven true to verify that a program meets its specifications. These proofs typically involve demonstrating that if the preconditions are met, then the postconditions will be satisfied after the program executes.

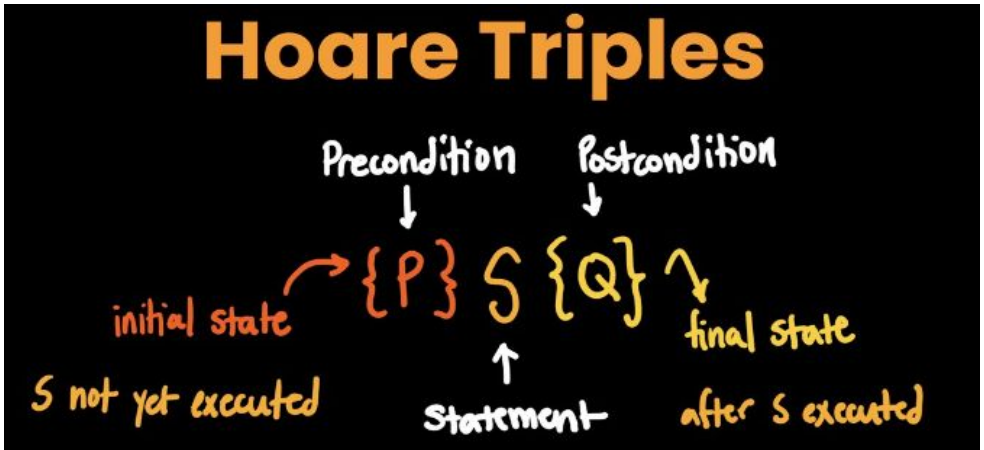
Hoare Logic as a Framework for APV

Hoare Logic is one of the most prominent frameworks used in Axiomatic Program Verification. It was introduced by British computer scientist Tony Hoare in the 1969 and remains a foundational approach to reasoning about program correctness.

Overview of Hoare Logic

Hoare Triples: The central concept in Hoare Logic is the **Hoare Triple**, which is a formal notation used to describe the behavior of a program or program fragment

The interpretation of a Hoare Triple is: "If the precondition **{ P }** is true before executing command **S**, then the postcondition **{ Q }** will be true after **S** has been executed."



Overview of Hoare Logic [Identifying Valid Hoare Triples]

1. $\{x > 4\}\ x:=x+3\ \{x>5\}$

Where:
E: $x+3$
Q: Post condition
To prove the validity, we try,

$\{Q[x:=E]\} =$ $x:=E\{Q\}$ (i.e we apply the assignment statement to the post condition).
$\{Q[x:=E]\} =$ $(x>5)[x:=x+3]$
$\{Q[x:=E]\} =$ $(x+3)>5$
$\{Q[x:=E]\} =$ $x>5-3$
$\{Q[x:=E]\} =$ $x>2$

⇒ since our answer is $(x>2)$, and $(x>4)$ satisfy the condition $(x>2)$, then it implies that it is a valid hoare triple.

Overview of Hoare Logic [Identifying Valid Hoare Triples]

2. {x≥0} x:=x-1 {x>0}

Where:
E: x-1
Q: Post condition = {x>0}

To prove the validity, we try,

{Q[x:=E]} = x:=E{Q} (i.e we apply the assignment statement to the post condition).
{Q[x:=E]} = (x>0)[x:=x-1]
{Q[x:=E]} = (x-1)>0
{Q[x:=E]} = x>0 +1
{Q[x:=E]} = x>1

Is this a valid hoare triple??

Does the condition x≥0 satisfy the condition x>1,

Recall that if (x≥0) it implies that (x=0 OR x>0)

What happens when x is a number between 0 and 1? (i.e if x=0)

x=0 does not satisfy the condition x>1, therefore it is not a valid hoare triple.

Overview of Hoare Logic [Identifying Valid Hoare Triples]

Overview of Hoare Logic [Reasoning About Hoare Triples]

Overview of Hoare Logic

Rules of Inference

Hoare Logic provides a set of rules (Axioms) of inference that allow for the construction of proofs of program correctness. These rules describe how to combine Hoare Triples and how to reason about different programming constructs, such as assignment, sequencing, conditionals, and loops

Overview of Hoare Logic (Assignment Axiom)

If you assign a value to a variable, the precondition must reflect how this assignment affects the program state.

$$\{Q(e)\}\, x := e\, \{Q(x)\} \text{ is valid.}$$

x has the value after execution that e has before, so $Q(x)$ is true after iff $Q(e)$ is true before.

Overview of Hoare Logic (Sequence Axiom)

For a sequence of commands S1 and S2, the postcondition of S1 should be the precondition of S2.

$$\frac{\{P\}\ S_1\ \{R\},\ \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1;\ S_2\ \{Q\}}$$

In general:

- if you know $\{P\}\ S_1\ \{R\}$ and
- you know $\{R\}\ S_2\ \{Q\}$
- then you know $\{P\}\ S_1;\ S_2\ \{Q\}$.

(So, to prove $\{P\}\ S_1;\ S_2\ \{Q\}$, find $\{R\}$.)

Overview of Hoare Logic (Sequence Axiom)

```python
def variable_swapper(variable_a: int, variable_b: int) -> tuple[int, int]:
    """Swap two variables."""

    temp = variable_a
    variable_a = variable_b
    variable_b = temp

    return variable_a, variable_b
```

We expect at the end that

P : {variable_a = 10 ^ variable_b = 20}

Q : {variable_a = 20^ variable_b = 10}

But there are 3 expression how do we prove these sequence of expression

```python
temp = variable_a
variable_a = variable_b
variable_b = temp
```

Overview of Hoare Logic (Sequence Axiom)

S1: `temp = variable_a`

S2: `variable_a = variable_b`

S3: `variable_b = temp`

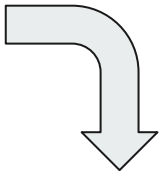
We can rewrite Hoare triple thus

{P} S1; S2; S3 {Q}

{P} S1 {R}
{R} S2 {T}
{T} S3 {Q}

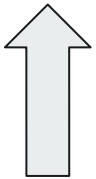
In order to prove {P} S1; S2; S3 {Q} We need to find some {R}, {T} such that they satisfy

Let's work backwards



Overview of Hoare Logic (Sequence Axiom)

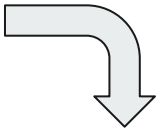
Step1 -> {variable_a = 10 ^ variable_b = 20} temp = variable_a { R }
Step2 -> { R } variable_a = variable_b { T }
Step3 -> { T } variable_b = temp {variable_a = 20 ^ variable_b = 10}



Walking backwards let find a precondition **{T}** that satisfies the post condition {**variable_a = 20 ^ variable_b = 10**}

Since S3 says **variable_b = temp** and the post-condition { Q } implies that **variable_b = 10** there for **temp = 10**

Hence we can get a precondition
{**T**} = {**variable_a = 20 ^ temp = 10**}



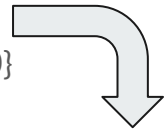
Overview of Hoare Logic (Sequence Axiom)

Step1 -> {**variable_a = 10 ^ variable_b = 20**} **temp = variable_a** { **R** }
Step2 -> { **R** } **variable_a = variable_b** {**variable_a = 20 ^ temp = 10**}
Step3 -> {**variable_a = 20 ^ temp = 10**} **variable_b = temp** {**variable_a = 20 ^ variable_b = 10**}

Next we need to find a precondition **{R}** that satisfies the post condition {**variable_a = 20 ^ temp = 10**}

Since S2 says **variable_a = variable_b** and the post-condition { T } implies that **variable_a = 20** and **temp = 10.** Substitting the information of the post condition into S2 we can extrapolate that **variable_b = 20**

Hence we can get a pre-condition {R}: {**variable_b = 20 ^ temp = 10**}



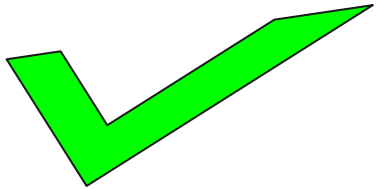
Overview of Hoare Logic (Sequence Axiom)

Step1 -> **{variable_a = 10 ^ variable_b = 20} temp = variable_a {variable_b = 20 ^ temp = 10}**
Step2 -> {**variable_b = 20 ^ temp = 10**} **variable_a = variable_b** {**variable_a = 20 ^ temp = 10**}
Step3 -> {**variable_a = 20 ^ temp = 10**} **variable_b = temp** {**variable_a = 20 ^ variable_b = 10**}

Lastly we show that **{ R }** implies **{ P }** i.e **{ P }** is a valid precondition for **{ R }**

{R} : {**variable_b = 20 ^ temp = 10**}
Since S1: **temp = variable_a** therefore
{**variable_b = 20 ^ variable_a = 10**}
I.e we can rearrange to {**variable_a = 10 ^ variable_b = 20**} : { **P** }

Which means
{R} => {P} i . e {R} implies { P }



PROVEN

Overview of Hoare Logic (Conditional Axiom)

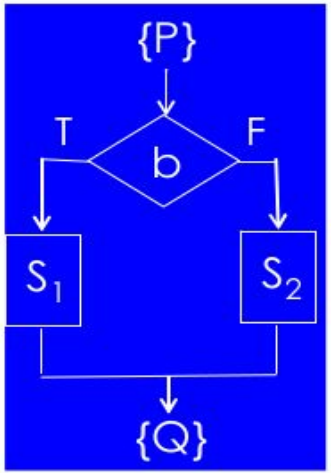
For conditionals, you must prove that the postcondition holds for both branches of the conditional.

S1

$$\frac{\{P \wedge B\}\, C_1\, \{Q\} \qquad \{P \wedge \neg B\}\, C_2\, \{Q\}}{\{P\}\ \text{if}\ B\ \text{then}\ C_1\ \text{else}\ C_2\ \{Q\}}$$

S2

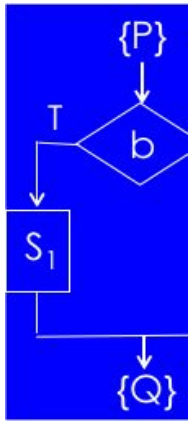
B is the boolean condition



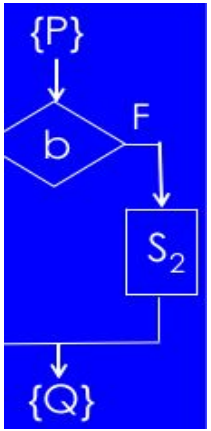
Overview of Hoare Logic (Conditional Axiom)

How do we use the conditional axiom to prove the correctness of a conditional statement ?

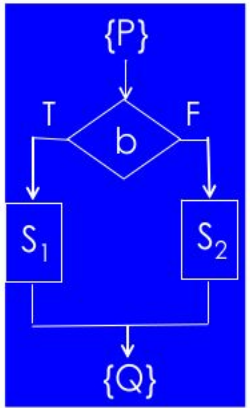
Prove that **{B ∧ P} S1 {Q}**



Prove that **{¬B ∧ P} S2 {Q}**



Conclude **{P}** if **B** then **S1** else **S2** **{Q}**



Overview of Hoare Logic (Conditional Axiom)

```python
def apply_discount(purchase_amount: int) -> float | int:
    """Apply discount of 10% for amounts greater than 100."""

    if purchase_amount > 100:
        discount = 0.1 * purchase_amount
    else:
        discount = 0
    result = purchase_amount - discount
    return result
```

Can you Identify the pre and post conditions of the program ?

Pre-condition **{P}**: **{purchase_amount ≥ 0}**
Post Condition **{Q}**: **{0 ≤ result ≤ purchase_amount}**
True branch **S1**: `discount = 0.1 * purchase_amount`
False branch **S2**: `discount = 0`
Boolean condition **B:** `purchase_amount > 100`

Recall In order to prove the validity of the code snippet we need to prove the validity of

1. **{B ^ P} S1 {Q}**
{purchase_amount > 100 ^ purchase_amount ≥ 0} discount = 0.1 * purchase_amount **{0 ≤ result ≤ purchase_amount}**
2. **{¬B ^ P}** S2 **{Q}**
{purchase_amount ≤ 100 ^ purchase_amount ≥ 0} discount = 0 **{0 ≤ result ≤ purchase_amount}**

Overview of Hoare Logic (Conditional **Axiom**)

Lets Prove

{B ^ P} S1 {Q}

{purchase_amount > 100 ^ purchase_amount ≥ 0}
discount = 0.1 * purchase_amount**;**

{0 ≤ result ≤ purchase_amount}

In-order to Prove **{B ^ P} S1 {Q}**, we assume **B** is true. Assuming P holds true then

Overview of Hoare Logic (Loop Invariant Rule)

For loops, you must find an invariant III that holds before and after each iteration.

$$\frac{\{I \wedge B\}\, C\, \{I\}}{\{I\}\ \text{while}\ B\ \text{do}\ C\ \{I \wedge \neg B\}}$$

Here, III is the loop invariant, and BBB is the loop guard.

Other Frameworks of APV

Dynamic Logic

- **Overview**: Dynamic Logic (DL) is a modal logic that extends classical logic by incorporating actions or programs. It is used to reason about the behavior of programs within a logical framework.
- **Application**: In DL, you can express properties of programs, such as "if a program terminates, a certain property will hold" or "after executing a program, a specific condition will always be true."
- **Key Concepts**:
    - **Program Modality**: DL introduces modalities $[P]$ and $\langle P \rangle$, where P is a program. $[P]\varphi$ means "after every possible execution of P, the formula φ holds." $\langle P \rangle \varphi$ means "there exists an execution of P after which φ holds."
    - **Syntax and Semantics**: The syntax and semantics of DL are more expressive than Hoare Logic, allowing reasoning about partial correctness, total correctness, and termination of programs.

Hoare Logic vs Dynamic Logic

Expressiveness	While Hoare Logic uses triples {P}C{Q} to reason about preconditions and postconditions {x > 0} y := x*x* *{y > 0}*	DL incorporates program modalities directly into the logical formulas. *x > 0 → [y :=* xx](y > 0) "if x is greater than 0, then after executing y := x*x, y will be greater than 0."
Reasoning	In Hoare Logic, termination is typically handled separately from partial correctness.	DL can express termination properties more directly using the ⟨P⟩ operator. *⟨P⟩true ∧ [P](x > 0)* “P always terminates and results in x being positive”

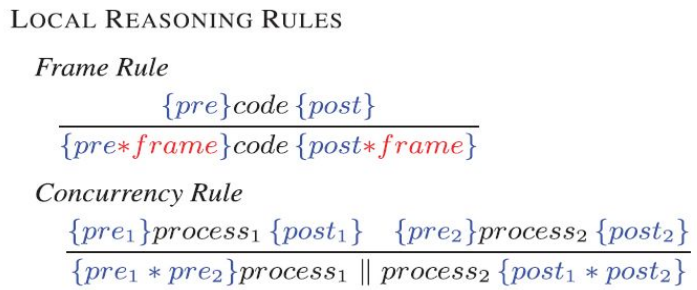
Hoare Logic vs Dynamic Logic

Partial and total correctness	In Hoare Logic, different proof systems are typically used for partial and total correctness.	DL can distinguish between partial and total correctness more easily within the same framework. *Partial correctness: $[P]\varphi$* *Total correctness: $\langle P\rangle true \wedge [P]\varphi$*
Modalities	DL introduces modal operators [P] and ⟨P⟩, allowing for more nuanced reasoning about program behavior.	Hoare Logic uses triples {P}C{Q}, which are less flexible in expressing complex program properties.

Other Frameworks of APV cont.d

Separation Logic

- **Overview**: Separation Logic is an extension of Hoare Logic designed to reason about programs that manipulate pointers and dynamically allocated memory.
- **Application**: It is particularly useful for verifying properties of programs that involve shared mutable data structures, such as linked lists, trees, or graphs.
- **Key Concepts**:
  - **Separating Conjunction (*)**: A key feature of Separation Logic is the separating conjunction, which allows reasoning about disjoint parts of memory. If `P * Q` holds, it means that `P` and `Q` hold for separate, non-overlapping parts of memory.
  - **Frame Rule**: The Frame Rule allows local reasoning, meaning you can reason about a small part of the program (and its corresponding part of memory) independently of the rest of the program.



Hoare Logic vs Separation Logic

Focus and Scope	Primarily designed for reasoning about sequential programs with simple variables.	Specifically tailored for reasoning about programs that manipulate heap memory and use pointers.
Handling of Memory	Struggles with complex memory structures and aliasing.	Excels at reasoning about shared mutable data structures.

Example:

Consider a simple linked list manipulation:

Hoare Logic would struggle to express and verify properties about this function due to the dynamic memory allocation and pointer manipulation.
Separation Logic, however, can handle this elegantly:

Here, `ls(head, NULL)` represents a linked list from `head` to `NULL`, and the postcondition states that we now have a linked list ending with a new node containing `new_data`.

```
{ls(head, NULL)}
  append(head, new_data)
{∃x. ls(head, x) * x ↦ (new_data, NULL)}
```

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def append(head, new_data):
    new_node = Node(new_data)

    if head is None:
        return new_node

    last = head
    while last.next is not None:
        last = last.next

    last.next = new_node
    return head
```

Hoare Logic vs Separation Logic

Modularity and Local Reasoning	Requires considering the entire program state.	Allows for local reasoning through the Frame Rule.
Expressiveness for Pointer Manipulation	Limited in expressing complex pointer relationships.	Provides intuitive ways to describe pointer-based structures.

Example:

Example of the power of Separation Logic: Consider verifying a function that swaps the contents of two memory locations:

```
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

In Separation Logic, we can specify this concisely:

```
{x -> a * y -> b}
swap(x, y)
{x -> b * y -> a}
```

This specification clearly expresses that x and y point to separate memory locations (thanks to *), and their contents are swapped. Proving this correct in traditional Hoare Logic would be more verbose and less intuitive.

Real world applications of APV

1. Rudder Control function.

As shown in the diagram, rudder pedals are part of the flight control system.

The rudder controls the yaw motion of the aircraft, which is the L or R movement of the aircraft's nose.

A rudder pedal is a foot-operated aircraft flight control interface for controlling the rudder of an aircraft. The usual set-up in modern aircraft is that each pilot has a pedal set consisting of a pair of pedals, with one pedal for each foot.

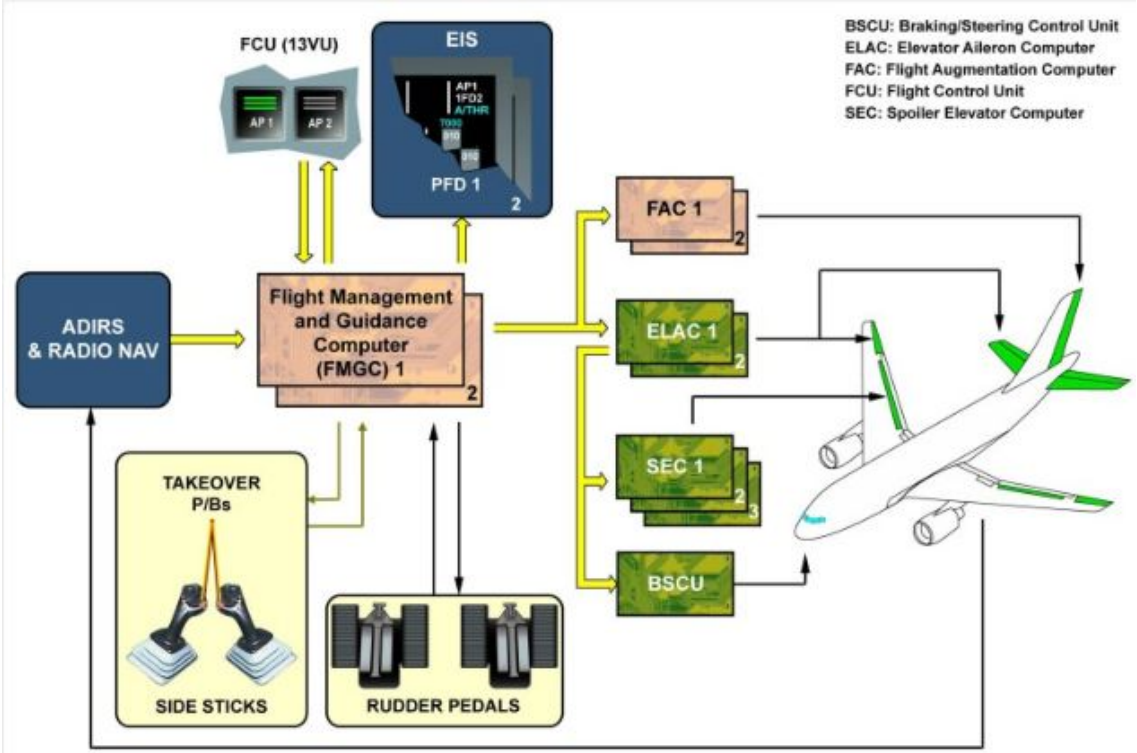


Figure: Diagram of an aircraft flight control system

Real world applications of APV

1. Ruder Control Function (contd.)

We focused on the rudder pedal because of the following important reasons:

1. From the definition provided earlier, the rudder is a key part of the flight control system.
2. Rudder control is crucial for aircraft stability, making it a safety-critical system that requires rigorous verification.
3. The rudder control function provides a clear and simple example to illustrate axiomatic program verification principles.
    a. The rudder's well-defined movement limits offer clear preconditions and postconditions for verification.
    b. This rudder example demonstrates concepts applicable to verifying other flight control surfaces and integrated systems.
    c. The rudder, as a continuous control system, showcases the application of axiomatic verification to real number inputs and outputs.

Real world applications of APV

Step-by-step breakdown of the axiomatic program verification process in Ruder Control Function:

1. Formal Specification:

Function: `setRudderAngle(angle: float) → void`

Precondition:

```
-30.0 ≤ angle ≤ 30.0  // Rudder angle limited to ±30 degrees
```

Postcondition:

```
currentRudderAngle = angle

-30.0 ≤ currentRudderAngle ≤ 30.0
```

Real world applications of APV

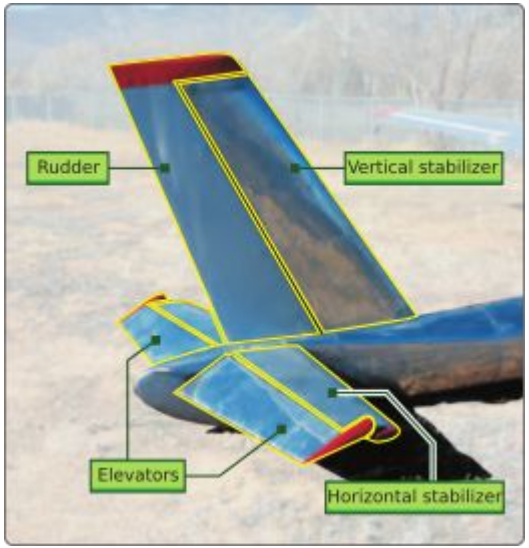
Step-by-step breakdown of the axiomatic program verification process in Ruder Control Function:

2. Implementation:

```
current_rudder_angle = 0.0

def set_rudder_angle(angle: float) -> None:
    global current_rudder_angle

    if angle < -30.0:
        current_rudder_angle = -30.0
    elif angle > 30.0:
        current_rudder_angle = 30.0
    else: current_rudder_angle = angle
```



Real world applications of APV

Step-by-step breakdown of the axiomatic program verification process in Ruder Control Function:

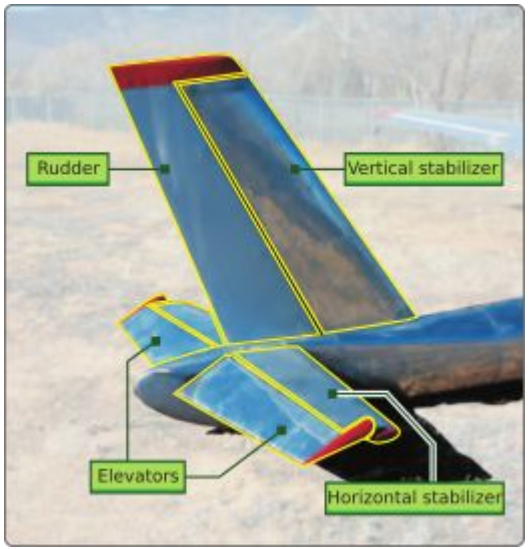
3. Axioms and Inferences Rules:

Axiom 1: For any real number x,
if x < -30.0, then max(-30.0, min(30.0, x)) = -30.0

If the input value x is less than -30.0, the function will return -30.0. This ensures that the rudder does not exceed its lower limit.

Axiom 2: For any real number x,
if x > 30.0, then max(-30.0, min(30.0, x)) = 30.0

If the input value x is greater than 30.0, the function will return 30.0. This ensures that the rudder does not exceed its upper limit.



Real world applications of APV

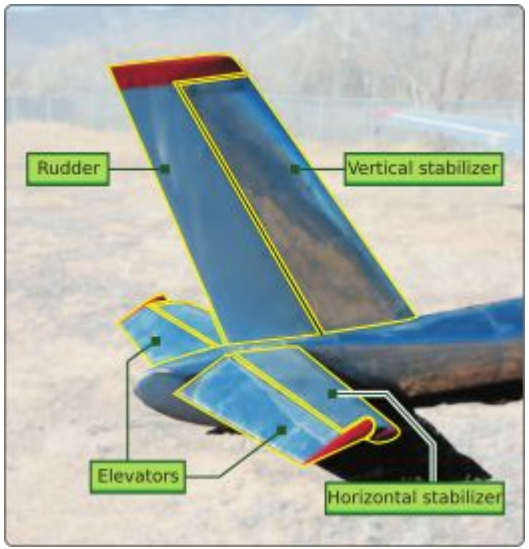
Step-by-step breakdown of the axiomatic program verification process in Ruder Control Function:

3. Axioms and Inferences Rules:

Axiom 3: For any real number x,
if -30.0 ≤ x ≤ 30.0, then max(-30.0, min(30.0, x)) = x

If the input value x is within the range of -30.0 to 30.0, the function will return xxx itself. This allows the rudder to operate normally within its defined range.

Inference Rule:
If P *implies* Q *and* Q *implies* R, *then* P *implies* R
I.e. If P => Q and Q => R, then P => R *(transitivity of implication)*



Real world applications of APV

Step-by-step breakdown of the axiomatic program verification process in Ruder Control Function:

4. Proof:

1. **Precondition:** -30.0 ≤ angle ≤ 30
2. **Case analysis:**
   a. **If angle < -30.0:**
      By the if statement, currentRudderAngle = -30.0
      This satisfies, -30.0 ≤ currentRudderAngle ≤ -30.0
   b. **If angle > 30.0:**
      By the else if statement, currentRudderAngle = 30.0
      This satisfies -30.0 ≤ currentRudderAngle ≤ 30.0
   c. **If -30.0 ≤ angle ≤ 30.0:**
      By the else statement, currentRudderAngle = angle
      Given the precondition, this satisfies
      -30.0 ≤ currentRudderAngle ≤ 30.0

- In all cases, -30.0 ≤ currentRudderAngle ≤ 30.0 is satisfied
- The assignment currentRudderAngle = angle (or its bounded version) ensures that currentRudderAngle equals the input angle (within the allowed range)
- Therefore, both postconditions are satisfied.

Future Directions: Current Research Trends

Scalability: Researchers are working on methods to apply APV to larger, more complex systems. This includes developing more efficient proof strategies and automated reasoning techniques.

Concurrency: As multi-threaded and distributed systems become more prevalent, there's increased focus on verifying concurrent programs using axiomatic methods.

Real-time systems: Applying APV to real-time and cyber-physical systems, where timing constraints are critical.

Domain-specific verification: Developing specialized logics and proof techniques for specific domains like cryptography, blockchain, or autonomous systems.

Integration with other Formal methods

Hybrid approaches: Combining APV with model checking to leverage the strengths of both methods. For example, using model checking for exploring state spaces and APV for proving properties about those states.

Abstract interpretation: Integrating abstract interpretation techniques with APV to handle more complex program properties and improve scalability.

Theorem provers: Enhancing the integration between APV and automated theorem provers to increase the level of automation in proofs.

Machine learning and AI in program verification

Automated proof generation: Using machine learning to guide proof search and generate proof steps automatically.

Property inference: Applying AI techniques to infer program properties, invariants, and specifications, which can then be verified using APV.

Proof assistance: Developing AI-powered tools that can assist human verifiers by suggesting proof strategies or identifying relevant axioms and lemmas.

Potential impact on software development practices

Continuous verification: Integrating APV into continuous integration/continuous deployment (CI/CD) pipelines to catch bugs earlier in the development process.

Industry adoption: As APV tools become more user-friendly and integrated into development environments, we may see increased adoption in industry, particularly for safety-critical systems.

Regulatory compliance: APV could play a larger role in demonstrating compliance with safety and security standards in regulated industries like aerospace, automotive, and healthcare.

Smart contract verification: With the rise of blockchain technology, APV could become crucial for verifying the correctness and security of smart contracts